Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)

Harald Störrle, Goetz Botterweck, Michel Bourdellès,
Dimitris Kolovos, Richard Paige, Ella Roubtsova,
Julia Rubin, Juha-Pekka Tolvanen (Eds.)

Joint Proceedings

Co-located Events at the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)

Harald Störrle, Goetz Botterweck, Michel Bourdellès, Dimitris Kolovos, Richard Paige, Ella Roubtsova, Julia Rubin, Juha-Pekka Tolvanen (Eds.)

Editors

Harald Störrle
Technical University of Denmark (DTU)
Richard Petersens Plads, 322.024
DK-2800 Kongens Lyngby
hsto@imm.dtu.dk

Goetz Botterweck
Michel Bourdellès
Dimitris Kolovos
Richard Paige
Ella Roubtsova
Julia Rubin
Juha-Pekka Tolvanen

ISBN: 978-87-643-1014-6
Publisher: Technical University of Denmark (DTU)
Printed by DTU Informatics
Technical University of Denmark (DTU)
Building 321, DK-2800 Kongens Lyngby
Copenhagen, Denmark
reception@imm.dtu.dk
www.imm.dtu.dk
2011

Preface

We are very glad to welcome you all at the Technical University of Denmark (DTU) in Kongens Lyngby for the events co-located with the 8th European Conference on Modelling Foundations and Applications.

Despite the economic downturn, we have a very strong program this year again, with six workshops and two tutorials. Among the workshops, we have two repeat workshops (BMFA and PMDE) that run for the fourth, and second consecutive time, respectively. We also have a healthy dose of new workshops picking up emerging trends and topics, like GMLD, ACME, and CloudMDE. Altogether, these workshops received 39 paper submissions, of which 23 were accepted, yielding an acceptance rate of 59%.

Furthermore, we also have a workshop providing an overview of academic-industrial collaboration projects in the area of Real-Time and Embedded Modelling (EIAC-RTESMA) with eight presentations, six tool demonstrations and four posters.

In the true spirit of the word "Workshop", the events co-located to ECMFA are working sessions, that is, they are intended as forums for constructive discussion, collegial criticism, and scientific openness. Together with the more classic layout of the main ECMFA conference, we believe this is an excellent way of promoting the science and practice of model based software development.

Thank you all for contributing, and thank you for joining us in Kongens Lyngby. We hope you enjoy ECMFA 2012 and all its co-located events!

July 2012

Harald Störrle
Goetz Botterweck
Michel Bourdellès
Dimitris Kolovos
Richard Paige
Ella Roubtsova
Julia Rubin
Juha-Pekka Tolvanen

Table of Contents

First International Workshop on Model-Driven Engineering for and in the Cloud

CloudMDE 2012
(co-located with ECMFA 2012)

Proceedings

2 July 2012

DTU Lyngby, Denmark

Editors: Richard Paige, Jordi Cabot, Marco Brambilla, Marsha Chechik, Parastoo Mohagheghi

Preface

The first workshop on Model-Driven Engineering (MDE) for and in the Cloud was held on 2 July 2012 at DTU Lyngby, Denmark, co-located with the 8th European Conference on Modelling: Foundations and Applications (ECMFA) 2012. Model Driven Engineering (MDE) elevates models to first class artefacts of the software development process. MDE principles, practices and tools are also becoming more widely used in industrial scenarios. Many of these scenarios are traditional IT development and emphasis on novel or evolving deployment platforms has yet to be seen. Cloud computing is a computational model in which applications, data, and IT resources are provided as services to users over the Internet. Cloud computing exploits distributed computers to provide on-demand resources and services over a network (usually the Internet) with the scale and reliability of a data centre.

Cloud computing is enormously promising in terms of providing scalable and elastic infrastructure for applications; MDE is enormously promising in terms of automating tedious or error prone parts of systems engineering. There is potential in identifying synergies between MDE and cloud computing. The workshop aimed to bring together researchers and practitioners working in MDE or cloud computing, who were interested in identifying, developing or building on existing synergies. The workshop focused on identifying opportunities for using MDE to support the development of cloud-based applications (MDE for the cloud), as well as opportunities for using cloud infrastructure to enable MDE in new and novel ways (MDE in the cloud). Attendees were also interested in novel results of adoption of MDE in cloud-related domains, as well as work-in-progress or experience reports, that provide insight into early adoption of MDE for building cloud-based applications, or in terms of deploying MDE tools and infrastructure on 'the cloud'.

The workshop received 10 paper submission (technical papers, position papers and work-in-progress papers), from which it accepted 6 for presentation at the workshop. Each paper was reviewed by 2-3 members of the program committee, and was selected based on its suitability for the workshop, novelty, likelihood of sparking discussion, and general quality. The workshop also featured a keynote presentation by Muhammad Ali Babar (ITU Copenhagen, Denmark) on migration to the cloud. The organisers thank all authors for submitting papers, our keynote speaker Ali Babar, the workshop participants, the ECMFA local organisation team, the workshop chair Harald Störrle, and the program committee for their support.

Workshop Organisers: Richard Paige (University of York, UK), Jordi Cabot (AtlanMod, École des Mines de Nantes, France), Marco Brambilla (Politecnico di Milano, Italy), Marsha Chechik (University of Toronto, Canada) and Parastoo Mohagheghi (ICT at NAV, Norway)

Program Committee: Danilo Ardagna, Aldo Bongio, Radu Calinescu, Marcos Didonet Del Fabro, Federico Facca, Xavier Franch, Esther Guerra, Sebastian Mosser, Alek Radjenovic, Louis Rose, Manuel Wimmer

Transforming Very Large Models in the Cloud: a Research Roadmap

Cauê Clasen[1], Marcos Didonet Del Fabro[2], and Massimo Tisi[1]

[1] AtlanMod team, INRIA - École des Mines de Nantes - LINA, Nantes, France
{caue.avila_clasen, massimo.tisi}@inria.fr
[2] C3SL labs, Universidade Federal do Paraná, Curitiba, PR, Brazil
marcos.ddf@inf.ufpr.br

Abstract. Model transformations are widely used by Model-Driven Engineering (MDE) platforms to apply different kinds of operations over models, such as model translation, evolution or composition. However, existing solutions are not designed to handle very large models (VLMs), thus facing scalability issues. Coupling MDE with cloud-based platforms may help solving these issues. Since cloud-based platforms are relatively new, researchers still need to investigate if/how/when MDE solutions can benefit from them. In this paper, we investigate the problem of transforming VLMs in the Cloud by addressing the two phases of 1) model storage and 2) model transformation execution in the Cloud. For both aspects we identify a set of research questions, possible solutions and probable challenges researchers may face.

1 Introduction

Model transformation is a term widely used in Model-Driven Engineering (MDE) platforms to denote different kinds of operations over models. Model transformation solutions are implemented in general purpose programming languages or transformation-specific (often rule-based) languages such as ATL [8], Epsilon [10], or QVT [11]. These solutions access and manipulate models using existing model management APIs, such as the Eclipse Modeling Framework API, EMF [2]. Current model transformation solutions are not designed to support very large models (VLMs), i.e., their performances in time and memory quickly degrade with the growth of model size, as already identified in previous works[9].

Moving model transformation tools to a cloud may bring benefits to MDE platforms. In a cloud, a large amount of resources is shared between users (e.g., memory, CPUs, storage), providing a scalable and often fault-tolerant environment. Distribution issues are transparent to final users, which see cloud-based applications as services. Some initiatives for improving the performance of the EMF have been conducted. For instance, the Morsa [7] framework enables loading larger models, by using a storage framework based on documents (MongoDb).

While MDE techniques have been used to improve cloud-based solutions [13,3], not much work has been done the other way around. Cloud-based platforms are relatively new and researchers still need to investigate if/how/when

MDE solutions can really benefit from a cloud. This area has been called Modeling *in* the Cloud, or Modeling As A Service [1].

In this article, we present a set of research questions, possible solutions and probable challenges we may face when coupling MDE and Cloud Computing. Specifically, we concentrate on two main tasks to ultimately accomplish the execution of model transformations on the Cloud:

1. **Model storage in the Cloud:** a cloud-based and distributed storage mechanism to enable the efficient loading of VLMs to the Cloud, for subsequent querying and processing.
2. **Model transformation execution in the Cloud:** intended to take advantage of the abundance of resources by distributing the computation of the transformations to different processing units.

We will present a set of questions, benefits, and challenges that have risen in both these aspects, and possible solutions that need to be further investigated.

As future work we plan to implement a proposed solution to the problems described in this paper, in the form of a model transformation tool based on EMF and ATL. For this reason we base the examples in this paper on this technological framework.

This article is organized as follows. Section 2 presents the problem of storing/accessing models in the Cloud. Section 3 focuses on distributed model transformations in the Cloud. Section 4 concludes the paper.

2 Storage of Models in the Cloud

One of the core principles of cloud computing is to distribute data storage and processing into servers located in the cloud. In MDE, a cloud-based framework for model storage and/or transformation could bring several benefits, e.g.:

Support for VLMs. Models that would be otherwise too large to fit in the memory of a single machine could be split into several different nodes located in the cloud, for storage, processing, or both.

Scalability. The execution time of costly operations on models (e.g., complex queries or model transformations on VLMs) can be improved by the data distribution and parallel processing inherent capabilities of the Cloud.

Collaboration. A cloud-based model storage can simplify the creation of a collaborative modeling environment where development teams on different locations could specify and share the same models in real-time.

Other topics, such as transparent tool interoperability, model evolution and fault-tolerance could also benefit from the cloud computing principles and have yet to be further investigated [1].

In the next subsections we identify and discuss two main research tasks that have to be addressed to obtain an efficient mechanism for VLMs in the cloud:

1. how to access models in remote locations in a transparent way, so that existing MDE tools can make direct use of them;
2. how to distribute the storage of a VLM on a set of servers, to make use of the resources offered by the cloud.

2.1 Transparent remote model storage

The use of models in the Cloud should not hamper their compatibility with existing modeling environments. All complexity deriving from the framework implementation, such as element/node location, network communication and balance should be hidden to end users and applications. This transparency towards the MDE clients can be obtained by implementing the network communication mechanism behind the model management API.

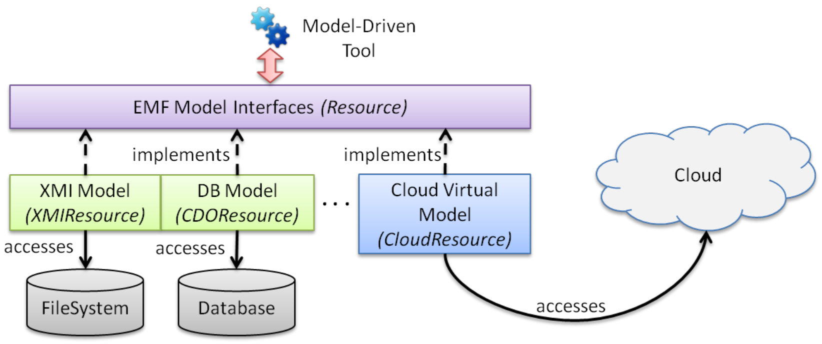


Fig. 1. Extending EMF with support of cloud storage for models.

The idea of providing alternative backends to model management APIs has already been used for local storage. For instance EMF allows applications to load and manipulate models stored as XMI files on disk when using its XMI backend, or the CDO[3] backend for models stored in databases. Clasen et al. [5] generalize this idea by introducing the concept of *virtual models* (with a direct reference to virtual databases) as a re-implementation of the EMF API to represent non-materialized models whose elements are calculated on demand and retrieved from other models regardless of their storage mechanism.

The same principle can be extended to support a cloud-based storage. The model management API can be extended/re-implemented to allow the access to a cloud-based persistence layer. Requests and updates of elements on this non-materialized model would be translated into calls to the web-services exposed by the cloud infrastructure. In our research agenda we plan to provide such a mechanism as a Cloud Virtual Model, illustrated in Figure 1.

[3] `http://www.eclipse.org/cdo/`

2.2 Distributed model storage

Data manipulated by a given cloud can come from a single data source (e.g., the client) or a distributed storage mechanism *in* the cloud. The second solution is especially useful to handle VLMs. The idea behind distributed model storage is to decompose a full model and to store subsets of its elements in different servers or physical locations (see Fig. 2). The sets of elements located in each node can be regarded as *partial models*, and from their composition the global model is constituted. The distribution strategy can be made invisible to the client application by using a virtualization layer as explained above. This way the persisted model is perceived as one single logical model.

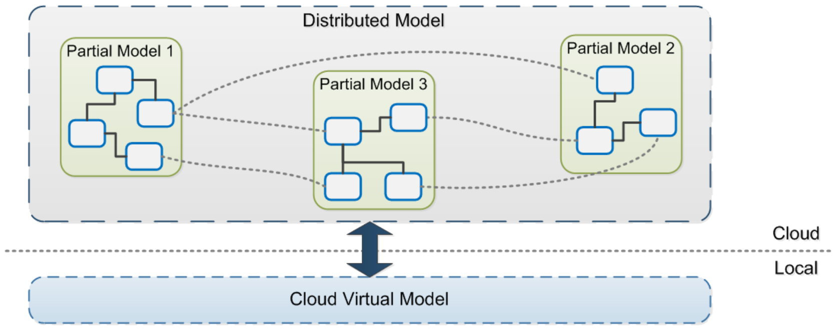


Fig. 2. A cloud virtual model that abstracts the composition of several distributed partial models. Dashed lines represent elements associations between cloud nodes.

Distributing model elements. The criteria used to define which elements are stored in each cloud node vary according to the context of use of the distributed model. For instance, when considering collaboration aspects, the model can be distributed to reduce the network costs, assigning to a given node the elements more likely to be accessed by the team located the closest to that node. As another example, in cases of parallel processing some knowledge about the computation algorithm may be used to assign model elements to nodes, to optimize parallelization. There are already approaches that study how to create partitions from graphs in a cloud for processing purposes (see the solutions from [12]). A study on the nature of MDE applications to identify correspondences with these existing approaches, in order to adapt them or to create novel solutions, has yet to be done.

Among the different model distribution policies two corner cases can be identified, analogous to the homonymous techniques in database systems:

– *Vertical Partitioning.* [14] Each partial model holds only elements conforming to certain types, i.e., each node has the responsibility to store only a certain subset of concepts of the global model. For instance, a first partial model may contain only structural aspects of a UML model whereas a second may correspond to dynamic aspects.

– *Horizontal Partitioning.* [4] Each partial model holds elements of any type, and the separation conforms to a property-based selection criteria. For instance, elements representing French customers may be allocated to one node, whereas elements representing Brazilian customers may be allocated to another.

The choice of the distribution policy is not limited to partitions of the original set of model elements. Element replication could be desired to optimize the balance of network vs. memory usage [17].

Distributing associations. In most cases it is not possible to determine a partitioning in partial models that can be processed by completely independent nodes. Model elements can have different types of relationships between them (e.g., single and multi-valued references, containment references) and the computation on one node could at one point need to access an associated element contained in another one.

We first need a mechanism to store information about associations between elements located in different partial models. This information has to contain 1) pointers to locate incoming and outgoing associated element/s, and 2) the relationship type, so the distribution infrastructure can correctly interpret it. The pointers must necessarily contain both identifiers to the referenced elements within a partial model, and the location of the node that holds this partial model.

Cross-node associations can be distributed in several ways in the cloud nodes. Two main topologies are used by distributed databases and filesystems [17]:

1. The relationship metadata is centralized in a single node. All partial nodes must ask this central node for the location of the partial model containing the referenced element.
2. The relationship metadata is known by all nodes. Partial models then can directly request the referenced element to the correct node when necessary.

Both topologies have their pros and cons. Sharing the metadata in all nodes implies in extra memory usage, whereas a centralized metadata node requires extra inter-node communications. The choice of the best solution depends on several factors, as for instance node location, network bandwidth, and the quantity of cross-node associations.

Wherever the cross-node associations are stored, this information has to be correctly *interpreted* to enable navigation of the distributed model across nodes. When each node has a transparent abstraction of the full model, this navigation has to happen seamlessly. A navigation call to the virtual model of the node would have to start a resolution algorithm to: 1) retrieve the information about the cross-node association, 2) locate the requested elements from another node, and 3) return it to the MDE tool mimicking a call to a local model. This way, when a node wants to access external model elements, it becomes itself a client of the cloud that contains it.

3 Model transformations in the Cloud

Model transformations are central operations in a MDE development process. A model transformation can be seen as a function that receives as input a set of source models and generates as output a set of target models. A transformation execution record is commonly represented in MDE as a set of *trace links*, each one connecting: 1) a set of source elements, 2) a set of corresponding target elements and 3) the section of code (e.g., rule, method) responsible for the translation.

Transformations can consume a lot of resources, in terms of memory occupation and computation time. Operations like traversing the full model or executing recursive model queries can be very expensive. When a centralized solution cannot handle the processing efficiently, one solution is to parallelize the execution of the transformations, for instance, within a cloud. The computation tasks have to be distributed on several nodes, each one in charge of generating partial outputs (i.e., models) that are later merged to obtain the full result. The expected result of a parallel computation must be the same result of its correspondent sequential transformation.

We sketch below a subdivision of the parallel transformation process in the three following steps, resulting on an overall conceptual view depicted in Fig. 3.

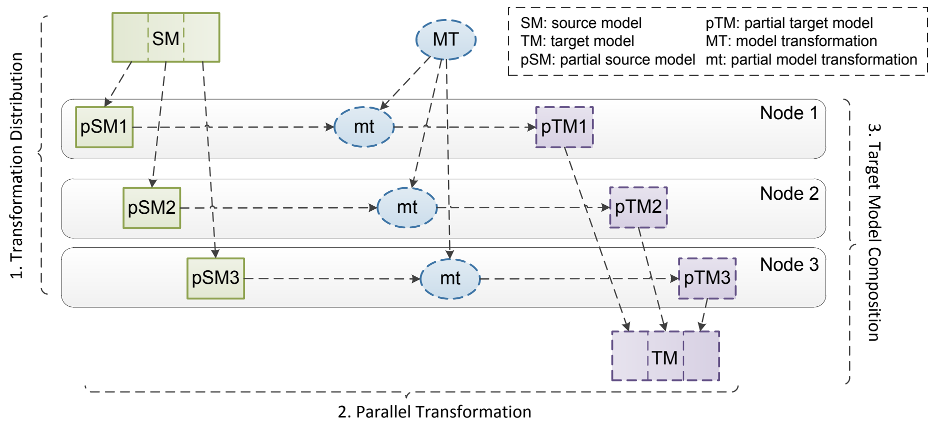

SM
MT
SM: source model
TM: target model
pSM: partial source model
pTM: partial target model
MT: model transformation
mt: partial model transformation
Node 1
Node 2
Node 3
pSM1
pSM2
pSM3
mt
pTM1
pTM2
pTM3
TM
1. Transformation Distribution
2. Parallel Transformation
3. Target Model Composition

Fig. 3. Parallel Transformation Overview.

This process is divided in three major steps:

1. **Transformation Distribution.** An algorithm is defined to distribute the transformation computation over the available nodes. This phase may include a physical partitioning of the source model into partial models to send

to each server. The step is optional, e.g., in the case of transforming source models that are already distributed on nodes.

2. **Parallel Transformation.** The transformation code of each node is fed with a source model and runs in parallel with the others, generating a set of partial target models. When implementing the transformation engine running on the nodes, it could be advisable to re-use its sequential implementation, and to add a communication mechanism between nodes to access unavailable information. Both aspects have been discussed in Section 2.2.
3. **Target Model Composition.** The partial target models generated by each node are composed into a full target model.

While in the following we describe the three steps as sequential, they don't have to be necessarily executed eagerly. For example, in scenarios requiring costly data processing, but where a cloud-based storage infrastructure is not available, source models can be loaded and transformed lazily (i.e., on demand). Even when the target partial model have been computed, they don't necessarily have to be returned to the client as a full model, but a virtual model can be used to lazily retrieve needed model elements only when they are requested. The subject of lazy execution has been investigated in [15], and its application to the phases of parallel transformations is another pointer for future research.

3.1 Transformation distribution

The phase of transformation distribution is responsible for the assignment of parts of the transformation computation to each node. At the end of the parallel transformation the global transformation record will be constituted by the same set of trace links of the equivalent sequential transformation, since the correspondence between source and target model elements is not changed by the parallelization. Each node is responsible for a subset of the trace links, having translated only a subset of the VLM. Thus, distributing the computation of the transformation is equivalent to partitioning the set of trace links in groups assigned to the nodes.

To implement this partitioning, a distribution algorithm has to communicate to the nodes the needed information to determine which trace links to generate. Being a trace link uniquely determined by a set of source elements and a section of transformation code, the nodes, in general, have to receive information about the model elements they are responsible for and the transformation code to apply to each of them.

This information can be determined according to several different strategies. We classify the possible strategies based on the knowledge they exploit:

– In a first class of strategies, the transformation distribution algorithm assigns computation to nodes without ever loading the model. These approaches avoid the problem of loading a VLM that could exceed the limited memory of the client. Distribution can be still performed based on:

    - A partitioning of the transformation code (e.g., each node executes a single transformation rule). All nodes have access to the full source model (or its replica), but only execute a subset of the transformation code. This approach is called *transformation slicing.*
    - A low-level parsing of the serialized model, that in some situations can be split in consistent chunks without being fully loaded in memory. The chunks are then only loaded when they arrive to their assigned nodes.
- A second class of transformation distribution algorithms allow to load the model (and its metamodel) and to select which computation to assign based on properties of the model elements. This category is called*model slicing.* Vertical and horizontal model partition algorithms (see Section 2.2) can be used as transformation distribution approaches of this type.
- A more sophisticated class of algorithms would be based on static analysis of the transformation code, to determine a partitioning that can optimize parameters of the parallel execution, e.g., total time, throughput, network usage. Static analysis can for instance identify dependencies between transformation rules that can be exploited to maximize the parallelization of the computation. Similar dependencies have already been computed in related work, e.g., in [16] with a focus on debugging.
- Finally several external sources of data can be used to drive the distribution, like usage statistics on model elements, or information about the cloud topology and resources.

An analogous characterization can be done for the algorithms of target model composition. For instance, for some transformations, the nodes could provide a perfect partition of the target model, without requiring the composition algorithm to load and analyze the produced partial models. Alternatively, some processing could be required during composition, e.g., to remove redundant elements, or to bind missing references.

Optimal algorithms in these classes may be a promising research subject.

3.2 Coupling model transformations with MapReduce

A well-known large scale data processing framework that may be adapted to implement distributed (especially on-demand) model transformations on the Cloud is the MapReduce framework[6]. MapReduce has three key aspects:

Input format: is always a pair *(key, value).* The *key* is used to distribute the data. The *value* is processed by the framework. However, it is necessary to implement import/export components to be able to inject different formats (e.g., text files, databases, streams) into the framework.

Map tasks: receives each *(key, value)* pair and processes them in a given node. The result is another *(key, value)* pair.

Reduce tasks: receives the output of the Map tasks and merges them into a combined result.

Once the *(key,value)* pairs are defined and these two tasks are implemented, the framework takes care of the complex distribution-inherent details such as load balancing, network performance and fault tolerance.

In order to use MapReduce to execute model transformations, we need to precisely define how to represent models and model transformations in terms of these components. We can identify a clear correspondence in Fig. 3 between those steps and the MapReduce mechanism:

1. The source model needs to be divided into appropriate *(key,value)* pairs. Values should be partial source models.
2. The Map functions execute the transformations rules in parallel, on each one of the partial source models, generating partial target models as output data.
3. The Reduce functions combine the result of all Maps (i.e. partial target models) into a final result (i.e. a full composed model).

The exploitation of the MapReduce framework seems a feasible starting point towards parallel model transformations by allowing the research focus to be on MDE-related issues, while the framework is in charge of handling all cloud-inherent complications.

4 Conclusion and Future Work

In this article, we have discussed a set of research questions to port modeling and transformation frameworks into a cloud-based architecture. We have described different paths that need further investigation. Based on our previous experience on the use, development and research of model transformations, we have identified key aspects and divided them in two phases. First, an efficient model storage and access mechanism on the cloud needs to be investigated. The main difficulties are related on how to efficiently distribute the model elements and the relationships between them. Second, a parallel processing mechanism by distributed model transformations has to be provided. The main difficulties are about the distribution of the transformations coupled with the models that are going to be processed and how to combine the distributed results.

In our future work we plan to propose a solution, among the illustrated alternatives, in the form of a cloud-based engine for the ATL transformation language. The main design features for this engine will be: cloud transparency, re-use of the standard ATL engine, node inter-communication, and support for pluggable distribution algorithms. We also want to study how the static analysis of ATL transformations can help in optimizing the distribution algorithm for our engine. Finally, we hope that this article will promote discussion and involve other researchers to the task of moving MDE to the Cloud.

References

1. H. Brunelière, J. Cabot, and F. Jouault. Combining Model-Driven Engineering and Cloud Computing. In *MDA4ServiceCloud'10 (ECMFA 2010 Workshops)*, Paris, France, June 2010.
2. F. Budinsky. *Eclipse modeling framework: a developer's guide.* Addison-Wesley Professional, 2004.
3. S. Ceri, P. Fraternali, and A. Bongio. "Web Modeling Language (WebML): a modeling language for designing Web sites". *Computer Networks*, 33(1–6):137 – 157, 2000.
4. S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *ACM 1982 SIGMOD International Conference*, pages 128–136, Orlando, USA, 1982. ACM.
5. C. Clasen, F. Jouault, and J. Cabot. VirtualEMF: A Model Virtualization Tool. In *Advances in Conceptual Modeling. Recent Developments and New Directions (ER 2011 Workshops)*, LNCS 6999, pages 332–335. Springer, 2011.
6. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
7. J. Espinazo Pagán, J. Sánchez Cuadrado, and J. García Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *MODELS 2011*, LNCS 6981, pages 77–92. Springer, 2011.
8. F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS 2005 Workshops*, LNCS 3844, pages 128–138. Springer, 2006.
9. F. Jouault and J. Sottet. An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In *5th International Workshop on Graph-Based Tools, Grabats*, Zurich, Switzerland, 2009.
10. D. Kolovos, R. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT 2008*, LNCS 5063, pages 46–60. Springer, 2008.
11. I. Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In *AGTIVE 2007*, LNCS 5088, pages 377–393. Springer, 2008.
12. J. Lin and C. Dyer. Data-Intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.
13. I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Model-Driven Design and Deployment of Service-Enabled Web Applications. *ACM Transactions on Internet Technology*, 5(3):439–479, Aug. 2005.
14. S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems*, 9(4):680–710, 1984.
15. M. Tisi, S. Martínez, F. Jouault, and J. Cabot. Lazy Execution of Model-to-Model Transformations. In *MODELS 2011*, LNCS 6981, pages 32–46. Springer, 2011.
16. Z. Ujhelyi, A. Horvath, and D. Varro. Towards Dynamic Backward Slicing of Model Transformations. In *ASE 2011*, pages 404 –407. IEEE, nov. 2011.
17. P. Valduriez and M. Ozsu. *Principles of Distributed Database Systems.* Prentice Hall, 1999.

Towards a Common Modelling Platform for the Migration to the Cloud

Alek Radjenovic[1] and Richard F. Paige[1]

Department of Computer Science, The University of York, United Kingdom
{alek.radjenovic,richard.paige}@york.ac.uk

Abstract. Cloud-based software is starting to replace the ubiquitous desktop applications. Software manufacturers are investigating ways of migrating their key assets (desktop software) to the cloud. Such migrations are not easy, as they must take into account migration of data, functionality and user interfaces. We propose an approach that supports abstraction and automation, leveraging a set of established Model-Driven Engineering technologies, in order to support migration. The approach intends to help define a common modelling platform that will formalise the migration process, and provide mechanisms to support partial and incremental migration. We argue that such systematic approach may lead to a significant reduction in cloud application development costs and, consequently, faster adoption of the cloud computing paradigm.

1 Introduction

Cloud-based software solutions are beginning to replace the previously ubiquitous desktop applications. Software manufacturers, who are aware of these changes, are investigating ways of protecting their long-term investments – *desktop software* – that are increasingly becoming legacy. Over the years, many of these desktop applications were re-engineered and migrated to multiple operating systems (OS), either by being made cross-platform, or spawning separate versions, one for each OS. Migration of desktop software to the cloud, however, has no straightforward solutions.

The cloud computing paradigm imposes a significant shift in design thinking. Although the computational ability or functionality of an application may remain the same/similar, the way in which storage, security, networking, off-line usage, and user interfaces (UI) of a cloud-based application are designed and implemented is substantially different from in desktop software. In this respect, migration of desktop applications to the cloud can no longer be regarded as a straightforward (software) evolution problem; a more complex, *transformational*, approach is needed. This is arguably best achieved at a high level of abstraction, e.g. at a model level.

We argue that a systematic approach supporting abstraction and automation, like those rooted in the disciplines of software architectures and Model-Driven Engineering (MDE), can take into account the essential aspects and deal with the critical challenges of cloud migration problems. Our hypothesis is that, by leveraging software architecture and MDE, the process of migration of desktop applications to the cloud will be easier to understand, raise awareness of potential problems at an early stage, and provide structured development, deployment and maintenance plans. This, we predict, could lead to a substantial reduction in cloud application development costs and faster adoption of the paradigm.

In this position paper, we highlight some of the key concerns and challenges associated with the migration process and propose an approach that will, we hope, kick-start the definition of a core set of principles, methods and formalisms unified under a common modelling platform. Our primary intention is to attempt to stimulate discussion within the MDE and cloud communities, and to bring them together in order to tackle the identified challenges.

2 Background

Cloud computing is a computational model which does not yet have a standard definition (nor standard application frameworks for their development). A working definition [10] is that clouds of distributed computers provide on-demand resources and services over a network with the scale and reliability of a data centre [7]. Though cloud computing may have a positive impact on organisations, the absence of widely accepted open standards is a risk to adoption; the Open Cloud Manifesto [12] aims to provide a minimal set of principles that may form a basis for an initial set of accepted standards.

Cloud computing acts as a catalyst [13] for: tool developers for better delivery, data as a service, creation of workflow standards, and metadata services and standards. A typical cloud architecture consists of three service layers [9]: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Clouds that provide on-demand computing instances (e.g. Amazon EC2 [2]) can use these to supply SaaS (e.g Salesforce.com), or to provide a PaaS (e.g Heroku [8]), often in the form of *tools* used for the development of SaaS (e.g. Google App Engine [5], or Microsoft Windows Azure [1]).

Cloud migration is growing in importance; increasing numbers of applications are moving to the Web, including office software or development tools. Some predictions suggest that eventually little software will run on a desktop [3].

Architecturally, differences between traditional web applications and cloud-based applications are significant. A web application normally resides in one location (e.g., a web server, an application pool). The data it manipulates is typically stored inside a single database within a single database server instance. (The term *single* is used loosely here, meaning a single *logical* unit – e.g. a traditional master/slave database architecture, provided in an ad–hoc and costly way). In contrast, cloud applications are distributed and scale differently.

There is also a strong case for the production of *hybrid* applications (part-desktop, part-cloud). One scenario sees an incremental conversion process, where during each stage only a portion of a desktop application is migrated, and where each stage ends with working software (e.g. using an Agile approach). Another scenario considers cases where the target application is only partially converted (e.g. part of the focus of the Diaspora* project [6], which aims to support local storage of data while using a cloud-based solution, e.g. Facebook [4]).

Existing research projects such as mOSAIC [11] and RESERVOIR [15] address interesting cloud computing challenges, but not legacy software. REMICS [14] on the other hand does focus on migration, but its wide scope and close integration with OMG standards may prove too impractical in the long run.

3 Scientific Questions and Objectives

Migration of applications to the cloud is generally ad-hoc. We would like to put migration to the cloud on a more rigorous footing, by proposing and developing novel MDE theories, tools and techniques that directly address the challenges of migration of desktop applications to the cloud, while providing reusable mechanisms that increase productivity. Amongst other things, this will help reduce the likelihood of errors in migration, allow non-CS experts to take advantage of the cloud more easily, and promote better understanding of the challenges of migration. Theoretically, we aim to enrich the field of software migration and maintenance by providing a rigorous methodology for transitioning to the cloud.

The key scientific questions to be addressed are: What MDE techniques to use in the process of migration? Can a generic MDE framework be provided to support incremental and partial migration? How can we assess the effectiveness and practicality of such approach? Can the migration process be formalised? Is it possible to identify when a migration is feasible (e.g., measured in terms of the proportion of the process that can be automated) and when it is not?

To provide answers to these questions, we have decided to focus on the following research objectives: (a) provision of new theories and practical implementations of modelling frameworks and technologies, (b) generation of a set of architectural blueprints using MDE for model driven migration that focuses on maximal code reuse, with guidelines on how to perform incremental and partial migration, (c) identification of scenarios in which the migration is either not possible, unnecessary, or not beneficial from the perspective of a cost-effort trade-off, and (d) formalisation of the migration process. The latter may ultimately be represented in the form of process models (e.g., using SPEM or activity diagrams) that can then be automated using a suitable workflow engine.

Central to the proposed approach are techniques and technologies, instantiated as a set of architectural blueprints and MDE tools. The blueprints could take the form of *MDE metamodels* (capturing key concepts and concerns of cloud application architecture) and *MDE migration operations* (automating the process of transition from a desktop architecture to a cloud architecture). The emphasis should be on delivering the metamodels and operations using tools that permit their automated application (preferably exploiting open-source standards).

4 Approach

We propose that the initial work addresses the following aspects of applications: **computation**, **data I/O and persistence**, and **user interface (UI)** (we call these *domains* in the sequel). These domains are not only core to all applications, but they are also significantly different between the two platforms. Typically, desktop *computation* is migrated into (web) services, application *data* is migrated to the network, and the cloud–based *UI* is either flattened inside a web browser, or reduced to a (smaller) mobile device screen. Features such as security, data integrity, or multi-tenancy may also be considered where appropriate but not at any great length during the initial work. Furthermore, the projected common modelling platform will need to define (at a minimum) the following components:

Metamodels, formalising: (i) domain modelling logic (components, relationships, composition and interaction rules) within as well as between the domains, (ii) relationships between domains, (iii) common architectural models for each platform (desktop and cloud), and (iv) transformation of architectural models from one platform to another with respect to the domains. The resulting unified metamodels for each domain *and* platform will identify, define, classify and formalise the relevant modelling components, intra-domain (within the domain) and inter-domain (between domains) relationships (dependencies, communication, messaging), as well as typical modelling operations within the domain.

Migration mechanisms, for each domain, allowing explicit expression of the dependencies with model elements from other domains (enabling domain detachment – e.g. a desktop application UI, and its migration to the cloud in support of incremental migration and hybrid applications). The mechanisms need to define how the newly created cloud components can work with the remaining desktop elements (e.g., migration of computation to web services has to consider how these can deal with local data storage and a desktop UI).

Migration scenarios, defining strategies for migration in the form of step-by-step guidelines that describe how to approach the migration process, and which architectural components and relationships are the right candidates for migration in each particular step. Furthermore, we propose the definition of hybrid architectural models that describe the necessary transformations required to achieve the migration from one phase to another. Finally, the common modelling platform needs to identify those scenarios where only partial migration may be the best or feasible option and to specify the utility of such approach. For instance, there may be situations in which only the user interface is migrated to a web browser while the computation and data storage is done locally.

The definition of the above components could best be reached in an iterative fashion, with each iteration broadly comprising the following steps: **analysis**, identifying relevant existing technologies, allowing us to minimise the amount of standard development and to focus on the novel aspects; **formalisation**, developing migration strategies e.g. as: (i) correspondence models (highlighting important relationships between platforms), and (ii) mechanisms required to achieve migration; **migration**, deploying desktop application parts to the cloud using case studies; **testing**, using identical test cases on the original and migrated applications where criteria is based on validating the applications' functionality, carrying out (at the same time) benchmark tests that compare the performance aspects of the original desktop application with its cloud equivalent; and **evaluation**, assessing previous step outputs and drawing further conclusions on the suitability of the approach(es) employed.

5 Conclusion

Traditional desktop software is steadily being replaced by the new cloud-based solutions. Software companies are seeking ways to migrate their exiting desktop applications to the cloud. This migration is generally ad hoc as there are no predefined mechanisms or strategies to help with the process.

In this paper we have outlined an approach that puts the migration on a rigorous footing, and that leverages the well-established theories and techniques from the software architecture and MDE arenas. The approach involves defining a common modelling platform that provides support for three core aspects of applications: *computation*, *data I/O and persistence*, and *user interface*. The common modelling platform also comprises three major components: *metamodels* – formalising modelling logic, architectures, and transformations; *migration mechanisms* – enabling incremental migration and hybrid applications; and, *migration scenarios* – providing strategies, guidelines and the feasibility studies for various types of applications.

We have also proposed a five-step iterative process for the definition of the common modeling platform components, which includes the *analysis*, *formalisation*, *migration*, *testing* and *evaluation* steps.

References

1. Microsot Windows Azure. http://www.microsoft.com/windowsazure/, 2011.
2. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2, 2011.
3. Hakan Erdogmus. Cloud Computing: Does Nirvana Hide behind the Nebula? *IEEE Software*, 26(2):4–6, March 2009.
4. Facebook. http://www.facebook.com, 2011.
5. Google App Engine. http://code.google.com/appengine, 2011.
6. Daniel Grippi, Maxwell Salzberg, Raphael Sofaer, and Ilya Zhitomirskiy. Diaspora* (https://joindiaspora.com/), 2011.
7. Robert L. Grossman. The Case for Cloud Computing. *IT Professional*, 11(2):23–27, March 2009.
8. Heroku. http://www.heroku.com/, 2012.
9. Ali Khajeh-Hosseini, Ian Sommerville, and Ilango Sriram. Research Challenges for Enterprise Cloud Computing. Technical report, LSCITS, 2010.
10. Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical report, 2011.
11. mOSAIC. http://www.mosaic-project.eu/, 2012.
12. Open Cloud Manifesto. http://www.opencloudmanifesto.org, 2011.
13. RCUK. 'Cloud Computing for Research' Workshop. Technical report, 2010.
14. REMICS. http://remics.eu/, 2012.
15. RESERVOIR. http://www.reservoir-fp7.eu/, 2012.

Towards CloudML, a Model-based Approach to Provision Resources in the Clouds*

Eirik Brandtzæg[1,2], Sébastien Mosser[1], and Parastoo Mohagheghi[1]

[1] SINTEF IKT, Oslo, Norway
[2] University of Oslo, Oslo, Norway
{firstname.lastname}@sintef.no

Abstract. The Cloud-computing paradigm advocates the use of resources available "in the clouds". In front of the multiplicity of cloud providers, it becomes cumbersome to manually tackle this heterogeneity. In this paper, we propose to define an abstraction layer used to model resources available in the clouds. This cloud modelling language (CloudML) allows cloud users to focus on their needs, *i.e.*, the modelling the resources they expect to retrieve in the clouds. An automated provisioning engine is then used to automatically analyse these requirements and actually provision resources in clouds. The approach is implemented, and was experimented on prototypical examples to provision resources in major public clouds (*e.g.*, Amazon EC2 and Rackspace).

1 Introduction

Cloud–Computing [2] was considered as a *revolution*. Taking its root in distributed systems design, this paradigm advocates the share of distributed computing resources designated as "*the cloud*". The main advantage of using a cloud-based infrastructure is the associated scalability property (called *elasticity*). Since a cloud works on a *pay–as–you–go* basis, companies can rent computing resources in an elastic way. A typical example is to temporarily increase the server–side capacity of an e–commerce website to avoid service breakdowns during a load peak. According to Amazon (one of the major actor of the Cloud market): *"much like plugging in a microwave in order to power it doesnt require any knowledge of electricity, one should be able to plug in an application to the cloud in order to receive the power it needs to run, just like a utility"* [15]. However, there is still a huge gap between the commercial point of view and the technical reality that one has to face in front of "*the cloud*".

The Cloud-computing paradigm emphasises the need for automated mechanisms, abstracted from the underlying technical layer. It focuses on the reproducibility of resource provisioning: to support the horizontal scaling of cloud-applications (*i.e.*, adding new computing resources on-the-fly), such a provisioning of on-demand resources will be performed by a program. The main drawback

* This work is funded by the European commission through the REMICS project (www.remics.eu), contract number 257793, with the 7th Framework Program.

associated is the heterogeneity of cloud providers. At the infrastructure level, more than ten different providers publish different mechanisms to provision resources in their specific clouds. It generates a *vendor lock-in* syndrome, and an application implemented to be deployed in cloud C will have to be re-considered if it now has to be deployed on cloud C'. All the deployment scripts that were designed for C have to be redesigned to match the interface provided by C' (which can be completely different, *e.g.*, shell scripts, RESTful services, standard API).

Our contribution in this paper is to describe the first version of CloudML, a cloud modelling language specifically designed to tackle this challenge. This research is done in the context of the REMICS EU FP7 project, which aims to provide automated support to migrate legacy applications into clouds [10]. Using CloudML, a user can express the kind of resources needed for a specific application, as a model. This model is automatically handled by an engine, which returns a "run-time model" of the provisioned resources, according to the models@run.time approach [3]. The user can then rely on this model to interact with the provisioned resources and deploy the application. The approach is illustrated on a prototypical example used to teach distributed systems at the University of Oslo.

2 Challenges in the cloud

To recognise challenges when doing cloud provisioning we use an example application [5]. The application (known as *BankManager*) is a prototypical bank manager system which support *(i)* creating users or bank accounts and *(ii)* moving money between bank accounts and users. *BankManager* is designed but not limited to support distribution between several nodes. Some examples of provisioning topologies is illustrated in Fig. 1, each example includes a browser to visualise application flow, a front-end to visualise executable logic and back-end represents database. It is possible to have both front-end and back-end on the same node, as shown in Fig. 1(a). In Fig. 1(b) front-end is separated from the back-end, this introduces the flexibility of increasing computation power on the front-end node while spawning more storage on the back-end. For applications performing heavy computations, it can be beneficial to distribute the workload between several front-end nodes as seen in Fig. 1(c), the number of front-ends can be increased n number of times as shown in Fig. 1(d). *BankManager* is not designed to handle several back-ends because of the relational database, this can solved on a database level with master and slaves (Fig. 1(e)) although this is out of the scope of this article.

We used bash scripts to implement the full deployments of *BankManager* against *Amazon Web Services* (AWS) [1] and Rackspace [13] with a topology of three nodes as shown in Fig. 1(c). From this prototype, it became clear that there were multiple challenges that we had to address:

- **Heterogeneous Interfaces**: The first challenge we encountered was to simply support authentication and communication with the cloud. The two

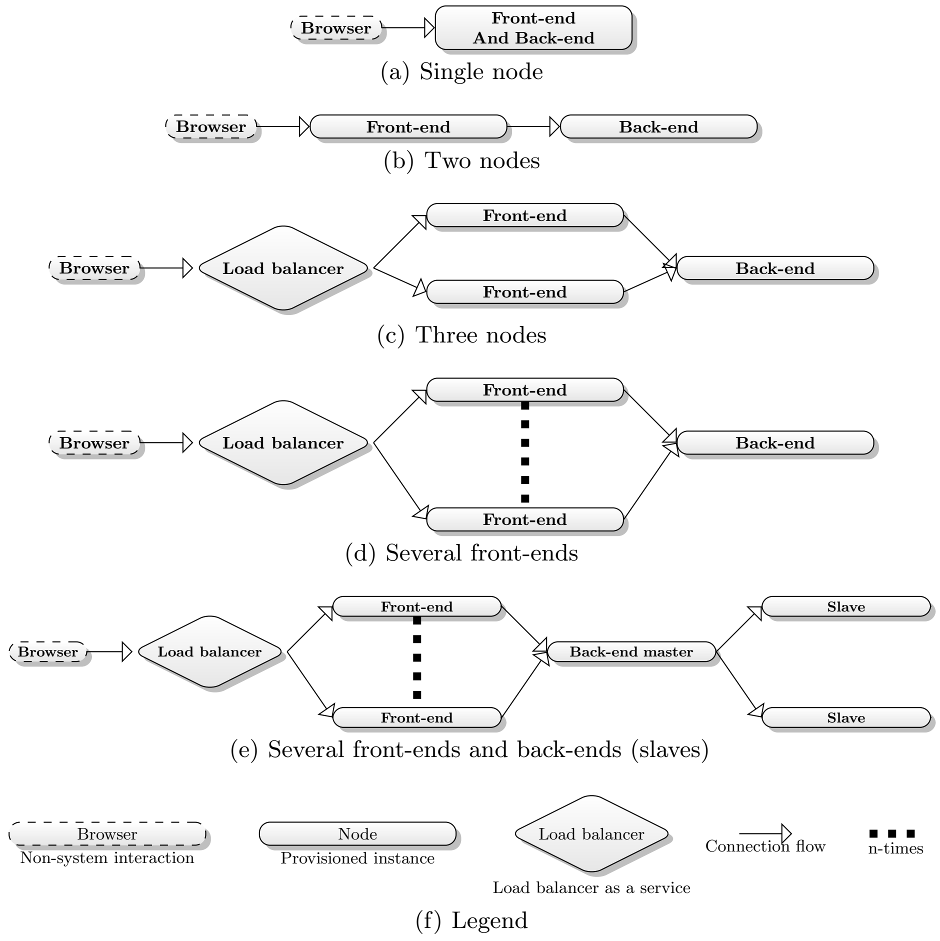


Fig. 1. Different architectural ways to provision nodes (topologies).

providers we tested against had different approaches, AWS [1] had command-line tools built from their Java APIs, while Rackspace [13] had no tools beside the API language bindings, thus we had to operate against the command-line tools and public APIs. As this emphasises the complexity even further, it also stresses engineering capabilities of individuals executing the tasks to a higher technical level.

– **Platform-specific Configuration**: Once we were able to provision the correct amount of nodes with desired properties on the first provider it became clear that mirroring the setup to the other provider was not as convenient as anticipated. There were certain aspects of vendor lock-in, so each script was hand-crafted for specific providers. The lock-in situations can, in many cases, have financial implications where for example a finished application is locked to one provider and this provider increases tenant costs[3]. Or availability decreases and results in decreases of service up-time, damaging revenue.

– **End-user Reproducibility**: The scripts provisioned nodes based on command-line arguments and did not persist the designed topology in any way, this made topologies cumbersome to reproduce. Scripts can be "re-executed" to redo a provisioning step, but they often rely on command-line arguments that differs from a computer to another one (*e.g.*, file paths), requiring technical knowledge to be correctly executed.
– **Shareable**: Since the scripts did not remember a given setup it was impossible to share topologies "as is" between coworkers. It is important that topologies can be shared because direct input from individuals with different areas of competence can enhance quality. Provisioning scripts can be shared as plain files, and the lack of modularity expressiveness in the underlying language does not support re-use as defined in the Object-Oriented community. The re-use of deployment script is empirically done through a copy-paste approach, and concerns are not modularised in shareable components.
– **Robustness**: There were several ways the scripts could fail and most errors were ignored. Transactional behaviours were non-existent.
– **Run-time dependency**: The scripts were developed to fulfil a complete deployment, and to do this it proved important to temporally save run-time specific meta-data. This was crucial data needed to connect front-end nodes with the back-end node. Shell scripts are usually executed in a *batch* mode, and will result in static files containing the information available from the cloud (*e.g.*, IP addresses) at deployment time. Thus, changes in the cloud (*e.g.*, IP re-allocation) cannot be easily propagated.

Vision: Towards a CloudML environment. Our vision is to tackle these challenges by applying a model-driven approach supported by modern technologies. Our objective is to create a common model for nodes as a platform-independent model [4] to justify *multi-cloud* differences and at the same time base this on a human readable lexical format to resolve *reproducibility* and make it *shareable*. The concept and principle of CloudML is to be an easier and more reliable path into cloud computing for IT-driven businesses of variable sizes. We envision a tool to parse and execute template files representing topologies of instances in the cloud. Targeted users are application developers without cloud provider specific knowledge. The same files should be usable on other providers, and alternating the next deployment stack should be effortless. Instance types are selected based on properties within the template, and additional resources are applied when necessary and available. While the tool performs provisioning meta-data of nodes is available. In the event of a template being inconsistent with possibilities provided by a specific provider this error will be informed to the user and provision will halt.

[3] For example, Google decided in 2011 to change the pricing policies associated to the GoogleAppEngine cloud service. All the applications that relied on the service had basically two options: *(i)* pay the new price or *(ii)* move to another cloud-vendor. Due to vendor lock-in, the second option often implied to re-implement the application.

Table 1. CloudML: Challenges addressed.

Challenge	**Addressed by**
Complexity	One single entry point to multiple providers. Utilizing existing framework. Platform-independent model approach used to discuss, edit and design topologies for propagation.
Multicloud	Utilizing existing framework designed to interface several providers.
Reproducibility	Lexical model-based templates. Models can be reused to multiply a setup without former knowledge of the system.
Shareable	Lexical model-based templates. Textual files that can be shared through mediums such as e-mail or version control systems such as Subversion or Git.
Robustness	Utilizing existing framework and solid technologies.
Metadata dependency	*Models@run.time.* Models that reflect the provisioning models and updates asynchronously.

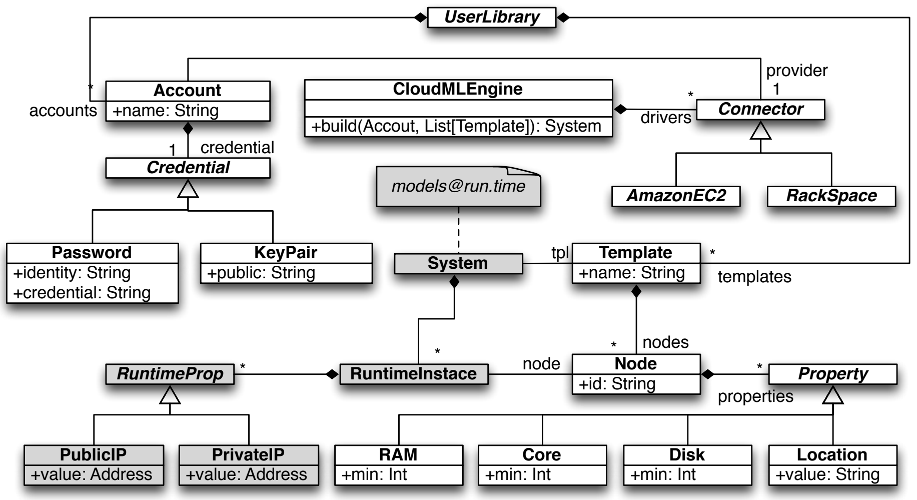


Fig. 2. Architecture of CloudML

3 Contribution

We have developed a metamodel that describe CloudML as a *Domain-Specific language* (DSL) for cloud provisioning. It addresses the previously identified challenges, as summarised in Tab. 1. We provide a class-diagram representation of the CloudML meta-model in Fig. 2. The scope of this paper is to describe the provisionning part of CloudML. The way application are deployed is described in [7].

Illustrative Scenario. CloudML is introduced using a scenario where an end-user (named Alice) is provisioning the *BankManager* to Amazon Web Services *Elastic*

Compute Cloud (EC2) using the topology shown in FIG. 1(c). It is compulsory that she possesses an EC2 account in advance of the scenario. She will retrieve security credentials for account and associate them with `Password` in FIG. 2. `Credential` is used to authenticate the user to supported providers through `Connector`. The next step for Alice is to model the appropriate `Template` consisting of three `Nodes`. The characteristics Alice choose for `Node Properties` are fitted for the chosen topology with more computational power for front-end `Nodes` by increasing amount of `Cores`, and increased `Disk` for back-end `Node`. All `Properties` are optional and thus Alice does not have to define them all. With this model Alice can initialize provisioning by calling `build` on `CloudMLEngine`, and this will start the asynchronous job of configuring and creating `Nodes`. When connecting front-end instances of *BankManager* to back-end instances Alice must be aware of the back-ends `PrivateIP` address, which she will retrieve from CloudML during provisioning according to *models@run.time* (M@RT) approach. `RuntimeInstance` is specifically designed to complement `Node` with `RuntimeProperties`, as `Properties` from `Node` still contain valid data. When all `Nodes` are provisioned successfully and sufficient metadata are gathered Alice can start the deployment, CloudML has then completed its scoped task of provisioning. Alice could later decide to use another provider, either as replacement or complement to her current setup, because of availability, financial benefits or support. To do this she must change the provider name in `Account` and call `build` on `CloudMLEngine` again, this will result in an identical topological setup on a supported provider.

Implementation. CloudML is implemented as a proof of concept framework [6] (from here known as *cloudml-engine*). Because of Java popularity we wrote cloudml-engine in a JVM based language with Maven as build tool. Cloudml-engine use jclouds.org library to connect with cloud providers, giving it support for 24 providers out of the box to minimize *complexity* as well as stability and *robustness*.

We represent in FIG. 3 the provisioning process implemented in the CloudML engine, using a sequence diagram. Provisioning nodes is by nature an asynchronous action that can take minutes to execute, therefore we relied on the actors model [9] using Scala actors. With this asynchronous solution we got concurrent communication with nodes under provisioning. We extended the model by adding a callback-based pattern allowing each node to provide information on property and status changes. Developers exploring our implementation can then choose to "listen" for updating events from each node, and do other jobs / idle while the nodes are provisioned with the actors model. We have divided the terms of a node before and under provisioning, the essential is to introduce *M@RT* to achieve a logical separation. When a node is being propagated, it changes type to `RuntimeInstance`, which can have a different *state* such as *Configuring*, *Building*, *Starting* and *Started*. When a `RuntimeInstance` reaches *Starting* state the provider has guaranteed its existence, including the most necessary metadata, when all nodes reaches this state the task of provisioning is concluded.

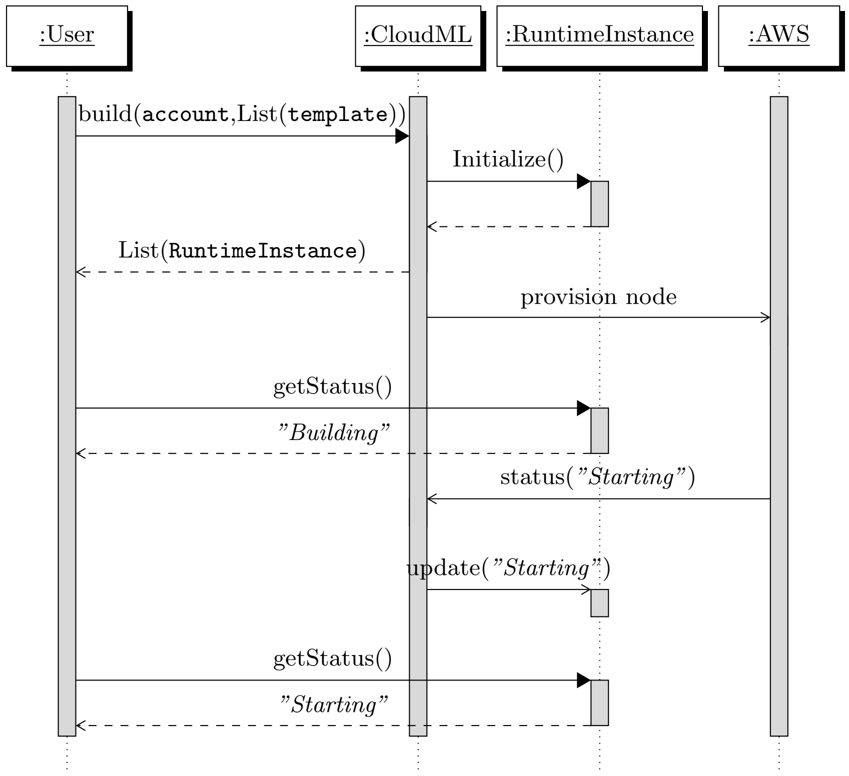


Fig. 3. CloudML asynchronous provisionning process (Sequence diagram).

4 First Experiments: Sketching Validation

Our objective here is to sketch the validation of the CloudML framework, by supporting the provisioning of several nodes into multiple clouds. To start the validation of the approach and the implemented tool, we provisioned the *BankManager* application using different topologies in Fig[1(a), 1(c)]. The implementation uses *JavaScript Object Notation* (JSON) to define templates as a human readable serialisation mechanism. The lexical representation of Fig. 1(a) can be seen in Listing. 1.1. The whole text represents the `Template` of Fig. 2 and consequently "nodes" is a list of `Node` from the model. JSON is textual which makes it *shareable* as files. We implemented it so once such a file is created it can be reused (*reproducibility*) on any supported provider (*multi-cloud*).

```
{ "nodes": [
    { "name": "testnode" }
  ]
}
```

Listing 1.1. One single node (topology: Fig. 1(a))

The topology described in Fig. 1(c) is represented in Listing. 1.2, the main difference from Listing. 1.1 is that there are two more nodes and a total of five more properties. The characteristics of each node are carefully chosen based on each nodes feature area, for instance front-end nodes have more computation power, while the back-end node will have more disk. The key idea is that the meta-model is extensible, and can support new properties in the language thanks to the extension of the `Property` class.

```
{
  "nodes": [
    { "name": "frontend1",
      "minRam": 512,
      "minCores": 2 },
    { "name": "frontend2",
      "minRam": 512,
      "minCores": 2 },
    { "name": "backend",
      "minDisk": 100 }
  ]
}
```

Listing 1.2. Three nodes (topology: Fig. 1(c))

5 Related Work

There already exists scientific research projects and technologies which have similarities to CloudML both in idea and implementation. First we will present three scientific research projects and their solutions, then we will introduce pure technological approaches. We also discuss how our approach differ from theres.

One project that bears relations to ours is mOSAIC [12] which aims at not only provisioning in the cloud, but deployment as well. They focus on abstractions for application developers and state they can easily enable users to *"obtain the desired application characteristics (like scalability, fault-tolerance, QoS, etc.)"* [11]. The strongest similarities to CloudML are *(i)* multi-cloud with their API [11], *(ii)* meta-data dependencies since they support full deployment and *(iii)* the robustness through fault-tolerance. The mOSAIC project works at a code-based level. Thus, it could not benefit from the use of models as interoperability pivot with other tools, to ensure verification for example. The *M@RT* dimension advocated by CloudML also tames the complexity of the technological stack to be used from an end-user point of view. However, model transformation can be designed from CloudML provisioning models to target the mOSAIC API, thus benefiting of the multi-cloud capabilities offered by the mOSAIC platform. Reservoir [14] is another project that also aim at multi-cloud. The other goal of this project is to leverage scalability in single providers and support built-in *Business Service Management* (BSM), important topics but not directly related to our goals. CloudML follows the same underlying approach, but brings the model@run.time dimension, considering that the keystone of such an approach should be at the model level. Vega framework [8] is a deployment framework aiming at full cloud deployment of multi-tier topologies, they also follow a model-based approach. Contrarily to Vega, CloudML supports multi-cloud provisioning.

There are also distinct technologies that bear similarities to CloudML. None of AWS CloudFormation and CA Applogic are model-driven, and actually focus on their own specificities to ensure vendor lock-in. Access to run-time data is bound to specific interface, where CloudML advocate a M@RT representation of the system, supporting reasonning at a higher level of abstraction. Others are plain APIs supporting multi-cloud such as libcloud, jclouds and DeltaCloud. The last group are projects that aim specifically at deployment, making *Infrastructure-as-a-Service* (IaaS) work as *Platform-as-a-Service* (PaaS) like AWS Beanstalk and SimpleCloud. The downside about the technical projects are their inability to solve all of the challenges that CloudML aims to address, but since these projects solve specific challenges it is appropriate to utilize them. Cloudml-engine leverages on jclouds in its implementation to support multi-cloud provisioning, and future versions can utilize it for full deployments.

6 Conclusions & Perspectives

In this paper, we presented the initial version of CloudML, a cloud modelling language used to model the resources that a given application can require from existing clouds. The approach is defined as a meta-model, associated to a reference implementation using the Scala language. This reference implementations is connected to several cloud providers, and we described preliminary experiments that address major cloud providers: Amazon EC2 and Rackspace.

The first perspective of this work is to emphasise its validation. In the context of the REMICS project, our partners provide us several case studies (tourism, banking, scientific computation) that require the provisioning of resources in the clouds. As CloudML is as a platform-independent meta-model to support cloud resource provisioning, one can consider it as a target of a model transformation. This point will be investigated in the context of REMICS: the migration chain results in SOAML models, to be deployed on provisioned resources. We are also interested in refining the set of properties available in the CloudML meta-model to properly categorise the available resources (as for now this mechanism is limited and can lead to sun-optimal provisioning). For now, we focus our effort on computational power, but other dimensions of clouds (*e.g.*, data location, costs) should be taken into account at the CloudML level. The next challenge to be tackled by the CloudML environment is to model the complete deployment of cloud-applications. By coupling the current version of CloudML with an architecture description language, it will be possible to model the needed resources and the deployment plan to be followed to support the automated deployment of the application.

References

1. Amazon: Amazon web services (2012), http://aws.amazon.com
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (Feb 2009), http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html
3. Aßmann, U., Bencomo, N., Cheng, B.H.C., France, R.B.: Models@run.time (dagstuhl seminar 11481). Dagstuhl Reports 1(11), 91–123 (2011)
4. Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: Proceedings of the 16th IEEE international conference on Automated software engineering. pp. 273–. ASE '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=872023.872565
5. Brandtzæg, E.: Bank manager (2012), https://github.com/eirikb/grails-bank-example
6. Brandtzæg, E.: cloudml-engine (2012), https://github.com/eirikb/cloudml-engine
7. Brandtzæg, E., Parastoo, M., Mosser, S.: Towards a Domain-Specific Language to Deploy Applications in the Clouds. In: Third International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING'12). pp. 1–6. Nice, France (Jul 2012)
8. Chieu, T., Karve, A., Mohindra, A., Segal, A.: Simplifying solution deployment on a Cloud through composite appliances. In: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on. pp. 1 –5 (april 2010)
9. Haller, P., Odersky, M.: Actors that unify threads and events. In: Proceedings of the 9th international conference on Coordination models and languages. pp. 171–190. COORDINATION'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1764606.1764620
10. Mohagheghi, P., Sæther, T.: Software Engineering Challenges for Migration to the Service Cloud Paradigm: Ongoing Work in the REMICS Project. In: SERVICES. pp. 507–514. IEEE Computer Society (2011)
11. Petcu, D., Crăciun, C., Neagul, M., Panica, S., Di Martino, B., Venticinque, S., Rak, M., Aversa, R.: Architecturing a Sky Computing Platform. In: Cezon, M., Wolfsthal, Y. (eds.) Towards a Service-Based Internet. ServiceWave 2010 Workshops, Lecture Notes in Computer Science, vol. 6569, pp. 1–13. Springer Berlin / Heidelberg (2011)
12. Petcu, D., Macariu, G., Panica, S., Crăciun, C.: Portable Cloud applicationsFrom theory to practice. Future Generation Computer Systems (2012), http://www.sciencedirect.com/science/article/pii/S0167739X12000210
13. Rackspace: Rackspace cloud (2012), http://www.rackspace.com/cloud
14. Rochwerger, B., Breitgand, D., Levy, E., Galis, A., Nagin, K., Llorente, I.M., Montero, R., Wolfsthal, Y., Elmroth, E., Caceres, J., Ben-Yehuda, M., Emmerich, W., Galan, F.: The Reservoir model and architecture for open federated cloud computing. IBM Journal of Research and Development 53(4), 4:1 –4:11 (july 2009)
15. Varia, J.: Architecting for the Cloud : Best Practices. Compute 1(January), 1–23 (2010)

Performance Evaluation of Model-based Data Access Layers in NoSQL Databases

Tamás Vajk[1], László Deák[1], Gergely Mezei[1], and Tihamér Levendovszky[2]

[1] Budapest University of Technology and Economics,
Magyar Tudósok körútja 2. QB-207., Budapest, H-1117, Hungary
tamas.vajk@aut.bme.hu, ladeak@windowslive.com, gmezei@aut.bme.hu
http://www.aut.bme.hu
[2] Vanderbilt University,
1025 16th Ave S, Suite 102, Nashville, TN 37212, USA
tihamer@isis.vanderbilt.edu

Abstract. Cloud service providers offer a great variety of computational power per virtual machines and several storage mediums, such as SQL databases, associative arrays to store BLOBs, schema-less data tables. These diverse services provide flexible and economical solutions for companies to store, access, and transform their data. Schema-less data tables offer a large number of possibilities to separate the data. These options are not trivial to choose from, performance may diverge greatly in different access scenarios. Thus, in this paper, we focus on the performance of schema-less tables based on different data separation patterns. Based on our measurements, we came to the conclusion that choosing appropriate keys has significantly higher impact on performance than data separation algorithms. Naturally, the development of the different data access layers can be efficiently automated with the use of a data model and a modeling environment that supports code generation from these artifices.

Keywords: NoSQL, Windows Azure table storage, Performance measurement

1 Introduction

Cloud computing has received significant attention recently. Companies can store their data and perform their computations off-premise in a highly available and scalable environment, where they only pay for the resources that they actually use. Compared to traditional infrastructures that only use in-house resources, cloud computing has many advantages that have been transforming the computing solutions used at companies. Naturally, already existing on-premise resources can still be used as part of the infrastructure by connecting them to cloud services and forming a hybrid environment.

From the service provider's perspective, companies with over-scaled computation and storage capacity can now profit from utilizing their unexploited resources and offering a cloud computing provider solution. Their previously gained knowledge of building and maintaining a large infrastructure is now being offered as a service to customers. The most widespread cloud computing platforms include Amazon AWS [1], Google App Engine (GAE) [2], The Rackspace Cloud [3], and Microsoft Windows Azure [4].

Platform as a Service (PaaS) cloud service providers offer several types of data storage mediums. Many software solutions store their data in a relational database, thus, cloud providers typically implement SQL servers for data storage, which makes the transition to the cloud easier [5]. However, if we need to scale out a relational database and spread across multiple servers, the relational constraints – which the database server can guarantee – are significantly reduced. Referential integrity of the data may have to be checked in application logic, which results in application code that is hard to maintain and performance issues arise. Thus, cloud providers tend to offer simpler, but more scalable storage options. Google App Engine, Microsoft Windows Azure, and Amazon AWS all offer BLOB (*Blobstore, BLOB storage, Amazon S3*), queue (*Task Queue, Queue, Amazon SQS*) and schema-less table (*Google Storage, Table storage, Amazon SimpleDB*) storage solutions. Typically binary large object (BLOB) stores are simple key-value pairs, where the key identifies the binary data (1 byte to TBs). Queues provide reliable storage and delivery of messages between application parts. And schema-less table storages or NoSQL databases [6] are containers for structured data, where the data is organized into tables, each table contains rows of data, and each row holds attribute name-value pairs, but attribute names may differ from row-to-row.

In this paper, we have analyzed a number of ways how data can be stored in NoSQL solutions. The history of NoSQL databases is not too long, thus, the design paradigms are not straightforward or commonly known contrary to SQL data scheme design. Database normalization has a strong mathematical background, however, in NoSQL databases, denormalization is a common method of providing better query performance. NoSQL databases are designed for distributed use, which makes indexing resource intensive, thus, most of the solutions lack secondary database indices, which forces developers to combine as much information into the primary key as possible. In this paper, we focus on two aspects of NoSQL databases: (i) the effect of separating the data into different tables and (ii) the selection of primary keys. We have selected a simple example with one-to-many relationships, which would be obvious to map to a SQL schema, and examined the effect of changing the underlying NoSQL-based data access layer on the query performance. For the performance evaluation, we have defined 10 queries with different access scenarios, and measured their execution time as we have increased the amount of stored data. Obviously, hand-coding the different data access layers would be error-prone, thus, we have modeled the data layer in the Visual Modeling and Transformation System (VMTS) [7], and used its *T4* template [8] processing capabilities to generate most of the different data access layers. These layers can be tested by the developer and if the best performing layer is found, later only that layer needs to be generated in the iterative development process. We have measured the query performance over the Microsoft Windows Azure table storage NoSQL implementation.

The structure of this paper is as follows. Sect. 2 gives an overview of cloud service and NoSQL performance evaluations. Also, Sect. 2 introduces promising cloud-based model driven engineering solutions. In Sect. 3, we provide the one-to-many relationship analysis implemented in Windows Azure. We have illustrated the used data model, introduced the examined data access layer variations, and execution time measurements have been provided. Finally, conclusions are drawn and future research options are described in Sect. 4.

2 Background and Related Work

As cloud-based applications are getting widely used even in industrial areas, the performance of cloud services should be evaluated to validate that they meet the expectations. Quantitative performance metrics should be provided to fully evaluate the pros and cons of moving an application to the cloud.

There are several cloud providers that can be chosen for an application. The offered services are typically similar between the providers, thus, selecting the most appropriate one can be a daunting task. On a high level, *CloudCmp* [9] is a tool to systematically compare the performance and cost of cloud providers. *CloudCmp* compares several popular providers (Amazon, Google, Microsoft, Rackspace, etc.) by executing benchmark tests against their computation and storage services. Table storage tests given in [9] consist response-time evaluations for single *get*, *put* and *entity listing* operations. Single *get* operation response time is typically more than 10ms, similarly to what we have measured.

In [10], the perfomance of Windows Azure services have been measured. Scalability, availability and performance have been tested. Table storage performance has been examined under stress test with 192 clients. The experiments show that the number of operations a client could perform against the table store does not depend on the entity size. Also, [10] evaluates the maximum throughput based on different entity sizes and number of clients.

General scalability issues in cloud solutions are introduced in [11]. The high scalability of NoSQL solutions is mentioned, with the drawbacks of not having traditional ACID transaction. Clustered relational databases are also mentioned as an alternative with compromised performance even for moderate loads when transactions are supported. In [12], traditional implementations of ACID transactions are named as the source of declined performance in relational databases. The author concludes that poor performance has nothing to do with SQL, thus, good performance can be achieved in either SQL or NoSQL context as well.

The weaving of model-driven engineering (MDE) and cloud technologies is producing new results mainly in two distinct areas: (i) flexible cloud-based application development, and (ii) unprecedented performance boost in model-based tools. The former one emerges from the fact that model-based tools provide flexible code generation options, thus, only the generator applications need to be developed to provide cloud-based applications from previously existing models

[13]. In the latter one, the previously unavailable computational power and storage space is used in MDE tools. For instance CPU intensive model transformations can be executed in the cloud, for instance [14] performs model verification as a hosted service.

In our work, we have focused on one-to-many relationships as their mapping to NoSQL tables are not evident. In a relational database, their representation would be two tables with one having a column for storing the other's ID and the referential integrity would be enforced by a foreign key. An example of transforming this solution to NoSQL tables can be found in [15]. However, each NoSQL solution has its own restrictions, thus, giving a universally optimal solution is not possible. For instance, in Windows Azure, tables have two keys: (i) a partition key and (ii) a row key. Row keys identify an entity in the table, while along the partition keys the environment may partition the data to several physical servers, which can greatly improve the performance under stress [16]. Keys provide a fast access to entities, having no additional indices in tables means that developers should choose keys that have information coded inside them. For instance, in one of the examples in [16], using a date as a row key is suggested, because that way selecting entries that are created after a certain date can be done efficiently. Also, as operations on keys are strictly limited, for example *Substring()* cannot be called on keys, only string-based comparison can be used for filtering. *Substring()* functionality can be easily implemented with *Compare()*, but it requires keys with fixed length and prefixing [17].

The importance of data partitioning is mentioned in [6] as one of the performance tuning options. Many large-scale cloud applications build on map-reduce [18] solutions, where the map phase emits key-value pairs and reducers consume these pairs. All the key-value pairs that have the same key go to the same partition and get processed by the same reducer. Map-reduce solutions can be efficiently parallelized because corresponding data is located in the same partition and there is no connection between different partitions. For the same reason, to achieve good query times in Windows Azure table storage, only the interrelated data should be stored in the same partition.

3 Contributions

The Visual Modeling and Transformation Tool (VMTS) [7] is a powerful domain-specific modeling and visual model transformation tool. In this work, we have utilized its modeling feature and its ability to traverse the models and generate output from them with transformations expressed in *T4* templates. Fig. 1 illustrates the overall development architecture. After creating the data model, VMTS generates several data access layers over the Windows Azure table store with *T4* templates. The application developer performs the stress test on the different layers, and selects the one that fits best the performance requirements of the application. Afterwards, if the data model is tweaked, the VMTS generator process only needs to produce the selected data access layer. Obviously, if proper performance metrics were found, the selection process could be automatized.

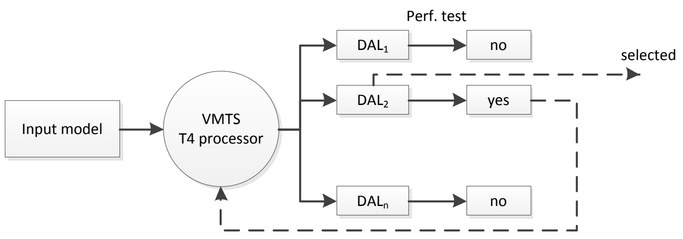


Fig. 1. Overall development architecture.

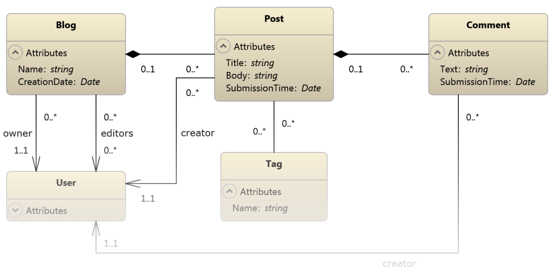


Fig. 2. Blog class diagram.

For our performance evaluation, we have modeled the simplified internal class hierarchy of a blog engine. Fig. 2 depicts the sample VMTS input model used in this paper. Note that in the performance measurements, we have only examined the *Blog*, *Post* and *Comment* elements. However, we have implemented a simple ASP.NET MVC 4 application [19] above the data access layer, where user authentication was necessary and we are currently examining many-to-many relations as well, hence the figure shows the other classes.

3.1 Generated Data Access Layers

From the model in Fig. 2, we have generated nine different data access layers to evaluate their query performance. The generation process has been implemented by *T4* templates [8], which traverse the models and generate C# code from them based on text-based templates. The data has been stored in (i) a single table (Si), (ii) three separate tables (Mu), and (iii) in a mixed way (Mi), where blogs are stored in a separate table, while posts and comments are stored in the same one. Orthogonally to this distinction, we have analyzed the effect of using different partition and row key selection logic. The following options have been implemented: (i) the blog ID is the partition key and a combination of the other keys are used as a row key (B, C), (ii) similarly, the blog ID is the partition key, but the row key is a random identifier (B, R), and (iii) both partition and row keys are randomly chosen (R, R). Table 1 summarizes the options with their reference names used in this paper. Note that although composition is a type of association in the UML standard [20], semantically it means a stronger connec-

Table 1. Examined data access layers.

	Single table	Multiple table	Mixed table (B-PC)
Partition key: Blog Id Row key: Combined	$Si(B, C)$	$Mu(B, C)$	$Mi(B, C)$
Partition key: Blog Id Row key: Random	$Si(B, R)$	$Mu(B, R)$	$Mi(B, R)$
Partition key: Random Row key: Random	$Si(R, R)$	$Mu(R, R)$	$Mi(R, R)$

tion between items, thus, combined row keys are generated based on this observation. In our experiment, combined row keys are using the following pattern: $\{Typename\ of\ element\}[\_\{ID\ of\ owners\}]^*\_\{ID\ of\ element\}$, e.g. the combined row key of a post whose ID is 26, and whose blog's ID is 74 is *post_74_26*. As mentioned in Sect. 2, it is worth using fixed length IDs or padding them to the same length, thus, in the application we are using the 36 character-long string representation of GUIDs.

Naturally, to facilitate the performance measurements, we have defined a common interface that is implemented by the data access layers. Thus, switching the actual implementation that is examined is handled simply by changing a type name in the configuration file.

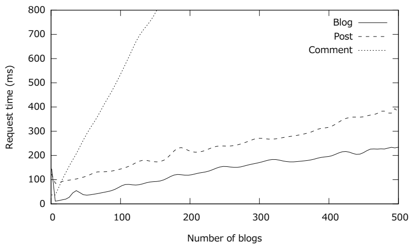
As stated before, in this work, we have only examined key selection and data separation in one-to-many relationships. We have not considered other relationship types and data redundancy, which would lead to other optimization options, has been kept at a minimum.

3.2 Performance Evaluation

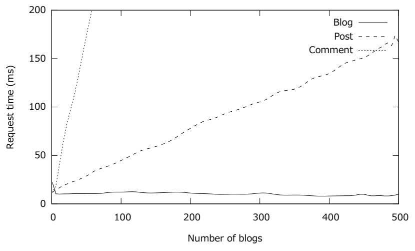
During the measurements, we have linearly increased the number of items in the data store. Each inserted blog contained 100 posts and each post had 10 corresponding comments, thus, with inserting a single blog, 1101 items have been inserted altogether. After each insertion, the execution time of the following 10 queries have been measured:

1. Selecting all the blogs
2. Selecting a single blog by its ID
3. Selecting all the posts for a blog
4. Selecting a single post by blog and post ID
5. Selecting all the comments for a post
6. Selecting a single comment by blog, post and comment ID
7. Selecting all the comments
8. Selecting a comment by ID
9. Selecting a comment by post and comment ID
10. Selecting a post by ID

The above 10 queries follow two navigation scenarios: (i) forward navigation, where we follow the containment relations, and (ii) backward navigation, where we start from the comments, and try to retrieve the container post and blog.

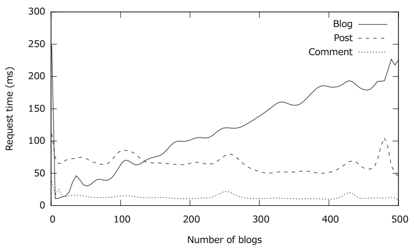


(Get all by container)

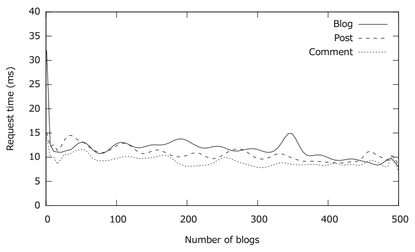


(Get single with IDs)

Fig. 3. Performance of RDBMS solution mapped to NoSQL tables ($Mu(R, R)$).



(Get all by container)



(Get single with IDs)

Fig. 4. Performance of multiple tables with meaningful partition keys ($Mu(B, R)$).

Naturally, in case of a blog engine, typically, only the forward navigations are required, because comments are not visualized without their posts. However, if we consider the same class hierarchy, but changing the blog, post, and comment elements to customer, order, and order item respectively, we gain a structurally similar class diagram, but in this case backward navigations are equally important. Thus, the last 4 items of Listing 3.2 are also required in some scenarios.

Forward navigations There are cases in Table 1 that are definitely not optimal, for instance using random keys in a table storage is rarely a viable way of storing data. As in this case, for instance returning all the comments of a post requires the server(s) to iterate through all the partitions and check items one-by-one if they need to be returned. This solution is obviously not going to provide good performance results, however, it can be considered a good baseline solution as this is exactly the schema that we would use in a relational database. Fig. 3 depicts the execution times of querying the first 6 queries from Listing 3.2 in data access layer $Mu(R, R)$. Note that we have fitted a Bézier spline to the measured raw data to visualize the incline trend.

Changing the partition key to the blog ID, but leaving the row key a random GUID improves the performance immensely as illustrated in Fig. 4. Note that querying all the blogs has not changed, as in both cases the servers need to iterate through the *Blogs* table and return all the elements. However, the execution

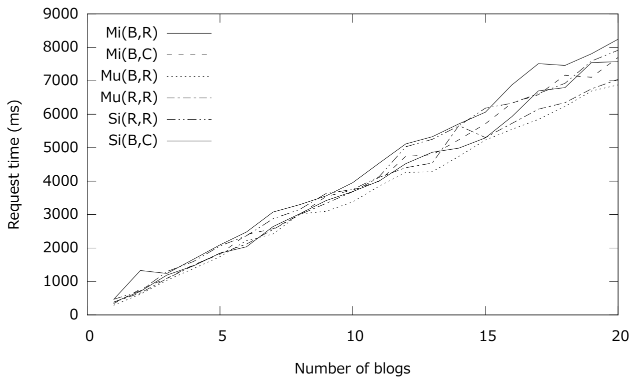


Fig. 5. Execution times of retrieving all the comments.

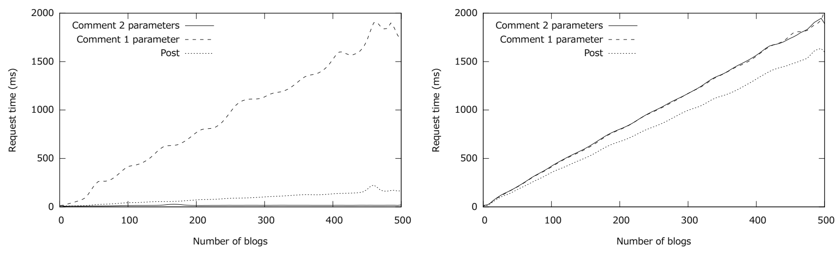
times of all the other queries become almost constants, as in the data access layer $Mu(B,R)$ the single element queries can operate on a single partition, which for instance contains 1000 entries for comments, thus, finding a single entry requires iterating through at most 1000 entries. Similarly, returning the corresponding posts to a blog requires a query to a single partition in the *Posts* table and returning all the elements from that partition.

Performing the same measurements in data access layers $Mu(B,C)$, $Mi(B,C)$, $Mi(B,R)$ results in almost the same execution times, as in those cases, the above observations still apply. Also, single table solutions $Si(B,C)$, $Si(B,R)$ perform similarly in Queries 2 to 6, however Query 1 becomes significantly slower. This performance decline is caused by the fact that Query 1 needs to iterate through all the elements in the table and check whether the row key starts with *"Blog_"* or not. Thus, in this case having too many irrelevant items (posts and comments) in the same table reduces the performance as we cannot restrict the search to a single partition. However, changing the partition key to a fixed value from the blog ID resolves this problem.

Backward navigations In case of a blogging engine, navigating from a comment to a post is rarely a request, however, other scenarios may require navigation from contained to container element. Thus, we have examined those navigations in our case study as well. In a schema-less environment, one might think that storing entities in a single table will not affect performance too much.

The performance of the backward navigation is evaluated with the last 4 items of Listing 3.2. Execution times of querying all the comments is depicted in Fig. 5 for data access layer $Mi(B,R)$, $Mi(B,C)$, $Mu(B,R)$, $Mu(R,R)$, $Si(B,C)$, and $Si(R,R)$.

As it can be seen in Fig. 5, the performance does not vary too much between the different implementations, because to return all the comments, the servers have to iterate through all the elements where the comments are stored. Obviously, this means that the less data is stored in the table that is traversed, the



$(Mu(B, C))$ $(Si(R, R))$

Fig. 6. Performance of backward queries without redundant data.

faster the execution is. Thus, the Multiple table solutions provide the best performance. However, this does not result in significant differences, as the number of comments is way more than the number of other items.

The execution times of the last 3 queries are depicted in Fig. 6 for $Mu(B, C)$ and $Si(R, R)$. The former option provides the best performance, while the latter one is the slowest. Note that finding a comment by its ID takes almost identical amount of time in both cases as there is no partition keys given in those queries, thus, iterating over all the entries in the tables is necessary.

As a general remark, we can conclude that if backward navigation is required in an application, data separation and lucid key selection is not going to provide good enough performance. Thus, other optimization strategies need to be chosen, such as storing data redundantly.

4 Conclusions and Future Work

With the proliferation of cloud services in industrial applications, the performance of these services need to be extensively tested, as applications have to predictably meet behavioral requirements. In this paper, we have examined the performance of the NoSQL solution available in the Microsoft Windows Azure. The performance effect of data separation into multiple tables and the importance of key selection in one-to-many relationships have been tested by generating several different data access layers from a data model.

We have found that data separation does not have a huge effect on performance, however, identifier selection has an immense impact on query execution times. Also we have found that without storing data redundantly, the execution time of many queries hugely depend on the size of the tables. Thus, using separate tables for the different type of entities may result in better performance.

In the future, we aim at exploring the effects of generating data access layers with different redundancy schemes. We believe that after key selection, redundancy has the second largest impact on performance. Also, currently the selection of the optimal data access layer is performed with manual testing. Based on the queries executed on the data, we could automatically select the optimal data access layer. Thus, if the queries were defined previously for instance in Object

Constraint Language (OCL) and the data model were extended with necessary usage performance metrics, the corresponding optimal data scheme could be selected automatically.

Acknowledgments

This work is connected to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" project, supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

References

1. Amazon: Overview of Amazon Web Services. Technical report, Amazon (2010)
2. Severance, C.: Using Google App Engine. 1 edn. O'Reilly Media (May 2009)
3. Rackspace: Cloud Computing, Cloud Hosting & Online Storage. http://www.rackspace.com/cloud/
4. David Chappell: The Windows Azure Programming Model. Technical report, DavidChappel & Associates (2010) Sponsored by Microsoft Corporation.
5. Brunetti, R.: Windows Azure Step by Step. Step by Step. Microsoft Press (2011)
6. Tiwari, S.: Professional NoSQL. Wrox Programmer to Programmer. Wiley (2011)
7. VMTS: Visual Modeling and Transformation System website. http://vmts.aut.bme.hu/ (2012)
8. Vogel, P.: Practical Code Generation in .Net: Covering Visual Studio 2005, 2008, and 2010. Addison-Wesley Microsoft Technology Series. Pearson Education (2010)
9. Li, A., Yang, X., Kandula, S., Zhang, M.: CloudCmp: comparing public cloud providers. In: ACM/USENIX Internet Measurement Conference. (2010)
10. Hill, Z., Li, J., Mao, M., Ruiz-Alvarez, A., Humphrey, M.: Early observations on the performance of windows azure. Scientific Programming **19**(2-3) (2011) 121–132
11. Vaquero, L.M., Rodero-Merino, L., Buyya, R.: Dynamically scaling applications in the cloud. SIGCOMM Comput. Commun. Rev. **41**(1) 45–52
12. Stonebraker, M.: Sql databases v. nosql databases. Commun. ACM **53**(4) (April 2010) 10–11
13. MoDisco: MoDisco Toolbox. http://www.eclipse.org/gmt/modisco/toolBox/
14. Verum: ASD:Suite For Rapid Software Design And Defect-Free Code. http://www.verum.com/
15. Krishnan, S.: Programming Windows Azure - Programming the Microsoft Cloud. O'Reilly (2010)
16. Windows Azure Storage Group: How to get most out of Windows Azure Tables. Technical report, Microsoft Corporation (2010)
17. Jeffrey Richter: Working with Azure Tables with Multiple Entity Schemas. Technical report, Wintellect (2012)
18. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM **51** (January 2008) 107–113
19. Vajk, T., Deák, L.: Sample blog application. http://tableperftest.cloudapp.net/ (2012)
20. Stevens, P., Pooley, R.: Using UML: Software Engineering with Objects and Components. Object Technology Series. Addison-Wesley (1999)

Cloud Computing Workload and Capacity Management Using Domain Specific Modelling

Rafidah Pakir Mohamad, Dimitrios S. Kolovos and Richard F. Paige

Department of Computer Science,
University of York, UK
{rafidah,dkolovos,paige}@cs.york.ac.uk

Abstract. Cloud computing is a technology through which software, platforms and infrastructure can be provided as services. Managing resources which enable this technology is a crucial task to maximise performance and benefits all related parties. Demand for resources starts with application requests initiated by the users. These trigger the creation of virtual machines, which in turn trigger the allocation of physical resources. This paper proposes a solution for cloud computing workload management and capacity planning by utilising domain specific modelling, model transformation, and time series analysis techniques with estimation algorithms. Each application's workload is different depending on its behaviour, software design and the technologies it builds atop. As such, each application needs to provide a custom workload domain specific language (WL-DSL) through which clients can express their anticipated workloads. Models conforming to different WL-DSLs can be transformed to a model conforming to a Virtual Machine domain specific language (VM-DSL) to assist the SaaS or PaaS service provider to manage their virtual resources in the cloud. The IaaS provider can also benefit from numbers of VM-DSL from their customer to manage their physical resources in the actual data centre by transforming multiple incoming VM-DSL models into a model that conforms to a Physical Machine Domain Specific Language (PM-DSL).

1 Introduction

Cloud computing generally offers three types of services: software as service (SaaS), platform as service (PaaS) and infrastructure as a service (IaaS). SaaS provides software applications such as e-mail, file backup and CRM; PaaS provides platforms for software development such as those provided by Microsoft Azure[1] and Google App Engine[2]; and IaaS is a complete computing infrastructure provided as a service for the customer to be used for software development and/or service hosting such as these provided by GoGrid[3] and Amazon EC2[4].

[1] http://www.windowsazure.com/en-us/
[2] http://www.google.com/enterprise/cloud/appengine/
[3] http://www.gogrid.com/
[4] http://aws.amazon.com/ec2/

These three types of services are provided with computing resources through the internet.

Capacity management is the process of assuring that the correct amount of resources is allocated to satisfy the required level of computing demand. The cost of a cloud computing service provider is closely related to resource usage and this renders effective capacity management essential in order to avoid resource over-provision and at the same time to maintain the required level of performance. This study focuses on the scenario in which SaaS and PaaS providers (SPSP) use a computing infrastructure provided by an IaaS provider (ISP) and provide their services to customers as application software or as a software development platform. In this context, capacity management is conducted by two parties. Firstly, by the SPSP, who estimate virtual machine resources demand based on their users' workload, and secondly, by the ISP who ensures that the physical resources are available for the SPSP to create virtual machines in the cloud data centre. Figure 1 shows the relationship between these parties. SPSP tries to minimise operational costs and to maintain the performance at an acceptable level by utilising auto-scaling to avoid resource over provisioning. On the other hand, the ISP also tries to minimise operation costs of data centre by optimising the utilisation of physical machines through consolidation.

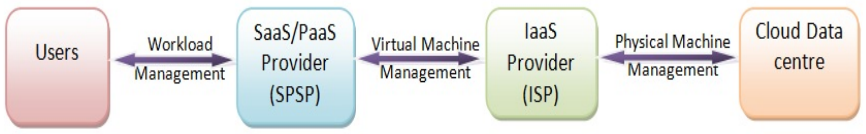

Users
Workload Management
SaaS/PaaS Provider (SPSP)
Virtual Machine Management
IaaS Provider (ISP)
Physical Machine Management
Cloud Data centre

Fig. 1. Relationship between Users, SPSP, ISP and Data Centre

Although auto-scaling is a cost-effective method for SPSP to run their services, it can be particularly challenging for ISPs to manage virtual machines demands on limited physical resources in an efficient manner. Both SPSP and ISP would try to avoid Service Level Agreements (SLAs) violations with their respective customers, but in some cases SLA violations occur with or without control. Managing SLA violations is closely related to capacity management, and several approaches [1, 3, 7] have been proposed for particular types of applications. In this work, we propose an integrated framework for capacity management in scenarios involving multiple applications using domain specific modelling. Domain specific modelling (DSM) is a methodology which advocates constructing and using modelling languages that are tailored to the domain of interest. DSM has been adopted in many fields [9] such as automotive, telecommunications and high-integrity systems and in this work, DSM is utilised in cloud computing resource management to facilitate rigorous specification and automated analysis of workloads.

The remainder of the paper is organised as follows. Section 2 discusses related techniques and tools for workload management and resource estimation methods and Section 3 clarifies the motivation for conducting this study, Section 4 explains the proposal of workload and capacity management by utilising domain specific modelling, and Section 5 concludes the paper by outlining the expected outcomes of this study.

2 Background

2.1 Techniques and Tools to Generate Workloads and to Estimate Resources

The characteristics of workloads are categorised based on application types, e.g., web applications [5, 11, 13], data intensive [8] and media streaming (audio and video) [3]. Synthetic workload generation for selected applications based on the analysis of existing log files is important to estimate the required resources for that particular application [2, 5, 13]. Various parameters are accessed from the application logs for each category of workload since the nature of each category is different. Furthermore, it is difficult to establish generic workload prediction mechanisms because the behaviour of the users of each application as well as its architecture and implementation style make each workload pattern unique.

However, having a standardised workload specification mechanism for each category has been shown to be possible by implementing domain specific modelling languages with identified parameters extracted from previous log information of the application. Bahga and Madisetti have developed the Workload Specification Language (WSL), to examine and compare IaaS package offerings before moving multi-tier web applications to the cloud [2].

Previous workload patterns and their associated system parameters are essential to estimate future workload. For this purpose, parameters required for each category of applications have been compiled in Table 1. Mainly statistical approaches have been used to estimate future workloads such as KCCA [8], PCA [17], regression analysis [13], and Maximum Likelihood Estimation [2]. Table 1 summarises the techniques and tools used for workload generation as well as the workload estimation methods and parameters used in these works.

2.2 Physical and Virtual Machine Capacity Management

To be cost-effective, the right amount of resources need to be allocated for the instantiation of virtual machines (VM) to ensure an acceptable level of performance for the hosted applications and to avoid over or under provisioning of resources. Workload demand prediction is essential for supporting resource autoscaling in cloud computing. However, assigning additional resources to SPSP virtual data centres from limited ISP physical resources can be challenging. An ISP needs to run a minimum number of physical servers with optimum utilisation to fulfil SPSP VM demands with the agreed response time for VM creation in the

Application Specification	**Workloads Generator**	**Workload Estimation**	**Parameter Access from Log File**
Web Application	Workload Specification Language (WSL) [2]. SPECweb99, SURGE, SWAT and httperf	Probabilistic Finite State Machine and Maximum Likelihood Estimation [2]	Input, output, states, transitions and probability of a transition
	Jean 2 model	Semantic descriptions [7]	*not applicable*
	KOOZA [5]	Markov Chain Models for storage, processor and memory. Simple queuing for network.	Storage: block size, type, randomness, inter-arrival times. Processor: CPU utilization. Network: arrival-rate
	not applicable	Autoregressive moving average method (ARMA) [13]	Number of visits to a single page from the total number of customers, number of machines providing the service demand and the think time for clients
Data Intensive	*not applicable*	Kernel Canonical Correlation Analysis [8]	Map time, reduce time, total execution time, map output bytes, HDFS bytes written, and locally written bytes
Media streaming	Medisyn [3]	Mathematical model in a tool called MediaProf	Time, file name, duration, file size, available users bandwidth and elapse end time

Table 1. Techniques and Tools for workloads generation and estimation

SLA. At the same time, the ISP offers the flexibility to increase VM resources with auto-scaling. But sometimes node failure or unpredicted VM demand might occur. Both SPSP and ISP need to perform capacity management to avoid or minimise SLA violations. Several methods for performing capacity management from the SPSP and ISP perspective are outlined in Table 2. SPSPs perform capacity management in virtual data centres where the computing resources are virtually accessed from the cloud, while ISPs perform capacity management in physical data centres.

The main computing resources of interest to capacity management are CPU, memory, storage, disk I/O and network use. Certain works have combined all those resources as a unit [1, 13] while others study only a selected component or a selection of specific components. Ejarque et. al. and Tan et. al. focus on CPU and memory usage [7, 17] and Sun et. al. include storage in their study [16] while [5] explore the combination of four components by extending their studies to include network resources. Ganapathi et. al explore execution time of data intensive workloads for scheduling and resource allocation with KCCA statistic-driven with Hadoop task [8].

Capacity Management	**Cost Model**	**Operational Models**
Virtual Data Centre	reward and penalty based on respond time [1]	Normal and surge operations model [1]. Markov Chain Models; for storage, processor and memory and simple queue model for network [5]. Decision retrieving media file from memory or disk [3]. Minimum resource requirement [7]. Gradually increase number of machine to identify the right number of resources required by calling the Mean Value Analysis [13]. Kalman filter, double exponential smoothing, and Markov prediction [12]. input, output, states, transitions and probability of a transition
Physical Data Centre	Customer Priority model [7]. Cost with energy consumption and cost of VM creation [6]	Surplus resource distribution [7]. Prebooted and preconfigured VM instances with common feature [6]. Markov chain technique [10]. Multi-dimensional Resource Integrated Scheduling algorithm [16]

Table 2. Physical and Virtual Machine Capacity Managements

2.3 SLA related issues in workload management

Quality of Service (QoS) in an SLA can refer to response time, throughput, availability, reliability, and security [10] of the provided service. Response time is more favourable in most studies [1, 5, 6, 10] as it can be easily measured and associated with resource demand.

2.4 Domain Specific Modelling

Domain Specific Modelling is a technique advocating the use of Domain Specific Modelling Languages (DSMLs) for modelling solutions in particular application domains [15]. The rationale behind DSM is that each application domain is characterised by its own set of abstractions which are represented more precisely and effectively using tailored modelling languages instead of generic languages such as UML. A Domain Specific Language consists of five fundamental components [4]: abstract syntax, concrete syntax, syntactic mapping, semantic domain and semantic mapping. In a DSML, these components are used to formally represent a specific set of structural, behavioural and requirement features of a particular domain in the form of a meta-model. A meta-model is a model that describes the abstract syntax and static semantics of a DSML. Concrete syntax is the concrete form of textual or graphical constructs used to create a model.

3 Motivation

Most of the previous works [1, 3, 6, 12] are focused on managing resources for a single application rather than considering a combination of applications. Work conducted in [3] proposed a VM demand estimation formula for media streaming and this formula can be adopted to estimate numbers of virtual machines needed by considering the cost of accessing media files from disk or memory. Workload forecasting methods such as ARMA [13], PCA [17], KCCA [8], Kalman Filtering [12] are good prediction methods for times series in statistics to predict future workload. In addition, Dougherty et. al [6] used Model Driven Engineering to assist in green auto-scaling which reduce energy consumptions resulting from idle machines. Feature models are used as an abstraction which describes the software platform's behaviour and its associated configurable variables for a single application in [6].

Three important phases are identified in providing services in cloud environments. Firstly, managing fluctuating workloads initiated by end-users and these workloads can be for different SaaS or PaaS provider using IaaS as IT solution. Secondly, managing virtual machine resources, which are used to run the services; and finally, managing physical resources in IaaS data centres. Previous research has focused on individual phases. This work proposes an integrated framework for capacity management from the end-user to the infrastructure service provider. A set of domain specific modelling languages (DSMLs) is used to facilitate the integrations of these three phases.

Each application hosted on a virtual machine has its own unique characteristics and as such workload specification models are unique to each application. The behaviour of web application and media stream application is not the same although they might have common attributes as an application. Furthermore, technology and the architecture used to construct the application also makes the application is unique to each other. This can be expressed using a DSM language tailored to the application. A DSM language can precisely capture all the parameters needed to express the estimated user workloads, and the workload models can then be analysed and consolidated in order to estimate the virtual machine and physical resource usage. While applications are unique and each of them requires its own DSML to specify its workload, it is anticipated that a single domain specific language will be sufficient to express virtual machine workload specifications by capturing requirements for CPU, memory and other resources over time. Virtual machine workload models can be used to perform capacity management, in order to achieve an acceptable balance between performance and cost. Moreover, the information captured in the virtual machine demand model (VM-DSL) can be used to optimise physical resources in the ISP data centre.

4 Proposed Work

Workloads generated by users utilise the virtual machine resources in SPSP virtual data centres. The SPSP needs to provide sufficient resources with the correct specifications of virtual machines offered by the ISP to assure the adequate performance of the hosted applications and minimise the cost by not over provisioning resources. Previous works focus only on a particular application while in practice SPSPs typically provide multiple applications as services sharing the virtual machines and computing resources. As discussed earlier, each application has its own unique characteristics and as such workload specification models are unique for each application and this can be expressed using a DSM language tailored to that application. A DSM language can precisely capture all the parameters needed to express users' estimated workloads, and workload models can then be analysed and consolidated in order to estimate VM and physical resource usage.

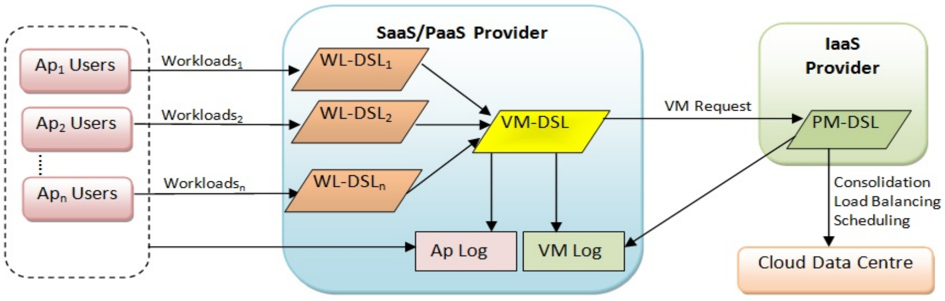


Fig. 2. Workload and Capacity Management Framework

While applications are unique and each of them requires its own DSML to specify its workload, we anticipate that a single domain specific language is sufficient to express virtual machines workload specifications by capturing requirements for CPU, memory, storage, disk I/O, network bandwidth and response time. VM workload models can be used to perform capacity management and achieve an acceptable balance between performance and cost. Moreover, the information captured in the virtual machine demand model (VM-DSL) can be used to optimise physical resources on the ISP data centre. Efficient physical resource utilisation with consolidation reduces the operating cost of data centres [14].

Figure 2 illustrates the framework proposed in this work to manage workload and capacity using domain specific modelling languages in three stages. First, each application will have their unique workload specification DSL. Secondly, application workload specification models can be transformed to virtual machine workload specification models (instances of the VM-DSL) to estimate the overall virtual machine resource demand and to perform a VM request from the ISP. Finally, the ISP can transform a collection of VM-DSL models from their customers to instances of the physical machine DSL (PM-DSL) to estimate the overall demand for physical resources in the data centre.

5 Expected Research Outcome

The proposed work is aimed at assisting capacity managers who are providing and/or using IaaS to plan future resource requirements in physical and virtual data centres. This task is performed by integrating the analysis techniques identified from the literature. It is anticipated that using DSML models to specify workloads will render capacity management more flexible, precise and effective.

References

1. Bruno Abrahao, Virgilio Almeida, Jussara Almeida, Alex Zhang, Dirk Beyer, and Fereydoon Safai. Self-adaptive sla-driven capacity management for internet services. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 557–568, April 2006.
2. Arshdeep Bahga and Vijay Krishna Madisetti. Synthetic workload generation for cloud computing applications. *Journal of Software Engineering and Applications*, 4(7):396–410, July 2011.
3. Ludmila Cherkasova, Wenting Tang, and Sharad Singhal. An sla-oriented capacity planning tool for streaming media services. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 743–752, Washington, DC, USA, 2004. IEEE Computer Society.
4. Tony Clark, Andy Evans, Stuart Kent, and Paul Sammut. The mmf approach to engineering object-oriented design languages. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications*, April 2001.
5. Christina Delimitrou and Christos Kozyrakis. Cross-examination of datacenter workload modeling techniques. In *31st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 72 –79, june 2011.
6. Brian Dougherty, Jules White, and Douglas C. Schmidt. Model-driven auto-scaling of green cloud computing infrastructure. *Future Generation Computer Systems*, 28(2):371–378, 2012.
7. Jorge Ejarque, Marc de Palol, Inigo Goiri, Ferran Julia, Jordi Guitart, Rosa M. Badia, and Jordi Torres. Sla-driven semantically-enhanced dynamic resource allocator for virtualized service providers. In *Fourth IEEE International Conference on eScience*, 2008.
8. Archana Ganapathi, Yanpei Chen, Armando Fox, Randy Katz, and David Patterson. Statistics-driven workload modeling for the cloud. In *IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, pages 87–92. IEEE, March 2010.

9. Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.
10. Hamzeh Khazaei, Jelena Misic, and Vojislav B. Misic. Modelling of cloud computing centers using m/g/m queues. In *31st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 87–92, June 2011.
11. Xue Liu, Xiaoyun Zhu, Sharad Singhal, and Martin Arlitt. Adaptive entitlement control of resource containers on shared servers. In *9th IFIP/IEEE International Symposium on Integrated Network Management*, 2005.
12. C.C.T. Mark, D. Niyato, and Tham Chen-Khong. Evolutionary optimal virtual machine placement and demand forecaster for cloud computing. In *IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 348–355, March 2011.
13. Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 500–507, July 2011.
14. Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, Berkeley, CA, USA, 2008. USENIX Association.
15. Thomas Stahl, Markus Vølter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development: Technology, Engineering, Managementb*. Wiley, 2006.
16. Xin Sun, Sen Su, Peng Xu, Shuang Chi, and Yan Luo. Multi-dimensional resource integrated scheduling in a shared data center. In *31st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 7–13, June 2011.
17. Jian Tan, P. Dube, Xiaoqiao Meng, and Li Zhang. Exploiting resource usage patterns for better utilization prediction. In *31st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 14–19, June 2011.

Model-Driven Cloud Data Storage

Juan Castrejón[1], Genoveva Vargas-Solar[2],
Christine Collet[3], and Rafael Lozano[4]

[1] Université de Grenoble, LIG-LAFMIA,
[2] Centre National de la Recherche Scientifique, LIG-LAFMIA
[3] Grenoble Institute of Technology,
681 rue de la Passerelle, Saint Martin d'Hères, France
`{Juan.Castrejon, Genoveva.Vargas}@imag.fr`,
`Christine.Collet@grenoble-inp.fr`
[4] Instituto Tecnológico y de Estudios Superiores de Monterrey,
Campus Ciudad de México, Calle del Puente 222, México, México
`ralozano@itesm.mx`

Abstract. The increasing adoption of the cloud computing paradigm has motivated a redefinition of traditional software development methods. In particular, data storage management has received a great deal of attention, due to a growing interest in the challenges and opportunities associated to the NoSQL movement. However, appropriate selection, administration and use of cloud storage implementations remain a highly technical endeavor, due to large differences in the way data is represented, stored and accessed by these systems. This position paper motivates the use of model-driven techniques to avoid dependencies between high-level data models and cloud storage implementations. In this way, developers depend only on high-level data models, and then rely on transformation procedures to deal with particular cloud storage details, such as different APIs and deployment providers, and are able to target multiple cloud storage environments, without modifying their core data models.

1 Introduction

Cloud computing represents one of the most promising paradigms for software development nowadays, due to its natural separation between users, applications and the services they require. In this utility computing model, resources are provided as services, easily accessible over a distributed network [12].

Cloud storage represents a paradigm to *store*, *retrieve* and *manage* large amounts of data, using highly scalable distributed infrastructures. This area has received a great deal of attention in recent years, due to a growing interest in the challenges and opportunities associated to the NoSQL movement [3]. However, unlike traditional environments, where the use of the *relational model* is pervasive, there is a wide variety of data models that can be used in cloud applications. These data models include [3]: *key-value*, *document*, *extensible record*, *graph* and *relational* repositories. Each of these data models are designed for different use cases, and provide different support for functional and non-functional requirements of distributed systems [3], such as different degrees of *consistency*, *scalability*, *replication* and *concurrency* [3]. Moreover, there is also a wide variety of both public and private providers for the distributed infrastructure that is required for cloud data storage [15]. These providers offer different combinations of *pricing*, *support*, *service levels*, and usually have different APIs to *store*, *retrieve* and *manage* data. These differences make it difficult to design and deploy applications targeting different cloud environments [16].

A key challenge in this heterogeneous environment is the appropriate selection of a data store that best matches the requirements of particular applications [15, 16]. This can be a daunting task, due to the high number of implementations in this environment, over *120* as of this writing [5], and the technical knowledge required to make an appropriate selection, as outlined in [15].

Furthermore, applications may require more than one type of data store, in order to support different use cases. In this regard, the appropriate use of data stores, either traditional or NoSQL, to support multiple use cases in a single application, is currently being studied as part of an emerging movement, named *polyglot persistence* [6]. Nonetheless, the selection and administration of suitable storage systems for each use case, remain an open challenge.

This paper motivates the use of model-driven engineering (MDE) techniques [10], in order to characterize cloud data storage requirements, and to effectively encapsulate the *selection*, *administration* and *use* of cloud data storage implementations, specially, in polyglot persistence environments. We believe that MDE is a natural fit for this purpose, due to its emphasis in relying on different levels of modeling notations [10], which can be ultimately used to generate the implementation of software systems [1]. In particular, these multi-level structures can be used to avoid dependencies between high-level data models, cloud storage implementations and deployment providers.

The remainder of this paper is organized as follows. Section 2 outlines collaborations between cloud data storage and MDE. Section 3 describes related work. Finally, conclusions and future challenges, are discussed in Section 4.

2 Model-driven cloud data storage

In this section, we outline a set of collaborations between cloud data storage and MDE, that are intended to avoid dependencies between high-level data models and cloud storage implementations. In particular, we strive for the following objectives: (i) provide adequate notations and environments to characterize cloud data storage requirements; (ii) selection of storage implementations and deployment providers; and, (iii) management of the required artifacts to work with different combinations of cloud storage implementations and providers.

2.1 Data modeling for the cloud

The specification of data models for software systems is traditionally performed using notations such as entity-relation (ER) or UML class diagrams. MDE techniques can currently be applied to transform these diagrams into their corresponding relational database models and programming language entities.

However, these notations are usually not enough to characterize all the possible cloud data models. For instance, consider the *document* data model [3], that lacks a rigid schema and in which semi-structured information can be stored. Another example would be the adequate modeling of *families of attributes* [3], usually associated to *extensible record* scenarios [3]. In this regard, different techniques are currently being proposed to overcome these limitations [9].

One of the objectives of our current work is the definition of adequate notations and environments for the modeling of datasets, and their associated functional and non-functional requirements, for cloud environments. For this, we intend to rely on the ISO/IEC Software Product Quality Requirements and Evaluation (SQuaRE) standards [8], that already define software quality characteristics, such as *performance efficiency*, *portability* and *functional suitability.* These international standards also provide guidelines for the association, and evaluation, of metrics associated to quality characteristics. In this case, we propose to define these characteristics through the association of metrics relevant to cloud scenarios. We can reuse previously proposed metrics [15], such as *performance*, *cost* and *access latency*, but further validation of these metrics is also required. In particular, our modeling environments would allow users to specify expected values for these metrics, according to their datasets requirements.

We intend to organise our modeling notations based on a traditional MDE structure of platform independent and specific models (PIM/PSM) [10], in regard to cloud storage implementations. In this way, we could integrate current research and industrial efforts, such as specification languages for modeling cloud environments [11], and different cloud data management interfaces [7, 18].

2.2 Data storage selection

In order to ease the selection of data storage implementations and providers, we propose a decision process based on the analysis of historic data and usage patterns, both in test applications and within systems generated in our modeling environment. This analysis could be performed in a non-intrusive manner, during application runtime, by automatically generating aspect-oriented programming (AOP) monitoring artifacts, as outlined in [2]. In particular, dynamic crosscutting techniques can be used to monitor the behavior of the selected data stores, regarding the metrics associated to the SQuaRE quality standards. This monitoring information could then be used to compare with the expected values specified by the users of our modeling notations and environments. In turn, this analysis could be automatically integrated in applications designed with our modeling notations, with the objective of sharing the results in an open and collaborative environment, that could be exploited by new users of cloud data storage, and by our own modeling tools, as input for the data storage recommendation engine.

Developers could also generate the artifacts to work, at the same time, with multiple combinations of implementations and providers, as outlined in [2]. For instance, this would be helpful to compare their performance in real scenarios.

2.3 Cloud artifacts generation and management

Once the data storage implementations and providers have been selected for the application datasets, we propose to use transformation procedures to generate the low-level artifacts to work with them, that is, configuration files for the deployment environments and cloud data management interfaces. This process could be performed using different levels of transformation procedures, each of them more dependent with particular storage implementations and providers, using a similar approach as modern application development tools [17].

For example, an initial transformation could be defined between the graphical data models and an intermediate domain specific language (DSL), possibly extending the work in [11] and [17]. From this DSL, we could generate configuration files for the particular storage implementations, the AOP monitoring aspects, and the configuration of data management interfaces [7, 18].

3 Related work

The *Modeling as a Service* (MaaS) initiative is proposed in [1] as an approach to deploy and execute model-driven services over the Internet. This initiative is aligned with SaaS principles, since consumers do not manage the underlying cloud infrastructure and deal mostly with end-user systems. Our work deals with lower level service models (PaaS and IaaS) by allowing control over the deployed applications and the configuration settings of the deployment environments.

A model-driven approach for designing and deploying scalable applications on cloud platforms is described in [13]. This approach promotes the use of graphical models in order to capture cloud requirements, in particular, scalability features. These models are then bundled into a generic platform that automatically deploys them into PaaS and IaaS environments. Instead of striving for a generic platform, our work is focused only on data storage features.

An approach for the automatic selection of cloud storage services is proposed in [15]. This approach relies on the characterization of storage systems, based on capabilities, such as *performance* and *cost*, and on the specification of requirements for application datasets, such as *expected dataset size*, *access latency* and the *number of concurrent clients*. Based on this information, an assignment of datasets to the storage systems is proposed, for example, using a mathematical model that strives for optimal data allocation [16]. In comparison, we propose a recommendation engine based on the monitoring of usage patterns.

4 Conclusions and Future work

This paper outlined collaborations between MDE and cloud data storage, intended to facilitate both the specification of cloud data storage requirements, and to encapsulate the *selection*, *administration* and *use* of cloud data storage.

We mentioned challenges that are required to make these collaborations possible, and that are currently being addressed by members of our research group.

For future work, we intend to use our approach in the context of networking applications, managed as part of the UBIQUEST [4] and CLEVER [14] projects.

Acknowledgments

This work is funded by the French National Research Agency, through the UBIQUEST project (http://ubiquest.imag.fr) ANR-09-BLAN-0131-01, and by the STIC-AMSUD program, within the CLEVER project (http://clever.imag.fr).

References

1. Bruneliere, H., Cabot, J., Jouault, F.: Combining Model-Driven Engineering and Cloud Computing. In: Modeling, Design, and Analysis for the Service Cloud Workshop. MDA4ServiceCloud '10 (2010)
2. Castrejón, J.: An Aspect Oriented Approach for the Synchronization of Instance Repositories in Model-Driven Environments. RCS 52, 179–189 (2011)
3. Cattell, R.: Scalable sql and nosql data stores. SIGMOD Rec. 39, 12–27 (May 2011)
4. Collet, C., Ahmad-Kassem, C., Bobineau, C., Double, E., Ma, F., Martínez, S., Grumbach, S., Ubéda, S.: A Data-Centric Approach for Networking Applications. In: International Conference on Data Technologies and Applications (2012)
5. Edlich, S.: List of NoSQL Databases. http://nosql-database.org/ (March 2012)
6. Fowler, M.: Polyglot Persistence. http://martinfowler.com/bliki/ PolyglotPersistence.html (November 2011)
7. jclouds Inc.: jclouds. http://www.jclouds.org/ (March 2012)
8. ISO/IEC:25010:2011: Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. ISO, Geneva, Switzerland (2011)
9. Katsov, I.: NoSQL Data Modeling Techniques. http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/ (March 2012)
10. Kent, S.: Model Driven Engineering. In: Butler, M., Petre, L., Sere, K. (eds.) Integrated Formal Methods, LNCS, vol. 2335, pp. 286–298. Springer Berlin (2002)
11. Liu, D., Zic, J.: Cloud#: A Specification Language for Modeling Cloud. In: Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing. pp. 533–540. CLOUD '11, IEEE Computer Society, Washington, DC, USA (2011)
12. National-Institute-Standards-Technology: The NIST Definition of Cloud Computing. http://csrc.nist.gov/publications/PubsSPs.html (September 2011)
13. Peidro, J.E., Muñoz-Escoí, F.D.: Towards the Next Generation of Model Driven Cloud Platforms. In: 1st International Conference on Cloud Computing and Services Science. pp. 494–500. CLOSER '11 (2011)
14. Portilla, A., Hernández-Baruch, V., Vargas-Solar, G., Zechinelli-Martini, J., Collet, C.: Building reliable services based mashups. In: IV Jornadas Científico-Técnicas en Servicios WEB y SOA. JSWEB 2008 (2008)
15. Ruiz-Alvarez, A., Humphrey, M.: An Automated Approach to Cloud Storage Service Selection. In: Proceedings of the 2nd International Workshop on Scientific Cloud Computing. pp. 39–48. ScienceCloud '11, ACM, New York, NY, USA (2011)
16. Ruiz-Alvarez, A., Humphrey, M.: A Model and Decision Procedure for Data Storage in Cloud Computing. In: Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. CCGrid '12 (2012)
17. SpringSource: Spring Roo. http://www.springsource.org/spring-roo (May 2012)
18. Storage-Networking-Industry-Association: Cloud Data Management Interface. http://www.snia.org/cdmi (September 2011)

First International Workshop on Academics Modelling with Eclipse

ACME 2012
(co-located with ECMFA 2012)

Proceedings

2 July 2012

DTU Lyngby, Denmark

Editors: Dimitris Kolovos, Davide di Ruscio, Louis Rose

Preface

The 1^{st} Academics Modelling with Eclipse (ACME) Workshop was organised as a satellite event of the 2012 European Conference on Modelling Foundations and Applications (ECMFA 2012) which was held at the Technical University of Denmark (DTU), Kgs. Lyngby, Denmark, during July 2-5, 2012.

The Eclipse platform has played a very significant role in the evolution of MDE over the last few years as it has provided mature infrastructure - predominately the Eclipse Modeling Framework - for the development of interoperable modelling and model management languages and tools. The academic community has in turn embraced Eclipse as the de-facto standard MDE environment and has contributed several modelling and model management tools back as open source projects - some of which have been brought under the umbrella of the Eclipse Foundation.

The aim of this workshop was to provide a venue where developers of research-oriented MDE tools built on top of Eclipse could demonstrate the most recent developments in their tools, provide insights on issues encountered when using these tools in practice, obtain feedback, exchange expertise, and engage in fruitful discussions with like-minded researchers.

In this first incarnation of the ACME workshop, the Program Committee received 10 paper submissions. From them, 7 were accepted for presentation at the workshop and publication in these proceedings. To maintain a strong Eclipse focus, all papers were asked to provide an Eclipse update site (and possibly more detailed instructions) through which the presented tool could be installed and tested on a fresh copy of the Eclipse Indigo Modelling distribution. Moreover, all submissions included a link to a screencast demonstrating the tool presented.

The keynote speaker of the workshop was Bruce Trask, from MDE Systems in the USA. We thank him very much for accepting our invitation and for his enlightening talk.

We are grateful to our Program Committee members for providing their expertise through high-quality and timely reviews. Their helpful and constructive feedback on all submitted papers is most appreciated. We also thank the ECMFA General Chairs for their advice and guidance.

Workshop Organisers: Dimitris Kolovos (University of York), Davide di Ruscio (University of L'Aquila), Louis Rose (University of York)

Programme Committee: Cedric Brun, Antonio Cicchetti, Nicholas Drivalos, Esther Guerra, Jendrik Johannes, Jan Koehnlein, Alfonso Pierantonio, Istvan Rath, Jess Snchez Cuadrado, Massimo Tisi, Antonio Vallecillo, Pieter Van Gorp, Juan Manuel Vara, Edward Willink, Steffen Zschaler

Conper: Consistent Perspectives on Feature Models

Julia Schroeter[1], Malte Lochau[2] and Tim Winkelmann[2]

[1] TU Dresden
Institute for Software- and Multimedia-Technology
`julia.schroeter@tu-dresden.de`
[2] TU Braunschweig
Institute for Programming and Reactive Systems
`{m.lochau,t.winkelmann}@tu-bs.de`

Abstract. Domain feature models express commonality and variability among variants of a software product line. For separation of concerns, e.g., due to legal restrictions, technical considerations, and business requirements, views restrict the configuration choices on feature models for different stakeholders. Our tool *Conper* allows to create views that obey different feature model semantics. We call such views perspectives on a feature model. We use a structured view model based on the Eclipse Modeling Framework (EMF) to define concern-relevant views as well as dependencies and hierarchies between views w.r.t. their concerns. Our tool supports the composition of views to create perspectives and offers an efficient algorithm to verify their consistency. The tool is applied in a staged configuration process for feature model preconfiguration. Additionally, as it is possible to define restricted perspectives per stakeholder, *Conper* supports customization on feature model level.

Keywords: Software Product Lines, Feature Models, Preconfiguration, Customization, Automated View Composition

1 Introduction

Feature models are used in software product line (SPL) engineering to express variability and commonality among product variants [3]. They specify the variant space. Due to various reasons the variant space is further restricted prior variant derivation. Reasons for this are driven by business concerns, e.g., to enable a variable pricing strategy for selling features in packages as well as by technical concerns, e.g., to identify a representative subset of variants for efficiently SPL testing (cf. [5]). It seems promising to express such concerns in a separate view model. Prior the derivation of a variant, concern-related views are selected and the domain feature model is filtered accordingly [6]. We call such a semantic preserving preconfiguration a *perspective* on the domain feature model. Various approaches to create views on feature models exist in literature [1, 2]. Though, these approaches focus on separation of concerns, whereas a particular view

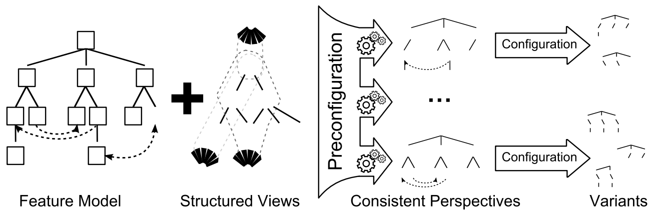


Fig. 1. *Conper* allows to define structured views on a feature model and create consistent perspectives by composing them.

is not intended to derive a complete variant, but rather to allow for specific configuration decisions only. Our tool *Conper* supports the definition of views on feature models in general and their aggregation to create consistent perspectives in particular.

The remainder of this paper is structured as follows. We briefly explain the concepts of perspectives on feature models in Sect.2. Sect. 3 describes the tool *Conper* followed by Sect. 4 which explains our findings and conclusion.

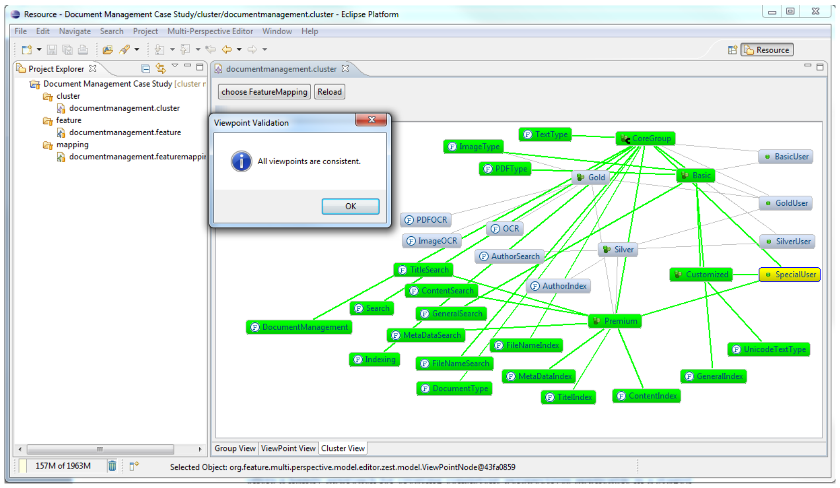
2 Consistent Perspectives on Feature Models

A consistent perspective imposes a specialization of the configuration space, i.e., a refinement of the original feature model semantics. A perspective is created by composing various views. A view represents a certain concern of the feature model. In our approach, a domain feature model and a view model are unified in a multi-perspective model, which imposes a conservative extension to the domain feature model. Due to the fact that only a valid subset of domain feature model variants is required to be derivable, not all possible view combinations form perspectives. Therefore, we use the concept of viewpoints to explicitly define valid view aggregations. The creation of a perspective is considered as a preconfiguration step in the variant derivation process as we show in Fig. 1. Furthermore, perspectives allow customization on feature model level in the way that stakeholder-specific features are only available to a particular stakeholder in the stakeholder's perspective.

Ensuring multi-perspective model consistency is, in general, hard to maintain due to the crosscutting nature w.r.t. the feature model, its cross-tree constraints and its potential overlapping of feature groups in a view model. Therefore, besides a comprehensive brute force approach, we developed an incremental heuristic for a scalable consistency verification of perspectives [6].

3 Conper: A Tool for Creating Consistent Perspectives

We implement *Conper* as plug-ins for the Eclipse Modeling IDE and extend the existing feature modeling and variant derivation environment *FeatureMapper* [4]. Further information, the source code as well as some example projects



All viewpoints are consistent.

Selected Object: org.feature.multi.perspective.model.editor.zest.model.ViewPointNode@43fa0859

Fig. 2. Conper: Visualization of the relation between features and views.

and various screencasts of *Conper* are provided online[3], as well as an Eclipse update-site[4].

Our tool integrates the concepts of view models, feature models and mappings between them, as well as mechanisms to derive consistent perspectives. A *View Model* captures the relationships between views and viewpoints. We implement this concept using the Eclipse modeling framework (EMF). As the *FeatureMapper* offers an EMF-based feature meta-model that allows to create feature models with group cardinalities, we were able to seamlessly combine our approach with this environment. In addition, the *FeatureMapper* defines mappings between feature models in the problem space and EMF-based solution space artifacts. We reuse this functionality in our approach to create the assignments between features of the feature model and groups of the view model.

Conper provides various editors focussing on different aspects of the multi-perspective model. In Fig. 2, we show a screenshot of *Conper*. In the shown *Cluster View*, a viewpoint named "SpecialUser" is selected, and related view groups and features are highlighted. Furthermore, the Group View is a top down view, showing the view model starting from the core group. It is used during domain engineering to create view groups and viewpoints. In combination with the mapping view provided by the *FeatureMapper*, features are assigned to view groups in this view. Another Eclipse view, the *Viewpoint View* complements the *Group View* as it represents the view model bottom up, starting from a selected viewpoint and therefrom showing those groups the viewpoint is directly and indirectly assigned to. The *Cluster View* visualizes the mapping between feature model and the view model. It shows the assignment of features to view groups

[3] https://github.com/multi-perspectives/cluster/wiki
[4] http://juliaschroeter.de/conper/update

and which features belong to a certain viewpoint. Finally, we use the *Mapping View* provided by the *FeatureMapper* to assign features to view groups. By selecting a viewpoint in one of the editors, it is possible to derive a perspective. If the consistency check succeeds the corresponding perspective is derived by automated view composition and persisted in the Eclipse workspace stating a preconfigured feature model for subsequent product configurations.

4 Lessons Learned

The Eclipse IDE is a convenient platform to easily integrate new functionality. We used the frameworks EMF, EMFText and Zest as they offer powerful graphical and textual modeling capabilities. An issue we experienced with EMF is that our view model is a lattice graph as we support multiple inheritance relations among group views, whereas the EMF meta-model is tree-based. Therefore, some implementation effort was needed. One key finding of our experiments considering very large feature models with up to $10,000$ features is, that our heuristic consistency check algorithm is efficient as it scales well. Depending on the size of the view model, it identifies within milliseconds which viewpoints are consistent and which are not. In general, our experiences with various case studies show that perspectives are a promising concept for tailoring and customizing the variant space of a domain feature model to multiple stakeholders' concerns. Another finding is that there is a strong need for an appropriate visualization concept of the assignment of features in the feature model to groups in the view model. Our Zest-based cluster view provides a good overview of a complete mapping, but we need to support the mapping process graphically as well. In general, our tool *Conper* offers a valuable approach for creating consistent perspectives integrable in a staged configuration process with support for customization on feature model level.

Acknowledgements The presented work is co-funded by the European Social Fund, Federal State of Saxony and SAP AG in the research project #080949335.

References

1. Abbasi, E., Hubaux, A., Heymans, P.: A toolset for feature-based configuration workflows. In: Proceedings of SPLC'11 (2011)
2. Clarke, D., Proença, J.: Towards a theory of views for feature models. In: Proceedings of FMSPLE'10 (2010)
3. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
4. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: Proceedings of AOPLE'07 (2007)
5. Lochau, M., Oster, S., Goltz, U., Schürr, A.: Model-based pairwise testing for feature interaction coverage in software product line engineering. Software Quality Journal (2011)
6. Schroeter, J., Lochau, M., Winkelmann, T.: Extended version of multi-perspectives on feature models. Tech. Rep. TUD-FI11-07-Dezember 2011, TU Dresden (2011)

FAMILE: Tool support for evolving model-driven product lines

Thomas Buchmann and Felix Schwägerl

Lehrstuhl Angewandte Informatik 1, University of Bayreuth
D-95440 Bayreuth
firstname.lastname@uni-bayreuth.de

Abstract. Model-driven development is a well-known practice in modern software engineering. Many tools exist which allow developers to build software in a model-driven way. Unfortunately, these tools do not provide dedicated support for the specific needs in software product line processes. Only recently some approaches tried to combine feature modeling and model-driven development. In this paper we present a new approach that allows for the combination of feature models and Ecore based domain models and keeps both models consistent during evolution.

1 Introduction

Software product line engineering [1, 2] deals with systematic development of products belonging to a common system family based on organized reuse of software artifacts. Thus, composing products from a library of reusable components, rather than developing each product instance from scratch is preferred. *Model-driven software engineering* [3] puts strong emphasis on the development of higher-level models rather than on the source code. Both techniques promise to increase productivity. In the past, several approaches have been made to combine both techniques to get the best out of both worlds. Only recently, *model-driven software product line engineering* has been established as an integrating discipline. This paper contributes to this discipline by presenting FAMILE (Features and mappings in lucid evolution), a new tool to combine feature models and domain models using an explicit mapping model. Furthermore, textual languages are provided to (a) annotate domain model elements with feature expressions and (b) specify dependencies and (automatically derived) repair actions to ensure the well-formedness of configured domain models.

2 Tool support

When developing software product lines in a model-driven way, various models are involved: Features have to be mapped onto corresponding domain model elements realizing them. Figure 1 shows the Ecore-based [4] models and metamodels part of our toolchain. For the *domain model*, arbitrary EMF modeling languages can be employed. In our running example, we use our own UML2

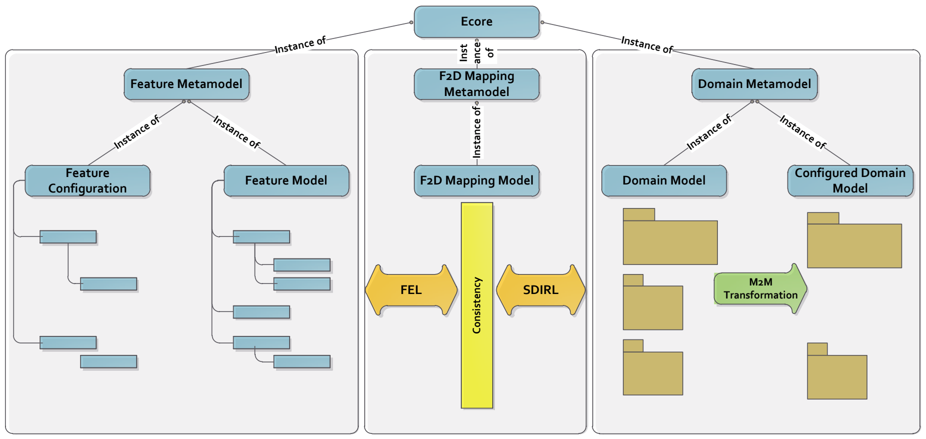
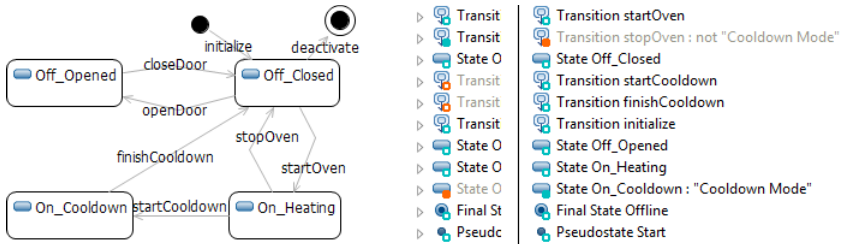


Fig. 1. Involved models and meta-models.

based domain modeling tool. The *feature model* [5] consists of a tree of features. A non-leaf feature may be decomposed in two ways: In the case of an AND decomposition, all of its child features have to be selected when the parent is selected. In contrast, for an OR decomposition exactly one child has to be selected. For one feature model, a number of *feature configurations* exist, each defining the selection of features for one product. A mapping model (F2DMM) is used to interconnect both feature model and domain model. The accompanying screencasts (see section 4) demonstrate the mapping editor's use.

In our previous work [6], we used implicit feature annotations and propagation of features to dependent model elements to ensure the well-formedness of configured domain models in case a selected element requires the inclusion of an unselected one. Contrastingly, in our current approach, selection states rather than annotations are propagated. The user can choose between different *propagation strategies*: The *suppress active* strategy propagates the selection state of the required element, resulting in a negative (*suppressed*) selection state of the previously selected context element. Contrastingly, the *enforce inactive* strategy changes the selection state of the unselected required element to *enforced*.

Dependencies are either defined as meta-model specific constraints covered by the SDIRL language based on OCL (see part 2 of the screencast), or generically determined by the Ecore containment hierarchy. Annotations of domain model elements are phrased with our Xtext-based *Feature Expression Language* (FEL) covered in part 3. As soon as a feature configuration is loaded, feature expressions are evaluated and the result is presented to the user: A filled cyan circle indicates that the corresponding model element is directly contained in the configured domain model because its associated annotation evaluates to `true`, whereas orange filled circles mark the exclusion of the respective element. Circles with cyan or orange border depict *enforced* or *suppressed* model elements. Model elements which are not annotated are decorated with a yellow circle. Based upon the user's choice, these elements are included or excluded from each product.



Transit | Transition startOven
Transit | Transition stopOven : not "Cooldown Mode"
State O | State Off_Closed
Transit | Transition startCooldown
Transit | Transition finishCooldown
Transit | Transition initialize
State O | State Off_Opened
State O | State On_Heating
State O | State On_Cooldown : "Cooldown Mode"
Final St | Final State Offline
Pseudc | Pseudostate Start

Fig. 2. Left: UML 2 state machine diagram from multi-variant input domain model. Right: Excerpts from F2DMM mapping model for two feature configurations.

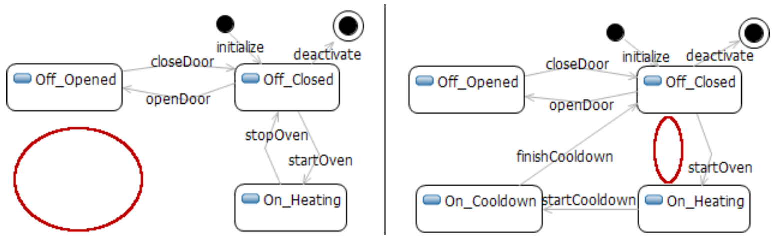


Fig. 3. Products derived from mapping using both feature configurations.

Figure 2 shows the annotation of two elements of a UML2 state machine, a state (`On_Cooldown`) and a transition (`stopOven`), positively or negatively associated with the Feature `Cooldown Mode`, which is only included in one of two example configurations. The derived products are visualized in Figure 3: In the left configuration, the adjacent transitions of state `On_Cooldown` are indirectly excluded (suppressed) due to the propagation of `On_Cooldown`'s selection state.

An advanced concept realized by our mapping editor are *alternative mappings*. These pseudo domain model elements are either defined in-place or reference objects located in a different resource. They are merged into the derived product in case the associated feature expression evaluates to true and no conflicts with existing domain model elements occur. This enables the definition of *variation points* even in case of a single-valued structural domain meta-model feature, e.g. a class name (see parts 4 and 6 of the screencast). During the mapping process, the consistency between feature model and configuration is preserved as well as the tree structure of the mapping model which is dependent on the domain model's containment structure except for alternative mapping elements. Changes in feature names may affect feature expressions. In this case, the user is free to accept or deny automatically derived renaming proposals.

3 Related work

Due to space restrictions, we limit our comparison to FeatureMapper [7], as it also allows to map features to Ecore-based domain models. For a more detailed comparison of model-driven software product line approaches, the reader is referred to [6]. FeatureMapper [7] is a tool that allows for the mapping of features to Ecore based domain models. Like our approach, it is very general and does not have any knowledge about the domain meta-model. In contrast to our approach, it provides only basic capabilites of checking well-formedness constraints of the Ecore metamodel [8] and does not provide automatic repair actions. In our previous work [6] we used the UML profile mechanism to implicitly annotate the domain model. Contrastingly to our new approach, the mapping information was persisted within the domain model rather than in a dedicated mapping resource. Automatic feature propagation in a suppress active way was employed, while feature expressions were limited to conjunction of features only.

4 Conclusion

In this paper we gave a short overview of our new tool support for the development of evolving model-driven product lines. We extended our previous approach by different directions of automatic feature propagation and feature expressions allowing arbitrary combinations of features. Alternative mappings allow the expression of variation points in domain models. To install our tool in a clean Eclipse Modelling distribution, please make sure to download Xtext 2.2.1, Acceleo 3.2.0, OCL Tools 3.1.2 and ATL 3.2.1 first. The update site can be found at: http://btn1x4.inf.uni-bayreuth.de/famile/update and the accompanying screencast is located at
http://btn1x4.inf.uni-bayreuth.de/famile/screencasts. Please note that the tool distribution also comprises our UML2 based domain modeling tool *Valkyrie*. However, Famile can be used for arbitrary Ecore based domain models.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, Boston, MA (2001)
2. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer Verlag, Berlin, Germany (2005)
3. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development : Technology, Engineering, Management. John Wiley & Sons (2006)
4. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. 2 edn. The Eclipse Series, Boston, MA (2009)
5. Batory, D.S.: Feature models, grammars, and propositional formulas. In Obbink, J.H., Pohl, K., eds.: Proceedings of the 9th International Software Product Line Conference (SPLC'05). Volume 3714 of Lecture Notes in Computer Science., Rennes, France, Springer Verlag (September 2005) 7–20
6. Buchmann, T., Westfechtel, B.: Model-driven Engineering of Software Product Lines with Feature Models and Graph Transformations. Software and Systems Modeling (2011) submitted for publication.
7. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping features to models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany (May 2008) 943–944
8. Heidenreich, F.: Towards systematic ensuring well-formedness of software product lines. In: In Proceedings of the 1st Workshop on Feature-Oriented Software Development, ACM Press (October 2009)

Developing a multi-panel editor for EMF trace models ⋆

Álvaro Jiménez, Juan M. Vara, Verónica A. Bollati, Esperanza Marcos

Kybele Research Group,
Department of Computing Languages and Systems, Rey Juan Carlos University,
C/ Tulipán s/n, 28933, Móstoles, Madrid (Spain).
{alvaro.jimenez,juanmanuel.vara,
veronica.bollati,esperanza.marcos}@urjc.es
http://www.kybele.urjc.es/

Abstract. Eclipse and more specifically the Eclipse Modeling Framework is probably the most commonly adopted meta-modelling framework to support model-based Software Engineering proposals. One of the main advantages of EMF is the ability to generate tree-like editors with basic capabilities for models conforming to a previously defined metamodel. Unfortunately, the generic nature of these editors does not always fit the needs of specific scenarios. For instance, that is the case with models owning a relational nature, such as weaving, trace or transformation models. To alleviate this problem, this work presents the development of a multi-panel editor for EMF models. In particular, it shows the development of an editor for trace models. It allows representing at the same time the different source and target models plus the *relational* model.

Keywords: Model-Driven Engineering, Eclipse, EMF, multi-panel editor, trace links.

1 Introduction

With the advent of Model-Driven Engineering (MDE) [4], the role of models has changed drastically since they become the main artefact all along the development process. As a consequence, modelling can be seen as the most important activity and tool support for such activity becomes also a cornerstone issue.

Indeed, as any other software paradigm, MDE should provide with the proper tooling in order to support the development process. As software developers use IDEs (Integrated Development Environments) to assist them in coding activities, modellers need from modelling environments to in the development of modelling activities. In this field, Eclipse and more concretely the Eclipse Modeling Framework (EMF, [7], [20]) is considered the *de-facto* standard to develop tool support for model-based proposals. Probably one of the most used facilities provided by EMF is the ability to generate tree-based model editors [20] for models conforming to a given metamodel stating from the metamodel itself. Nevertheless, the generic nature of such editors does not always fit with the specific nature of specific scenarios.

⋆ Update-site: `http://www.kybele.es/research/tools/ACME2012/T-Trace_UpdateSite`
Screencast: `http://www.kybele.es/research/tools/ACME2012/screencast.htm`

In other words, the generative nature of EMF implies a compromise between the level of automation (100%) and the level of adjustment to specific purposes. i.e. EMF generates an efficient though not perfect diagrammer in reasonable time and manner.

As a consequence, EMF-generated code use to be modified in order to adapt the generated editor to enrich modelling experiences in specific scenarios [9]. See for instance, the [10], [17]. The same applies for GMF-generated diagrammers: Graphiti [19], EuGENia [16] and Kybele GMFGen [15] being examples of frameworks to help on adapting GMF-based editors to the specific needs of a given domain.

One such scenario is constituted by *relational* models, i.e. models whose main purpose is to represent the relationships between some other models, such as weaving models [3]. To address this issue, this work presents the development of an EMF-based editor for trace models.

The rest of this paper is structured as follows. Section 2 presents the motivation behind the development of an *ad-hoc* editor to model and represent relationships between EMF model elements. Section 3 digs into the modification of EMF-generated code and finally, Section 4 concludes by summarizing the main contributions and providing directions for further work.

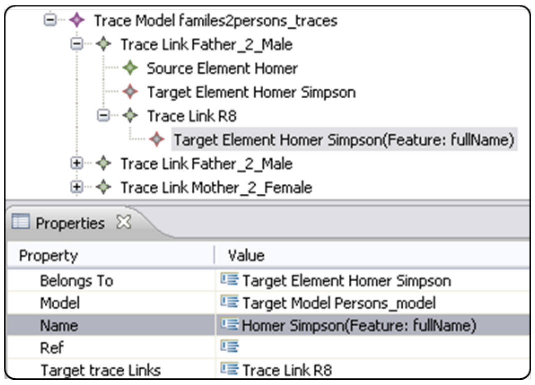
2 Motivation

The impact of the MDE paradigm has resulted in the advent of a number of methodological proposals for Model-Driven Software Development (MDSD) [21], where the key role of models can positively influence the management of traceability information. In a MDSD process the traces between software artifacts that have to be maintained are mainly the links between the elements of the different models handled along the process. Furthermore, such traces can be collected in other models to process them using any model processing technique, such as model transformation, model matching or model merging [3].

In this context we addressed the development of a methodological and technical proposal to improve traceability management in model transformations development [14]. Since we use to lean on EMF as underlying metamodeling framework, as part of such proposal we have developed a new EMF-based DSL, so-called `t-Trace`, to model the trace-links derived from the execution of model transformations (a trace-link is an instance of a traceability relation between source elements and target elements [1]). Note that it is not our intention to discuss here the need for such DSL or the state of the art of traceability management [14]. Here we focus just on the development of an EMF-based editor for such DSL.

As we have already mentioned, EMF is capable of generating a fully functional tree-based editor for models conforming to a previously defined Ecore metamodel. However, a plain view of nested elements is not enough to provide an optimal visualization of the different relationships that can arise between the elements of different models. That is the case of trace models: in the tree-based editor generated by EMF, the elements linked are showed as children of the trace-link objects, while it would be much more intuitive to display them as referenced items, i.e. source and/or target elements.

Fig. 1 illustrates this scenario by showing the use of an EMF tree-based editor for trace models. Though functional, this representation is not intuitive at all. We



```
Trace Model familes2persons_traces
  Trace Link Father_2_Male
    Source Element Homer
    Target Element Homer Simpson
    Trace Link R8
      Target Element Homer Simpson(Feature: fullName)
  Trace Link Father_2_Male
  Trace Link Mother_2_Female
```

Properties

Property	Value
Belongs To	Target Element Homer Simpson
Model	Target Model Persons_model
Name	Homer Simpson(Feature: fullName)
Ref	
Target trace Links	Trace Link R8

Fig. 1. EMF tree-based editor for trace models

believe that a much more desirable editor would be that where source elements are showed on the left-hand side of the editor where as target elements are showed on the right (as we will describe later, Fig. 3).

Reviewing existing works focused on the management of traceability information in the context of MDE, we have found that some of them ([2], [6], [8], [13]) advocates in favor of using the editors provided by existing tools such as AMW [8] or ModeLink [11], [13]. This kind of editors, usually referred to as **multi-panel editors**, allow showing views of several related models plus the model that relates them in an integrated way. Although they improve the capabilities of the generic EMF editors to display *relational* models, they still own some generic nature that results in some limitations when used to display trace models. They were devised to work with any kind of *relational* model, thus they still leave space for customization when used for more specific purposes.

In particular, the ModeLink editor just supports the visualization of two/three models, including the relationships model. Therefore, the user can only define relationships between the elements of one source and one target model. As a consequence, a limitation arises when we require defining traces between elements of several (more than one) source and/or target models.

On the other hand, we found two main issues regarding the use of AMW. The first one is that the AMW project is not very active, hence it is not updated as much as other EMF-based plug-ins. As a consequence, compatibility problems with other EMF-based tools can be arisen. Nevertheless, this can be seen as a minor issue given that are in contact with the author and we have been considering to join efforts to produce updated versions of AMW. The main limitation of AMW when used to display trace models is that it hampers the distinction between source and target models when dealing with more than two related models. For instance, Fig. 2 shows the use of AMW in a scenario composed by three source models and two target models. Note that the generic layout of the different views in the AMW multi-panel editor is as follows: first referenced model, weaving model (which contains the objects that represent the relationships) and the rest of referenced models. As figure shows, this layout can be confusing for the users.

The most immediate solution to address the above-commented issues would be extending AMW or ModeLink. However, we preferred to avoid technological-dependency. Therefore, we decided to develop an *ad-hoc* multi-panel editor for trace

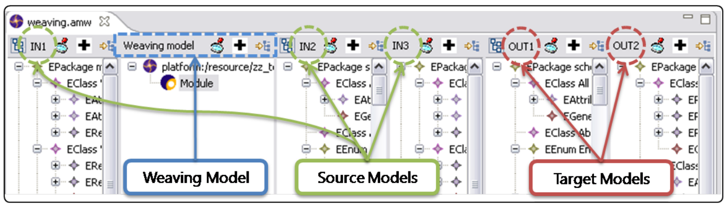


Fig. 2. Using the AMW editor with several source models

models. Concretely, we aimed at supporting the behavior shown in Fig. 3. Such an editor should satisfy the following requirements:

– It should bundle three different panels to show separately the source models, the trace model and the target models. If there are several source or target models, they should be co-located vertically in their corresponding panel.
– The user should be able to drag elements from source and target models and drop them on the trace model to establish new relationships (create new trace-link objects).
– If the user selects a trace-link object, the editor has to highlight automatically the elements referenced by the selected link. They should be highlighted in the models containing the referenced elements and not in the trace model itself.
– If the user selects a source or target element, the editor must highlight the trace-link objects that reference it.

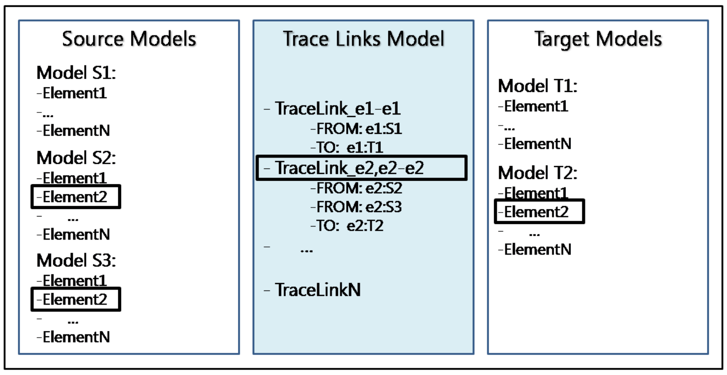


Fig. 3. Desired functionality for the multi-panel editor

The following section present the development of a multi-panel editor for trace models fulfilling the above mentioned requirements.

3 Development process

To illustrate the development of the multi-panel editor for EMF trace models, we will show the development of the editor for the `t-Trace` DSL. As we have already mentioned, `t-Trace` is a specific DSL to model trace-links whose metamodel is shown in Fig. 4. The multi-panel editors we present in this work can be used to provide any DSL owning a relational nature with a graphical syntax. The only restrictions are that the metamodel of such DSL has to contain some particular concepts: metaclasses that allow representing source and target models (`SourceModel` and `TargetModel` in the `t-Trace` metamodel) and elements to represent the linked source and target objects (`TraceElement`). Moreover, note that each element registers the ID of the traced element it represents (`ref` attribute).

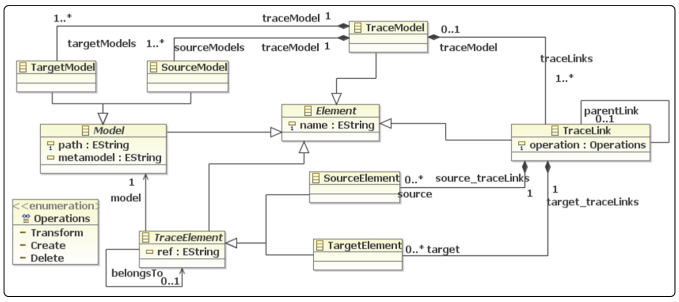
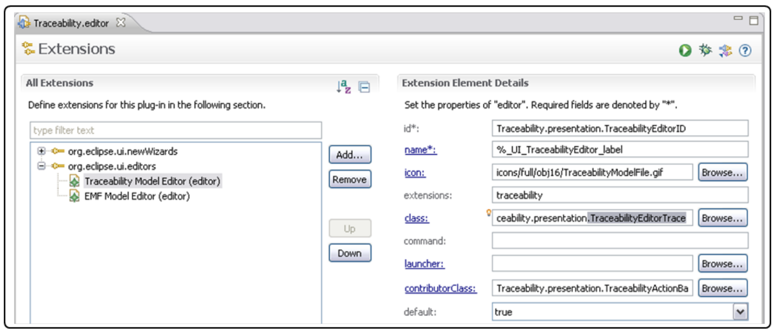


Fig. 4. t-Trace Metamodel

The multi-panel editor developed to ease the edition of models conforming to the previous metamodel is the result of modifying the implementation of the tree-based editor generated by EMF [1].

The first step of the process consists of modifying the `plugin.xml` file of the `editor` project. In particular, one of the extension in the `extensions` tab corresponds to the EMF tree-based editor. One can replace it for the new editor or keep it and create a new one instead. In this case we have opted for the second option. Thus, we add an extension on the `org.eclipse.ui.editors` extension so-called `Traceability Model Editor` (Fig. 5). The `class` property of this extension (see left-hand side of the figure) establishes that the Java class that implements the extension is the `TraceabilityEditorTrace` class, that we have to add to the `editor` project.



All Extensions

Define extensions for this plug-in in the following section.

type filter text

- org.eclipse.ui.newWizards
- org.eclipse.ui.editors
  - Traceability Model Editor (editor)
  - EMF Model Editor (editor)

Extension Element Details

Set the properties of "editor". Required fields are denoted by "*".

id*:	Traceability.presentation.TraceabilityEditorID
name*:	%_UI_TraceabilityEditor_label
icon:	icons/full/obj16/TraceabilityModelFile.gif
extensions:	traceability
class:	ceability.presentation.TraceabilityEditorTrace
command:	
launcher:	
contributorClass:	Traceability.presentation.TraceabilityActionBa
default:	true

Fig. 5. Creating a new editor extension

Probably, the easiest way to start with the coding tasks of the new editor is to copy-paste the code that implements the tree-based editor in the new class

[1] To generate a tree editor with EMF, please see: `http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html`

(TraceabilityEditorTrace). The result is obviously two different editors providing exactly the same functionality. So, next step is to modify the duplicated code of the TraceabilityEditorTrace class in order to get the desired functionality from the new editor.

To define the structure of the editor, which will bundle three different panels (source, trace and target models), we define three TreeViewer attributes, namely sourceViewer, traceabilityViewer and targetViewer. Next step is to modify the createPages() method that defines the appearance of the editor. This method serves to establish the shape of the container and define the functionality of the three different viewers. Note also that in order to display all the source models (respectively target models) in the source viewer (respectively target), it must load all the source models as Eclipse resources and it must register their corresponding metamodels. These operations are done by retrieving their path as well as the path of the metamodels from the trace model. As example, the following code excerpt shows the implementation of the source viewer:

```
// Creation of the viewer for source models.
ViewerPane viewerPane = new ViewerPane(
  getSite().getPage(), TraceabilityEditorTrace.this){

 public Viewer createViewer(Composite composite) {
  Tree tree = new Tree(composite, SWT.MULTI);
  return new TreeViewer(tree);
  }

 public void requestActivation() {
  super.requestActivation();
  setCurrentViewerPane(this);
 }
};

 viewerPane.createControl(topSashForm);
 sourceViewer=(TreeViewer)viewerPane.getViewer();
 sourceViewer.setContentProvider(new AdapterFactoryContentProvider(
   adapterFactory));
 sourceViewer.setLabelProvider(new AdapterFactoryLabelProvider(
   adapterFactory));
 Transfer[] transfers = new Transfer[] {LocalTransfer.getInstance()};
 sourceViewer.addDragSupport(DND.DROP_LINK , transfers,
   new ViewerDragAdapter(sourceViewer));
 for(int i=0;i<sources.size();i++){  //If metamodel is not null
  if (sources.get(i).getMetamodel() != null) {  //If it is not empty
   if (!sources.get(i).getMetamodel().equals("")) {
     String metamodelRegistered=Actions.registerMetamodel(
       sources.get(i).getMetamodel(), sources.get(i).getName());
     sources.get(i).setMetamodel(metamodelRegistered);
   }
  }
 }
 sourceRs = Actions.createResourceSet_Sources(sources);
 Actions.setSourceModels(getEditingDomain().getResourceSet(), sources);
 sourceViewer.setInput(sourceRs);
 sourceViewer.setSelection(new StructuredSelection(
   sourceRs.getResources().get(0)), true);
 viewerPane.setTitle("Source Models");
 sourceViewer.addSelectionChangedListener(
   new ISelectionChangedListener(){
   // This ensures that we handle selections correctly.
     public void selectionChanged(SelectionChangedEvent event){
        handleContentSourceSelection(event.getSelection());}});
 new AdapterFactoryTreeEditor(sourceViewer.getTree(), adapterFactory);
```

At this point, the editor is composed of three tree-based panels displaying the source, trace-links and target models respectively. The next step consists of connecting the information displayed in these panels. To do so, we created some methods to connect the model elements contained in the different viewers (in order to support model elements highlighting):

– `handleContentOutlineSelection()`: whenever a user selects an element on the Outline viewer of Eclipse, this method is responsible of selecting the same element on the trace model.
– `handleContentTraceabilitySelection()`: this method is invoked when the user selects a trace-link or an element of a trace-link. whether it represents a source or a target element. When the selected element is a source one, the method search for its ID (`ref` attribute) among the elements of the different source models, highlighting those with the same ID. The same applies for target elements. Finally, if the selection is a trace-link object, the previous functionalities are combined to search for the IDs of all the elements referenced by the selected link.
– `handleContentSourceSelection()`: this method is called when the user selects an element from a source model. It searches for the ID of the element among the source elements of the trace-link objects to identify the trace-links in which the selected element is involved.
– `handleContentTargetSelection()`: this method provides the same functionality for target models.

As a result, the multi-panel does not only allocate each model in the corresponding panel, but also supports bi-directional highlighting of model elements: selections in trace models produce highlighting actions in source/target models and vice-versa.

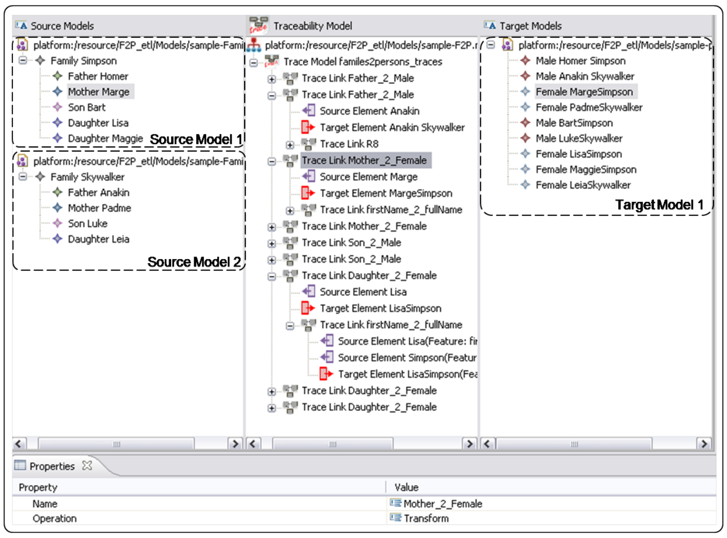
Last task to address is the implementation of drag&drop functionality. This way, the user should be able to create trace-links by dragging elements from source and target models and dropping them on the trace-links panel. To support this functionality, we add a new class so-called `TraceabilityDragDrop` in the `editor` project. It extends the `EditingDomainViewerDropAdapter` class, so that its constructor receives as arguments the domain, the viewer and the source and target models. It implements the following methods:

– `helper()`: this method receives the drag&drop event, identifying the element dragged as well as the trace-link over which it has been dropped. By identifying which the element dragged is, the proper response to the event can be derived, i.e. whether a source or a target element has to be added to the corresponding trace-link object.

– `drop()`: this method determines whether an action should be carried in response to the event or if it must be discarded. To that end, it uses the `selectionType` attribute of the `helper()` method.
– `createTraceElement()`: this method receives as arguments the dragged element, the trace-link which receives the element and the type of the element dragged (source or target). With this information, it decides which method to invoke (`handleSetSourceElement()` or `handleSetTargetElement()`) in order to create a source or a target element in the corresponding trace-link object.
– `handleSetSourceElement()`: it adds an element to the trace-link object pointing to the source element that was dragged.
– `handleSetTargetElement()`: it adds an element to the trace-link object pointing to the target element that was dragged.

The `TraceabilityDragDrop` class is instantiated from the `createPages()` method mentioned before, just after the code implementing the viewers.

At this point, the multi-panel editor for trace models provides the desired structure and functionality (section 2). Moreover, we have added some minor improvements, mainly related with icons and text-descriptions. Fig. 6 serves to illustrate the final result. It shows a trace model between two source models (`Family Simpson` and `Family Skywalker` models) and one target model (`People` model). In particular, when the user selects the `Mother_2_Female` trace-link, the corresponding source and target elements (`Mother` and `Female`) are identified by the editor.



```
platform:/resource/F2P_etl/Models/sample-Fami
  Family Simpson
    Father Homer
    Mother Marge
    Son Bart
    Daughter Lisa
    Daughter Maggie
platform:/resource/F2P_etl/Models/sample-Fami
  Family Skywalker
    Father Anakin
    Mother Padme
    Son Luke
    Daughter Leia
```

```
platform:/resource/F2P_etl/Models/sample-F2P.r
  Trace Model familes2persons_traces
    Trace Link Father_2_Male
    Trace Link Father_2_Male
      Source Element Anakin
      Target Element Anakin Skywalker
      Trace Link R8
    Trace Link Mother_2_Female
      Source Element Marge
      Target Element MargeSimpson
      Trace Link firstName_2_fullName
    Trace Link Mother_2_Female
    Trace Link Son_2_Male
    Trace Link Son_2_Male
    Trace Link Daughter_2_Female
      Source Element Lisa
      Target Element LisaSimpson
      Trace Link firstName_2_fullName
        Source Element Lisa(Feature: fir
        Source Element Simpson(Featur
        Target Element LisaSimpson(Fea
    Trace Link Daughter_2_Female
    Trace Link Daughter_2_Female
```

```
platform:/resource/F2P_etl/Models/sample-p
  Male Homer Simpson
  Male Anakin Skywalker
  Female MargeSimpson
  Female PadmeSkywalker
  Male BartSimpson
  Male LukeSkywalker
  Female LisaSimpson
  Female MaggieSimpson
  Female LeiaSkywalker
```

Property	Value
Name	Mother_2_Female
Operation	Transform

Fig. 6. Multi-panel editor to show trace models

It worth mentioning that this multi-panel editor has been used in more complex scenarios such as traces models between UML conceptual data models and XML Schemas [14].

4 Conclusion

Models play a key role in any MDE proposal [4]. Therefore, MDE practitioners should have at their disposal the proper and mature tooling to create and handle models. In this context, Eclipse and EMF [7][20] are widely used as underlying basis for the development of new tools. Probably due to being an open-source project and its extensive nature, EMF has been massively adopted as metamodelling framework by researchers from academia. Another relevant advantage of EMF is its generative nature: from the abstract syntax of a DSL collected in an Ecore metamodel, it provides with a basic Java implementation that includes an API to handle models programmatically as well as a full-fledged (but basic) editor.

The generative nature of EMF implies that produced editors have to be generic. In some sense, they are general-purpose editors since they should work for models conforming to any given metamodel. However, such editors leave much room for improvement. Indeed, they are devised to be adapted to the needs of specific domains.

All this given, in this work we have combined the main features of EMF to overcome the disadvantages brought by the generic nature of EMF tree-based editors when they are used to display trace models. So, we have shown how to refine EMF-generated code to produce a multi-panel editor that fits better with the *relational* nature of trace-link models. The result is an *ad-hoc* editor composed by three tree-based panels which shows several source models, a trace model and several target models in an integrated manner. Moreover, it supports drag&drop creation of trace-link elements and automatic bi-directional selection of model elements and trace-links. As well, this work has served to show that EMF-generated code can be used as starting point to produce high-level tools and it can be easily refined to adapt EMF-based software artefacts to specific needs. Indeed, EMF itself can be seen as the perfect example of how to put into practice MDE principles.

Regarding directions for further work, the most immediate is to add new functionality to the multi-panel editors already developed. For instance we are integrating element searching and ordering capabilities. A more interesting direction for future work is to follow the method presented here to develop multi-panel editors to support other kind of models for model management operations. For instance, we are already developing ad-hoc editors for transformation models [5]. Besides, the refining of EMF-generated code is subject to automation. In this sense, we will analyse Xpand to address this task.

Acknowledgements

This research was carried out in the framework of the MASAI (TIN-2011-22617) project, financed by the Spanish Ministry of Science and Innovation.

References

1. Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., and Shaham-Gafni, Y. (2006): Model traceability, IBM Systems Journal, vol. 45 (3), pp. 515-526.

2. Barbero, M., Del Fabro, M. D., and Bezivin, J. (2007): Traceability and provenance issues in global model management. Proceedings of International Conference on Systems Engineering and Modelling (ICSEM07), 2007.
3. Bernstein, P. (2003): Applying model management to classical meta data problems. First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA.
4. Bézivin, J. (2004): In search of a basic principle for model driven engineering. Novatica Journal, Special Issue, vol. V(2), pp. 21-24.
5. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A. (2006): Model Transformations? Transformation Models!. Model Driven Engineering Languages and Systems. vol. 4199, Springer, pp. 440-453.
6. Boronat, A., Carsí, J., and Ramos, I. (2005): Automatic Support for Traceability in a Generic Model Management Framework. 1st European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'05), Nuremberg.
7. Budinsky, F. (2004): Eclipse modeling framework: a developer's guide. Addison-Wesley Professional.
8. Didonet Del Fabro, M. (2007) Metadata management using model weaving and model transformation. PhD thesis, University of Nantes.
9. Ehrig, K., Ermel, C., Hänsgen, S., and Taentzer, G. (2005): Generation of visual editors as eclipse plug-ins. The 20th IEEE/ACM international Conference on Automated Software Engineering (ASE'05), Long Beach, USA, pp. 134-143.
10. EMF Facet Project. `http://www.eclipse.org/modeling/emft/facet/`
11. Epsilon ModeLink, `http://www.eclipse.org/epsilon/doc/modelink/`
12. Fowler, M., Parsons, R. (2010): Domain-specific languages. Addison-Wesley Professional.
13. Guerra, E., de Lara, J., Kolovos, D.S., and Paige, R. F. (2010): Inter-modelling: From Theory to Practice. Model Driven Engineering Languages and Systems. vol. 6394, Springer, pp. 376-391.
14. Jiménez, Á. (2012): Integrating traceability management in a framework for MDD of model transformations. PhD thesis, Rey Juan Carlos University.
15. Jiménez, Á., Vara, J. M., Bollati, V., and Marcos, E. (2010): Mejorando el nivel de automatización en el desarrollo dirigido por modelos de editores gráficos. DSDM 2010 (JISBD), Valencia, Spain, pp. 29-37.
16. Kolovos, D. S., Rose, L. M., Paige, R. F., Polack, F. A. C. (2009): Raising the level of abstraction in the development of GMF-based graphical model editors. Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering, 2009, pp. 13-19.
17. Langer, P., Wieland, K., Wimmer, M., Cabot, J. (2011): From UML Profiles to EMF Profiles and Beyond. Objects, Models, Components, Patterns. vol.6705, Springer.
18. Levendovszky, T., Balasubramanian, D., Smyth, K., Shi, F., Karsai, G. (2010): A transformation instance-based approach to traceability. Proceedings of the 6th ECMFA Traceability Workshop, Paris, France, 2010, pp. 55-60.
19. Maxime, P., Jonathan, P., Matthieu, W., Slaheddine, A. (2009): An Open Framework for Rapid Prototyping of Signal Processing Applications. EURASIP Journal on Embedded Systems, vol. 2009.
20. Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008): EMF: Eclipse Modeling Framework. 2nd Edition ed.: Addison-Wesley Professional.
21. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006).

An Integrated Tool Chain for Software Process Modeling and Execution

Ralf Ellner[2], Samir Al-Hilank[2], Martin Jung[2],
Detlef Kips[1,2], and Michael Philippsen[1]

[1] University of Erlangen-Nuremberg, Computer Science Department,
Programming Systems Group, Martensstr. 3, 91058 Erlangen, Germany
`philippsen@cs.fau.de`
[2] develop group Basys GmbH, Am Weichselgarten 4, 91058 Erlangen, Germany
`ellner|alhilank|jung|kips@develop-group.de`

Abstract. The Eclipse Process Framework (EPF) allows for a detailed modeling of software development processes and methods based on the Software and Systems Process Engineering Metamodel (SPEM) standard. A comprehensive electronic process guide may be generated from such a model. However, EPF and SPEM only support a rather coarse description of the behavior of software processes. As there is no support for automated enactment or simulation of these software process models, one cannot benefit from context-sensitive process guidance, automated process conformance checking, or automated progress tracking when enacting detailed software process models.
eSPEM (enactable SPEM) is an extension of the SPEM standard that supports UML activities and state machines for fine-grained behavior modeling. Its operational semantics is based on OMG's fUML (Semantics of a Foundational Subset for Executable UML Models) and may be used to instantiate, simulate, and enact software process models. However, without a reasonable tooling for eSPEM the benefit for end users is still limited.
This paper presents an integrated tool chain based on eSPEM and Eclipse. The tool chain not only supports process modelers in modeling fine-grained eSPEM-based software processes, but also guides and supports project staff in working according to the process in a context-sensitive manner. It automates repetitive and cumbersome work like checking process conformance or progress tracking. Hence, it lets end users benefit from process modeling and enactment.

1 Introduction

Software development processes (SDPs) are widely accepted as a critical factor in the efficient development of complex and high-quality software and systems. Beginning with Osterweil's process programming [1] many process modeling languages (PML) have been proposed to describe SDPs in more or less abstract, (semi-)formal ways, see [2–4] for an overview.

SPEM [5] is a standardized PML; it is based on the UML Infrastructure [6] and defines a graphical notation. Due to its familiar notation, practitioners can pick up SPEM easily. The Eclipse Process Framework (EPF) provides a reference implementation of SPEM. However, SPEM and EPF have been primarily designed to model and document the static structure of SDPs. Thus, when a SDP is modeled with SPEM, this results in a thorough informal documentation of the process. With EPF, one may generate an electronic process guide from a SPEM-based SDP model. Such a process documentation is valuable or may even be required, for example in safety critical projects, but it does not provide much additional value for the project staff as there is no help in executing the process.

Although it has been a requirement, executability is not in the scope of the current version 2.0 of SPEM, even though it would provide the following additional benefits (see [1, 7]): First, executable software process models can be simulated and can thus more easily be validated before they are used in a project. Second, a process execution machine (PEX) can guide and support the project staff. Third, since a PEX can automatically check conformance of the executed process with the modeled process, it can detect and prevent process violations. Finally, a PEX can track progress of the executed process. This is of great use for process audits, because it is possible to partially automate the proof that the actually executed process conforms to the modeled process.

In [8] we presented eSPEM, a SPEM extension based on UML activities and state machines [9]. In addition to the behavior modeling concepts of UML (for example, decisions, exceptions, and events), eSPEM also provides behavior modeling concepts that are specific to SDPs (e.g., task scheduling). These behavior modeling concepts can be used to describe the behavior of SDPs in a fine-grained, formal, but intuitive way. The formality of the SDP behavior description is required in order to execute it. Another requisite of SDP execution is that there is a rigid definition of the operational semantics used to describe a SDP. In [10] we presented the operational semantics of eSPEM based on the OMG standard *Semantics of a Foundational Subset for Executable UML Models* (fUML) [11]. fUML defines the operational semantics of UML activities and actions. But fUML execution is limited to a single machine, and fUML does not support human interaction nor can its execution model be extended. While fUML is suitable for local simulation, it is insufficient for distributed process execution which is needed for typical team-driven software projects. Thus, we added support for distributed execution, human interaction, and user specific extensions to the operational semantics of eSPEM. We also implemented the operational semantics of UML state machines which are missing in fUML.

Our extensions to SPEM and fUML are the conceptual foundation of a detailed software process modeling and automated enactment. However, a reasonable tool support for eSPEM is required to let end users benefit. This papers presents such an integrated tool chain for software process modeling and execution. Fig. 1 gives an overview of the tool chain from a user's perspective.

A *Process Designer* models, validates, and simulates an executable software process using a *Process Modeling Environment (PME).* The *Process Designer*

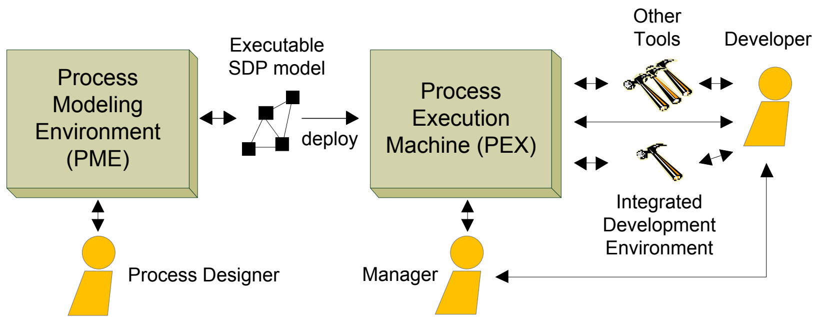

Process Modeling Environment (PME)
Executable SDP model
deploy
Process Execution Machine (PEX)
Other Tools
Developer
Integrated Development Environment
Process Designer
Manager

Fig. 1. Overview of the Integrated Tool Chain

may then deploy the resulting model to a *PEX*. A *Manager* may use the PEX to create development projects that follow the model. The PEX interprets the software process model and guides and supports the *Developer*s through the process. Depending on a task a developer performs, the PEX offers context sensitive help. The PEX also cooperates with tools used in the process and manages artifacts as specified in the process model. It can report deviations between the modeled process and the process actually executed. In addition, the PEX tracks all actions performed throughout the process and supports process traceability.

Section 2 presents the overall architecture of our integrated tool chain and shows how it achieves the mentioned benefits. In section 3 we use the tool chain to model and enact an exemplary process. Section 4 covers related approaches and tools. In section 5 we conclude and preview future work.

2 Architecture of the Integrated Tool Chain

The tool chain is built from three major components (see Fig. 2). First, a Process Modeling Environment (PME) to model, simulate and deploy software processes, a Process Enactment Server (PES) that manages deployed process models and distributed process model instances, and a Process Enactment Client (PEC) that is the graphical front end for process enactment. PEC and PES employ a traditional client/server architecture, together providing the functionality of a PEX (see Fig. 1). All three components are based on the Eclipse Equinox OSGi framework [12] and several other Eclipse components [13] (see Fig. 2) to help distributed development teams in their process enactment.

2.1 Process Modeling Environment (PME)

The PME implements the abstract and concrete syntax of the modeling language eSPEM. The abstract syntax is realized as an EMF (Eclipse Modeling Framework) Ecore meta-model [14] accompanied by mostly generated Java code. Since typical

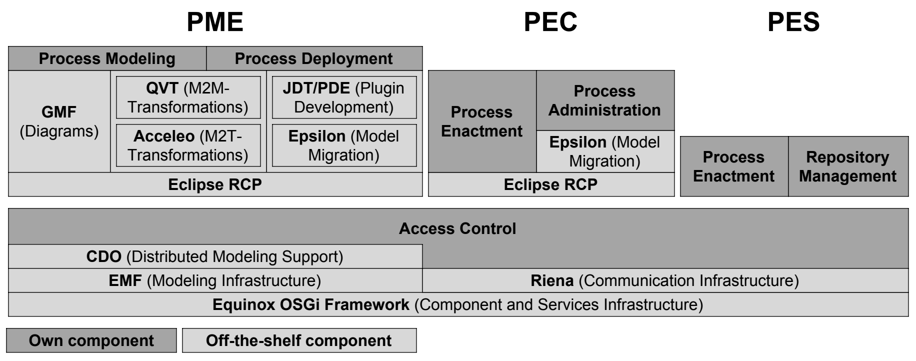

PME
PEC
PES
Process Modeling
Process Deployment
GMF (Diagrams)
QVT (M2M-Transformations)
Acceleo (M2T-Transformations)
JDT/PDE (Plugin Development)
Epsilon (Model Migration)
Eclipse RCP
Process Enactment
Process Administration
Epsilon (Model Migration)
Eclipse RCP
Process Enactment
Repository Management
Access Control
CDO (Distributed Modeling Support)
EMF (Modeling Infrastructure)
Riena (Communication Infrastructure)
Equinox OSGi Framework (Component and Services Infrastructure)
Own component
Off-the-shelf component

Fig. 2. Overall Architecture of the Integrated Tool Chain

SDP models tend to be large and complex, the PME strives to reduce that complexity. First, a view-based diagram editing approach for eSPEM's concrete syntax makes modeling of processes more intuitive and easier. Four different types of diagrams depict different aspects of the SDP model. There are a state machine diagram editor and an activity diagram editor for the dynamic behavior of software processes. Aspects of the static structure of software processes can be modeled with a method content diagram editor and a team profile diagram editor. All diagram editors are realized with GMF (Graphical Modeling Framework) [15] and may be used concurrently on a single process model.

In order to further reduce complexity of process models and to provide fast and easy access to frequently used information, we also provide a variety of Eclipse views that filter the model by the type of model elements. For example, the activity view shows all activities within a model. These views also show the semantic relations among model elements, e.g., which roles perform an activity. As a result, these views provide fast access to a model element and its usage within the model. In addition, there is an electronic process guide (EPG) preview that provides a comprehensive, navigable electronic process documentation in natural language. It uses a model-to-text (M2T) transformation to generate an EPG and to display it in a browser.

For general purpose modeling, the model explorer provides an unfiltered view of the abstract syntax tree of a process model. It can be used to generically create, select, move, and delete elements. In addition, model validation can be triggered from the model explorer. Our validation support is based on the EMF Validation Framework [14] and various OCL [16] constraints in the eSPEM meta-model. This automated validation support is a great help in checking a large model for structural and semantic correctness. For even more validation support, there is the process simulation view. It may be used to locally simulate the behavior of a process on top of the eSPEM execution machine. Process simulation greatly improves debugging and testing SDPs and finding bottlenecks in them.

As most software processes have to be tailored to the needs of a concrete project, eSPEM provides language constructs to adapt and select processes and development methods, and to bring them together as a process configuration [8, 5]. To ease process configuration, the configuration view helps to bind a process to methods by simple drag-and-drop.

Similar to process tailoring, a PEX must be adaptable to the needs of a concrete project or organization. For this kind of adaption the tool chain provides a lot of extension points for plugins. Plugins may be developed with the PME that also includes all Eclipse components for Java and Eclipse plugin development. There are extension points for tool adapters, version control systems, extensions of our process runtime infrastructure, and process plugins. To deploy and enact a process model, a corresponding process plugin for our PES/PEC has to be generated first. This task is fully automated. Model-to-model and M2T transformations turn an eSPEM-based process model into an Ecore model, Java code, and meta data (e.g., OSGi meta data, EPG) that form a process plugin. Finally, process plugins may adapt an already running process to a new version of its process model. For this purpose, we use Epsilon Flock [17], a transformation language for model migrations. Flock greatly simplifies model migrations because only rules for transforming structural differences have to be written explicitly at the level of the generated Ecore models. Unchanged parts of two meta-models are adapted automatically.

2.2 Process Enactment Server (PES)

The PES provides secure shared access to process instances. It manages eSPEM execution models with CDO [13], a framework that allows for concurrent remote access to models and supports model versioning. We use the latter to implement traceability of process enactment. This is useful in process audits and for process improvements. Since versioning leads to large amounts of data, we use a relational database as back-end to store execution models. To secure access to process instances, we implemented an access control layer on top of CDO and Riena [18].

The eSPEM process execution machine relies on the observer design pattern to track changes of the execution model. Upon each change, the next execution steps are computed and performed. For performance reasons, the PES manages the observer object that has direct access to process instances in the Java virtual machine of the PES.

2.3 Process Enactment Client (PEC)

The PEC provides views and editors to instantiate, manage, and execute deployed software process models and their instances. It is an Eclipse rich client application and may be directly integrated into the development environment used by the project staff. The PEC presents two predefined perspectives: First, the administration perspective with views and editors to create and administrate projects, users, and plugins. Second, the process enactment perspective that the project staff uses to execute the process, i.e., manage activities, tasks, and work products. Context-sensitive help from the electronic process guide is seamlessly integrated into those perspectives to allow a direct access to the relevant part of the process description.

Process audits often require a thorough documentation of each process step. Without an automated enactment this results in a lot of additional manual labor. The PEC/PES fully automates this work and automatically generates log messages that make all changes to the process state reproducible. The PEC's process log view is another benefit as it provides easy access to the generated log messages.

Due to the complexity of software processes, it is often a nontrivial task to follow them. It is even more difficult to fix conformance problems after things have gone wrong. eSPEM uses (OCL) constraints to detect such problems. Checking of conformance is bundled with a transaction system that performs rollbacks in case of severe problems or issues warnings in case of less severe problems. This automated process instance validation and rollback helps project staff in working according to the process by reducing time to detect problems.

3 Walkthrough of an Exemplary SDP

To demonstrate the functionality of our integrated tool chain, we first cover the Scrum SDP [19] as an exemplary process. Scrum is an agile, iterative-incremental process. An iteration is called *sprint*. Within each sprint, a potentially shippable product increment is developed. A sprint starts with a *sprint planning meeting*. During this meeting, tasks are estimated and written down in the *sprint backlog*. After the sprint planning meeting, the planned tasks are executed by project staff. Each day starts with a *daily scrum meeting* followed by development work. At the end of a sprint, the developed product increment is presented during the *sprint review meeting*. Afterwards, the sprint is discussed and process improvements are elaborated during the *sprint retrospective meeting*.

3.1 Modeling Scrum

After this sketch of Scrum, we illustrate how to model a part of Scrum with a PME, deploy the model to a PES, and enact Scrum with a PEC. We provide a more detailed version of this example as screencast in [20]. Our tool chain can be downloaded from [21].

Since software development is a creative endeavour, not all details of a process can be anticipated and modeled. In fact, most processes (including Scrum) use dedicated planning activities (e.g., the sprint planning meeting) to react upon changing requirements or risks within a project. However, current process modeling languages have no means to express such process situations. In contrast, eSPEM provides task schedulers to model the planning of dynamically instantiated tasks. Fig. 3 shows how a task scheduler (`Backlog Task Scheduler`) may be used to model an execution strategy (e.g., based on the priority or dependencies of a task) for tasks within the `Sprint Backlog`.

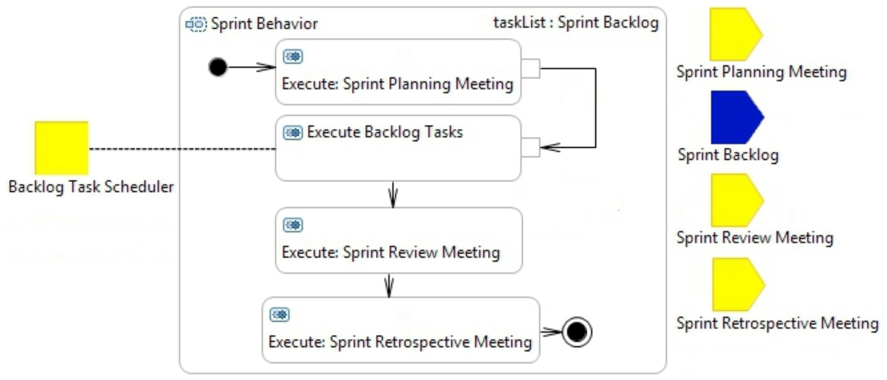

Sprint Behavior
taskList : Sprint Backlog
Execute: Sprint Planning Meeting
Execute Backlog Tasks
Execute: Sprint Review Meeting
Execute: Sprint Retrospective Meeting
Backlog Task Scheduler
Sprint Planning Meeting
Sprint Backlog
Sprint Review Meeting
Sprint Retrospective Meeting

Fig. 3. Behavior model of a Scrum sprint modeled with eSPEM and our PME

The special action `Execute Backlog Tasks` accepts the `Sprint Backlog` and executes its tasks in the order determined by the `Backlog Task Scheduler`. Using this modeling concept, we integrated the main project planning behavior of Scrum into our process model. The two additional actions in Fig. 3 have simple call semantics executing the tasks they refer to (`Sprint Review Meeting` and `Sprint Retrospective Meeting`). In the same way, the behavior of a Sprint shown in Fig. 3 may be called by another activity.

Although this example lacks some detail (see [20] for a more detailed step-by-step presentation), we expressed the main behavior of a Scrum sprint with just a few model elements. This behavior may be simulated with a PME or enacted with a PES to drive a Scrum-based project. For the latter, a user must deploy the process model as a process plugin to a PES.

3.2 Enacting Scrum

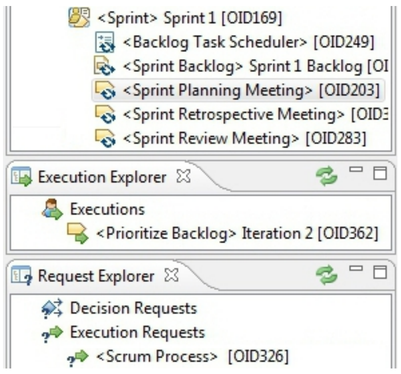

<Sprint> Sprint 1 [OID169]
<Backlog Task Scheduler> [OID249]
<Sprint Backlog> Sprint 1 Backlog [OI
<Sprint Planning Meeting> [OID203]
<Sprint Retrospective Meeting> [OID:
<Sprint Review Meeting> [OID283]
Execution Explorer
Executions
<Prioritize Backlog> Iteration 2 [OID362]
Request Explorer
Decision Requests
Execution Requests
<Scrum Process> [OID326]

Fig. 4. Scrum Enactment with our PEC

After a process plugin is successfully deployed to a PES, a PEC may be employed to create a development project that uses this process. When a project is created the PES automatically creates an initial activity, artifact repositories, and proxy objects for the process roles and tools. Afterwards, the behavior of the initial activity must be started manually to execute the process. In our example, the `Sprint Behavior` is executed. As shown in Fig. 3, a sprint planning meeting has to be executed first in a sprint. To guide the user, the PES interprets the process model and creates a request to execute a sprint planning meeting. A PEC displays these pending execution requests in a dedicated view called `Execution Explorer` (see Fig. 4). This view provides context-sensitive actions to mark a task as finished. When a user marks the sprint planning meeting as finished, the PES executes the next action (`Execute Backlog Tasks`) by inspecting the `Sprint Backlog` (provided as input parameter) and scheduling the tasks in the backlog according to the `Backlog Task Scheduler`. This results in further execution requests in the `Execution Explorer` that must be handled by a user. When all requests are handled, execution of a `Sprint Review Meeting` and a `Sprint Retrospective Meeting` is requested by the PES. Finally, the control flow returns to the caller of the `Sprint Behavior` that may start another sprint. With this short example, we demonstrated how to model a SDP with our PME, deploy it, and enact it with a PES and a PEC.

4 Related Work

Many authors have identified software process modeling and execution as relevant for producing software. Early approaches of software processes modeling are executable [2] but had limited impact in industry due to their complex formalisms, low level of abstraction, or inferior tool support [22]. Therefore, below we focus on high-level modeling languages with up-to-date tool support.

The Microsoft Team Foundation Server (TFS) [23], IBM Rational Team Concert (RTC) [24], and Method Park Stages [25] are popular Application Lifecycle Management (ALM) tools that integrate other tools, e.g., configuration management systems, change management systems, and IDEs into a distributed development and collaboration platform. Although they can be adapted to a process by using templates, their template languages are limited to static aspects of the processes only. In contrast to our work, these tools cannot guide and support project staff in executing a custom process based on a process template, e.g., they cannot determine the next possible steps in a process and guide project staff accordingly.

UML4SPM [26] extends SPEM 1.1 with UML 2.0 behavior modeling concepts. Although its operational semantics has been implemented on top of fUML to simulate and execute UML4SPM-based process models [27], there is no support for distributed execution that is needed for realistic team-based development.

The Eclipse Process Framework (EPF) [28] offers a reference implementation of SPEM 2.0. With the IBM Rational Method Composer [29] a commercial version of EPF is also available. Although comprehensive EPGs may be generated from such EPF models, they lack a description of the behavior as SPEM lacks suitable language constructs. Hence, EPF cannot simulate or enact a process and thus cannot help exploit the mentioned benefits, e.g., it makes process validation more complex.

The Process Enactment Toolkit (PET) [30] is a framework to enact process models. PET is a generic model-to-model transformation framework with input adapters for different process modeling languages and output adapters for different issue management, collaboration, and ALM tools like TFS. As stated above, TFS and other ALM tools fall short in guiding and supporting project staff. Moreover, in contrast to our work PET cannot simulate process execution.

5 Conclusion and Future Work

In this paper, we presented an integrated tool chain for modeling and enacting software processes. The Eclipse-based tool provides a reasonable support for modeling, documenting, simulating, and enacting eSPEM-based software processes. It also comes with a process simulator to easily validate a process before it is enacted. Our context-sensitive help provides easy access to the relevant part of the process documentation. Our automated process logging keeps accurate data for process audits and improvements without any additional effort for project staff. Furthermore, the tool automatically checks for process conformance and proactively prevents illegal process states. Our tool chain relies on the Eclipse architecture to integrate other Eclipse-based extensions and third party tools. Currently no other tool provides such a completely integrated approach that leverages all mentioned benefits of automated process enactment.

Our future work will extend the scope of our integrated tool chain to support process assessments and to check conformance of process models to process reference models, e.g., CMMI [31], and standards like ISO 26262 (Road vehicles – Functional safety) [32]. Since our tool chain provides access to the process model and runtime data, it is an ideal environment to check conformance of processes. We integrate standard models and trace models with our tool chain in order to support assessments of process models, and assessments of executed processes in multi-certification contexts (i.e., processes must conform to more than one standard).

References

1. Osterweil, L.: Software processes are software too. In: Proc. 9th Intl. Conf. Software Eng., Monterey, CA. (Apr. 1987) 2–13
2. Zamli, K., Lee, P.: Taxonomy of Process Modeling Languages. In: Proc. ACS/IEEE Intl. Conf. Computer Sys. and Appl., Beirut, Lebanon. (Jun. 2001) 435–437
3. Acuña, S.T., Ferré, X.: Software Process Modelling. In: Proc. World Multiconf. Systemics, Cybernetics, and Informatics, Orlando, FL. (Jul. 2001) 237–242
4. Bendraou, R., Jezequel, J.M., Gervais, M.P., Blanc, X.: A Comparison of Six UML-Based Languages for Software Process Modeling. IEEE Trans. Softw. Eng. **36**(5) (2010) 662–675
5. Object Management Group: Software & Systems Process Engineering Meta-Model Specification, Ver. 2.0. (Apr. 2008)
6. Object Management Group: OMG Unified Modeling Language, Infrastructure, Ver. 2.3. (May 2010)
7. Almeida da Silva, M., Bendraou, R., Blanc, X., Gervais, M.P.: Early Deviation Detection in Modeling Activities of MDE Processes. In: Proc. 13th Intl. Conf. Model Driven Eng. Lang. and Sys., Oslo, Norway. Volume 6395 of LNCS. (2010) 303–317

8. Ellner, R., Al-Hilank, S., Drexler, J., Jung, M., Kips, D., Philippsen, M.: eSPEM - A SPEM Extension for Enactable Behavior Modeling. In: Proc. 6th Europ. Conf. Model. Foundations and Appl., Paris, France. Volume 6138 of LNCS. (2010) 116–131
9. Object Management Group: OMG Unified Modeling Language, Superstructure, Ver. 2.3. (May 2010)
10. Ellner, R., Al-Hilank, S., Drexler, J., Jung, M., Kips, D., Philippsen, M.: A FUML-based Distributed Execution Machine for Enacting Software Process Models. In: Proc. 7th Europ. Conf. Model. Foundations and Appl., Birmingham, UK. Volume 6698 of LNCS. (2011) 19–34
11. Object Management Group: Semantics of a Foundational Subset for Executable UML Models, Ver. 1.0 Beta 3. (Mar. 2010)
12. McAffer, J., VanderLei, P., Archer, S.: OSGi and Equinox. Addison-Wesley Longman (2010)
13. Eclipse Project: `http://eclipse.org/`. (Apr. 2012)
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. 2nd edn. Addison-Wesley Longman (2009)
15. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Longman (2009)
16. Object Management Group: Object Constraint Language, Ver. 2.2. (Feb. 2010)
17. Rose, L., Kolovos, D., Paige, R., Polack, F.: Model Migration with Epsilon Flock. In: Theory and Practice of Model Transform. Volume 6142 of LNCS. (2010) 184–198
18. Riena Platform Project: `http://eclipse.org/riena/`. (Apr. 2012)
19. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press (2004)
20. Al-Hilank, S., Ellner, R.: Modeling and Enacting Scrum (Screencast), `http://www2.cs.fau.de/research/IWKMMASWEP/screencasts/`. (Apr. 2012)
21. Al-Hilank, S., Ellner, R.: eSPEM and tool chain download page, `http://www2.cs.fau.de/research/IWKMMASWEP/download/`. (Apr. 2012)
22. Gruhn, V.: Process Centered Software Engineering Environments – A Brief History and Future Challenges. Annals of Softw. Eng. **14**(1-4) (2002) 363–382
23. Microsoft: VS Team Foundation Server, `http://microsoft.com/vs/`. (Apr. 2012)
24. IBM: Rational Team Concert, `http://ibm.com/rational/rtc/`. (Apr. 2012)
25. Method Park Software AG: Stages, `http://methodpark.com/en/product.html`. (Apr. 2012)
26. Bendraou, R., Gervais, M.P., Blanc, X.: UML4SPM: A UML2.0-Based Metamodel for Software Process Modelling. In: Proc. 8th Intl. Conf. Model Driven Eng. Lang. and Sys., Montego Bay, Jamaica. Volume 3713 of LNCS. (2005) 17–38
27. Bendraou, R., Jezéquél, J.M., Fleurey, F.: Achieving process modeling and execution through the combination of aspect and model-driven engineering approaches. J. of Softw. Maintenance and Evolution: Research & Practice (2010) Preprint.
28. Eclipse Process Framework Project: `http://eclipse.org/epf/`. (Apr. 2012)
29. IBM: Rational Method Composer, `http://ibm.com/rational/rmc/`. (Apr. 2012)
30. Kuhrmann, M., Kalus, G.: Providing Integrated Development Processes for Distributed Development Environments. In: Workshop on Supporting Distributed Team Work at Computer Supported Cooperative Work (CSCW 2008). (Nov. 2008)
31. CMMI Product Team: CMMI for Development, Ver. 1.3. Technical Report CMU/SEI-2010-TR-033, Carnegie Mellon Univ. – Software Eng. Inst. (Nov. 2010)
32. Intl. Org. for Standardization: ISO 26262: Road vehicles – Functional safety. (Nov. 2011)

Transient View Generation in Eclipse*

Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden

Real-Time and Embedded Systems Group,
Department of Computer Science, Christian-Albrechts-Universität zu Kiel
{chsch,msp,rvh}@informatik.uni-kiel.de

Abstract. Graph-based model visualizations can effectively communicate information, but their creation and maintenance require a lot of manual effort and hence reduce productivity. In this paper we build on the concept of Model Driven Visualization by presenting a meta model for graphical views and an infrastructure for configurable automatic layout. This enables the *transient views* approach, in which we efficiently derive and draw graph representations from arbitrary models.

1 Introduction

Graphical modeling languages such as the UML and its dialects as well as domain-specific modeling languages (DSMLs) can be effectively employed for a multitude of purposes, including documentation, communication, and code generation. However, an important amount of productivity is wasted with drawing and beautification activities while working with state-of-the-practice modeling environments. Even when automatic layout is available, it is often unable to properly consider domain-specific requirements on the layout. Furthermore, the development of a graphical notation can be unnecessarily tedious, especially when it comes to the details.

Our objective is to allow *lightweight* and *transient* graphical representations, following the concept of Model Driven Visualization (MDV) introduced by Bull et al. [4]. This means that views are completely specified using model transformations, hence no manual user interaction is required. Our main contribution is a meta model for graphs, their layout, and their graphical representation. We employ this meta model both for the view model of generated graphical views and for the interface to layout algorithms. Furthermore, we present an infrastructure for automatic layout that is statically and dynamically configurable and provides the foundation for transient view synthesis. Both contributions are realized within the KIELER project,[1] which provides an Eclipse update site. A more detailed version of this paper is available as technical report [12].

This paper is organized as follows. We discuss previous work on view synthesis and layout integration in Sect. 2. The meta models and basic approaches for transient views are presented in Sect. 3. The integration and configuration of layout algorithms in Eclipse is described in Sect. 4, after which we conclude.

* This work was funded in part by the Program for the Future Economy of Schleswig-Holstein and the European Regional Development Fund (ERDF)

[1] http://www.informatik.uni-kiel.de/rtsys/kieler/

2 Related Work

The work presented here builds on Model Driven Visualization as proposed by Bull et al. [4, 2], which is an extension of the Model Driven Engineering (MDE) approach to the creation of views. This helps to lift the development of graphical tools to a more abstract level. However, Bull et al. focus on view models for different kinds of data visualization, which does not only include graphs, but also tables and charts. The *Zest* toolkit [3] employed in their contribution mainly addresses the SWT integration of graph viewers and offers only few graph layout algorithms. In our approach we go one step further and add rendering specification as well as layout directives to the graph view model and hence allow to express all details of the generated view using MDE methods.

A very simple tool for visualizing EMF models is offered by the *EMF To Graphviz* project.[2] Being restricted to drawing boxes with lists of attributes and using Graphviz as layout engine [8], this tool it is very limited in terms of rendering and layout, which makes it useful for debugging and rapid prototyping, but insufficient for more complex visualizations.

The established graphical modeling frameworks GMF and Graphiti are not well suited for transient model visualization in the sense of this paper. Both are designed for composing models by dragging and dropping figures onto a diagram canvas. They require a fully-fledged editor setup in order to simply *show* diagrams, which is a waste of resources that is felt bitterly for large diagrams. Although Graphiti maintains the description of figures in a view model (the *Pictogram* model), some characteristics, such as the bend point rendering of edges (angular or rounded), are configured in the editor code. GMF's *Notation* model has no means at all for specifying rendering primitives, but points to predefined edit part classes using integer identifiers. The arrangement of figures (*micro layout*) must be realized in Java code in both frameworks. While GMF relies on the *layout manager* concept of Draw2D, Graphiti requires to implement *layout features*. This inconsistent use of view models for the specification of graphics impedes the application of model transformations and other model-based techniques for full-automatic view generation.

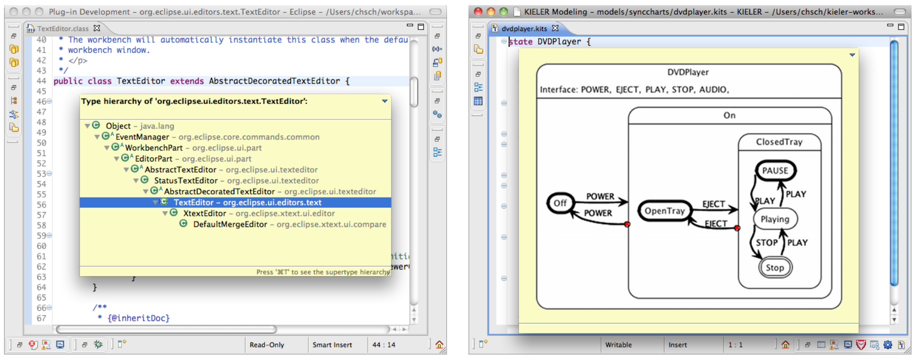
Although editor code generation front-ends such as *GMF Tooling*[3] and *Spray*[4] offer means for model-based view specification, the generated code suffers from the same problems as described above. The additional level of abstraction makes it even harder to customize and fine-tune the views. Furthermore, GMF Tooling requires a tight coupling of model and view. This imposes strong requirements on the structure of the meta model, i. e. the abstract syntax of the language, which is not acceptable, since we aim for multiple views on the same model.

Modeling tools such as GME [10] and VMTS [11], which are built on Windows instead of Eclipse, allow the creation of graphical editors with custom graphics, but the graphics must be created by either using a specific API or editing a

[2] http://sourceforge.net/projects/emf2gv/

[3] http://www.eclipse.org/modeling/gmp/?project=gmf-tooling

[4] http://code.google.com/a/eclipselabs.org/p/spray/


Plug-in Development – org.eclipse.ui.editors.text.TextEditor – Eclipse – /Users/chsch/workspa...
TextEditor.class
Read-Only
Smart Insert
44 : 14
KIELER Modeling – models/synccharts/dvdplayer.kits – KIELER – /Users/chsch/kieler-works...
dvdplayer.kits
DVDPlayer
Interface: POWER, EJECT, PLAY, STOP, AUDIO,
On
ClosedTray
PAUSE
PLAY
PLAY
Playing
STOP
PLAY
Stop
Off
POWER
POWER
OpenTray
EJECT
EJECT
Writable
Insert
1 : 1

```
 * The workbench will automatically instantiate this class when the defau
 * workbench window.
 * </p>
 */
public class TextEditor extends AbstractDecoratedTextEditor {
```

```
Type hierarchy of 'org.eclipse.ui.editors.text.TextEditor':

Object – java.lang
  EventManager – org.eclipse.core.commands.common
    WorkbenchPart – org.eclipse.ui.part
      EditorPart – org.eclipse.ui.part
        AbstractTextEditor – org.eclipse.ui.texteditor
          StatusTextEditor – org.eclipse.ui.texteditor
            AbstractDecoratedTextEditor – org.eclipse.ui.texteditor
              TextEditor – org.eclipse.ui.editors.text
                XtextEditor – org.eclipse.xtext.ui.editor
                  DefaultMergeEditor – org.eclipse.xtext.ui.compare

Press '⌘T' to see the supertype hierarchy
```

```
        }
    }

    /**
     * {@inheritDoc}
```

```
state DVDPlayer {
```

(a) Inheritance hierarchy of a class. (b) Proposal: transient Statechart diagram.

Fig. 1. Examples of transient views

visualization model in the UI. The focus of these projects is on meta-modeling and model transformations, and they do not cover automatic view generation and layout.

3 Towards Lightweight Graphical Modeling

The common graphical modeling approaches, either based on generic modeling environments or customized editing tools, require the modeler to manually put each single element on the canvas and determine its position. If an element shall be characterized with respect to different facets, e. g. a class as part of a software system, multiple diagrams must be drawn, each representing that element from different points of view. Those diagrams are often persisted in separate files, which may lead to consistency issues if elements are reordered or deleted. Regarding this dissatisfying situation we believe that it can be improved by exploiting the ability to automatically arrange diagram elements.

3.1 Transient Graphical Views

We propose to employ the *transient views* approach, which consists of the direct synthesis of graphical views out of existing models. This inverts the traditional *graphical editing* approach, in which a model is constructed using a graphical view. In our vision a modeler works with an arbitrary editor, e. g. based on a textual DSL, and requests and dismisses graphical views like Java programmers hit *ctrl+T* to see the inheritance hierarchy of a class, see Fig. 1. This way the benefits of graphical modeling are preserved, while disadvantages such as time consuming composition are avoided.

The transient graphical view synthesis process comprises the following steps.

1. Select models to be represented, possibly with manual or automatic filtering.

2. Construct a view model according to mapping rules from the domain model.
   a. Identify the essential graph elements (nodes, edges, labels, ports).
   b. Create each element's graphics by composing rendering primitives.
   c. Arrange the rendering primitives (*micro layout*).
3. Arrange the graph structure of the view model (*macro layout*).
   a. Analyze the view model and derive a layout graph.
   b. Configure the layout by choosing layout algorithms and setting options.
   c. Execute the layout algorithms.
   d. Transfer the computed layout back to the view model.
4. Render the view model by means of a 2D graphics framework.

In our approach we aim to optimize the synthesis process in terms of performance and simplicity in order to justify the predicate "lightweight". This would enable truly transient views, eliminating the necessity of persisting the view models. We achieve this by employing the same meta model for layout algorithms and for the view model, and extending it with annotations for expression of rendering primitives and their arrangement. This yields the following benefits:

– The view model is based on EMF and thus allows to use model transformation as well as other model-based techniques, which is the basic idea of MDV [4]. For instance, this enables the employment of interpreted transformations that could be formulated by the tool user. This advantage applies to all three parts of Step 2 in the view synthesis process.
– In common graphical editors the micro layout (Step 2.c) is implemented in Java (see Sect. 2). By including the micro layout specification in the rendering model we are able to express it on an abstract level. While this may seem like a trivial matter, it turns out to be crucial for the consistent use of automatic layout: as illustrated in Fig. 2, changes of the macro layout may require recomputation of the micro layout. Therefore a close coupling of both levels of layout is beneficial.
– Step 3.a is often an intricate task, which concerns the extraction of the graph structure as well as the initial macro layout. We obtain the simplest possible solution by using the same graph structure for the view model and the layout process, hence no transformation or adaption is needed. This also applies to Step 3.d, since the concrete layout attached to the graph instance during execution of layout algorithms directly affects the view model.

3.2 Use Cases

Applications of transient views are manifold. As motivated in Sect. 3.1, modelers may want to get certain information on their system under development. Similarly, modelers continuously want to check the correctness of their work, e. g. the reachability of states in Statecharts, by reviewing it in an alternative notation. In case of an error the fix shall be performable directly in the view.

In practice, specifications of large systems are usually created in a component-based way. This often occurs in form of declaring and referencing elements separately. An example, found in the railway signaling domain, looks as follows

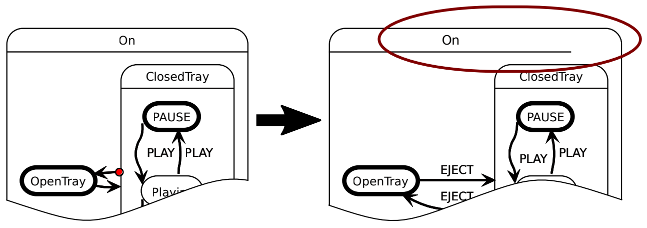


Fig. 2. Broken figure rendering after an update of the Statechart diagram: inserting labels on the transitions between the OpenTray and ClosedTray states causes the On state to be enlarged; afterwards the state label is not centered anymore and the line below is too short.

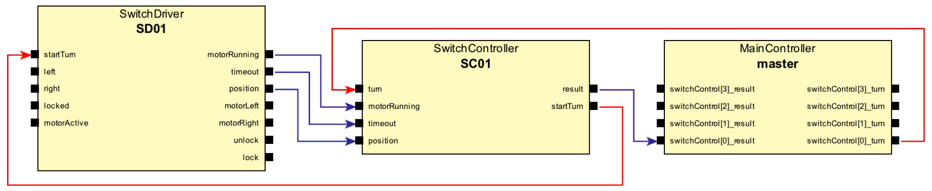


Fig. 3. Component architectural outline on a specification excerpt of a complex railway signaling system. It has been composed based on various single description parts.

(simplified). There are three types of components, each of them specified in a separate document: a MainController, SwitchControllers, and SwitchDrivers. The components communicate via dedicated interfaces described in further specification parts. Finally, instances of the components are introduced and connected in an additional statement. Although those particular descriptions may be simple, the resulting networks can become quite complex and difficult to browse, understand, and maintain. By means of transient views the tool can offer specific compound representations, which are built upon multiple parts of the specification, and provide a component architectural outline as shown in Fig. 3.

3.3 The KGraph and KRendering Meta Models

The *KGraph* meta model describes the graph as used in steps 2.a and 3.a of the view synthesis process. Its class diagram, derived from an EMF Ecore model, is shown in Fig. 4. The graph structure is represented by the classes KNode, KEdge, KPort, and KLabel. Each instance of these graph elements contains an attached KEdgeLayout (for edges) or KShapeLayout (for other elements), which are both able to hold concrete layout data as well as abstract layout data represented by *layout options* (see Sect. 4). A graph is represented by a KNode instance with its content stored in the children reference.

Fig. 5 depicts an excerpt of the *KRendering* notation meta model, which is an extension of KGraph. Basic figure shapes are instances of KRendering, which inherits from KGraphData and thus can be attached to KGraphElements. KRen-

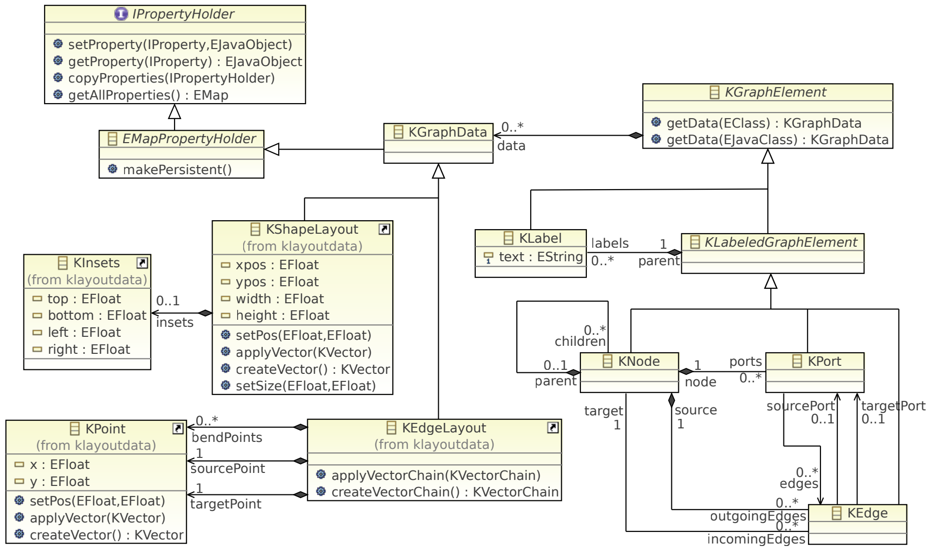


Fig. 4. KGraph meta model with basic graph structures and layout data.

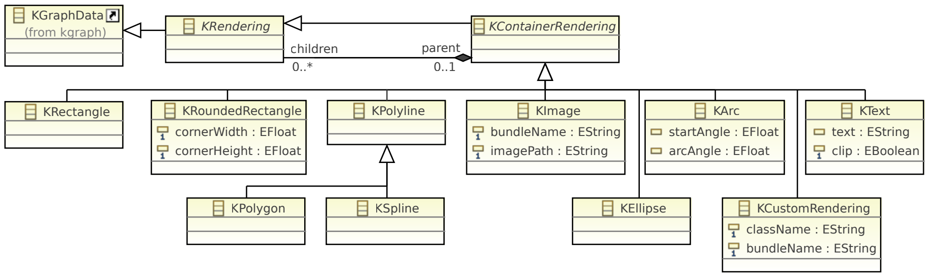


Fig. 5. KRendering basic shapes to be composed to diagram figures.

derings can be configured in terms of KStyles for specification of properties such as line width or foreground and background color (see Fig. 6). KRenderingRefs refer to other rendering definitions, and templates of KRenderings may be stored in a KRenderingLibrary. The placement of KRenderings can be defined by means of KDirectPlacementData or KPolylinePlacementData (see Fig. 7), or by attaching them to grid- or stack-based micro layouts, which are omitted here. The direct placement definitions consist of two points, given by KX-/KYPositions, that are related to the borders of the parent KRendering. Polyline placements consist of a number of bend positions, respectively. These implicit coordinates are resolved in Step 4 of the view synthesis process.

The composition of diagram figures by means of the KRendering primitives is exemplified in Fig. 8, showing a description excerpt of the SwitchController diagram node from Fig. 3. The figure consists of a KNode comprising a KShapeLay-

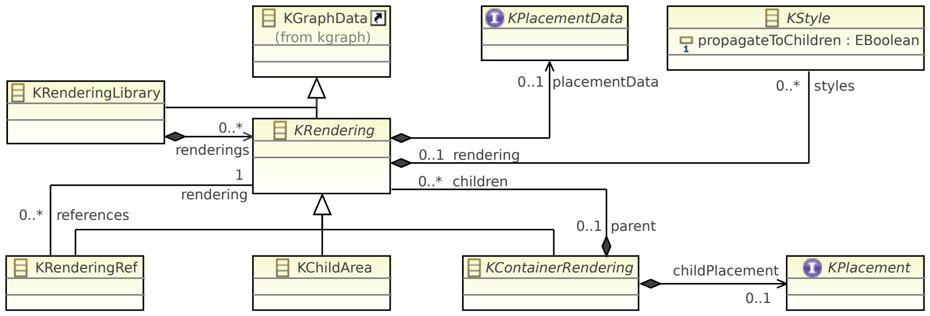


Fig. 6. KRendering core elements.

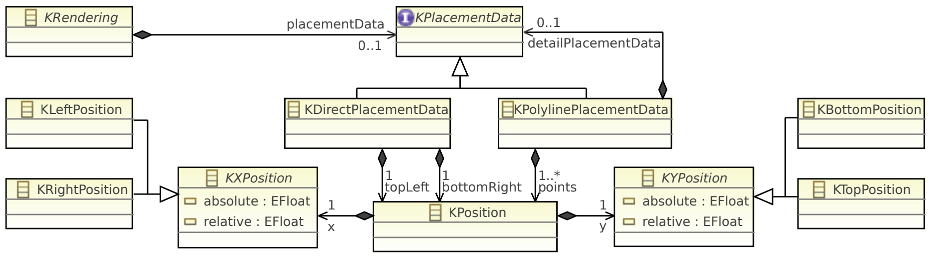


Fig. 7. KRendering placement elements for specifying micro layout directives.

out defining its size and a layouter hint, a KRectangle, and a bunch of KPorts (only one is shown for the sake of brevity). The rectangle covers the whole figure, since no placement data are given, and contains two horizontally centered text fields showing the type and instance name of the depicted element (the second text field's description is omitted, too). They are modified in terms of the text's vertical alignment, the transparency of the background color, and their size and position in the figure that is characterized by the KDirectPlacementData entry. Thus, the first text field spans a rectangle ranging from the left to the right and from the top to 14pt below the top border of the diagram figure. The horizontally centered alignment is set by default.

The figures of the ports are basically constructed in the same way, apart from horizontal alignment of the port label text field. In addition, one observes that KText elements determine their minimal size by their font size and text length. Hence, the horizontal part of the bottomRight position does not matter for the left aligned port label. Observe furthermore that child KRenderings need not to be placed within the bounds of its parent, which is the case for the port label.

The view synthesis process is realized in the KIELER Lightweight Diagrams (KLighD) project, our test bed for investigating the topic of transient graphical representations, which is integrated in the KIELER view management [7]. KLighD provides infrastructure to manage the mapping rules needed in Step 2, which currently have to be provided by the tool developer. Step 3, the macro layout, is

```
KNode {
 KShapeLayout {
  width 200 height 100
  mapProperties:
  "de.cau.cs.kieler.portConstraints"
      = "FIXED_POS"
 },
 KRectangle {
  KBackgroundColor 255 250 205
  KText "SwitchController" {
   KVerticalAlignment TOP
   KBackgroundVisibility false
   KDirectPlacementData {
     topLeft KLeftPosition abs 0.0 rel 0.0
           / KTopPosition abs 0.0 rel 0.0
     bottomRight KRightPosition abs 0.0 rel 0.0
               / KTopPosition abs 14.0 rel 0.0
   }
  },
  ...
 }
```

```
KPort {
 KShapeLayout {
  xpos -8 ypos 31 width 9 height 9
 },
 KRectangle {
  KBackgroundColor 0 0 0
  KText "turn" {
   KFontSize 9,
   KHorizontalAlignment LEFT
   KBackgroundVisibility false
   KDirectPlacementData {
    topLeft KLeftPosition abs 12.0 rel 0.0
          / KTopPosition abs 0.0 rel 0.0
    bottomRight KLeftPosition abs 0.0 rel 0.0
          / KBottomPosition abs 0.0 rel 0.0
   }
  }
 }
},
...
}
```

Fig. 8. Excerpt of the KRendering-based specification of the SwitchController diagram element depicted in Fig. 3.

delegated to the KIELER Infrastructure for Meta Layout (KIML), see Sec. 4, and Step 4, the rendering of the representation, is performed by the graphics framework *Piccolo2D* [1]. Since the view model (KGraph + KRendering) does not rely on any specific graphics framework, we will add support for other frameworks such as Draw2D in the future.

4 Configurable Automatic Layout

Research on graph drawing algorithms has led to a rich variety of methods over the past 30 years [6, 9]. In theory, these layout methods should equip users of graphical modeling tools adequately to satisfy their need for automatic diagram layout. However, today's modeling tools are still quite far from the point where diagram layout would be available with the same flexibility as textual formatting such as the Java code formatter of Eclipse JDT. The problem is not the lack of appropriate algorithms or libraries for graph layout, but rather their integration.

Two examples of excellent libraries are OGDF [5] and Graphviz [8], both of which offer several layout methods with plenty of options for customization. The former offers a C++ API and support for GML and OGML graph formats, while the latter offers a C API and support for the DOT graph format. Connecting one of these tools to a Java application is a costly task, since it consists either in the intricacy of directly executing native code or in the communication with a separate process using one of the supported graph formats.

The KIELER Infrastructure for Meta Layout (KIML) provides a bridge between diagram viewers and layout algorithms and offers interfaces for layout configuration. Graphviz, OGDF, and a collection of Java-based algorithms are included, thus offering a wide variety of layouts to Eclipse based diagram editors and viewers. Graphiti and GMF have been connected generically such that layout can be done in many editors that are based on these frameworks without the need of adding or changing any code. However, as explained in Sect. 3.1, the layout process is a lot more efficient for KLighD views.

4.1 Layout Configuration

We consider two levels of automatic layout: concrete layout and abstract layout. A concrete layout determines the exact position and size of all elements of a graph, including nodes, labels, and edge bend points, whereas an abstract layout consists of options for the selection and configuration of layout algorithms. When an algorithm is executed on an input graph, it reads these options and considers them in the calculation of graph element positions.

Layout options can be set for each graph element independently. This allows to modify general settings of an algorithm, to set constraints for specific graph elements, or even to apply different layout algorithms for different hierarchy levels of a compound graph. These layout options are set in Step 3.b of the view synthesis process described in Sect. 3.1 by executing a set of *layout configurators* on each graph element. Some of the currently defined layout configurators are described in the following.

- The **Default** configurator has the lowest priority and returns default layout settings that are acceptable for most graphs.
- The **Eclipse** configurator manages an extension point and a preference page, which can both be used to override default values for specific element types. The edit part class or the domain model class can be used to specify the type of a graph element.
- The **Semantic** configurator is an extensible mechanism for deriving layout settings from the domain model. This is used when different layout option values are chosen depending on properties of the domain model instance.
- The **GMF / Graphiti** configurators allow to customize the layout for a single diagram, which can be done through an Eclipse view named *Layout*. For GMF diagrams the options are stored as Style annotations in the *Notation* model, while for Graphiti the options are stored as Property annotations in the *Pictogram* model.

5 Conclusion and Future Work

We presented a continuation of the MDV approach by allowing the view model to express graph structure as well as rendering and layout directives. The KGraph meta model includes structural information that is relevant for layout algorithms, properties for abstract layout specification, and concrete layout data calculated by algorithms. The KRendering meta model adds rendering primitives with style and micro layout annotations. The system is backed by a flexible and configurable automatic layout infrastructure. Putting these building blocks together yields a tool that is well suited for the visualization of complex models.

Our future work will target various aspects. Diagram specifications shall be expressed in a textual language, similarly to Köhnlein's *Generic Graph View*.[5] This involves describing the rendering of elements, as well as composing diagrams, which may be understood as *queries* on a model base. The synthesized diagrams shall be equipped with *semantic zoom*, i. e. the ability to change their amount of detail according to the user's focus. Finally, intuitive means for modifying the abstract layout, e. g. in form of sliders or gestures, shall be investigated.

References

1. Bederson, B.B., Grosjean, J., Meyer, J.: Toolkit design for interactive structured graphics. IEEE Transactions on Software Engineering 30(8), 535–546 (Aug 2004)
2. Bull, R.I.: Model Driven Visualization: Towards A Model Driven Engineering Approach For Information Visualization. Ph.D. thesis, University of Victoria, BC, Canada (2008)
3. Bull, R.I., Best, C., Storey, M.A.: Advanced widgets for Eclipse. In: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange. pp. 6–11. ACM, New York, NY, USA (2004)
4. Bull, R.I., Storey, M.A., Litoiu, M., Favre, J.M.: An architecture to support model driven software visualization. In: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06). pp. 100–106. IEEE (2006)
5. Chimani, M., Gutwenger, C., Jünger, M., Klein, K., Mutzel, P., Schulz, M.: The Open Graph Drawing Framework. Poster at the 15th International Symposium on Graph Drawing (GD07) (2007)
6. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall (1998)
7. Fuhrmann, H., von Hanxleden, R.: Taming graphical modeling. In: Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10). LNCS, vol. 6394, pp. 196–210. Springer (Oct 2010)
8. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software—Practice and Experience 30(11), 1203–1234 (2000)
9. Kaufmann, M., Wagner, D. (eds.): Drawing Graphs: Methods and Models. No. 2025 in LNCS, Springer-Verlag, Berlin, Germany (2001)
10. Lédeczi, Á., Maróti, M., Bakay, Á., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Völgyesi, P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing (2001)
11. Mezei, G., Levendovszky, T., Charaf, H.: Visual presentation solutions for domain specific languages. In: Proceedings of the IASTED International Conference on Software Engineering. Innsbruck, Austria (2006)
12. Schneider, C., Spönemann, M., von Hanxleden, R.: Transient view generation in Eclipse. Technical Report 1206, Christian-Albrechts-Universität zu Kiel, Department of Computer Science (Jun 2012)

[5] http://koehnlein.blogspot.de/2012/01/discovery-diagrams-for-generic.html

CommentTemplate: A Lightweight Code Generator for Java built with Eclipse Modeling Technology

Jendrik Johannes, Mirko Seifert, Christian Wende, Florian Heidenreich, and Uwe Aßmann

DevBoost GmbH
D-10179, Berlin, Germany

Technische Universität Dresden
Chair of Software Technology
D-01062, Dresden, Germany

{firstname.lastname}@devboost.de

Abstract. In this paper we present CommentTemplate, which realizes code generation features, as known from many model-driven development tools, as a Java language extension. This paper first introduces CommentTemplate and discusses its features and limitations. Second, it discusses CommentTemplate as an example of (a) a tool that makes concepts from model-driven development easily accessible for Java programmers and (b) a powerful and stable tool that was realized in a short time thanks to Eclipse modeling technology.

1 Introduction

One of the fundamental technologies of model-driven development approaches is code generation. Hence, many code generation technologies emerged over the years. In the Eclipse modeling project alone, multiple solutions exist [1–6]. Furthermore, there exist numerous code generation tools and framework for Java outside Eclipse, which can also be combined with Eclipse modeling technology (e.g., [7–9] and many more).

An issue model-driven development approaches, and code generation in particular, face when originating from academia, is their adoption by the "common" developer in industry. First, an approach needs adequate tool support above the level of academic prototyping. Second, such tools should be easily accessible for the developer. That is, the developer should be able to immediately start working with the tools without the need to acquire more theoretical knowledge in the beginning (flat learning curve).

Eclipse, and the Eclipse Modeling Framework (EMF), have greatly supported these two points in the past. First, the Eclipse plugin mechanism and the extendability of EMF allow a rapid development of new high-quality modeling tools by reusing existing, well-tested functionality. Second, Eclipse, being the development platform of choice for many Java developers, gives Java developers a quicker access to modeling technologies, because they are integrated into the tool (Eclipse) they use on a daily basis. However, this second point so far mainly addressed integration with the Eclipse IDE and not with the Java language itself. This makes it still difficult for Java developers to easily understand and use modeling tools which, at a first glance, are not connected to Java programming.

We previously developed the Java Model Printer and Parser (JaMoPP) [10], to bring Eclipse modeling and the Java language closer together. JaMoPP consists of a Java metamodel (defined in EMF's Ecore) and the tooling to parse Java source (and byte) code into instances of that metamodel as well as to print instances back to Java source code. This increases integration of modeling and Java both on the modeling and metamodeling level. On the modeling level, JaMoPP is used to handle the Java language equally to other modeling languages, which allows Eclipse modeling tools to work on Java source code as on other models (e.g., model-to-model transformations can be used to generate Java code). On the metamodeling level, JaMoPP allows the integration of Java and other (modeling) languages. New elements can be inserted into Java by extending the Java metamodel and syntax or parts of Java can be reused in other (modeling) languages by importing the Java metamodel and syntax.

In this paper, we present CommentTemplate, which is a code generation extension for the Java language. CommentTemplate is both an example of a tool that transfers concepts of model-driven development to the programming tool world and an example of a powerful, stable tool that was developed in a short time by taking advantage of Eclipse and Eclipse modeling technology. CommentTemplate takes the important features of existing code generation languages, which are not offered by Java itself already, and implements those as a lightweight extension for Java instead of providing a new language. This demonstrates how fundamental features of model-driven technology can be realised closely to an existing and well-accepted programming language to make these features, which are valuable in their own right, more accessible for programmers. CommentTemplate is developed based on JaMoPP and consists of approximately 800 lines of code. This shows how JaMoPP can indeed be used on the metamodeling level for developing tools that integrate into the Java language and thus benefit from the existing Java tooling in the Eclipse IDE. Although, we are both modeling enthusiasts and Java programmers, we prefer using CommentTemplate over other code generation tools in certain situations. We motivate our reasons for that by discussing the advantages and disadvantages of CommentTemplate.

The paper is structured as follows: Section 2 introduces CommentTemplate an explains its features. Section 3 discusses how CommentTemplate is implemented using Eclipse modeling technology. Section 4 takes a critical look at the usefulness of CommentTemplate and discuss its application areas. Section 5 discusses limitations of CommentTemplate and Sect. 6 points at related work, before a conclusion is given in Sect. 7.

```
 1 @CommentTemplate
 2 public String helloWorld() {
 3     String greeting = "Hello";
 4     /*<html>
 5         <head><title>greeting world!</title></head>
 6         <body>*/
 7         for (int i = 1; i <= 5; i++) {
 8             String greeted = "World" + i;
 9             /*
10                 greeting greeted!<br/>*/
11             if (greeted.equals("World2")) {
12                 /*
13                     greeted, you are the best!<br/>*/
14             }
15         }
16         /*
17         </body>
18     </html>*/
19     return null;
20 }
```

Listing 1. HelloWorld @CommentTemplate method

2 CommentTemplate Syntax and Features

CommentTemplate is a Java language extension for code generation. It makes use of and extends the following Java language elements: *multi-line comments*, *methods*, *local variables* and *annotations*. A code generation template written in CommentTemplate is shown in Listing 1.

A template is defined as a Java method annotated with `@CommentTemplate` that has the return type String.[1] Inside the Method, one can use multi-line comments (`/* */` notation) to define fragments of the template. Around these fragments, arbitrary Java code can be written and used to formulate, for example, loops (Line 7) or conditions (Line 11). Furthermore, one can refer to local variables of type String that are declared before the corresponding template fragment. In the example, the variable `greeting` (declared in Line 3) is referred to two times inside template fragments (Lines 5 and 10).

A `@CommentTemplate` method can be called as any Java method inside arbitrary Java code. However, it will return the expanded template as String. That is, all template fragments are appended and the variables inside the fragments are filled with their values. In the example of Listing 1, the String shown in Listing 2 is produced.

This sums up the basic features of CommentTemplate. However, CommentTemplate offers two additional annotations which help with syntax conflicts between templates and output syntax.

First, one can observe, that no special quotation is used to mark variables in a template fragment in the example (e.g. `greeting` in Line 5). Other code generation tools usually define a fixed symbol for escaping such variables. This is sometimes problematic, because depending on which output is generated, escape symbols can conflict with the output syntax. CommentTemplate does

[1] To write compilable code, a `return null;` statement is needed at the end of the method, which will be replaced by the CommentTemplate compiler (cf. Sect. 3).

```
<html>
    <head><title>Hello world!</title></head>
    <body>
        Hello World1!<br/>
        Hello World2!<br/>
        World2, you are the best!<br/>
        Hello World3!<br/>
        Hello World4!<br/>
        Hello World5!<br/>
    </body>
</html>
```

Listing 2. Expanded Hello World template of Listing 1

not define such a symbol itself. However, it allows the user to do so by offering the `@VariableAntiQuotation` annotation which takes a String formatting pattern as argument. In the example, we could add an annotation like `@VariableAntiQuotation("#%s#")` to define that variables should be enclosed in `#` characters. In the example in Line 5, we would then need to write `#greeting#` to a access the *greeting* variable. In such a case, CommentTemplate could be extended to check if a variable is declared and report a missing declaration to the user (cf. Sect. 5.3). If `@VariableAntiQuotation` is not used, this kind of feedback can not be provided.

Second, CommentTemplate still relies on one problematic fixed symbol, which is `*/` to end a template fragment. To generate this symbol in the output (e.g., when generating Java code with comments), an additional feature is needed. Again, usually, a fixed escape symbol, to escape such symbols which are part of the template language itself, is offered. CommentTemplate makes this configurable by offering the `@ReplacementRule` annotation. This annotation takes two arguments, a *pattern* and a *replacement*, which allows the specification of a replacement for a certain String. For the problem described above, one can use `@ReplacementRule(pattern="#/", replacement="*/")`, which replaces all occurences of `#/` with `*/`. `#/` can then be used as an alternative for `*/`.

Both `@VariableAntiQuotation` and `@ReplacementRule` can be applied on the level of single `@CommentTemplate` methods but also on the level of classes. This allows a fine grained control of which escape characters are used where and helps to avoid syntax conflicts with the output syntax.

3 CommentTemplate Compiler

CommentTemplate is implemented as a Java-source-to-Java-source compiler using JaMoPP. Since JaMoPP handles Java source code as models based on an Ecore metamodel of Java, one could also look at the CommentTemplate compiler as a model-to-model transformation with Java as input and output metamodel. The implementation is realized in Java but could also be realized in a model transformation language such as QVT.

The following transformation is performed by the compiler for each method annotated with `@CommentTemplate`: An instantiation of a StringBuilder is added to the beginning of the method body and the method's return statements are

```java
public String helloWorld() {
    StringBuilder __content = new StringBuilder();
    String greeting = "Hello";
    __content.append("<html>\n");
    __content.append("\t<head><title>");
    __content.append(greeting.replace("\n", "\n\t"));
    __content.append(" world!</title></head>\n");
    __content.append("\t<body>");
    for (int i = 1; i <= 5; i++) {
        String greeted = "World" + i;
        __content.append("\n");
        __content.append("\t\t");
        __content.append(greeting.replace("\n","\n\t\t"));
        __content.append(" ");
        __content.append(greeted);
        __content.append("!<br/>");
        if (greeted.equals("World2")) {
            __content.append("\n");
            __content.append("\t\t");
            __content.append(greeted.replace("\n","\n\t\t"));
            __content.append(", you are the best!<br/>");
        }
    }
    __content.append("\n");
    __content.append("\t</body>\n");
    __content.append("</html>");
    return __content.toString();
}
```

Listing 3. The Compiled HelloWorld @CommentTemplate method of Listing 1

modified to return the content of that StringBuilder as String. Furthermore, all elements inside the method body are checked for multi-line comments.[2] This accessibility of comments is an important feature of JaMoPP exploited in the CommentTemplate realization, which is not offered by mechanisms such as plain Java reflection. If a comment is found, it is split into lines and for each line, a *StringReference* (Metaclass in the Java metamodel) is instantiated, which is filled with the text from the comment line. Each String that is generated this way, is passed to the StringBuilder by adding an *append()* call to the position in the method, where the comment was located before. For the example, the result of this transformation is shown in Listing 3.

An additional feature of the compiler is the indentation handling. It is a well known problem in code template development, that formatting of the template and formatting of the output are mixed. This basically concerns indentations, tab (`\t`) characters, at the beginning of a line. In the example of Listing 1, one can observe this issue. In Lines 3–19, one tab character is used to indent the method body. This is formatting of the template, but not of the output. The generated `<html>` (Line 4), for instance, should not be indented in the output (cp. Listing 2; Line 1). Additional indentations of blocks inside the method (e.g., for-loop in Lines 8–14) should also be ignored in the output. Therefore, the CommentTemplate compiler keeps track of indentation and corrects the indentation in each String line. This behavior was adopted from Xtend2 [5].

[2] In the Java metamodel, every metaclass is a subclass of *Commentable* which contains the comments that are located before the element in the source file as String.

4 Application and Discussion

One can look at CommentTemplate from two viewpoints. First, one can regard it as yet another code generation tool. Second, one can look at it as a Java language extension that is so slim, that it does not even extend the syntax of the language. With a critical look from these two viewpoints, two questions arise:

1. Why do we need another code generation tool if there are so many already?
2. Is a language extension that does not extend the Java syntax powerful enough to add any significant new functionality that is not offered by Java itself or can be added through a library alone?

4.1 CommentTemplate vs. Code-Generation Tools

We developed CommentTemplate out of a need we had ourselves, which was not met by existing code generation solutions. For us, the most important properties for a code generation tool were: (a) compilation of templates to Java such that we can run them as plain Java applications without requiring a template interpreter at template expansion time, (b) no dependencies in the template code to (possibly changing) runtime frameworks or libraries, (c) tight integration into Java, since this is the language we are working with most, (d) the ability to reasonably format both the output and the template directly (i.e., without using an additional formatting post-processor).

While many existing solutions meet one or more of these criteria, none of them satisfied us completely. Before we decided to develop CommentTemplate, we were going with Xtend2 [5], because it fulfilled criteria (a) and (d) as well as (c) to some degree.

A problem we had with Xtend2 was that it did not satisfy requirement (b) by providing a runtime library which was evolving with each version of Xtend2 over the last year. This made it difficult to develop and use two code generators, which were developed with different Xtend2 versions, side-by-side inside one Eclipse installation, since it is not possible to run two Xtend2 versions in the same Eclipse. This violated our requirement that the more than 700 Eclipse plugins of our open-source toolbox DropsBox [11], which are integrated by a Continuous Integration system and deployed on a single update-side, should all be able to run together in one Eclipse installation.

Regarding criteria (c) one still has to consider that Xtend2 is a separate language and not Java. Although, the barrier for Java developers to get started with Xtend2 is low, since it compiles to Java source code and provides tooling that is oriented at and integrated with the Eclipse Java Development Tools (JDT), developers still need some time to familiarize with the syntax and programming model of Xtend2. If a Java developer only needs a code generation feature, CommentTemplate provides this in the familiar Java syntax and tools. For CommentTemplate, the JDT Java editor can be reused directly with all its established features. No additional editor or other kind of UI tooling (e.g., buttons or menus) are needed, since the compiler runs automatically in the background when a Java file is saved. On the downside, CommentTemplate does not offer other additional languages features which are sometimes useful for code generation (cp. limitations of CommentTemplate discussed in Sect. 5).

Instead of developing CommentTemplate, we could have extended or patched an existing open source code generation solution. However, thanks to model-driven technology, in particular EMF and JaMoPP, CommentTemplate was developed in a very short period of time (the initial working version was written on one afternoon). This way, we obtained a solution that exactly met our requirements with little effort.[3]

This shows that Eclipse modeling and metamodeling technology can be used for rapid tool development in a quality that exceeds the level of prototypes.

4.2 CommentTemplate vs. Java libraries

Before we used a code generation tool, we performed code generation with Java directly. In fact, all templates (approx. 230) of our modeling tool EMFText [14] are written in plain Java. The code of these templates bears a likeness to compiled CommentTemplate code (cp. Listing 3). To make it better readable, we wrote a small Java library that helped us with the formatting. However, the main issue remains, which is that there is no way to write a block of output code as a String in Java, since Strings do not support multiple lines. This renders the templates difficult to write and read, because there is a lot of boilerplate between the lines.[4]

Consequently, *multi-line template fragments* is the feature that CommentTemplate provides, which a Java library alone can not provide. Although, we do not extend the Java syntax, we use an existing element of the syntax — multi-line comments — for this feature, by altering the semantics of this element. The advantage of avoiding syntax extensions is that the existing tooling can be used without adjustment. Such an adjustment, in case of the JDT, would be possible but complex, time consuming and error prone; let alone that other Java IDEs would also need adjustment.

Nevertheless, altering the syntax of an existing element can be difficult or even dangerous. For multi-line comments, however, this is not the case. From the compilers point of view, comments have no existing semantics. From a user's point of view, there is still the alternative of single-line comments (`//`) to use inside `@CommentTemplate` methods. Furthermore, outside of `@CommentTemplate` methods, no new semantics are given to multi-line comments.

[3] Currently, we use CommentTemplate for example to implement the generator of our Hibernate DSL HEDL [13] (which in turn is used in multiple customer projects) and for our HTML5/JavaScript based company website `http://www.devboost.de`.

[4] The minimal way to write multi-line Strings in Java is to use String concatenation with the plus (+) operator, which still requires opening and closing the String in each line with quotes (").

5 Limitiations and Future Work

As described in the previous section, we designed CommentTemplate as a minimal extension to Java that met our requirements for a code generation tool. Compared to other code generation languages, CommentTemplate misses certain features. Currently, we believe that these features are out of scope for CommentTemplate as we discuss in the following.

5.1 Expressions inside Template Code

First, CommentTemplate does not allow complex expressions inside the template fragments (multi-line comments). Only String variables can be referenced to inject content into the template. Many code generation languages have an escape mechanism, which allows the definition of expressions inside an anti-quotation.

In cases where the `@VariableAntiQuotation` annotation is used in CommentTemplate, we could also support arbitrary Java expressions. For this, we would need to extend the CommentTemplate compiler such that it takes the String from the anti-quoted part and parses it as an expression. Since most of the tooling for this is provided by JaMoPP already, implementing this would probably be possible with reasonable effort.

Currently however, we feel that this feature is not necessary. One can always define a new local variable and derive its value with an arbitrary expression. This forces the developer to keep the template fragments clean from complex calculations. A similar separation of data calculation and template code is also deliberately enforced by StringTemplate [8] with the same argument. However, future experiences with CommentTemplate might lead us to rethink this issue.

5.2 Support for closures/lambda expressions

Most of the mentioned code generation languages provide closure/lambda expression support. Many rely on OCL, or a variant of it, for this. This is convenient in code generation, for example to iterate collections or to map a collection of Objects to a collection of Strings.

We agree that this is a helpful language feature for code generation, but also for other tasks. Thus, we see the responsibility to provide such features at Java itself and they might indeed be provided in Java 8 [15]. Currently, Java classes that use CommentTemplate can be combined with other JVM-based languages which already provide such features such as Scala [16] or Xtend2 [5]. However, this does not allow to use features of these languages and CommentTemplate inside the same class. For this, the CommentTemplate would need to be ported to work directly on the source of these languages.

5.3 Future Development

In the future, we plan to add some additional Eclipse tool support for CommentTemplate. Although, the existing JDT tooling can be reused without extension, the usability of CommentTemplate can be improved. For instance, when anti-quotation is used, CommentTemplate can report errors about variables that do not exist directly in the source (currently these errors are only seen in the compiled version) and could offer code completion for variables inside templates. Since JaMoPP/EMFText do already offer infrastructure to add this kind functionality, we will constantly improve on it.

Furthermore, we plan to port the templates of EMFText, which are currently written in plain Java (cp. Sect. 4.2), to CommentTemplate. This gives us the possibility to further assess CommentTemplate on a collection of 230 templates.

6 Related Work

CommentTemplate was motivated by the functionalities and ideas behind existing code generation languages and language features such as JET, EGL, Acceleo, Xpand, Xtend2, MOFScript Velocity, StringTemplate or JSP [1–9]. The idea of compiling the templates to Java source code can be found in JET [1], Xtend2 [5] and JSP [9]. The separation of template and output formatting by a smart handling of tab characters was adopted from Xtend2 [5]. The limitation that CommentTemplate does not support expressions inside templates, can be seen as a strength which forces the user to separate model and view as publicized by StringTemplate [8]. Unique to CommentTemplate is its closeness to Java and its lightweightness reflected in its low number of new features added to Java and the fact that compiled templates consist of plain Java code without any dependencies despite Java itself.

7 Conclusion

This paper demonstrated CommentTemplate, a light-weight Java language extension for code generation. CommentTemplate is developed with Eclipse modeling technology on the basis of EMF and JaMoPP. It is an example of both a powerful, stable tool that is developed in short time thanks to Eclipse modeling technology and a tool that brings important concepts of model-driven development closer to the programming world.

Installation Instructions and Screencast

CommentTemplate can be installed from the DropsBox update-site available at `http://www.dropsbox.org/update_trunk` (category CommentTemplate).
A screencast of the CommentTemplate installation and usage is available at `http://www.dropsbox.org/CommentTemplate`

Acknowledgments

This work is supported by:

Gefördert durch:
Bundesministerium für Wirtschaft und Technologie
aufgrund eines Beschlusses des Deutschen Bundestages

eXIST Existenzgründungen aus der Wissenschaft

EUROPÄISCHE UNION

ESF Europäischer Sozialfonds für Deutschland

References

1. Eclipse Foundation: JET Project. www.eclipse.org/emft/projects/jet (April 2012)
2. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon Generation Language. In: Proc. of ECMDA-FA'08. Volume 5095 of LNCS., Springer (2008)
3. Eclipse Foundation: Acceleo Project. www.eclipse.org/acceleo (April 2012)
4. Eclipse Foundation: Xpand Project. www.eclipse.org/modeling/m2t/?project=xpand (April 2012)
5. Eclipse Foundation: Xtend Project. www.eclipse.org/xtend (April 2012)
6. Eclipse Foundation: MOFScript Project (April 2012)
7. Apache Software Foundation: Apache Velocity Project. velocity.apache.org (April 2012)
8. Parr, T.: StringTemplate. www.stringtemplate.org (April 2012)
9. Oracle: JavaServer Pages Technology. www.oracle.com/technetwork/java/javaee/jsp (April 2012)
10. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. of SLE'09. LNCS, Springer (March 2010)
11. DevBoost GmbH and Software Technology Group Dresden: The Dresden Open Software Toolbox (DropsBox). www.dropsbox.de (April 2012)
12. DevBoost GmbH: DevBoost Website). www.devboost.de (April 2012)
13. DevBoost GmbH and Software Technology Group Dresden: HEDL - Hibernate Entity Definition Language. www.emftext.org/language/hedl (April 2012)
14. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Proc. of ECMDA-FA'09. Volume 5562 of LNCS., Springer (June 2009) 114–129
15. Java Community Process: JSR 337: Java SE 8 Release Contents. www.oracle.com/technetwork/java/javaee/jsp (April 2012)
16. École Polytechnique Fédérale de Lausanne (EPFL): The Scala Programming Language. www.scala-lang.org (April 2012)

Model2Roo: Web Application Development based on the Eclipse Modeling Framework and Spring Roo

Juan Castrejón[1], Genoveva Vargas-Solar[2], and Rafael Lozano[3]

[1] Université de Grenoble, LIG-LAFMIA,
[2] Centre National de la Recherche Scientifique, LIG-LAFMIA
681 rue de la Passerelle, Saint Martin d'Hères, France
{Juan.Castrejon, Genoveva.Vargas}@imag.fr
[3] Instituto Tecnológico y de Estudios Superiores de Monterrey,
Campus Ciudad de México, Calle del Puente 222, México, México
ralozano@itesm.mx

Abstract. Inherent complexity in web application development is continually increasing, due to technical challenges, like new programming frameworks and tools. In this context, model-driven techniques can currently be used to guide the development of web systems, by focusing on different levels of modeling abstractions that encapsulate both implementation details and the definition of system requirements. This paper presents Model2Roo, a tool intended for Java web application development, that relies on the Eclipse Modeling Framework and on the Spring Roo project. In particular, this paper outlines key issues highlighted by previous users of the tool, and demonstrates recent implemented features.

1 Introduction

Web application development is one of the most evolving industries in software engineering [3]. However, this evolution also represents an increased complexity in functional and non-functional requirements associated to web applications [3]. In this environment, software engineers need to constantly evaluate new technical solutions for the development and maintenance of web applications, including a wide variety of programming languages, frameworks and tools.

To help overcome this complexity, the Model2Roo [1] project was presented in early 2011. Model2Roo uses a model-driven approach, in particular Model to Text (M2T) techniques, to transform UML and Ecore [10] class diagrams into appropriate Spring Roo commands [9], which can then be used to generate a full-blown Java web application. The generated applications are automatically built with a set of architecture patterns and industry best practices [9], and contain not only the static structure of the system, but also comprehensive support for the functionality associated to the Spring Framework module[4]. Model2Roo has also been available as an open-source tool since early 2011[5].

[4] http://www.springsource.org/
[5] http://code.google.com/p/model2roo/

This paper describes related work (Section 2), the main issues encountered by previous users of the tool (Section 3), demonstrates implemented features (Section 4), and finally, outlines conclusions and future work (Section 5).

2 Related work

Web application development based on model-driven techniques is a widely researched topic in software engineering. Representative results of this research include the Web Modeling Language (WebML) [2] and the UWE approach [5]. Both projects provide tool support to generate fully functional web applications, including not only presentation content, but also complex navigation features.

In comparison, tool support for Model2Roo is not yet as complete and mature as the one provided by similar projects, however the main advantage of our tool is that it provides an association to modern application development tools, by means of the Spring Roo project. As a result, developers can easily access the full potential of the Spring framework, and associated projects.

3 Previous tool issues

Three main issues have consistently been highlighted by previous users of the tool: (i) Insufficient support for graphical environments; (ii) Basic edition of Spring Roo properties; and, (iii) Troublesome installation procedure.

The first issue, support for graphical environments, was mainly associated to the use of UML tools within the Eclipse Modeling Framework (EMF). In this case, though Model2Roo could generate Spring Roo commands either from Ecore and UML files, users found it troublesome to generate these files with the basic *UML* and *Ecore Model Editors* [10], and wanted to use more complex tools, such as Papyrus [8]. However, when trying to use Papyrus to generate UML class diagrams, several incompatibilities with our tool were discovered, such as inappropiate use of numeric data types.

Model2Roo allows the specification of Spring Roo properties through *Ecore Annotations* and *UML Profiles*. However, in previous versions of the tool, the value for each property could only be set as free text. This led to subtle errors, such as assigning a *String* value to a property that required a *numeric* value, but more importantly, users that were not familiar with Spring Roo, did not know the possible domains for each property. For example, consider the property *debugLevel*, that configures the level of application debug traces. In previous versions of the tool, users had to know that this particular property could only be assigned one of the following values: *Debug, Info, Warn, Error* and *Fatal*.

The third issue highlighted by the users, was a troublesome installation of the tool. In this regard, though Model2Roo was distributed as a single plugin file, the main issue was the installation of the required dependencies, in particular, users were required to manually install the ATL [4] and Papyrus distributions.

These three main issues have been fixed in the last version of the Model2Roo tool. The details will be demonstrated in the following section.

4 Features demonstration

This section demonstrates the Model2Roo features, using the Spring PetClinic sample application[6]. This system is intended for clinic employees who need to view and manage information regarding veterinarians, clients, and pets.

A screencast detailing this demonstration is available in the project web site[7]. To try this demonstration locally, install Spring Roo, an Eclipse Modeling distribution[8], and finally, Model2Roo, using the project update site[9].

It should be noted that the update site already references all of the features that are required to execute Model2Roo, both over Ecore and UML projects.

To generate the test application, we must define a class diagram containing the system *Classes* and *Enumerations*, as well as their *properties* and the *associations* between them. In this case, we are going to use the Papyrus project, applying the *rooCommand* and *rooStructure* UML profiles, which are part of the Model2Roo installation. Fig. 1 depicts a fragment of the resulting class diagram.

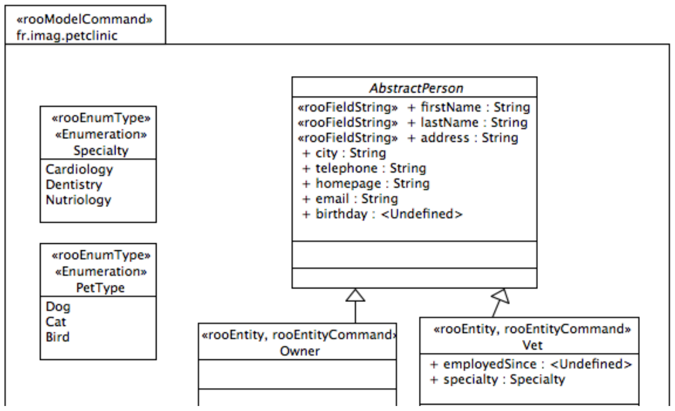
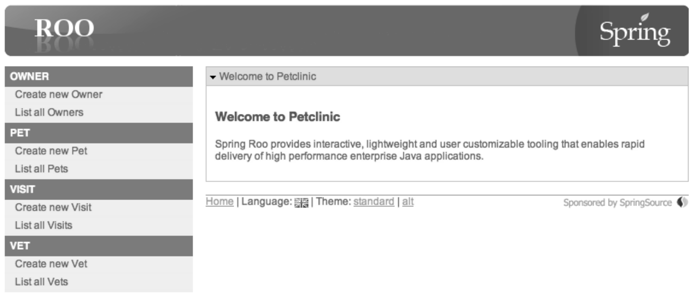


Fig. 1. Fragment of the PetClinic class diagram.

As opposed to previous versions of Model2Roo, the current version fully supports Papyrus profiles. As a consequence, property values can now be specified using drop-down menus. Also, in the current version of Model2Roo, the Acceleo project [6] is used to generate the Spring Roo commands that correspond to UML diagrams, as opposed to ATL queries in previous versions. This change was motivated to increase maintainability, considering that Acceleo provides an implementation of the MOF Model to Text Language (MTL) standard [7].

Finally, the generated commands can be executed by the Spring Roo console, in order to create the corresponding Java web application, as depicted in Fig. 2.

[6] http://static.springsource.org/docs/petclinic.html
[7] http://code.google.com/p/model2roo/
[8] http://eclipse.org/downloads/packages/eclipse-modeling-tools/indigosr1
[9] http://model2roo.googlecode.com/svn/trunk/fr.imag.model2roo.update.site/



Welcome to Petclinic

Spring Roo provides interactive, lightweight and user customizable tooling that enables rapid delivery of high performance enterprise Java applications.

Fig. 2. Generated web application.

5 Conclusions and Future work

This paper described Model2Roo, a tool for web application development that relies on UML and Ecore class diagrams, in order to generate Spring Roo commands. The main issues highlighted by previous users were discussed, as well as recent implemented features. Support for the full gamut of Spring Roo commands is intended for future work.

Acknowledgments

This work is funded by the French National Research Agency, through the UBIQUEST project (http://ubiquest.imag.fr) ANR-09-BLAN-0131-01, and by the STIC-AMSUD program, within the CLEVER project (http://clever.imag.fr).

References

1. Castrejón, J., López-Landa, R., Lozano, R.: Model2Roo: A Model Driven Approach for Web Application Development based on the Eclipse Modeling Framework and Spring Roo. In: Electrical Communications and Computers (CONIELECOMP), 2011 21st International Conference on. pp. 82 –87 (March 2011)
2. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann, San Francisco, CA (2003)
3. Jazayeri, M.: Some Trends in Web Application Development. In: Future of Software Engineering, 2007. FOSE '07. pp. 199 –213 (May 2007)
4. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (June 2008)
5. Kraus, A., Knapp, A., Koch, N.: Model-Driven Generation of Web Applications in UWE. In: 3rd International Workshop on Model-Driven Web Engineering (2007)
6. Obeo: Acceleo. http://www.eclipse.org/acceleo/ (November 2011)
7. Object-Management-Group: MOF Model to Text Transformation Language, v1.0. http://www.omg.org/spec/MOFM2T/1.0/ (January 2008)
8. Papyrus: Papyrus. http://www.eclipse.org/modeling/mdt/papyrus/ (April 2012)
9. SpringSource: Spring Roo. http://www.springsource.org/spring-roo/ (May 2012)
10. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley Professional, Boston, Massachusetts, 2nd edn. (2008)

The Fourth Workshop on Behaviour Modelling - Foundations and Applications

Ella Roubtsova[1],
Ashley McNeile[2],
Ekkart Kindler[3],
Mehmet Aksit[4]

[1] Open University of the Netherlands,
ella.roubtsova@ou.nl
[2] Metamaxim Ltd,UK,
ashley.mcneile@metamaxim.com
[3] Technical University of Denmark,
eki@imm.dtu.dk
[4] TU Twente, the Netherlands,
aksit@ewi.utwente.nl

Dynamics of changes and trends of todays systems show the growing role of behaviour modelling in system life cycle. Growing variety of E-businesses: E-commerce, E-logistics, E-procurement, E-government and collaborative services result in the consequent emphasis on well defined interaction between software components, importance of interfaces, contracts and service level agreements in defining and managing behavioural system integration both within and across organizational boundaries. New and new processes are covered with services accumulating business intelligence to work with the Cloud.

Does the changes in software system size and architecture influence the requirements for the behaviour modelling paradigms? Are there elements of system behaviour that we need to model but cannot model easily? What kind of analysis has to be fulfilled on models? These and other related questions are discussed at the Fourth International Workshop on Behaviour Modelling - Foundations and Applications.

After the evaluation of existing modelling paradigms during the three previous workshops the Fourth Workshop is aimed to analyse the requirements for the behaviour modelling paradigms formulated by the new trends in system design and organization. The workshop will focus its discussion on the need of design of interactive systems of independently maintained services and has an aim to formulate requirement patterns for the behaviour modelling techniques used in different domains of behaviour modelling.

Organizing Committee

Program Committee

Table of Contents

Defining and Verifying Behaviour of Domain Specific Language with fUML

Qinan Lai and Andy Carpenter

School of Computer Science, the University of Manchester
laiq@cs.man.ac.uk, Andy.Carpenter@manchester.ac.uk

Abstract. The behavioural semantics of a Domain Specific Language (DSL) are the instructions on how to execute the language. In practice, such semantics are often documented by text, which leads to ambiguity and tool generation problems. Although some formal frameworks have been proposed to address these drawbacks, they only allow the correctness of a specification to be tested at a later stage, usually when the semantics are implemented. This paper presents a new framework for implementing the behavioural semantics of meta-model based DSLs and tools. The framework uses the foundational subset of executable UML (fUML) as its semantic base, and uses the fUML meta-model for modelling the abstract syntax and operational semantics of a DSL. The semantics specification can be verified at design time without the need to execute behaviour models. Thus, it can provide useful feedback to the DSL designer. The framework is demonstrated in a Petri-net example.

1 Introduction

In Model-Driven Development, it is common to define a Domain Specific Language (DSL) for an application domain. A DSL definition consists of four parts [11]. The first one is abstract syntax, which defines the concepts of the language; it is often defined as a meta-model. The second part is concrete syntax, which defines the notations for representing these concepts. The notations could be developed with plenty of tools, such as Graphiti[1] for graphical syntax and Xtext [8] for textual syntax. The third part is static semantics. It adds further well-formedness rules to the language concepts; this can be achieved by using OCL constraints. The last part is behavioural semantics that defines how to execute a DSL program. Unfortunately, there is no universally agreed way to define the behavioural semantics. It is still unclear how to produce a high quality specification of behavioural semantics for a DSL [4].

Two challenges must be solved when defining the behavioural semantics of a DSL. The first challenge is that it is difficult to describe the behavioural semantics [4]. Informal descriptions lead to ambiguous and non-analysable specification. On the other hand, existing formal approaches have still not gained wide support. One of the possible reason is they are hard to learn. The other challenge is how to maximise the qual-

[1] http://www.eclipse.org/graphiti/

ity of the semantics specification, or in other words, minimise the defects in the specification [23].

Although a number of approaches to defining the behavioural semantics have been proposed, none of them addresses both of these challenges. For example, translational semantic and formal semantic frameworks require the DSL designers to combine an understanding of technological spaces with mathematical knowledge; but they can provide a better tool chain for verification. Whereas, weaving behaviour approaches uses an action language for capturing the behavioural semantics, which is easier to learn and use, but there is a lack of research on how to verify properties of the language.

In this paper, a new framework for defining DSLs is presented, with particular emphasis on behavioural semantics. Our framework aims to address both of the highlighted issues; firstly, the approach applies an intuitive and widely used notation for expressing behavioural semantics, thus making it easier to understand for both language designers and domain experts. Secondly, the framework allows language design-time validation of the DSL, which can detect and help fix anomalies in semantics specification. This validation includes static verification and integrated formal verification.

Our approach uses the newly published OMG standard foundational subset of executable UML (fUML) [13] and action language for fUML (ALF) [12] as the definition language. Due to its adoption of UML graphical syntax and ALF scripts, the method only demands limited prerequisite knowledge. It is also possible to integrate existing methods for verifying UML behavioural models; verification at design time helps to assure the quality of the semantics specification. In addition, how to add additional static verification rules to our framework is demonstrated. Finally, a fUML executor can take the language specification and a DSL program (an instance model) as inputs; whether the behaviours of the DSL are expected can be tested by reading the outputs of the executor.

The contributions of the paper are listed below. Firstly, we proposed a new way for defining both the syntax and semantics aspects of a DSL by OMG standards. Compared to existing weaving behaviour approaches such as Kermeta [3] and XOCL[5], our approach support a richer vocabulary, including parallel, signal send/receive and sequence expressions. Secondly, in the framework, the ALF code editor, the transformation from ALF code to fUML model, and the model merger that combines the behavioural aspects and structural aspects are implemented by us. Among them, the transformation is the first one that can translate most of the concepts defined in ALF standard.

Paper organisation. The second section summarises the work related to the two challenges of behavioural semantics development. The third section describes how our fUML-based framework addresses both these challenges. Furthermore, the technical details of the framework are discussed in this section. Section 4 demonstrates our framework by specifying and verifying the behavioural semantics of a Petri net language. The last section concludes the paper with discussion of further work.

2 Related works

This section discusses existing approaches to defining the behavioural semantics of DSLs and minimising defects in these specifications.

2.1 Behavioural Semantics Specification

A formally specification of the behavioural semantics of a language helps avoid ambiguities between language designers and users; it also makes automatic tool generation possible [4]. Existing approaches can be categorised as translational approaches and operational approaches.

Translational approaches map DSL behavioural concepts to a language that is intended for the definition of semantics (a semantic domain); semantic domains include Abstract State Machine [9], Alloy [16] and Maude [15]. The advantage of this approach is that existing tools associated with the semantic domain such as simulators, verifiers and compilers can be used; disadvantages of translational approaches include that the DSL developer must be an expert in both the application and semantic domains. It is also hard to define the mapping between the two domains. In comparison, our approach does not require DSL developers to understand a semantic domain or a mapping to a semantic domain.

Operational approaches derive from the idea of Plotkin's structural operational semantics [18]; with behavioural semantics defined as transition rules between system states. The original notation was a mathematical language that was not easy to learn or execute on a platform. Later works use graph transformation [7], general model-to-model transformation language [25] or an action language (also known as weaving behaviour approach) for describing the transition rules.

Researchers are using graph transformation for building the operational semantics of DSLs, for example the Dynamic Meta Modelling (DMM) [14] approach. When compared to translational approaches, graph transformation provides an intuitive mechanism for capturing operational semantics; additionally, its mathematical foundation makes it possible to analyse certain properties of descriptions. Although graph transformation represents a promising approach, it does have limitations [19] in particular learning curve for language designers, scalability (dealing with large models), maturity, and tool support. Our approach provides maintains the intuitive interface of the graphical transformation approaches while combining this with a better tool chain support.

Weaving behaviour approaches use an action language to describe operational semantics. Consequently, their usability depends on the capabilities of the action language. For instance, Kermeta [3], XOCL [5], and [24] use bespoke action languages that are unfamiliar to DSL developers and on which limited work has been done on design time verification of the semantics that they capture. In comparison, our work uses OMG standard language with which developers should be more familiar. Additionally, our chosen language includes more abstract concepts such as parallel block and signal sending/receiving which avoids the need to compose these from similar primitives.

Scheidgen and Fischer [20] use an UML activity diagram like notation for capturing behavioural semantics. However, their notation is only UML-like and they use their own implementation of CMOF of the defining the meta-model, the result is that their work does not integrate easily into existing tool chains. Furthermore, their work does not highlight any further work for verification. Compared to them, our approach provided better tool support and support more concepts in UML. Thus, it offers the prospect of a seamless tool chain for DSL development that combines the syntax and semantics defined together with verification to detect anomalies in the language definition.

2.2 Minimising defects in the Behavioural Semantics Specification

In section 2.1, the benefits of having a behavioural semantics specification were summarised. However, they can only be achieved when the specification is correct. In fact, some formal semantics specification, such as the Specification and Description Language (SDL), contains a large amount of errors [10]. While minimising defects is an important issue in developing DSLs, limited work has been done on the checking of semantics specification. Soltenborn and Engels [23] propose the application of Test-Driven Development (TDD) principles. This approach requires the execution of DSL programs; thus, its coverage is dependent on the tests used. Combemale et. al. [6] illustrated an idea to develop reference semantics for the original semantics, thus the DSL has one translational and one operational semantics. Using different approaches means that misunderstanding can be identified.

These approaches are methodologies for assuring the quality of semantics specification and have the potential for fulfilling their aim. However, they are not sufficiently simple to ensure that this would be the case when they are applied in a practical situation.

In practice, static verification is widely applied in developing software using general-purpose languages. Methods such as type system, dataflow analysis, model checking, logic based approaches and approaches based on pattern matching can detect potential or actual errors, enforce the developers to apply good programming style, and report bad practice [21]. Weaving behaviour approaches are criticised for their similarity to applying general-purpose language. On the other hand, it can be seen as a benefit, because known approaches in general-purpose languages can be adapted to use. To our knowledge, no work has provided a lightweight static verification tool for DSL behavioural semantic specification in the way that we propose.

3 DSL Behavioural Semantics as Behavioural Models

Whether creating a DSL from scratch or formalising an existing one, applying a Model-Driven approach makes developing DSL and its tools simpler than traditional approaches based around manipulating an abstract syntax tree. The abstract syntax of a DSL is captured as a meta-model. The languages for meta-modelling such as MOF or Ecore reuse the concepts and notations of UML class diagrams. Furthermore, UML is not only a language for modelling structural aspects; it also provides languages for behaviour modelling. However, UML behaviour diagrams are a family of modelling languages; their semantics are defined as text and many semantic variation points still remain. Because different UML CASE tools provide different execution semantics of UML activities, UML activity diagrams and action scripts are not compatible. To capture the behavioural semantics of a DSL using UML would result in a language valid for one particular UML dialect. This makes using UML for behavioural semantics modelling the same complexity as using a special modelling framework such as Kermeta or XOCL.

3.1 fUML and ALF

The new OMG standard fUML (and its action language, ALF), which is almost finalised, is intended to give UML formal execution semantics. It defines a basic subset (bUML) and its axiomic semantics using Process Specification Language (PSL) [2]. The semantics of bUML are first-order logic axioms that constrain bUML concepts to the process concepts defined in PSL. These basic models serve as vocabulary for defining the operational semantics of a larger subset of UML activities and actions. Therefore, fUML execution model is a model of itself; just like a meta-meta-model that could be used for define behaviour models. fUML specification gives execution semantics to three packages: `CommonBehaviors`, `Activities` and `Actions`. Although some behaviours existed in UML are excluded from fUML, it still provides behaviour models in a platform-independent and abstract way.

The ALF language can be seen as a textual notation of fUML. Its primary goal is for specifying executable behaviours that are not amenable to the UML graphical notation, or there is no nominated graphical notation (e.g. LoopNode) for the concept. Basic syntax of ALF is a Java-like syntax that is familiar to behaviour modelling community. It also supports the syntax of UML such as colon for type and double colon for qualified names. When dealing with sequences, ALF support OCL-like syntax, such as `select/reject/collect` [12]. It does not have any naming constraints; names could be declared as a string that contains any possible character. Therefore, the UML graphical model does not need to change any names. ALF syntax is easy to understand and learn when users have done some UML/OCL/Java beforehand. The execution semantics of ALF is defined by translating ALF meta-concepts to fUML, ALF code could be compiled to fUML; hence, semantically ALF code has no difference between fUML behaviour models. The concept of ALF code and fUML model will be treated as the same concept in the following part.

fUML has shown its ability to define the semantics of general purpose languages by defining its own behaviour. Hence, as a DSL usually has a simpler syntax and semantics than a general purpose modelling language, it is possible to use fUML to model a complete language definition [22].

A complete language specification based on fUML can gain these benefits: (1) well-known benefits of model-driven development can be achieved because the language specification is fully model-based; we can reuse, translate, validate these models or generate code from it. (2) fUML has execution semantics, which makes verification at design time possible. Language designers can test their models as early as creating them rather than finding design errors when implementing a compiler or interpreter. (3) Unlike other action language based approaches, much research has been done in UML behaviour modelling, including theory on how to verify particular property or tools for composing models. Existing knowledge can be adapted to fUML with a small effort, because a fUML model is still a valid UML model. (4) fUML models can be represented by the UML graphical syntax, which is intuitive for the users (domain experts) to understand.

3.2 Framework overview

To gain the benefits mentioned before, our framework (**Fig. 1**): works like this

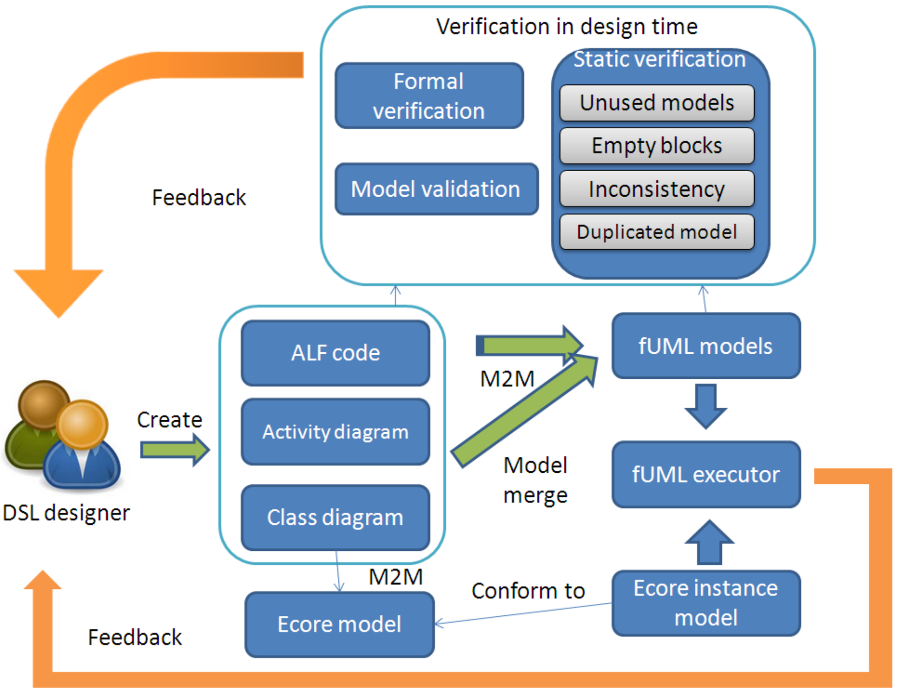


Fig. 1. Overview of fUML behavior semantics framework

1. The language designer creates the language specification using any UML CASE tool that supports UML2. The abstract syntax is modelled as a class diagram and the behaviours are modelled as activity diagrams. Complex operations that are cumbersome to create using the graphical syntax can be written in ALF.
2. The abstract syntax (class diagram) can be imported, as an Ecore model, into the Eclipse Modeling Framework (EMF), where tools, such as Xtext or Graphiti, can be used to create the concrete syntax of the DSL.

3. ALF code is transformed to fUML behaviour model through an ATL model-to-model transformation. This transformed model and the abstract syntax model are then woven together using a simple model merger.
4. Currently the framework can perform static verification (such as [17] for executability, and we proposed to implement our own static verifier later) in the fUML models. Feedback helps to reduce design error as early as possible.
5. The above steps support the DSL developer in defining their language and creating the tooling needed to support the execution of a DSL program. An end-user's DSL program is transformed into a model that becomes instance data for the behavioural model executed by the fUML execution engine.

To be clear, the complete framework has not been finished yet. In the next section, details of the implementations are introduced.

3.3 Compilative Execution of ALF

ALF standard [12] defines three types of Semantic Conformance of ALF: Interpretive Execution, Compilative Execution, and Translational Execution. Interpretive execution means that the executor directly interprets ALF code. Compilative Execution transforms ALF code to fUML models which are then executed. Translational Execution translates ALF code to another language which is then run to execute it. As our framework allows a combination of textual ALF and graphical UML syntax, Compilative Execution is used. Unfortunately, no tool is currently available that can editing of ALF code and its compilation to fUML. To give the DSL designers the ability to use ALF as an action language for describing semantics, such a tool is implemented as part of our tool chain.

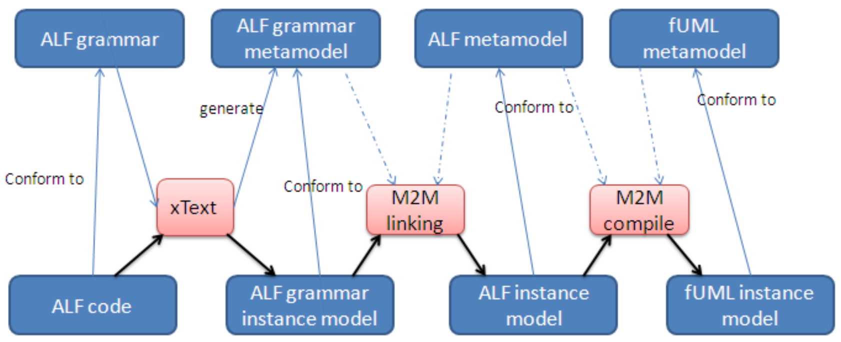


Fig. 2. Transformation from ALF to fUML

Xtext is used to create an ALF code editor. As Xtext is part of EMF, the process shown in **Fig. 2** can be used to create an equivalent fUML model.

To ensure that the ALF grammar is not LL recursive, the Ecore model generated by Xtext is very verbose. Moreover, the mapping between the grammar model and fUML is not specified. The second transformation transforms a grammar instance model to an instance of the ALF meta - model. In this process, verbose structures in the grammar model are simplified. This is equivalent to the difference between a parsed tree and an abstract syntax tree in programming language development.

To illustrate the process, here is an example.

```
activity test(){
  startA();
  //@parallel
  {
    startB();
    startC();
  }
}
```

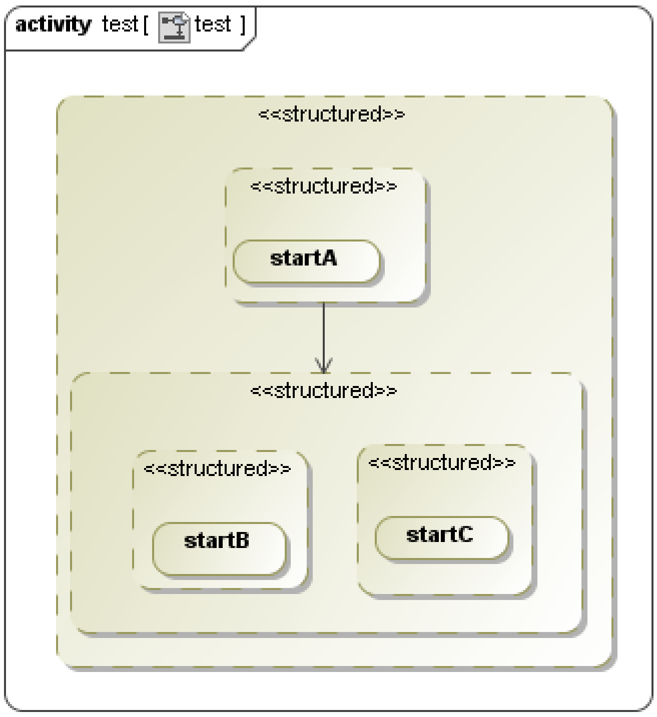


Fig. 3. Result of the example

Assume that `startA`, `startB`, and `startC` are pre-defined by UML graphical syntax. The activity test firstly calls the activity `startA`. Then `startB` and `startC` are started in parallel. In the example only `InvocationExpression`, `Statement` and `AnnotatedStatement` are included. Therefore, the Grammar meta model and ALF meta-model are similar. Due to the meta model are too large, they are not included in the paper.

The last transformation transforms ALF models to fUML models. The mapping rules are specified in ALF standard. The resulted fUML model is represented in **Fig. 3**. An ALF statement maps to a `StructuredActivityNode`, and the

InvocationExpression is mapped depends on what is really invocated. In our case it is mapped to an CallBehaviorAction in fUML. A normal block is mapped as each of the statements become a StructuredActivityNode, and each of them has a ControlFlow between them. A parallel block is the same, only without the ControlFlow between them.

Finally, the models in **Fig. 3** is merged to the models defined as UML diagrams. In this stage, the internal references are resolved.

3.4 Verification at Design Time

When defining a behavioural semantics specification, incorrect behaviours and potential design anomalies should be identified as early as possible. Leaving these errors to implementation will lead to higher costs than correct them at design time. Some certain errors such as deadlock can be identified by model checking or data flow analysis. Another widely used way for static verification is to define an error pattern library; the verifier can check whether these patterns exist in the target code. This lightweight approach is applied by many static code analysers such as Findbug[2] and PMD[3]. Apart from the built-in rules, these tools also support to customise the pattern database by adding new rules that are defined as XPath or Java code. Compared to them, the targets of our framework are models, not text, thus the languages for defining rules for text are not easy to apply.

Seifert and Samlaus [21] proposed to use OCL for defining verification rules. It is suitable for any language that has a meta-model, which is appropriate for our fUML models. In addition, OCL is a standard technology in model-driven development, there is no need for the developers to learn a new language. In their approach, OCL is used to query models and add constraints to models; when a rule is violated, the generated code editor would report it.

Unused activities. In fUML, the only behaviour that users can customise is Activity. The Semantics specification is formed with a set of activities. If one activity is defined but is never called in other activities, this activity is an unused activity. Unused activities should be removed when they are redundant. Another possibility is that the semantics is deficient, additional activities should be added.

The following ATL helper checks whether an activity is used or not. It returns false when there is no CallBehaviorAction calls the activity.

```
helper context UML!Activity def: isUsedActivity(): Boolean =
  UML!CallBehaviorAction.allInstances()
    ->exists(e|e.behavior=self);
```

[2] http://findbugs.sourceforge.net/
[3] http://pmd.sourceforge.net/

Unused class members. Similarly, when a property of a class is declared, but is never used in any of the behaviour models, it is defined as an unused member. When an unused member appears, it is possible that either the member is redundant, or there is a deficiency in the behavioural semantics.

To check whether a property is an unused member, an ATL helper can be defined as:

```
helper context UML!Property def: isUsedProperty(): Boolean =
   UML!ReadStructuralFeatureAction.allInstances()
      ->exists(e|e.structuralFeature=self)
   or UML!AddStructuralFeatureValueAction.allInstances()
      ->exists(e|e.structuralFeature=self);
```

This helper reports an unused member when there is no action that read or modifies the class member.

The plan of the static verifier is to establish a library of suspicious models and bad practices. As the library grows larger, the verifier can report a huge amount of warning and errors. Thus, an algorithm for controlling false positive is also needed to explore. The next step of the verification framework is applying formal verification technology. This could be done by translating the DSL specification to an input language of a model checker. For example, [1] proposed a method to translate fUML models to process algebraic specification language and check the models are deadlock free. Due to the well established research on UML, other ways of verification, such as verifying executability and data-flow analysis also exists. Unfortunately, this has not been integrated with the framework. What are the expected features are still under research.

4 Petri-Net Example

In the previous sections, the technology basis is introduced. How the proposed framework could deal with specification and verification challenge is illustrated. In this section, the abstract syntax and behavioural semantics of a Petri net language is specified using our framework, then how to verify bad practices in the models and execute instance models in the executor are demonstrated.

4.1 Petri net language specification

The meta-model of the DSL is firstly developed. **Fig. 4** shows the meta-model of a Petri net language. A `Net` class contains a set of `places` and `transitions`, each transition has access to its source places (`src`) and target places (`trg`). Associations between `Transition` and `Place` is bi-directional, so a Place also has access to its sources and targets. The property `token` is an integer number, once all sources of a `Place` have a token greater than 0, this `Place` is active.

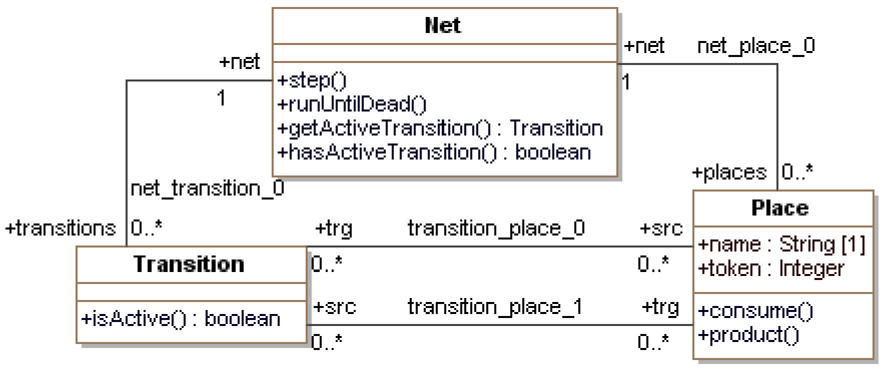


Fig. 4. Petri net meta-model and behaviour semantics definition

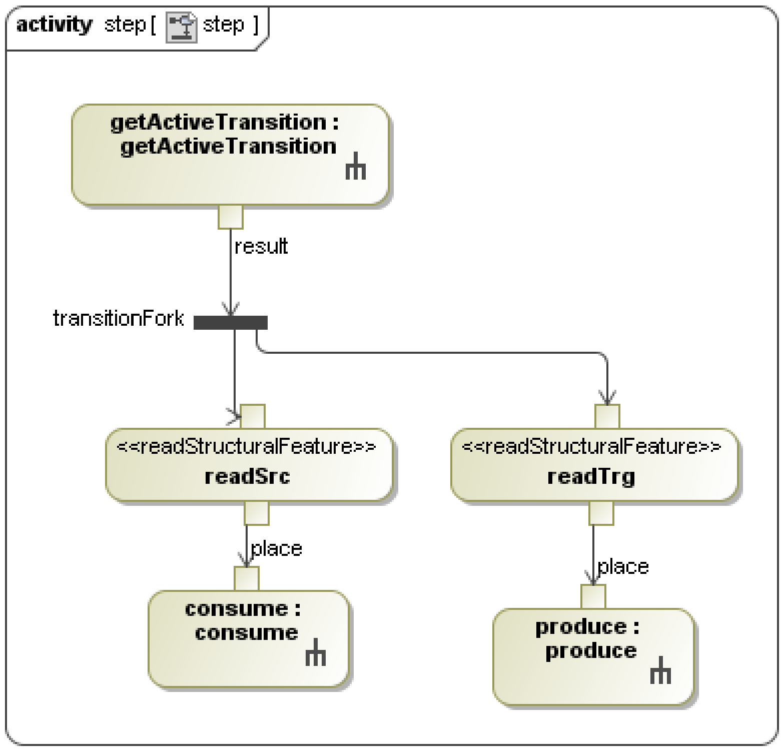


Fig. 5. Activity of step

The necessary operations are also declared in the UML class diagram. It is possible to use a general-purpose language like Java to specify the body of the operation. However, general-purpose language can contain too much detail or not sufficient for

UML concepts (like associations, multiplicity, and concurrency). In addition, a graphical language is preferable if the reader is a domain expert – not a programmer. Therefore, the DSL developer can select whether to use graphical or textual syntax of fUML by themselves. For example, **Fig. 5** presents the graphical part of the behavioural semantics. The activity step firstly finds an active transition (of which all its source places has a token property that is greater than 0; this done by activity `selectActiveTransition`), and then makes all the token of source places decrease by one (done by activity `consume`) and all its target places' token increase by one (done by activity `produce`).

The other activities are all presented as ALF programs (see **Fig. 6**). The activities are the body of the operations defined in the class diagram. Because they are defined in separate tools, to be convenient, the class name of the operation is added before each activity. When using the merger, the activities will be assigned as the method of the operations that has the same class name and the same operation name. Activity `runUntilDead`, `isActiveTransition`, `consume`, `produce` and `getActiveTransition` are modelled as ALF script rather than graph diagram. As mentioned before, the graphical notation tends to be very verbose for complex behaviour; equivalent representation in a diagram of OCL-like expressions in ALF, such as `select` and `forAll`, has to be mapped to an expansion region with all its condition expressions inside that region. Diagrams are preferred when the diagram is intuitive, but if the diagram is more verbose than a textual representation, and then textual is preferred.

The activity `runUntilDead` use a while loop. It repeats to do the `step` activity when there is an active transition. The `isActiveTransition` returns true if all tokens of source places are greater than zero. The activity `selectActiveTransition` returns the first active transition because the semantics of this Petri net is simplified by always firing the first active transition. The activities `consume` and `produce` modify the token of a fired place.

```
@class=Net
activity runUntilDead(){
  while (transitions->exists e (e.isActive())){
    step();
  }
}
@class=Net
activity getActiveTransition():Transition{
  return transitions ->select e (e.isActive()).at(1);
}
@class=Place
activity product(){
  token = token + 1;
}
@class=Place
```

```
activity consume(){
  token = token - 1;
}
@class=Transition
activity isActive(): boolean{
  return src->forAll e (e.token>0);
}
```

Fig. 6. ALF code

4.2 Reduce Anomalies in Petri net behavioural semantics specification

While creating the behavioural semantics, it is possible that developers made some flaws in the specification. Without static verification and test in design time, these flaws could be left until implementation. By using the static verifier, simple deficiency and redundant can be detected. For example, in the Petri net example, the verifier would identify that the activity `runUntilDead` is never used, because it is the main activity that executes the net. The verifier also reported that properties `transition_place_0`, `net` in class `Place`, and `transition_place_1`, `net` in class `Transition` are never used. This is because the current fUML standard requires an association to be bi-directional, these properties are opposite properties of used properties.

The logic errors can be identified by testing. For example, the Petri net model can be created by the Ecore instance model editor, then the behavioural semantics specification (fUML models) and the instance are loaded to fUML executor, and whether the result is expected can be checked.

Now it is possible to relate the example to the overview of the framework defined in section 3.2. The definition of the DSL responds to the first step mentioned. The transformation from ALF code to fUML and merged with fUML models (in the example, the `step()` activity) is step three. The verification relates to step 4 and 5.

5 Conclusion and Further work

It is a challenging work to develop a high quality behavioural semantics specification for a DSL. In this paper, a new framework for defining behavioural semantics using the newly published OMG standard fUML is introduced. Tools for supporting the framework are developed, including an ALF code editor, a transformation from ALF code to fUML model and a rule-based static verifier. The reference implementation of fUML is modified to be able to execute an Ecore instance model of a DSL, whose Ecore model is derived from fUML models.

Two contributions are highlighted here. Firstly, it is true that using UML activity diagrams or similar notation for capturing behavioural semantics is not a new idea; nevertheless, this paper is the first one that uses all OMG standard-based technology for doing that. Secondly, we identified that the existing work for UML behaviour model verification can be adapted to help to minimise the errors in a semantics specification. Currently the framework only supports to report some structural errors, for example, unused models and empty models. General model attributes, such as checking deadlocks, reachability and other properties requiring model checking are left as future works of this project.

References

[1] Islam Abdelhalim, James Sharp, Steve Schneider, and Helen Treharne. Formal verification of tokeneer behaviours modelled in fuml using csp. In Jin Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 371–387. Springer Berlin / Heidelberg, 2010.
[2] Conrad Bock and Michael Gruninger. Psl: A semantic domain for flow models. *Software and Systems Modeling*, 4:209–231, 2005.
[3] Lionel Briand, Clay Williams, Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. *Weaving Executability into Object-Oriented Meta-languages*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer Berlin / Heidelberg, 2005.
[4] B.R. Bryant, J. Gray, M. Mernik, P.J. Clarke, R.B. France, and G. Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems*, 2011 OnLine-First(00):–, 2011.
[5] Tony Clark, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language driven development, second edition, 2008.
[6] Benot Combemale, Xavier Crégut, Pierre-Loc Garoche, and Xavier Thirioux. Ësssay on semantics definition in mde - an instrumented approach for model verification. *Journal of Software*, 4(9):943–958, 2009.
[7] Juan de Lara and Hans Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages & Computing*, 15(3-4):309–330, 2004.
[8] S. Efftinge and M. Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.
[9] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages. *Automated Software Engineering*, 16(3):415–454, 2009.
[10] U. Glässer, R. Gotzhein, and A. Prinz. The formal semantics of sdl-2000: Status and perspectives. *Computer Networks*, 42(3):343 – 358, 2003.
[11] R.C. Gronback. *Eclipse modeling project: a domain-specific language toolkit*. The Eclipse series. Addison-Wesley, 2009.
[12] Object Management Group. Action language for foundational uml (alf) 1.0 - beta 1. www.omg.org/spec/ALF/, 2010.
[13] Object Management Group. Semantics of a foundational subset for executable uml models (fuml), version 1.0. http://www.omg.org/spec/FUML/1.0/, 2011.
[14] JH Hausmann. *Dynamic meta modeling: a semantics description technique for visual modeling languages*. PhD thesis, Universität Paderborn, Germany, 2005.

[15] Peter Őlveczky, José Rivera, Francisco Durán, and Antonio Vallecillo. *On the Behavioral Semantics of Real-Time Domain Specific Visual Languages*, volume 6381 of *Lecture Notes in Computer Science*, pages 174–190. Springer Berlin / Heidelberg, 2010.

[16] Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 690–704. Springer Berlin / Heidelberg, 2008.

[17] Elena Planas, Jordi Cabot, and Cristina Gomez. Lightweight verification of executable models. In *30th International Conference on Conceptual Modeling (ER 2011)*, 2011.

[18] G. D. Plotkin. A structural approach to operational semantics, 1981.

[19] Arend Rensink. *The Edge of Graph Transformation - Graphs for Behavioural Specification*, volume 5765 of *Lecture Notes in Computer Science*, pages 6–32–32. Springer Berlin / Heidelberg, 2010.

[20] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *Proceedings of the 3rd European conference on Model driven architecture-foundations and applications*, ECMDA-FA'07, pages 157–171, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] M. Seifert and R. Samlaus. Static source code analysis using ocl. In J. Cabot and P Van Gorp, editors, *Proc. of the MoDELS 2008 Workshop on OCL Tools: From Implementation to Evaluation and Comparison*, 2008.

[22] Bran Selic. The theory and practice of modeling language design for model-based software engineering - a personal perspective. In João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 290–321. Springer Berlin / Heidelberg, 2011.

[23] Christian Soltenborn and Gregor Engels. *Towards Test-Driven Semantics Specification*, volume 5795 of *Lecture Notes in Computer Science*, pages 378–392. Springer Berlin / Heidelberg, 2009.

[24] Gijs Stuurman and Ivan Kurtev. Action semantics for defining dynamic semantics of modeling languages. In *Proceedings of the Third Workshop on Behavioural Modelling*, BM-FA '11, pages 64–71, New York, NY, USA, 2011. ACM.

[25] Guido Wachsmuth. Modelling the operational semantics of domain-specific modelling languages. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 506–520. Springer Berlin / Heidelberg, 2008.

Consistency Checking Scenario-Based Specifications of Dynamic Systems by Combining Simulation and Synthesis

Joel Greenyer[1]*, Jens Frieben[2]

Politecnico di Milano,
Dipartimento di Elettronica e Informazione, DeepSE Group
Piazza Leonardo Da Vinci, 32, 20233 Milano, Italy
greenyer@elet.polimi.it

Fraunhofer Project Group Mechatronic Systems Design
Software Engineering Department
Zukunftsmeile 1, 33102 Paderborn, Germany
Jens.Frieben@ipt.fraunhofer.de

Abstract. Modern technical systems often consist of multiple components that must fulfill complex functions in diverse and sometimes safety-critical situations. Precisely specifying the behavioral requirements for such systems is a challenge, especially because there may be inconsistent requirements in possibly unforeseen component configurations. We propose a scenario-based specification approach based on Modal Sequence Diagrams and a novel technique for finding inconsistencies in such specification based on a combination of simulation and synthesis techniques. The simulation via the play-out algorithm can be used to analyze the scenario requirements in large and dynamic systems. Play-out, however, may run into avoidable violations, so that the engineer cannot assume the specification's inconsistency nor its consistency. We thus propose to check specification parts for static component configurations via synthesis. Then, if the part specifications are consistent, the resulting controllers can guide the play-out for the complete specification, avoiding more avoidable violations in the next simulation run.

Keywords: scenario-based specification, dynamic systems, consistency checking, simulation, synthesis

1 Introduction

Modern technical systems in areas like transportation, traffic, or production typically consist of multiple components that must interact to fulfill complex functions in diverse and sometimes safety-critical situations. Moreover, these systems

* This work was elaborated mainly as part of the author's dissertation thesis [8], while he was working at the University of Paderborn, Germany

are often *dynamic*, i.e., the relationships among the components may change or components may leave or enter the system. Precisely and consistently specifying the requirements for such dynamic systems is a major challenge, especially because there may be many, possibly unforeseen configurations of components where components are involved in multiple use cases at once and conflicts among the components' interaction specifications are possible to occur. If such inconsistencies remain undetected, this may lead to costly iterations in the development or to flaws in the final product.

In this paper, we propose first (1) a use case- and scenario-based approach for specifying the interaction behavior of components in a dynamic system. The approach is based on Modal Sequence Diagrams (MSDs) [18,12], a recent variant of Live Sequence Charts (LSCs) [6], that allows the engineer to formally specify what may, must, or must not happen in a system. Second (2), we propose a novel technique for finding inconsistencies in MSD specifications, which is based on the symbiosis of simulation and synthesis techniques.

As an example, we consider the specification of the RailCab system, which is developed at the University of Paderborn. Here, small, autonomous rail vehicles, called *RailCabs*, transport passengers and goods on demand. This system is highly dynamic as relationships among RailCabs and control stations change when for example RailCabs move along track sections, switches, and crossings.

First, to model such systems, we propose a special specification scheme where the requirements described in use cases are formalized by scenario-based *use case specifications*. A use case specification captures the structure and interaction behavior described in a use case formally by using UML collaboration diagrams and MSDs. We extend the MSDs with OCL binding expressions that can be attached to lifelines and allow the engineer to specify precisely which components in a dynamic system shall play which role in a use case. Within ScenarioTools, we have implemented an Eclipse/EMF&UML-based simulation engine for executing such use case specifications via the play-out algorithm [14,18]. This helps the engineer understand the interplay of different MSDs as environment events occur in a particular, maybe structurally evolving, system instance.

The play-out algorithm typically has to make many non-deterministic choices when executing the MSDs. In doing so, it may reach a state where a number of MSD require that something must happen that is forbidden by other MSDs. Such a violating state may indicate that the specification is inconsistent, but it may also be consistent and just the play-out algorithm did not "look ahead" to avoid the violation. Finding this out manually can be a very difficult.

We observe in our example that use cases typically describe the interaction of a fixed set of participants. For this case, we developed a *synthesis* technique that can effectively determine whether such a use case specifications is inconsistent or not. If it is consistent, we can synthesize a *strategy* that demonstrates that there exists a system that can always react to all possible sequences of environment event in a way that satisfies the use case specification.

However, even if all use case specifications are consistent, it may be that conflicts among MSDs of different use case specifications occur if use cases *overlap*, i.e., components are involved in multiple use cases at the same time. To further analyze the specification, we therefore still rely on the simulation via play-out. To improve the play-out, we developed a mechanism that guides the play-out by the strategies that could be successfully synthesized from single use case specifications. This improves the effectiveness of the simulation, giving the engineer more reason to suspect an actual inconsistency if a violation occurs. In the future, this approach could even be extended to successively eliminate all false negatives by synthesizing strategies also for overlapping use case occurrences.

This paper is structured as follows. We explain the foundations of MSDs in Sect. 2 and present an example use case specification in Sect. 3. A strategy that could be synthesized from a use case specification is explained in Sect. 4. In Sect. 5 we then describe our extended play-out algorithm and overview our tool implementation in Sect. 6. We discuss related work in Sect. 7 and conclude in Sect. 8.

2 Foundations

MSDs were proposed by Harel and Maoz as a formal interpretation of UML sequence diagrams, based on the concepts of LSCs [12]. In the following, we first explain the basics of MSDs and the play-out algorithm with respect to static systems. In Sect. 2.3 we then explain extensions to MSDs and their interpretation in the context of dynamic systems.

2.1 MSD Specifications

An MSD specification consists of a set of MSDs. An MSD can be *existential* or *universal*. Existential diagrams specify sequences of events that must be possible to occur in the system. Universal diagrams specify requirements that must be satisfied by all sequences of events that occur. During specification, the focus typically lies on universal MSDs, since they allow the engineers to express mandatory behavior. We also focus on universal MSDs in this paper.

Each lifeline in an MSD represents an object in an *object system* that consists of *environment objects* and *system objects*. The set of system objects is called the *system*; the set of environment objects is called the *environment*.

The objects can interchange messages. Here we consider only *synchronous* messages where the sending and receiving of the message is a single event. Our approach can, however, be extended also to asynchronous communication. We call the sending and receiving of a message a *message event* or simply *event*.

The messages in a universal MSD can have a *temperature* and an *execution kind*. The temperature can be either *hot* or *cold*; the execution kind can be either *monitored* or *executed*.

The semantics of these messages is as follows: An event can be *unified* with a message in an MSD iff the event name equals the message name and the sending and the receiving objects are represented by the sending resp. receiving lifelines of the message. When an event occurs in the system that can be unified with the first message in an MSD, an *active copy* of the MSD or *active MSD* is created. (We consider that an MSD has only one first message.) As further events occur that can be unified with the subsequent messages in the diagram, the active MSD progresses. This progress is captured by the *cut*, which marks for every lifeline the locations of the messages that were unified with the message events. If the cut reaches the end of an active MSD, the active copy is terminated.

If the cut is in front of a message on its sending and receiving lifeline, the message is *enabled*. If a hot message is enabled, the cut is also *hot*. Otherwise the cut is *cold*. If an executed message is enabled, the cut is also *executed*. Otherwise the cut is *monitored*. An enabled executed message is called an *active* message.

A *safety violation* occurs iff in a hot cut a message event occurs that can be unified with a message in the MSD that is not currently enabled. If this happens in a cold cut, it is called a *cold violation*. Safety violations must never happen, while cold violations may occur and result in terminating the active copy of the MSD. If the cut is executed, this means that the active MSD must progress and it is a *liveness violation* if an active MSD never terminates or progresses to a monitored cut.

As an example, Fig. 1 shows an MSD and an illustration of the considered example. Cold monitored messages are shown as blue, dashed arrows; hot executed messages are shown as red, solid arrows. The dashed horizontal lines in the MSD RequestEnterAtEndOfTrackSection also show the reachable cuts, which are accordingly cold and monitored (c/m) or hot and executed (h/e). Intuitively, this MSD expresses the following requirements. We consider a scenario where a RailCab moves along its current track section. At some point the RailCab rc detects that it reaches the end of the current track section. This is modeled as the message `endOfTS` sent between the environment and the RailCab rc. Now the RailCab rc must send `requestEnter` to the next track section control tsc2, which must reply with `enterAllowed`. These two messages must be sent before the RailCab reaches a point where it is possible for the last time to safely break before entering the switch (modeled by the environment message `lastBrake`).

Messages can also have parameters of certain types. Message events must then carry according parameter values. Here the message `enterAllowed` has a Boolean parameter, representing the choice to allow or deny the RailCab to enter. In this MSD, the required parameter value is not specified, which allows the parameter value to be either true or false. For more details on the interpretation of parameter values, we refer to [8, pp. 33].

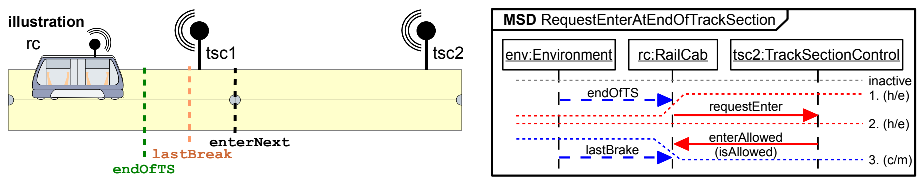


Fig. 1. The MSD RequestEnterAtEndOfTrackSection with illustration

We assume that the system is always fast enough to send any finite number of messages before the next environment event occurs. An infinite sequence of message events is called a *run* of the system and its environment. A run *satisfies* an MSD specification consisting of a set of universal MSDs if it does not lead to a safety or liveness violation in any MSD. (Multiple MSDs may be active at the same time.) We say that an MSD specification is *consistent* or *realizable* iff it is possible for the system objects to react to every possible sequence of environment events so that the resulting run satisfies the MSD specification.

2.2 Play-Out

Harel and Marelly defined an executable semantics for the LSCs, called the *play-out* algorithm [13], that was later also defined for MSDs [18]. The basic principle is that if an environment event occurs and this results in one or more active MSDs with active (enabled executed) system messages, then the algorithm non-deterministically chooses to send a corresponding message if that will not lead to a safety violation in another active MSD. The algorithm will repeat sending system messages until no active MSDs with an active message remain. Then the algorithm will wait for the next environment event, and this process continues.

If the play-out algorithm reaches a state where there are active messages, but they would all lead to safety violations, this is called a *violation.* If the MSD specification is inconsistent, this implies that there exists a sequence of environment events that will lead the play-out algorithm to a violation. Such a situation can, however, also occur if the specification is consistent. That is because the play-out algorithm will often make non-deterministic choices without "looking ahead" if they guarantee it not to run into violations later.

2.3 MSDs and Dynamic Systems

When specifying the behavior of dynamic systems, it is often impractical to consider MSDs where each lifeline refers to a concrete object. Instead, *symbolic lifelines* were introduced by Marelly et al. [19,14], which refer to a class. MSDs with symbolic lifelines are also called *symbolic MSD*; MSDs with non-symbolic lifelines, also called *concrete lifelines*, are called *concrete MSDs.* Here, concrete lifelines have an underlined label; the label of symbolic lifelines is not underlined.

In an active copy of an MSD with symbolic lifelines, a symbolic lifeline can be *bound* to an object that is an instance of the class referenced by the lifeline. For a given object system, the semantics of a symbolic MSD is equivalent to a set of concrete MSDs where for each possible combination of bindings of the symbolic lifelines, there exists a concrete MSD with lifelines corresponding to this possible combination of bindings.

Typically, we want to restrict a symbolic MSD to specify the behavior only for combinations of objects that have certain relationships or properties. Then, *binding expressions* can be added to the MSD in order to restrict the possible lifeline bindings. Harel and Marelly define that binding expressions are expressions over object properties or relationships between objects that evaluate to a

Boolean value [14]. A symbolic MSD then only specifies the behavior for the combinations of objects where there exists a set of lifeline bindings where all binding expressions evaluate to true.

Instead of translating symbolic MSDs to sets of concrete MSDs, Harel and Marelly extended the play-out algorithm to handle the dynamic binding of symbolic lifelines, supporting a simple form of binding expressions [14, pp. 209]. In ScenarioTools, we implement similar mechanisms and consider binding expressions of the form `<lifeline-name> := <expr>` where `<lifeline-name>` is the name of a lifeline, also called the *slot lifeline*, and `<expr>` is an OCL expression, also called the *value expression*. The value expression can evaluate to an object that is an instance of the slot lifeline's class.

Lifeline names can be used as variables within value expressions. If a lifeline is bound to an object, so is the corresponding variable. Also other variables can be used in value expression. In the course of progressing an active MSD, there may be for example variables that are assigned parameter values, like the variable isAllowed show in Fig. 1. The details of these mechanisms are not relevant here. Important is that value expressions can only be evaluated when all the variables appearing in the expression are bound.

During play-out, symbolic MSDs and binding expressions are interpreted as follows: As a message event can be unified with a first message in an MSD, an active copy of the MSD is created with the sending and receiving lifelines of the first message bound to the sending and receiving object of the message event. Then the value expressions of the binding expressions are evaluated as soon as that is possible, and the corresponding slot lifelines are bound to the resulting objects. It must not happen that a message is enabled and the sending or receiving lifeline is unbound.

As an example, consider the symbolic variant of the MSD RequestEnterAtEndOfTrackSection shown on the right of Fig. 2, executed in the context of an object system as illustrated on the left. If the message `endOfTS` is sent from the environment object e to the RailCab rc1, an active copy of the MSD is created where the lifeline env is bound to the object e and the lifeline rc is bound to the object rc1. Now the binding expression can be evaluated, which results in binding the lifeline next to the object tsc2.

We also consider that the value expression can evaluate to a set of objects. Then for each object in the set a copy of the active MSD is created with the slot lifeline bound to that object (see also [14, pp. 215]).

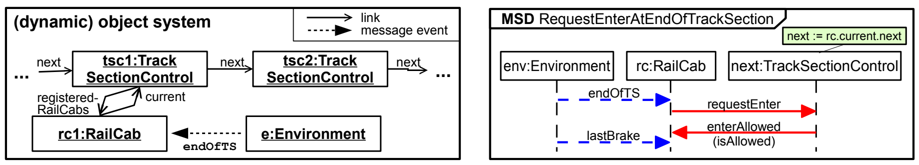


Fig. 2. A dynamic object system and the symbolic version of the MSD RequestEnterAtEndOfTrackSection

3 Use Case Specifications

We observe that the early, informal requirements of a dynamic system are often structured in use cases that describe (1) a particular configuration of objects and (2), by a number of scenarios, how these objects may, must, or must not react to certain, usually external, events. Instead of specifying the behavior of a dynamic system with a plain set of symbolic MSDs, we thus propose a more systematic approach where an engineer first captures the objects involved in the use case and then specifies the MSDs based on this structure.

3.1 Use Case Specification Structure

Figure 3 shows the example of a use case specification for the use case RailCab Obstacle Detected. The use case describes that a RailCab that detects an obstacle must report a hazard and its position to its current track section control, which then must warn the other RailCabs on that track section. The MSDs are in fact an example where the play-out algorithm may choose an execution that inevitably leads to a violation—but we will return to that in Sect. 4. We first explain the structure and semantics of a use case specification.

A use case specification consists of a package with class definitions and a collaboration (dashed ellipse) [1, Sect. 9.3.3]. The collaboration captures the objects participating in a use case and it contains a set of MSDs. The nodes in the collaboration diagram are called *roles*, and each role represents a system or an environment object. Here environment roles are represented by a cloud symbol. The roles are typed by classes, which are modeled in the class diagram. The classes can define attributes, associations, and operations; the latter indicate which messages an instance can receive.

Each lifeline of an MSD represents one role in the collaboration. Connectors between the roles can be used for indicating structurally which roles interact in the use case. In the MSDs it can then be ensured that messages are only modeled between lifelines where their roles are connected.

3.2 Use Case Specification Semantics

The advantage of this specification scheme, besides supporting a more structures modeling approach, is that it allows for two different interpretations that are crucial for the symbiosis of simulation and synthesis.

Symbolic interpretation: The MSDs are interpreted as symbolic MSDs, as if their lifelines would directly reference the classes that type the roles represented by each lifeline. The object system can be any valid instance of the class model. The collaboration has no particular semantics.

Static interpretation: Here we assume an object system where for each role in the collaboration there is a corresponding object of the class typing the role. The MSDs are then interpreted as static MSDs where each lifeline represents the object that corresponds to its role.

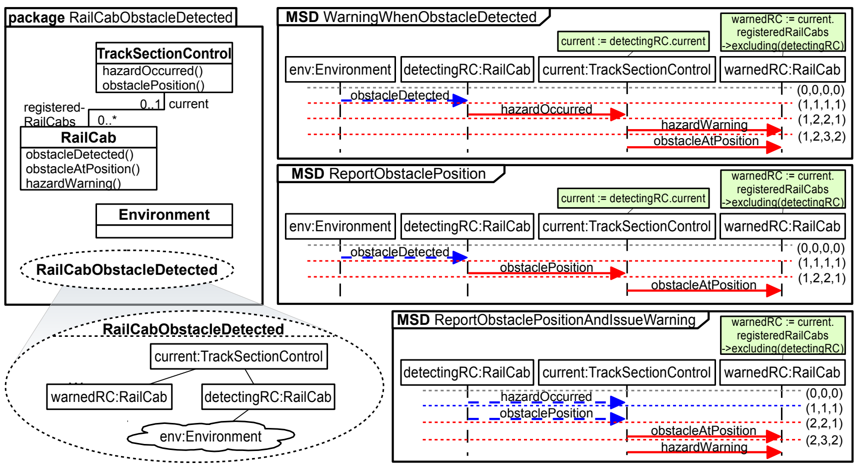


Fig. 3. The specification for the use case RailCab Obstacle Detected

The static interpretation makes the formal analysis of the use case specification feasible, which would not be the case with the symbolic interpretation, if we would have to consider many different object systems with different possible bindings of symbolic lifelines. The second-listed author has developed such a analysis technique in his thesis [8], which will be explained Sect. 4.

3.3 Combining Use Case Specifications

This modeling scheme also allows different engineers to specify different use case specifications in parallel. These can later be composed as follows.

(1) The class models of the use case specifications can be composed by merging them into one package using UML *package merge* [1, Sect. 7.3.41]. Package merge copies the contents of one or multiple *merged* packages into a *merging* package. Equally named UML elements (classes, attributes, operations, etc.) in the merged packages are mapped to the same element in the merging package.

(2) The UML package merge only defines how to merge structural (class) models. It could probably be extended easily to merge also the MSDs—but instead, we just slightly modify the symbolic interpretation of the MSD: We assume that the object system can be any valid instance of the merged class model. Then we interpret the sets of MSDs of all use case specifications like a plain set of (symbolic) MSDs where we interpret an MSD lifeline as if it was typed by the class that its role's class was merged into.

We call the merged package the *integrated package*. With this symbolic interpretation of the MSDs, it forms the *integrated specification* of the system.

If one use case depends on another, for example because one refers to message types (i.e., operations) or object properties already specified in another, this can also be expressed by a package merge where the depending use case specification package merges the package of the use case specification it depends on.

4 Synthesis

As already proposed by Bontemps et al. [3], the problem of deciding whether an MSD/LSC specification is consistent can be mapped to a two-player game problem. Intuitively, this means that environment events and system reactions are mapped to “moves” in a game that lead from one game state to another. A game state in this case is essentially a set cut configurations of currently active MSDs. Then it is checked whether there exists a *strategy* for the system against the environment such that a certain winning condition is satisfied. The winning condition here is that never a safety or liveness violation occurs and that there is no infinite sequence of system events, i.e., always eventually a next environment event can occur.

Today there exist a number of tools with efficient algorithms for finding winning strategies in two-player games. Similar to Bontemps et al, we have developed a synthesis approach [8,7] where MSD use case specifications are mapped to the input of Uppaal Tiga [2], a tool based on the Uppaal model-checker that implements an efficient game-solving algorithm [4]. Novel in our approach is that it also supports timed MSDs and MSDs that formulate *environment assumptions*, i.e., properties that the environment must satisfy to “win” against the system. But these novelties are not relevant in the scope of this paper.

In our synthesis approach, if an MSD use case specification is consistent, Uppaal Tiga will synthesize a strategy that shows us how the system can always react to the environment such that the specification is satisfied. Uppaal Tiga can even synthesize a *complete* strategy that shows all the moves to all states in which the system will be able to win. Furthermore, if an MSD use case specification is inconsistent, Uppaal Tiga can synthesize a *counter-strategy* that shows how the environment can always violate the MSD specification.

Listing 1.1 shows a excerpt from a complete strategy synthesized from the use case specification RailCab Obstacle Detected; Uppaal Tiga generates such a textual output to the console or a file. Here only one state in the game and the two winning transitions for this state are shown. As shown here, the sate is essentially a particular configuration of cuts of the active MSDs (see [8, App. C.2] for more information). Here it is the cuts reached in the MSDs after the environment event `obstacleDetected` occurred (compare also with Fig. 3).

Listing 1.1. Excerpt from the controller synthesized from the specification of the use case Warn RailCabs On Track

```
Strategy to win:
...
State: (...) ...
WarningWhenObstacleDetected_env=1
WarningWhenObstacleDetected_detectingRC=1
WarningWhenObstacleDetected_current=1
WarningWhenObstacleDetected_warnedRC=1
ReportObstaclePosition_env=1
ReportObstaclePosition_detectingRC=1
ReportObstaclePosition_current=1
ReportObstaclePosition_warnedRC=1
ReportObstaclePositionAndIssueWarning_detectingRC=0
ReportObstaclePositionAndIssueWarning_current=0
```

```
ReportObstaclePositionAndIssueWarning_warnedRC=0
When you are in true, take transition
   systemProcess.systemActive->systemProcess.produceEvent
      { 1, tau, event := detectingRC_current_reportHazard }
When you are in true, take transition
   systemProcess.systemActive->systemProcess.produceEvent
      { 1, tau, event := detectingRC_current_obstaclePosition }
...
```

According to this strategy, the system can satisfy the use case specification against any environment if in this state the RailCab detectingRC sends the message reportHazard or obstaclePosition to the track section control current. As it is a complete strategy, we know that sending any other system message will not allow the system to "win" against any possible environment.

The possible violation is follows: Sending the message hazardOccurred, which in this state is enabled in the MSD WarningWhenObstacleDetected, must be followed by sending obstaclePosition. This then leads to a situation where there is an active copy of MSD WarningWhenObstacleDetected in cut (1,2,2,1) and an active copy of MSD ReportObstaclePositionAndIssueWarning in cut (2,2,1). In the former, hazardWarning must occur, but obstaclePosition must not occur. In the latter, obstaclePosition must occur, but hazardWarning must not occur. Thus a safety or liveness violation is inevitable.

5 Combining Play-Out and Synthesis

The naive play-out of an integrated specification containing the use case RailCab Obstacle Detected could easily run into the above-mentioned, avoidable violation. In many cases, this could be avoided by using the strategies that could be successfully synthesized from single use case specifications. In the following, we explain an extension of the play-out algorithms that is guided by these strategies.

The principle of this extension is shown in Fig. 4. At the bottom, it sketches a RailCab object system where the environment just sent the message obstacleDetected to the RailCab rc2. On the top left, two active MSDs are shown, which are activated as a result. The lifeline bindings are indicated by the small labels on the lifelines. The remaining parts of the figure are explained in the following.

The main challenge in employing the strategies synthesized from use case specifications is to determine in a state during play-out which state in which strategy (or which states in which strategies) to inquire about which message event can be safely executed next. Intuitively, we have to determine where a use case "occurs". We define a *use case occurrence* as a set of active MSDs where the MSDs belong to the same use case occurrence and the lifelines representing the same role are bound to the same object.

After finding the active MSDs that make up a use case occurrence (1), we create an *active copy* of the corresponding strategy (also called *active strategy*). Then we find a state in the this strategy that corresponds to the cuts of the active MSDs (2). This state is also called the *current state* of the active strategy for the use case occurrence.

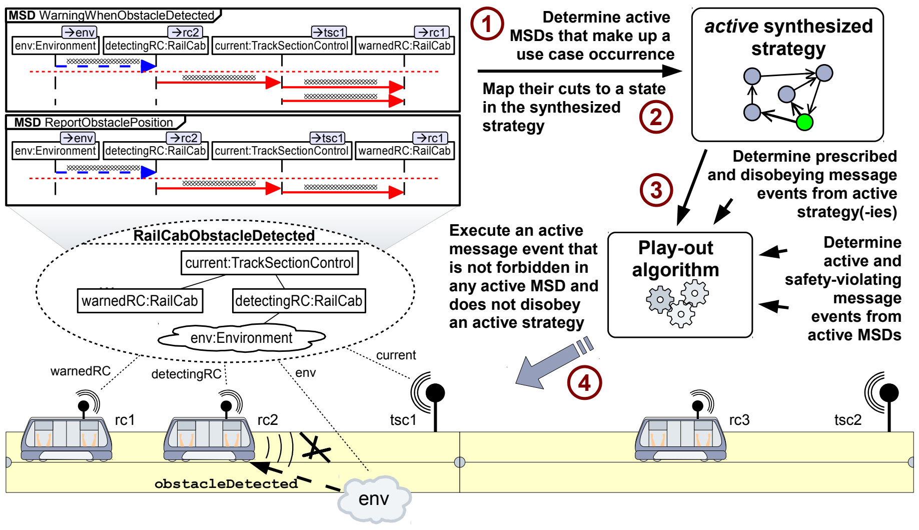

MSD WarningWhenObstacleDetected
env:Environment
detectingRC:RailCab
current:TrackSectionControl
warnedRC:RailCab
MSD ReportObstaclePosition
RailCabObstacleDetected
1 Determine active MSDs that make up a use case occurrence
Map their cuts to a state in the synthesized strategy 2
active synthesized strategy
3 Determine prescribed and disobeying message events from active strategy(-ies)
Play-out algorithm
Execute an active message event that is not forbidden in any active MSD and does not disobey an active strategy
Determine active and safety-violating message events from active MSDs
4
warnedRC
detectingRC
env
current
rc1
rc2
tsc1
rc3
tsc2
obstacleDetected
env

Fig. 4. Guiding the play-out by strategies synthesized from use case specifications

Once such corresponding states are found for all occurrences of use case for which a strategy is provided, we determine the *prescribed* and *disobeying* message events (3). A message event is *prescribed* if it corresponds to an event that labels a transition leaving the current state of the active strategy; a message event is *disobeying* if it corresponds to a message event that labels a transition that is not leaving the current state. (For brevity, we skip a more formal definition when message events correspond in this case.)

As in regular play-out, we also have to determine which message events are active and safety-violating in the active MSDs. We can now execute an active event that is not safety-violating any other active MSD, and not disobeying any active strategies (4). This process is repeated until the there are no more active events. Then the system waits for the next environment event.

In this extended play-out it is not guaranteed that never an active strategy must be disobeyed or never a safety violation occurs. This may still happen when use cases overlap, i.e., objects participate in multiple use case occurrences at once. Then again safety violations and events disobeying active strategies may not necessarily mean that the specification is inconsistent—it could still be that the system in the past could have chosen another sequence of steps to avoid this. The second-listed author also describes an extension of this approach for employing strategies synthesized from *composed use case specifications*, but these concepts are not yet implemented.

After an active strategy was disobeyed, the play-out can still continue, but then it may no longer be possible to find current state for the disobeyed active strategy. Note that if the strategies used in this process are not complete, it becomes more likely that active strategies must be disobeyed.

Note also that we can only identify use case occurrences if all lifelines of an active MSD are bound. For this process to work properly, there should thus not be an active MSD with unbound lifelines. It remains to be investigate if maybe the play-out can follow multiple active strategies in parallel for different “candiadate” use case occurrences as long as lifelines are unbound.

6 Realization and Evaluation

The concepts introduced here have been implemented in an Eclipse-based tool suite called ScenarioTools[1]. Figure 5 gives an overview of ScenarioTools and the supported modeling and analysis process.

In the first step, a UML-based MSD specification of the system is modeled. For modeling, ScenarioTools extends the Topcased UML-Editor. The figure shows a number of packages that represent use case specifications (1). As mentioned before, use case specifications can be modeled in separate packages, dependencies can be expressed by package merge relationships, and finally all use case specifications are merged into an integrated package.

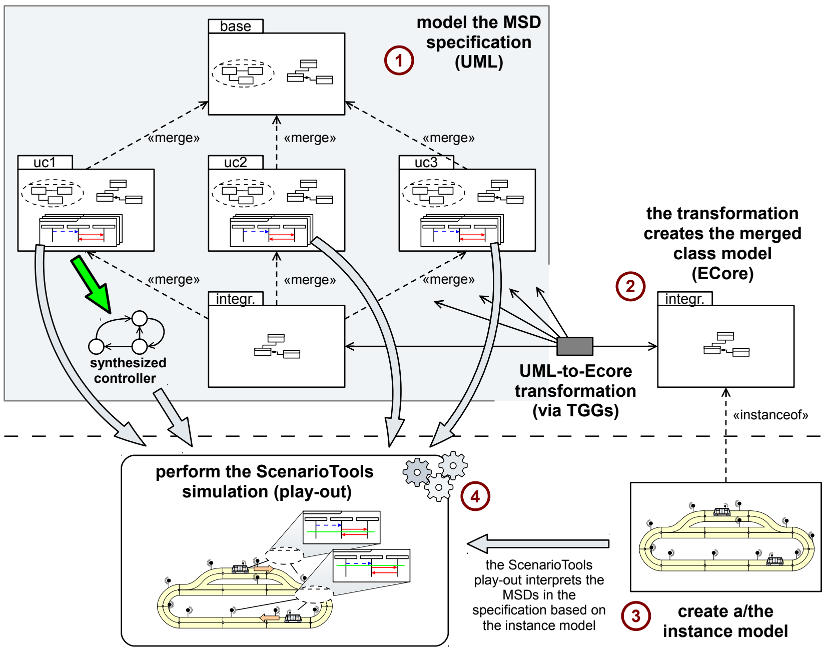
The figure here also shows a *base package*. We suppose that sometimes, prior to specifying the use cases, the requirements engineers already want to formally capture a structural (class) model of the system. This is also called *domain modeling*, and fosters a common understanding of the domain. This can be done in this separate package from where classes, associations, and attributes can be reused in the use case specifications. To do this, the use case specifications define merge relationships to the base package.

In the second step, after formally specifying the use cases, we want to create an instance system, or possible many instance systems, to carry simulations of the specified behavior. To be able to do this, we create an EMF/ECore[2] class model that corresponds to the merged class model of the UML-based MSD specification. This transformation is described by Triple Graph Grammar (TGG) that can be executed using the TGG Interpreter[3]. The transformation not only creates the ECore class model, but also a *correspondence model* that stores a detailed mapping between classes, properties, associations and operations in the UML and ECore class models.

The Eclipse/EMF framework allows us to automatically generate simple editors from the ECore class model. With the help of these editors, instance models can easily be created (3). In the RailCab case, this could be a particular RailCab track system with a particular number of RailCabs currently on certain track sections.

Based on an instance model, we can now simulate the behavior defined in the UML-based MSD specification (4). To know which MSD lifelines can be bound to which objects and which messages can be sent between the objects, we exploit the information in the above-mentioned correspondence model.

[1] `http://www.cs.uni-paderborn.de/index.php?id=scenariotools`
[2] `http://www.eclipse.org/emf/`
[3] `http://www.cs.uni-paderborn.de/index.php?id=tgg-interpreter`



base
model the MSD specification (UML)
1
«merge»
«merge»
«merge»
uc1
uc2
uc3
«merge»
«merge»
«merge»
integr.
synthesized controller
the transformation creates the merged class model (ECore)
2
integr.
UML-to-Ecore transformation (via TGGs)
«instanceof»
perform the ScenarioTools simulation (play-out)
4
the ScenarioTools play-out interprets the MSDs in the specification based on the instance model
3
create a/the instance model

Fig. 5. Overview of the ScenarioTools simulation

The figure also illustrates that the play-out can be guided by strategies that could be successfully synthesized from use case specification.

ScenarioTools supports different simulation modes: a user-guided step-by-step selection of system and environment events and a random execution.

7 Related Work

In the past, many approaches for the scenario-based specification of system requirements have been proposed. Many, however, did not regard that scenarios can be overlapping [15] or they only regarded existential scenarios, i.e., descriptions of what must be possible to occur [17]. Others considered combining existential scenarios with pre-and post-conditions on messages [21], automata [20], or safety properties in temporal logic [5] for expressing also mandatory requirements. With these additions also came the problem of checking the consistency of the specification, which is addressed in these papers. To the best of our knowledge, all these approaches did not consider the specification of dynamic systems.

LSCs introduced a rigorous semantics for expressing universal and existential requirements [6] and only with symbolic lifelines [19], the behavior of dynamic systems could be specified in a formal scenario-based way. Many approaches for consistency checking LSC specifications and synthesizing controllers from them were proposed [9,11,3,16], but they only consider static systems.

Another approach for improving the play-out of LSC specifications is smart play-out [10]. Here model-checking is employed for finding a sequence of steps for the system in reaction to an environment event that avoids avoidable violations. The problem here is that this approach can only “look ahead” until the next environment event occurs, thus not all avoidable violations can be anticipated. Also, smart play-out only works in a static setting.

Maoz et al. presented an alternative implementation of the play-out algorithm using AspectJ [18]. This implementation is extensible to plug-in different play-out “strategies”, which for example allows for integrating smart play-out. This implementation of the play-out algorithm is also used in the PlayGo tool[4]. The website also mentions that synthesized strategies and counter-strategies can be executed using this tool, but no details have been published thus far. Kugler et al. also mention to execute synthesized controllers [16], but these are not combined with play-out.

8 Conclusion and Outlook

We presented a novel extension of the play-out algorithm which combines play-out of MSDs in a dynamic object system with strategies synthesized from specification parts. This helps avoid more avoidable violations and improves the play-out because engineers now have more reason to suspect an actual inconsistency when violations occur.

We are currently developing a new version of the ScenarioTools play-out and synthesis and plan to extend this approach to also support environment assumptions and parametrized messages. We also plan to further investigate the symbiosis of synthesis and play-out for composed use cases.

References

1. UML 2.4.1 superstructure specification (August 2011), OMG document `formal/2011-08-06`
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) Proc. 19th Int. Conf. on Computer Aided Verification (CAV'07). LNCS, vol. 4590, pp. 121–125. Springer, Berlin, Germany (July 2007)
3. Bontemps, Y., Schobbens, P.Y.: Synthesis of open reactive systems from scenario-based specifications. In: Proc. 3rd Int. Conf. on Application of Concurrency to System Design (ACSD 2003), 18-20 June 2003, Guimaraes, Portugal. pp. 41–50 (2003)
4. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) Proc. 16th Int. Conf. on Concurrency Theory (CONCUR'05). LNCS, vol. 3653, pp. 66–80. Springer, San Francisco, CA, USA (August 2005)

[4] `http://www.weizmann.ac.il/mediawiki/playgo/index.php`

5. Damas, C., Lambeau, B., van Lamsweerde, A.: Scenarios, goals, and state machines: a win-win partnership for model synthesis. In: Proc. 14th Int. Symp. on Foundations of Software Engineering (ACM SIGSOFT '06/FSE-14). pp. 197–207. ACM (2006)
6. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. In: Formal Methods in System Design. vol. 19, pp. 45–80. Kluwer Academic Publishers (2001)
7. Greenyer, J.: Synthesizing modal sequence diagram specifications with Uppaal-Tiga. Tech. Rep. tr-ri-10-310, University of Paderborn (February 2010)
8. Greenyer, J.: Scenario-based Design of Mechatronic Systems. Ph.D. thesis, University of Paderborn (October 2011)
9. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. In: Foundations of Computer Science. vol. 13:1, pp. 5–51 (2002)
10. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design, FMCAD 2002, Portland, OR, USA, November 6-8, 2002. pp. 378–398 (2002)
11. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenario-based requirements. In: Kreowski, H.J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 309–324. Springer (2005)
12. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling (SoSyM) 7(2), 237–252 (May 2008)
13. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: The play-in/play-out approach. Software and System Modeling (SoSyM) 2, 2003 (2002)
14. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (August 2003)
15. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In: Proc. IFIP WG10.3/WG10.5 Int. Workshop on Distributed and Parallel Embedded Systems (DIPES '98). pp. 61–71. Kluwer Academic Publishers, Norwell, MA, USA (1999)
16. Kugler, H., Plock, C., Pnueli, A.: Controller synthesis from LSC requirements. In: Chechik, M., Wirsing, M. (eds.) Proc. 12th Int. Conf. of Fundamental Approaches to Software Engineering, FASE 2009. LNCS, vol. 5503, pp. 79–93. Springer (2009)
17. Maier, T., Zündorf, A.: The Fujaba statechart synthesis approach. In: Proc. 2nd Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ICSE '03 (2003)
18. Maoz, S., Harel, D.: From multi-modal scenarios to code: Compiling LSCs into AspectJ. In: Proc. Int. Symp. on Foundations of Software Engineering (FSE'05). pp. 219–230 (2006)
19. Marelly, R., Harel, D., Kugler, H.: Multiple instances and symbolic variables in executable sequence charts. In: Proc. 17th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02). ACM SIGPLAN Notices, vol. 37, pp. 83–100 (November 2002)
20. Sikora, E., Daun, M., Pohl, K.: Supporting the consistent specification of scenarios across multiple abstraction levels. In: Wieringa, R., Persson, A. (eds.) Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol. 6182, pp. 45–59. Springer (2010)
21. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proc. 22nd Int. Conf. on Software Engineering, ICSE '00. pp. 314–323 (2000)

The Event Coordination Notation: Execution Engine and Programming Framework

Ekkart Kindler

Informatics and Mathematical Modelling, Technical University of Denmark
DK-2800 Lyngby, DENMARK
eki@imm.dtu.dk

Abstract. *ECNO* (*Event Coordination Notation*) is a notation for modelling the behaviour of a software system on top of some object-oriented data model. ECNO has two main objectives: On the one hand, ECNO should allow modelling the behaviour of a system on the domain level; on the other hand, it should be possible to completely generate code from ECNO and the underlying object-oriented domain models.
Today, there are several approaches that would allow to do this. But, most of them would require that the data models and the behaviour models are using the same technology and the code is generated together. By contrast, ECNO can be used for modelling the behaviour on top of any object-oriented model – or even on top of manually written object-oriented code. This way, it is easy to integrate ECNO models with other technologies, to use ECNO on top of code generated by other technologies or with code that was written manually.
In this paper, we rephrase the main concepts of ECNO. The focus of this paper, however, is on the architecture of the ECNO execution engine and its programming framework. We will show how this framework allows us to integrate ECNO with object-oriented models, how it works without any explicit control, and how it easily integrates with traditional programming.
Keywords: Model-based Software Engineering, Local and global behaviour modelling, Event coordination, Model integration.

1 Introduction

The *Event Coordination Notation* (*ECNO*) [1–3] aims at modelling the behaviour of software systems on top of object-oriented data models. This way, domain models can not only cover the data part, but also the behaviour of a system. ECNO's main concept are *events* and the *coordination* of the joint execution of events among different objects. For each class, it is defined in which events the objects of the class can participate, and the *local behaviour* for the class defines when an object can participate in an event, and what happens within that object when the event happens. The joint execution of all the participating objects is called an *interaction*. The main concepts of ECNO have been presented earlier already [2, 3]; therefore, we only rephrase these ideas in

Sect. 2. For lack of space, however, we cannot discuss the related work again; for a detailed discussion of the related work we refer to [3].

In a first prototype [2], we had shown that the coordination of events defined by ECNO's *coordination diagrams* can be handled by an execution engine, and that the code for the local behaviour of each class can be generated from an extended version of Petri nets, which we called *ECNO nets* [3]. This prototype, however, required that all the object-oriented models or the manually written code followed the principles of ECNO and used the ECNO interfaces; for this purpose, the prototype was equipped with a light-weight version of object-oriented models. The first prototype of ECNO did not meet one of the most important objectives of ECNO yet: it should be easy to integrate different modelling technologies, and to use ECNO on top of existing software, which might be any manually written code or code generated by some other technology.

In this paper, we discuss the design and architecture of a second ECNO prototype, which consists of a ECNO engine and a programming framework that supports the integration of ECNO with other technologies. As an example, ECNO can now be used on top of the Eclipse Modeling Framework (EMF) [4].

2 The Event Coordination Notation

In this section, we rephrase the main concepts of ECNO by the help of an example, which is taken from [2] with minor modifications.

2.1 Coordination Diagrams

This running example is the eternal coffee and tea vending machine. Figure 1 shows a class diagram with some extensions concerning *events* and their *coordination*. Therefore, we call it *coordination diagram*.

Before explaining the extensions that concern the coordination, let us have a brief look at it as a class diagram – ignoring the operations compartment for a moment. The diagram shows the different possible *elements*[1] that are part of the system: A coin can be close to a slot, which is represented by the reference from Coin to Slot, or a coin can be in the slot, which is represented by the reference from Slot to Coin. There is a Safe to which a coin is passed when a coffee or tea is dispensed. There is a Panel for the user to interact with the vending machine. This panel is connected to a control, which is represented by the reference from Panel to Control. The control is connected to brewers, which can be either coffee or tea brewers. At last, there is an output device for the beverage, which is connected to the brewers.

As a class diagram, Fig. 1 can have instances, which would define a concrete configuration of the vending machine. In our example, we assume that there initially are three coins (not inserted yet to the slot), and that there are two

[1] In order to point out that our objects are a bit more than objects in the traditional sense of object orientation, we call them *elements* throughout this paper.

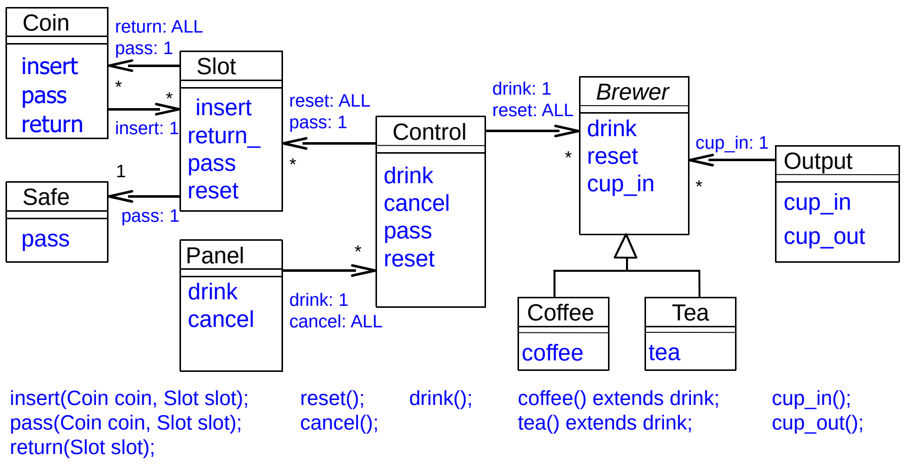


Fig. 1. A class and coordination diagram

coffee brewers and one tea brewer; for all other classes, we assume that there is exactly one instance. Figure 2 shows such such an instance at a later stage – after two of the coins were inserted to the slot. The overlays on top of the object diagram will be discussed later.

Now, let us explain the extensions of Fig. 1 that concern events and their coordination: First, there are some events defined at the bottom of the diagram, such as insert or drink, which will be explained in some more detail later. For now, we just use their names. These events occur in the operations compartment of the classes again: for example, the events insert, pass and return for Coin. They define in which *events* the different elements could participate. More importantly, the references between the different elements are annotated with events and an additional quantifier: 1 or ALL. We call them *coordination annotations*. These annotations define the coordination of events and the participating elements: more precisely, for an element executing some event, it defines which other elements need to participate in the execution of this event. We call a combination of all the required elements and events and their execution an *interaction*. Fig. 2 indicates two examples of such interactions in the given situation, which we will use to explain the meaning of coordination annotations below; note that there would be more interactions in this situation, which are not shown.

Let us assume that some element is involved in the execution of some event and that, in the coordination diagram, the type of this element has a reference that is annotated with that event. Then, some elements at the other end of the respective links also need to participate in the interaction. In Fig. 2, for example, the panel is involved in the event drink (actually, it is the more specific coffee event), the coordination annotation drink:1 at the reference to control, implies that the control also needs to participate in the interaction with that event; in turn, the reference from control to brewer annotated with drink:1 requires

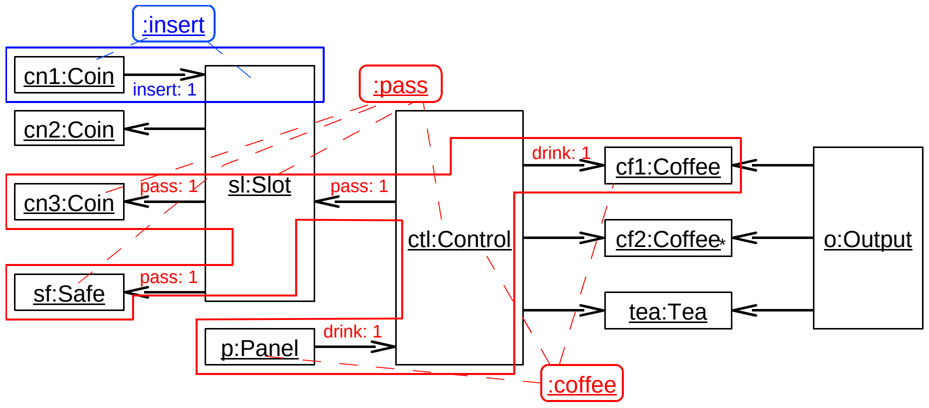


Fig. 2. A vending machine configuration with two interactions high-lighted

that also one brewer must participate. In the interaction shown in Fig. 2, it is the coffee brewer cf1; but choosing cf2 would also be fine. For reasons that will become clear later, the control is required to participate in a pass event together with a drink event. Therefore, the annotation pass:1 at the reference from class Control to class Slot requires a slot to participate too. The slot, in turn, has two references annotated with pass:1, one to the class Coin and one to the class Safe, both of which must be met. Therefore, one coin and one safe need to participate in the execution – this way, the coin will be passed from the slot to the safe, when the interaction is executed, which will be discussed in more detail later in Sect. 2.3. Altogether, this gives us an interaction with six elements and two events as shown in Fig. 2. Note that, though the synchronisations are via the drink event, the actually event in this interaction is a coffee event, since the coffee machine specializes the drink event to a coffee event.

As discussed above, a coordination annotation refers to an event and has a quantifier, which can be 1 or ALL. In the example, above, we have seen the quantifier 1 already: it means that one partner at the other end of the links corresponding to that reference must participate. If the event is quantified by ALL, all the elements at the other end of these links need to participate.

In our example from Fig. 2, there are two coins inserted to the slot. If, in this situation, the slot participates in a return event, the annotation return:ALL at the reference from Slot to Coin means, that both coins must participate in the execution of the event return.

2.2 Event Types and Parameters

Up to now, events have, basically, been names, which were used in coordination annotations to identify other partners that need to participate in an interaction. In addition to that, events can also be used to exchange information between the partners of an interaction. To this end, events can have parameters. The

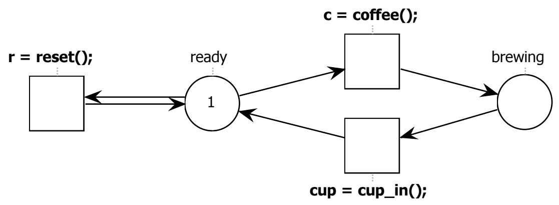


Fig. 3. Local behaviour of the coffee brewer

declarations of the events of our vending machine along with their parameters is shown at the bottom of Figure 1. In order to distinguish them from the concrete instances in interactions, we actually call them *event types*. On a first glance, the definition of event types looks very much like method declarations. In contrast to methods, however, event types or events do not have behaviour of their own. Events are used only for synchronizing participants in an interaction and to share information between them. Moreover, events are shared between different elements and do not belong to a particular element. This is why events are declared outside a specific element and are types in their own right. In particular, events do not have a caller or callee. Therefore, event parameters can be contributed in many different ways, and by different elements. It is not defined in advance, who will provide and who will use the parameters or in which direction the values are propagated. The execution engine of ECNO, however, guarantees that all elements participating in an interaction have the same parameters for the same event – if two partners contribute inconsistent values to an event, the interaction would not be possible.

In our example, we can also see *inheritance on events*: the coffee and tea event extend the more general event drink. This is an extension with respect to [2], but, we do not go into details here.

2.3 Local Behaviour

The *local behaviour* of an element defines when the element can participate in an event or a combination of events, and it defines what happens when the element participates in such an interaction. We use a slightly extended version of Petri nets for modelling the local behaviour of an element, which we call *ECNO nets* [3]. We will discuss the main concepts of ECNO nets by the help of our example.

We start with the ECNO net for the coffee brewer, which is shown in Fig. 3. Except for the annotations associated with the transitions, this is a conventional P/T-system. A transition annotation relates each transition of the ECNO net to one or more events; we call this annotation an *event binding*. After the event coffee, which represents the user pressing the coffee button, the coffee is brewed, which will be dispensed, when a cup is inserted (event cup_in). The reset event is possible only when the coffee machine is in the initial state (no coffee is being made). In this example, the notation for event bindings might appear overly

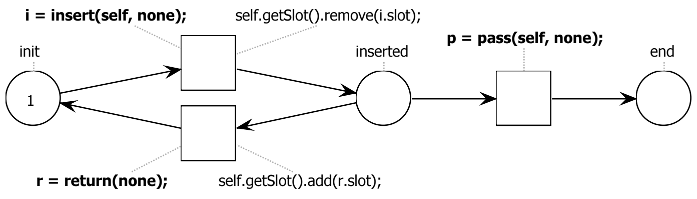


Fig. 4. Local behaviour of the coin

verbose. The reason for denoting an event binding as an assignment will become clear in the next example.

To explain some more details, let us have a look at the local behaviour of a coin, which is modelled by the ECNO net shown in Fig. 4. Here, the event bindings and the involved events have parameters. Let us consider the transition that is bound to the insert event first: as we have seen in Fig. 1, the event insert has two parameters: a coin and a slot. The annotation refers to these two parameters. The first one, self, assigns the coin itself as the first parameter (coin) to this event. The second parameter is none, which is the keyword indicating that, in this case, the coin does not assign a parameter to the event insert (in this case, this parameter is provided by another partner of the interaction). The other annotation of this transition is the *action*, which will be executed when all partners of an interaction are found and the interaction is executed. In this case, the coin deletes the link to the slot. The link to the slot is removed by using the API generated by EMF from the class diagram; self is an ECNO construct, which represents the object to which this behaviour is attached, the coin. The method getSlot() is the EMF generated API of the coin, returning a list of all slots the coin is close to, from which the involved slot is removed. This slot is denoted by i.slot, where i is the variable to which the insert event was assigned, and slot refers to the respective parameter, which, in this case, is assigned to the event by the element slot (see Fig. 5), which will be part of that interaction.

Once the coin is inserted, the ECNO net for the coin allows two things to happen: either the coin can be passed to the safe by the transition that is bound to event pass, or the coin is returned by the transition bound to event return. In the case of a pass event, the coin assigns itself (self) as the coin parameter; and there is no action. In the case of a return event, no parameter is assigned to the event (none), but the action will add a link from the coin to the slot again, where the slot is coming from the parameter r.slot of the event return.

Figure 5 shows the local behaviour of the slot. This is a rather degenerated Petri net. As a P/T-system, all transitions would be enabled all the time since their presets are empty. Due to the event bindings, however, the local behaviour becomes a bit more interesting. We start with explaining the bottom transition: This transition has two events bound to it: reset and return. This implies that the events reset and return must be executed together; this way, the slot defines that coins are returned during a reset. The slot assigns itself as a parameter to the return event. In the action, the slot deletes all the links to the coins it contains

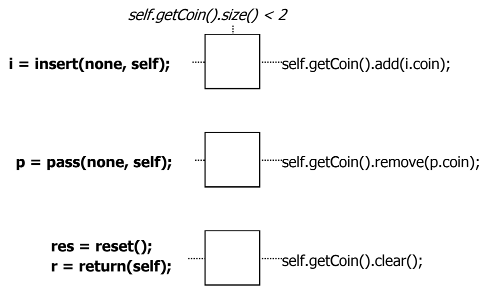


Fig. 5. Local behaviour of the slot

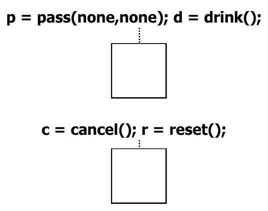


Fig. 6. Local behaviour of the control

using the EMF generated API: self.getCoin().clear(). At last, let us discuss the top transition of Fig. 5. It is bound to an insert event, where the slot assigns itself to the event's slot parameter. In the action, it adds a link to that coin. In this transition, another modelling concept is used: the *condition* which is shown above the transition. This condition guarantees that an insert event can happen only when there are less than two coins in the slot; to this end, the condition refers to the list of coins of the slot (getCoin()). In general, a condition can refer to the parameters of all involved events and to anything that the element can access via the API of its object-oriented implementation.

The ECNO nets for the other elements are similar. In particular, the ECNO net for the control uses two transitions (see Fig. 6). The first transition is bound to the two events drink and pass, which is the reason why in the large interaction of Fig. 2 must contain both of these events – this way making sure that a drink is only be dispensed together with passing a coin from the slot to the safe. The second transition combines events cancel and reset.

2.4 Discussion and Summary of Concepts

Altogether, the ECNO extends class diagrams by the explicit definition of *events*, resp. *event types*, and coordination annotations, which define in which way different elements need to participate in an interaction. We call this extension on top of class diagrams *coordination diagrams.*

The basic mechanism for defining these coordination requirements is annotating references of the class diagram with an *event type* and a *quantification.* Each of these annotations defines a bilateral coordination. In combination, however, they might require that many different elements participate in an *interaction* (cf. Fig. 2): First, there might be different references for the same event, which require different other elements to participate. Second, the other elements that are required to participate might have references with annotations, which require further elements to participate; in this way, establishing a chain or network of required elements until all requirements are met. Third, an event annotation with quantification ALL requires that all the elements at the other end of the respective links participate. Forth, the local behaviour of an element can require synchronization of two or more different events that all need to be part of the interaction (see Fig. 6 for example); the required other event, may in turn impose additional requirements on participating partners. This way, coordination diagrams define the *global behaviour* of a system by coordination annotations based on the *local behaviour* of its elements.

A coordination diagram does not say anything about the possible local behaviour of the elements. In our example, the local behaviour was defined by an ECNO net; but this could be done with other formalisms too. In essence, the local behaviour answers the following questions: when can an event be executed by an element (to be more precise, when can an element participate in an event), what is the local effect when the element participates in such an interaction, and which events need to be executed together. The ECNO framework provides an API for programming the local behaviour for every element [2]. This code, however, can be generated fully automatically from *ECNO nets*.

3 ECNO Engine: Concepts, Algorithms, and Architecture

In this section, we discuss the main concepts and the architecture of the ECNO execution engine: the algorithm for calculating interactions, the control mechanisms for automatically updating possible interactions, and their execution.

3.1 Local Behaviour

In Sect. 3.2, we will briefly discuss how the execution engine calculates the possible interactions for some element. The possible interactions depend on the information from the coordination diagram concerning global behaviour as well as on the local behaviour. In our example, we used ECNO nets for defining the local behaviour of an element. In order to be independent from a specific modelling notation, however, the ECNO provides a general programming interface for the local behaviour, which was discussed in [2]. For the calculation of the possible interactions this programming interface is used. Here, we briefly recapitulate the main functions of this interface.

The local behaviour of an element is represented by an interface ElementBehaviour. Its most important method provides, in any given situation, a list of all possible *choices*, each of which is represented by an instance of class Choice. In ECNO nets, each enabled transition would correspond to a choice. In general, a choice defines events of which type are involved in the choice and methods for assigning values to the parameters of the involved events. Since the value assigned to a parameter of some event might depend on other event parameters, the details are slightly more involved. The details of the parameter assignment, however, are not relevant for understanding the computation of the possible interactions. Therefore, we do not discuss this here (see [2] for more details). The class Choice also has a method that, after all parameters have been assigned to the events, checks the additional condition (as we have seen in Fig. 5). At last, each Choice provides a method to execute the choice; this will make all the local changes to the element, when the interaction eventually is executed. In ECNO nets, this would be the change of the net's marking as well as the execution of the transition's action.

3.2 Computing Interactions

Next, we discuss how the ECNO engine computes all the possible interactions for a given element and for a given event type. The basic idea of this algorithm is quite simple: it is a search in an AND/OR-Tree, which starts from the element together with an event type, following systematically up all the possible alternatives. When the search starts or arrives at an element with some event type, the possible alternatives are all the possible choices of the element with an event of the respective type. Each of these choices represents one possible interaction that needs to be followed up. Once a choice is fixed, we need to follow up all the event types of the choice; note that for a choice with more than one event type, we need to follow up all of them for the same interaction; therefore, there is no alternative here. Following up an event type for an element means that we need to follow up all the coordination annotation for that type – again all of them need to be followed for a single interaction. Following up a coordination annotation has two different cases: if the coordination annotation has cardinality `1`, each link of the element with respect to that reference is a possible alternative. If the coordination annotation has cardinality `ALL`, there is no alternative; we need to follow up all of the respective links of the element. Following a link, will lead us to an element and an event type again, were the search continues. If this element is already part of the currently computed interaction, we just need to check whether the respective event type is already part of the choice for that element. If it is, this branch of the search is successful. If the event type is not part of that choice, the currently computed interaction fails unsuccessfully.

During the computation, a potential interaction is represented as a set of elements that are determined to be part of the potential interaction along with the respective choice of the local behaviour for that element and the events. We call this a *partial interaction*. A partial interaction is complete, if all search branches terminate successfully. Since the computation, ultimately, should be able to compute all possible interactions for a given element and event type, the search is actually not done recursively. It is done iteratively, where the stack of all possible alternatives is stored explicitly along with each partial interaction in a list. Whenever the search has more than one alternative, the partial interaction along with its current stack of alternatives is copied for every possible new alternative and the new alternatives are put on top of the stack.

The above algorithm will systematically compute all interactions that are possible from the coordination point of view. For simplicity, the event parameters will not be assigned during the search in the current implementation. The parameters will only be assigned at the end of the search. This assignment could result in failures too, when different elements assign different values to the same event parameter. If the assignment of the event parameters was successful, the conditions of all the choices are checked. If this is successful for all the choices of all the interaction's elements, the interaction is actually *valid.* Only in that case, the computed interaction will be returned as a result.

Note that, technically, the ECNO engine does not compute and return the set of all valid interactions in a single go. It returns an iterator that, on demand, computes and returns one valid interaction after the other. We call this the *interaction iterator.*

Executing an interaction is very simple, once it is computed: The execution method on the choice attached to every element of the interaction is called one after the other – in an arbitrary order.

3.3 Controllers, Updates, and Notifications

Based on the algorithm from Sect. 3.2, the ECNO engine provides a method that returns an interaction iterator that can compute all currently possible interactions for some element and some event type. Other parts of the program can obtain these interaction iterators from the engine, obtain interactions and then execute them. The idea of ECNO, however, is that there is no need for a program that globally controls and selects the possible elements and events, and which executes interactions. The idea is, that as soon as some interaction is possible, it would be offered at some GUI so that the end-user could select it – and it should automatically be disabled again when the interaction becomes invalid again. To this end, ECNO has some features that allow us to indicate which kind of elements and which event types should show up on the GUI.

But, it should not only be possible to trigger interactions from the GUI; also other parts of the software should be able to trigger an interaction on some element, once it become enabled. Moreover, this should work, even when objects are added, removed, or its links are changed by some other parts of the software – even when these changes are made independently from the ECNO engine. To this end, the ECNO framework provides *controllers*, which can register for some element and event type, and which are automatically notified when interactions become enabled or disabled. The GUI mentioned above is just one specific extension of these ECNO controllers. Other applications can use and extend these controllers to be notified about enabled interactions and for issuing their execution.

In order to make the controllers aware of any changes in the underlying objects and their links, ECNO makes use of some notification and listener mechanisms. Here, we give an overview of these mechanisms and their interplay with the controllers. For lack of space, however, we cannot go into the subtle details of making these mechanisms re-entrant and thread-safe – most of which use standard concepts of concurrent programming.

Since ECNO should work independently from the underlying implementation of object-oriented models, the listeners actually register with the behaviour on top of the object-oriented part of the software. In order to be aware of changes in the underlying object-oriented parts, adapters need to be implemented, which will be discussed in Sect. 3.5. For now, we assume, that the element's behaviour is notified of any change that might have an effect on the enabledness of some interactions it could be involved in.

Note that also an interaction iterator must be aware of changes that potentially invalidate the interactions it is computing. To this end, an interaction iterator (see Sect. 3.2), registers itself as a listener with any new element, as soon as it comes across it during the computation of its interactions. As soon as the iterator receives a notification from one of the elements it had registered with, it assumes that its computed interactions and partial interactions are invalid; and it will not return any interactions anymore. Instead, it will raise an exception, when asked for another interaction. Moreover, the iterator de-registers itself from all the elements, since it is invalid already and nothing can change that.

In turn, a controller registers as a listener on the interaction iterator when it obtains it from the engine. This way, the controller is notified as soon as the interaction iterator becomes invalid. In that case, the controller, will obtain a fresh interaction iterator from the engine. This way, the controller stays up to date – and can select a new valid interaction from the new iterator.

Note that an iterator that did not find any interaction, remains registered with all elements it came across in its search for possible interactions. The reason is that any change in these elements could result in new possible interactions – and only changes in these elements can make new interactions possible. This way, an "empty" interaction iterator can notify a controller about possible new interactions, once they become enabled. Then, the controller can create a new interaction iterator which provides it with the currently possible interactions.

Once a controller has obtained a possible interaction, it should also be notified when this interaction becomes invalid. Therefore, the interaction iterator registers an interaction with all the elements the interaction is involved in before the interaction is returned to the controller. In turn, the controller registers with the interaction that it obtained from the interaction iterator, so that the controller can be updated when this interaction becomes invalid.

As long as such an interaction is not invalidated, the controller could execute this interaction anytime. Since changes in the objects and their structure can be made concurrently from different threads, it could happen that an interaction is invalidated while it is executing; to avoid this, the ECNO engine comes with a transaction mechanism, which will be discussed in Sect. 3.4.

The above mechanism result in a lot of notifications, which in turn would result in many re-computations of possible interactions in the respective controllers. The execution of a single interactions will make many changes: changes of attributes of the involved elements, state changes in the local behaviour of every element, and addition or deletion of links between some elements. Updating the controllers after every of these intermediate changes would just waste computation power. Therefore, notifications that happen due to changes of an interaction should be deferred until the execution of the interaction is finished. To this end, the notifiers of the ECNO engine are implemented in such a way that, when they encounter a change resulting from the execution of an interaction, they do not send out the notifications right away; they will register themselves with the end of the interaction (actually with the end of the respective transaction – see Sect. 3.4), and send out only one notification to all listeners in the end. We call this a *deferred notification*.

3.4 Transactions

As discussed before, controllers should be as independent from each other as possible, it should be possible that controllers run concurrently in different threads, and interactions should be computed and executed concurrently. As long as the sets of elements that are involved in two interactions are disjoint, there is no harm in executing interactions concurrently. When two interactions have elements in common, however, this might cause some problems. The change made by an action of one element might make the condition of the same element in the other interaction invalid; likewise, links that are part of an interaction might be deleted by the other, etc. The ECNO engine should make sure that interactions do not interfere with each other. In the sense of the ACID principle, we want to make sure that an interaction runs *atomically* (either completely or not at all) and in *isolation* (no other interaction interferes with it once it is started).

To this end, the ECNO engine introduces a transaction concept with a simple locking mechanism. When the execution of (a still valid) interaction is started, a transaction will be started and locks on all the elements of this interaction will be acquired. Acquiring the locks is actually done in some canonical order, in order to avoid deadlocks due to cyclic waiting for a lock. This way, we are sure that an interaction resp. its transaction eventually will be able to acquire all locks. Once the transaction has obtained all locks, the interaction will be executed – and we can be sure that it terminates. When the execution of the interaction is finished, the locks on all its elements are released again. Note that, this way, ECNO's transaction mechanism provides slightly more than atomicity: once successfully started[2] the interaction will successfully terminate.

As long as interactions are executed by controllers derived from ECNO controllers, interactions are executed atomically and in isolation – even when these controller are running concurrently. When other parts of the software make changes on the objects, they should also use transactions acquiring locks on all changed elements. If they do, ECNO's engine guarantees atomicity and isolation. If changes are made outside of transactions, ECNO's updating and notification mechanism would still work. But, atomicity and isolation might be compromised.

As pointed out in Sect. 3.3 already, transactions are also used for another purpose: In order to avoid unnecessary updates of the possible interactions of the controllers, the notification of the invalidation of an interaction iterator or an interaction is deferred to the end of the transaction (interaction), when a change on the element is made from within a transaction.

[2] It might happen that, while the transaction is still acquiring the locks, another interaction invalidates the interaction. In that case, the interaction will abort. So, "successfully started" for an interaction means, that the interaction is still valid after the transaction has acquire all locks.

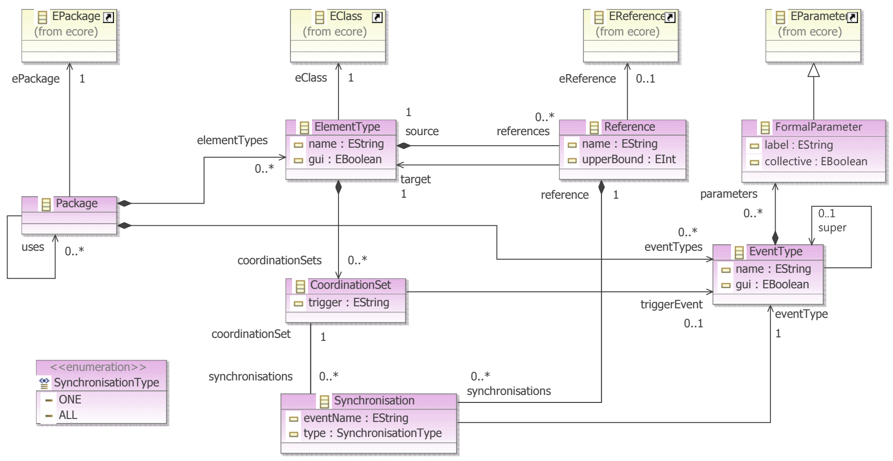


Fig. 7. ECNO model and EMF adapters

The ACID principle of transaction theory would also cover persistence or *durability* of changes in data – meaning that the changes would be persisted in a file or database, once the interaction is successfully completed. The ECNO engine does not (yet) take care of that. The reason is that persisting data is tightly coupled to the technology underlying the data model. Therefore, the implementation of durability depends on the underlying object-oriented technology. For models in EMF technology, for example, we are planning to use Hibernate for that purpose. A general infrastructure for persisting object-oriented models spanning different technologies, however, is beyond the scope of ECNO for now.

3.5 ECNO: Meta Model and Adapters

The concepts discussed up to now are independent from any technology and implementation issues. One of the objectives of ECNO was that models for local and global behaviour could be added on top of any other object-oriented technology. In this section, we discuss how this technology independence is achieved in the ECNO engine and programming framework.

ECNO comes with its own light-weight meta model of object-oriented models, on top of which the additional concepts for coordination diagrams are introduced. For a particular object-oriented technology, there will be a meta model that maps the concepts of ECNO's light-weight meta model of the object-oriented concepts to the concepts of the specific technology. Figure 7 shows the meta model of ECNO and the mapping of its object-oriented concepts to the corresponding concepts of the EMF technology. In the top row, you see the concepts of EMF, which we ignore for a moment. The rest of the diagram shows ECNO's meta model. Actually it shows already the mapping, but since there is almost a 1:1-correspondence, we use this diagram to explain the concepts of ECNO's meta model, which technically is just a set of Java interfaces (not visible in this diagram).

The main concept of ECNO's meta model for objects are the *element types*, which can have *references* to other element types. Note that there are no multiplicities for references, since these would come from the actual object-oriented model (the EMF model in our case).

The concepts specific to coordination diagrams are coordination sets (which reflect the different possibilities an *event type* requires synchronisations with). This is represented by the *synchronisations* that are contained in a reference and are attached to exactly one coordination set of an element type. Each synchronisation represents the coordination annotations of a reference (cf. Fig. 1), which consists of an event type and a *synchronisation type* (either ONE or ALL). Note that an event type can be derived from another event type (in the meta model represented by super) and event types also can have parameters.

As mentioned above, Fig. 7 shows how the ECNO concepts are mapped to the Ecore concepts: most importantly, an element type is referring to an EClass, and an ECNO reference is referring to an EReference.

The important point here is, that the ECNO mapping refers to concepts of the Ecore model, but there are no references in the opposite direction. Therefore, the ECNO engine works on code that was generated from the Ecore model. The Ecore objects, however, do not know anything about ECNO's element types, coordination sets, or synchronisations. In order for the ECNO engine to be able to obtain that information when it calculates the possible interactions, the ECNO engine uses so-called *package adapters*. For the EMF technology, these adapters can be generated automatically from the coordination diagrams.

A package adapter, basically, does the following: for a given object, it will check whether the object is of a type for which it provides a mapping; if so, it returns the object's element type. Based on the object, its element type, and some of its references, the adapter will provide all the links of that object with respect to that reference. This will be used by the engine to navigate to the respective related elements when computing the interactions.

Another important function of the package adapter is to create an object that represents the element's local behaviour (ElementBehaviour) the first time the engine encounters a new object. The ECNO engine will then use this element behaviour throughout the life-cycle of this object. The main concepts of the element behaviour have already been explained in Sect. 3.1. The package adapter is responsible only for creating it, when a new object is encountered.

As mentioned above, the package adapter can be automatically generated from a coordination diagram. The relevant package adapters are then registered with the ECNO engine, when it is started.

For the prototype version 0.2.0 of this ECNO implementation and some examples, we refer to the ECNO home page `http://www2.imm.dtu.dk/~eki/projects/ECNO/`.

4 Conclusion

We briefly discussed the goals and objectives of ECNO, and its main ideas and concepts. The ideas of ECNO and the related work have been discussed before with the focus on the programming interface for local behaviour [2] and with the focus on ECNO nets for modelling the local behaviour [3].

In this paper, we presented the ECNO engine and its architecture and design in order to show that ECNO also technically can be used on top of different object-oriented technologies. By the use of adapters, the ECNO engine can be integrated with different technologies.

In addition, we have discussed the main ideas of the algorithm for computing the possible interactions, and the control mechanisms that make it possible to update and execute interactions completely independently of each other. No additional control code is needed; the ECNO engine provides all the control mechanisms for making the interactions update and execute – just triggered by changes on the underlying model. The concepts of events and interactions allow systems to be integrated without using method invocation or function calls at all – and without explicitly thinking in terms of threaded programming.

The algorithm for computing the possible interactions has still much potential for optimizations. Since interactions are typically not very large and since the computation of interactions is local, the complete approach should scale in principle. Still, optimizations would depend on the kind of application, the size of typical interactions, how different interactions are intertwined, and on the timescale of their execution. Getting more experience with that, would require case studies that are larger than our simple vending machine. With the current version of the ECNO engine, we can now start working on examples – and case studies – on a large scale, which would provide the necessary input for optimizations and an evaluation of the ECNO approach in practice.

Acknowledgements
I would like to thank David Schmelter for comments on earlier versions of this paper.

References

1. Kindler, E.: Model-based software engineering: The challenges of modelling behaviour. In Aksit, M., Kindler, E., Roubtsova, E., McNeile, A., eds.: Proceedings of the Second Workshop on Behavioural Modelling - Foundations and Application (BM-FA 2010). (2010) 51–66 (Also published in the ACM electronic libraries).
2. Kindler, E.: Integrating behaviour in software models: An event coordination notation – concepts and prototype. In: Third Workshop on Behavioural Modelling - Foundations and Application (BM-2011), Proceedings. (2011)
3. Kindler, E.: Modelling local and global behaviour: Petri nets and event coordination. In Duvigneau, M., Moldt, D., Hiraishi, K., eds.: Petri Nets and Software Engineering. International Workshop PNSE'11, Newcastle upon Tyne, UK, June 2011. Proceedings. Volume 723 of CEUR Workshop Proceedings. (2011) 42–56
4. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. 2nd edition edn. The Eclipse Series. Addison-Wesley (2006)

Motivation Modelling for Human-Service Interaction

Ella Roubtsova

Open University of the Netherlands
ella.roubtsova@ou.nl

Abstract. Web services are goal-oriented software systems and often need to influence or motivate particular behaviour of their communication parties: humans and other services. This paper investigates modeling of motivation for human-service interaction. It shows why motivation needs a separate model different from the service process model, how to specify motivation and compose the motivation model with the service process model. Depending on the goals, the same service process model may have different motivation models. We provide an example of a service model with different motivation models that stimulate different behaviour of humans interacting with the web service.

Keywords: Motivation Model, Service, Process, Protocol Model

1 Introduction

In 2010, the Object Management Group and the Business Rules Group completed their work on the Business Motivation Model (BMM), Version 1.1 [13, 15]. The BMM provides a scheme or structure for developing, communicating, and managing business plans. The schema covers four related elements:
1. The *Ends* of a business plan. "Among the *Ends* are things the enterprise wishes to achieve, for example, *Goals* and *Objectives*" [15].
2. The *Means* of a business plan."Among the *Means* are things the enterprise will employ to achieve the *Ends*, for example, Strategies, Tactics, Business Policies, and Business Rules".
3. "The *Influences* that shape elements of a business plan".
4. "The *Assessments* that are made about the impacts of such *Influencers* on *Ends* and *Means* (i.e., Strengths, Weaknesses, Opportunities, and Threats)."

The OMG predicts that "three types of people are expected to benefit from the Business Motivation Model: developers of business plans, business modelers, and implementers of software tools and repositories". The Business Rule Group believes that "Eventually specifications such as the Business Process Modeling Notation (BPMN) together with the Business Motivation Model (BMM) should be merged into a single business-oriented modeling architecture, and implemented in integrated tool suites"[15]. The BMM is not a full business model and it does not prescribe in detail business processes, workflows and business vocabulary. However, business processes are key elements of business plans and the BMM does include a placeholder for Business Processes. The relations between *Goals* and other elements of BMM are left open.

In this paper we make a step in direction of relating BMM with business processes and show how business modelers can benefit from motivation modelling. The motivation modelling is especially important for the modern electronic business that covers any area of human life. Web services govern job application, purchasing orders, booking requests, testing and requesting official documents - the list of web services is endless. Depending on their goals, web services need to motivate their users to choose particular actions among all possible actions. If business services are provided by people, these people motivate actions of customers. The web services themselves favour the choices of their customers and therefore they should benefit from having well designed motivation models built into them. This motivation of users is a some sort of intelligence that we need to add to services. The first step to systematic use of this intelligence is propagating the business *Ends (Goals and Objectives)* to the business process.

Goals are usually formulated as non-functional requirements. They are usually abstract. The goals can be even unrealisable. The *Objectives* corresponding to goals are specific and measurable. They show realisability of goals. The motivation modelling can be seen as transformation of *Goals* into the corresponding *Objectives* described using elements of business processes and as a way to estimate realisability of *Goals*.

This paper presents a model of Business Motivation in Business Processes. We show how the semantics of Protocol Modelling [10] allows for localizing the motivation model in the business processes.

The structure of the paper is the following.

In Section 2 we formally define a process and introduce a motivation model on a process. Section 3 shows how to propagate the business goals and combination of goals to the motivation model of the process. Section 4 discusses the advantages of our approach to motivation modelling. Section 5 discusses related work. Section 6 concludes the paper and identifies future work.

2 Motivation Modelling

2.1 Process with can-semantics

In order to relate motivation and business process, we need a model of a process, a state transition system. We take a state transition system which is usually presented as a triple of $P = (S, A, T)$, where
- S is a finite set of states $\{s_1, ...s_i, ...s_j...\}$,
- A is the alphabet of P, a finite set of environmental actions or events ranged over $\{a, b, ...\}$,
- T is a finite set of transitions (s_i, a, s_j).

The set of transitions can be presented as a set of two relations [11]

$$T = \{C, U\} :$$

- $C \subseteq (A \times S)$ is a binary relation, where $(a, s) \in C$ means that action a is a possible action for P when in state s. C is called the *can-model* of P because it models the actions that P "can do" in each state.
- U is a total mapping $C \to S$ that defines for each member of C the new state that P adopts as a result of the action. $U(a; s_i) = s_j$ means that if P engages in action a when in state s_i it will then adopt state s_j. U is called the *update-model* of P because it models the update to the state of P that results from engagement in an action.

With separation of the can- and update-models a process P is a tuple:

$$P = (S; A; C; U).$$

This is the same process, it does not contain motivation yet.

2.2 Motivation Modelling

There sequences of states in the process that lead to achievement of a particular goal. There are also sets of states when a particular goal is achieved. We name these states goal states.

From the goal perspective the events triggering transitions to a goal state are the priority actions in the states preceding the goal state. They need to be motivated. So, a state preceding a goal state and the event that triggers a transition that may lead to the goal state, form a new binary relation:

– $M \subset (A \times S)$, $(a; s) \in M$ means that event a is a motivated action for P when in state s. We call relation M the *motivation-model* to show its semantic difference from the relation C [8].

In order to model motivation we propose to add the motivation-model M to the process:

$$P_M = (S; A; C; U; M).$$

The can- and motivation-models of a process are independent of each other, so when a process is in a given state, an action can have different combinations of can- and motivation- alternatives:

$$\{\textit{can happen};\ \textit{can not happen}\} \times \{\textit{motivated}; \textit{not motivated}\}$$

In this paper we base the modeling of motivation on this extra relation M added to the process.

2.3 Human-Computer Interaction

As motivation-models do not contribute to behaviour of the systems but motivate the human communication with the service, the most simple application of motivation models is the justified design of human-computer interfaces.

A service presents to a human the possibilities and wishes in form the can- and motivation-model. For example, two events can be submitted, but only one of them is motivated:

$$((a; s_P) \in C) \wedge ((b; s_P) \in C) \wedge ((a; s_P) \in M).$$

The human can choose any possible action, but the action indicated by the motivation-model leads to achieving a goal of the service:

$$((a; s_P) \in C) \wedge ((a; s_P) \in M)$$

Having a chosen goal in mind it is possible to favour the paths leading to the goal states by indicating motivated actions in any state of the process.

2.4 Several Goals

The goals can be OR-composed or AND-composed [14] in requirement specifications. In this case several motivation-models should be taken into account.

Two goals forming an AND-composition are conflicting if the system has a state from which it is impossible to reach a state where both goals are satisfied simultaneously. In the case of conflicting goals the motivation model should identify the subsequences in the process that lead to states from which achieving of all AND-composed goals is possible.

2.5 Requirements for a Semantics that Relates Motivation and Behaviour Models

Process P_M contains a new relation $M = (A \times S)$.

Usually $M \subseteq C$ and in this case the semantics of M means highlighting the transitions that lead to the goals state.

However, the new goals emerging in the life cycle of the modeled system may challenge the can-update process and may need transitions to other states caused by both the events from the alphabet A and new events. In this case the motivation model introduces a new subprocess.

Conventional behaviour modelling techniques use only can-update semantics and therefore they do not provide means for motivation modelling.

For example, if a process is presented as a workflow, as an activity diagram, then, to specify a motivation-model, extra means are needed to identify the motivated outgoing transitions in each state. For, example, we can colour the motivated transitions leading to a goal. The state of a workflow is a set of marked nodes, so the combinations of nodes have to be built to formulate a motivation-model. If several motivation-models should be presented, then an incomprehensible spaghetti of coloured sub-diagrams will cover paths of the workflow. The events submitted by the human are accumulated in this model as tokens in places. The tokens are handled in a non-deterministic way and handling of them cannot be motivated.

In another semantics, when a system is specified as a composition of communicating state machines, the model often contains states that cannot be described as composition of states of composed state machines. Such states appear because the semantics of state machines includes queues to keep the events which were submitted to the system when the system was not able to accept them. Events in queues are waiting for acceptance and may affect the motivated transitions in any state. The non-determinism takes control of the process from the human.

These observations lead to requirements for the modeling semantics that relates Motivation and Behaviour Models.

- The semantics should allow separating can-update and motivation models.
- The semantics should present all states and transitions of the process explicitly. The states should be visible as they are used for specification of goal states and for motivation modelling. Having invisible states in the model the user loses control over the process. The transitions should be visible as they are used for specification of motivation.
- For the purpose of transformation of a goal into a objective the semantics should be able to present abstractions on sets of goal states.
- The semantics should present deterministic models as the motivation specifies the choice of transitions. The human communicating with the model should be able to choose, not the machine. Any queues of events submitted by the human and the non-determinism caused by them take control from the human.
- The determinism of the semantics should not restrict the concurrency of the model as both can-update and motivation models should work in parallel in the model.

3 Motivation Models In Protocol Modelling

The semantics of the Protocol Modelling approach [10] offers an easy and practical way to model motivation separately from the can-update model of the process. The Modelscope tool [9] supporting Protocol Modelling enables execution of the can-update models with motivation models.

A Protocol Model is a synchronous CSP parallel composition of protocol machines [10]. This composition has its roots in the algebra Communicating Sequential Processes (CSP) proposed by Hoare [4]. McNeile [10] extended this composition for machines with data.

The CSP parallel composition means that a Protocol Model accepts an event if all the protocol machines recognizing this event accept it. Otherwise the event is refused.

A protocol model is deterministic, but this does not restrict concurrency as the synchronized protocol machines work concurrently. A state of a protocol model is always a composition of states of its protocol machines, so there are no states hidden for specification and observation. All the transitions are caused by the events initiated by environment, i.e. humans in our case.

We will introduce the relevant semantics of Protocol Modelling on a simple example. We show how reflect goals in a can-update protocol model, how to model motivation and how the motivation model justifies different human-computer interface for the same can-update model.

3.1 Web service: Pay by Credit Card.Goals and Requirements

Our simple case study is a *Pay by Credit Card* web service that can be seen in many electronic booking systems. The user of the service instantiates the service. The user is asked accept the privacy conditions of the service and to fill in his credit card number. The user may fill in the credit card number without accepting the privacy conditions and after accepting the privacy conditions. If the user has filled in the credit card number, he is not able to accept the privacy condition anymore (the service does not have the strict rule "first accept and then fill in"). The service can always be cancelled before the credit card number is filled in.

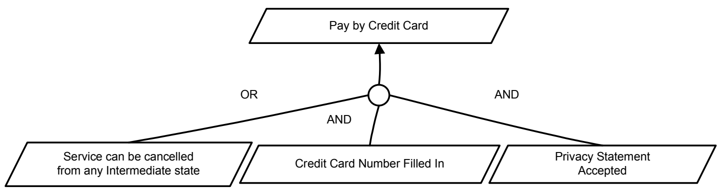

Pay by Credit Card
OR
AND
AND
Service can be cancelled from any Intermediate state
Credit Card Number Filled In
Privacy Statement Accepted

Fig. 1. Goal model of the web service: Pay by Credit Card

The goal-oriented methods for requirements engineering (GORE) emphasize the relations between the goals and requirements [3]. Usually, the top of a goal tree represents abstract goals that are refined with sub-goals and requirements. Requirements are the leaves of goal trees. In any set of requirements there are goals and other concerns.

The goal of a seller having the service `Pay by Credit Card` is to receive payment. This goal is refined by the sub-goal "Pay by Credit Card". We don't show the complete goal tree. Figure 1 shows a sub-tree of the goal tree in the notation of the KAOS method [3]. We recognize two goals (requirements) for this service, namely,

1. credit card number is filled in and
2. privacy statement accepted by the user.

The possibility of service cancelation is yet another concern. It is obvious that cancelation cannot be called a goal of the service.

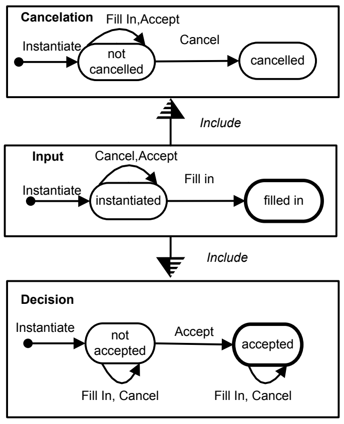


Fig. 2. Can-Update-Model: Pay by Credit Card

```
MODEL Pay by Credit Card
OBJECT Input
    NAME Session
    ATTRIBUTES Session: String,
    Credit Card Number: Integer
    STATES instantiated,filled in
    TRANSITIONS @new*Instantiate=instantiated,
                instantiated*Fill In=filled in,
                instantiated*Accept=instantiated,
                instantiated*Cancel=instantiated

BEHAVIOUR Decision
    STATES instantiated,
    not accepted, accepted, final
    TRANSITIONS @new*Instantiate=not accepted,
                not accepted*Accept=accepted,
                not accepted*Cancel=not accepted,
                not accepted*Fill In=not accepted,
                accepted*Cancel=accepted,
                accepted*Fill In=accepted,
BEHAVIOUR Cancelation
    STATES not cancelled, cancelled
    TRANSITIONS @new*Instantiate=not cancelled,
                not cancelled*Cancel=cancelled,
                not cancelled*Accept=not cancelled,
                not cancelled*Fill In=not cancelled,
EVENT Instantiate
    ATTRIBUTES Input:Input, Session:String,
EVENT Fill In
    ATTRIBUTES Input: Input,
    Credit Card Number: Integer,
EVENT Accept
    ATTRIBUTES  Input:Input,
EVENT Cancel
    ATTRIBUTES  Input:Input,
```

Fig. 3. Meta code of the Can-Update-Model: Pay by Credit Card

3.2 Process Model of the Service

We model the can-update process as a CSP composition of protocol machines corresponding to goals and concerns: `Input`, `Decision` and `Cancelation`. The graphical presentation of the protocol model is shown in Figure 2. The executable meta code is presented in Figure 3.

The CSP composition of protocol machines allows us presentation abstract goal states. State `filled in` of protocol machine `Input` is the goal state of the first goal and state `accepted` of the protocol machine `Decision` is the goal state of the second goal. The state space of a protocol model is a subset of the Cartesian Product of the states of allprotocol machines. Every tuple from this subset of states that contains a goal state is a goal state.

For example,
`(not accepted, not cancelled, filled in)`,
`(accepted, not cancelled, filled in)`
are the goal states for the first goal.

The manually written meta code describing protocol machines (Figure 3) is executable in the Modelscope tool. The generic user interface is generated allowing submission events and observing the state of the model in form of visible states and attributes.

The protocol machine `Input` is coded as `OBJECT`; every instance of it has its identification name. The protocol machines `Decision` and `Cancelation` are `BEHAVIOURS`. This means that their instances do not have identification names. `BEHAVIOURS` present only parts of objects behaviour (so-called mixins). They are included into each instance of object `Input`. This is shown as `INCLUDE` relations depicted as arcs with half-dashed ends. If a `BEHAVIOUR` and an `OBJECT` have an `INCLUDE` relation than for any instance of this `OBJECT` an instance of this `BEHAVIOUR` is generated and the traces of this instance are CSP composed with the traces of the `OBJECT`.

A human interacts with the service and with the protocol model by submitting events. Each protocol machine has an alphabet of recognized events. The events recognized by protocol machines are specified as data structures. Each instance of an event type contains own values of specified types. For example, each instance of event `Fill In` contains own identifier `Input:Input` and `Credit Card Number: Integer`.

All three machines `Input`, `Decision` and `Cancelation` are synchronously instantiated accepting event `Instantiate`.

Similar to a state machine, a protocol machine has a set of states and the local storage presented with attributes. However, the semantics of a protocol machine is different:

– A transition label of a state machine presents the pre-condition and the post-condition for enabling event to run to completion. A transition from state s_1 to state s_2 is labeled by

$$(s_1,\ [\mathit{precondition}]\ \mathit{event}/\ [\mathit{postcondition}],\ s_2)[12].$$

The label shows that the transition in a state takes place only if the pre-condition is satisfied. If the pre-condition is not satisfied, the behaviour is defined by the semantic rules. Namely, the event is kept in a queue and waits for a state change to fire the transition.

– A transition label of a protocol machine presents an *event* that causes this transition. The storage information is localized in the state. Being in a quiescent state in which the protocol machine can accept the submitted event, the protocol machine accepts one event at a time and handles it until another quiescent state. If the protocol machine cannot accept the event in its current state, the event is *refused* [7, 10].

The default type of protocol machines is `ESSENTIAL`. Essential protocol machines are composed (synchronized) using the CSP parallel composition and these machines are used to present the can-update model, i.e. the the business process.

3.3 Protocol Machines of Motivation Models

There are some semantic properties of Protocol Modelling that allow for localization of motivation models and separation them from the can-update model.

1. Thanks to the abilities of protocol machines to read but not modify the state of other protocol machines and to have an associated state function, it is possible to build protocol machines with derived states.
   A *derived state* is a state that is calculated from the states of other machines using the state function associated with the protocol machine.
2. Thanks to different types of protocol machines, the use of composition can be changed. The protocol machines of type `DESIRED` are not composed using the CSP parallel composition technique. These machines can be used to model the motivated behaviour.
3. It is also important that the refusal of events that arrive, when the system is not able to accept them, guarantees that any state of a protocol model is always described as a composition of states of a final subset of composed protocol machines.

According to the definition given in section 2.2, a state of a motivation model is related to some states of the process. Therefore, a motivation-model is presented as a protocol machine that does not have stored states but only derived states.

A motivation model cannot forbid any transition in the can-update model and does not participate in the event synchronization with the can-update models. Therefore, the motivation models are not composed using the CSP parallel composition and have type `DESIRED`.

Summarizing, we can say that the semantic of Protocol Modelling meets the requirements to specify motivation.

The motivation models for goals 1 and 2 are depicted in Figure 4,5.

The meta code and the corresponding call-back functions in Figure 6 show how motivation models corresponding to each goal are modelled as protocol machines. The code is added manually and it executed together with the meta code of protocol machines in the Modelscope tool.

The meta code presentations of protocol machines `Motivate Insert` and `Motivate Accept` have exclamation marks that show to the Modelscope tool that there are call-back functions in java files with the same names. Each call-back function derives state of the motivation model from the state of the objects and behaviours of the can-update model.

For example, if the state of object `Input` is `instantiated` or the state of the behaviour `Decision` is `accepted` then state `motivate fill in` is derived for protocol machine `Motivate Fill In`.

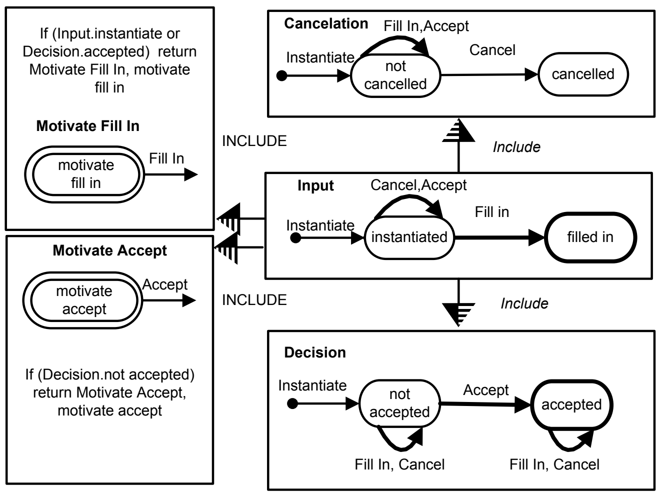


Fig. 4. Graphical Presentation: OR-combination of goals

3.4 Combination of Goals

If the goals are OR-composed then achieving any of the goals is a goal on its own and both call-back functions shown in Figure 6 are CSP parallel composed with the can-update model.

Motivation model of the AND-combination of goals should not direct to states where at least one of goals cannot be achieved.

The motivation model should lead to goals states. The goal state of the AND-composition of goals in our case is

$$\{Input.filled\,in\} \times \{Decision.accepted\}.$$

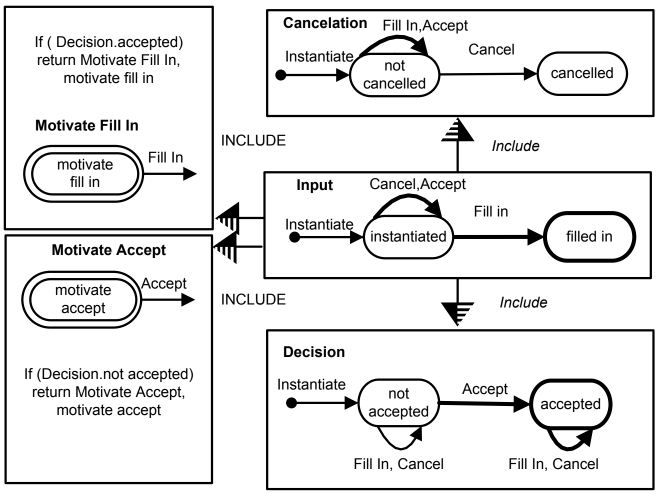


Fig. 5. Graphical Presentation: AND-combination of goals

Event *Fill In* should not be motivated in state

$$\{Input.instantiated\} \times \{Decision.notaccepted\}$$

because acceptance of this event in this state leads to the state

$$\{Input.filled\, in\} \times \{Decision.not\ accepted\}$$

where the privacy statement can not be accepted anymore.

The new call-back function `MotivateFillIn` as shown in Figure 7.

4 Discussion

4.1 Context-Dependent Decisions

It is known from the phycology studies that decisions of people are context-dependent [2]. The human-computer interface may provide the context that leads to the choices that lead to goal states.

The motivation model can be transformed into human-computer interface of different sort: different visual elements, different colour or different position on the screen or another output device. In the generic interface of the the Modelscope tool, the wanted events are presented in green.

The visual elements of the human interface can be generated from the motivation model with the context related to the specified goals. The user of the system gets extra context information to choose the right action.

```
MODEL Pay by Credit Card
OBJECT Input
    NAME Session
    INCLUDES Decision, Cancelation, Motivate
        ATTRIBUTES Session: String, Card Number: Integer
    STATES instantiated,filled in
    TRANSITIONS @new*Instantiate=instantiated,
                instantiated*Fill In=filled in
BEHAVIOUR Decision
    STATES instantiated ,not accepted, accepted, final
    TRANSITIONS @new*Instantiate=not accepted,
                not accepted*Accept=accepted,
                accepted*Rethink=not accepted,
                accepted*Finalize=final,
                not accepted*Finalize=final
BEHAVIOUR Cancelation
    STATES not cancelled, cancelled
    TRANSITIONS @new*Instantiate=not cancelled,
                not cancelled*Cancel=cancelled,

EVENT Instantiate
    ATTRIBUTES Input:Input, Session:String,
EVENT Fill In
    ATTRIBUTES Input: Input, Credit Card Number: Integer,
EVENT Accept
    ATTRIBUTES  Input:Input,
EVENT Rethink
    ATTRIBUTES  Input:Input,
EVENT Cancel
    ATTRIBUTES  Input:Input,
GENERIC Finalize
    MATCHES Fill In, Cancel

```

```
package PayByCreditCard;
import com.metamaxim.modelscope.callbacks.*;
public class MotivateFillIn extends Behaviour {
    public String getState() {
       String y=this.getState("Input");
       String x=this.getState("Decision");
            if (y.equals("instantiated") || x.equals("accepted")
             )  return "motivate fill in";
            else return "other";
            }
}
```

```
package PayByCreditCard;
import com.metamaxim.modelscope.callbacks.*;
public class MotivateAccept extends Behaviour {
    public String getState() {
            String x=this.getState("Decision");
            if (x.equals("not accepted")
             )  return "motivate accept";
            else return "other";
            }

}
```

Fig. 6. Protocol Model with Motivation Model for OR-combination of goals

```
package PayByCreditCard;
import com.metamaxim.modelscope.callbacks.*;
public class MotivateFillIn extends Behaviour {
    public String getState() {
           String x=this.getState("Decision");
            if (x.equals("accepted")
             )  return "motivate fill in";
            else return "other";
            }
}
```

Fig. 7. Call-back function Motivate Fill In for AND-composition of goals.

4.2 BMM and BPM

Relating the OMG Business Motivation and Business Process Models serves to better understanding between managers making strategic decisions and requirement engineers preparing requirements for implementation.

In our approach the `Ends` of the Business Motivation model are presented as abstract goal states.

The `Means` are events included into motivating protocol machines. They present the strategies.

The `Influences` can be modelled as protocol machines. The choice of the objects included into the model as `Influences` is made on the basis of the `Assessments` about the impact of `Influences` relevant for the business process.

5 Related Work

There are many approaches that try to relate goals and processes.

The User Requirements Notation (URN) [5] is a standard that recommends languages for software development in telecommunication. The URN consists of the Goal-Oriented Requirements Language (GRL), based on i* modelling framework [17], and Use Case Maps (UCM) [1], a scenario modelling notation. The GRL provides a notation for modelling goals and rationales, and strategic relationships among social actors [18]. It is used to explore and identify system requirements, including especially non-functional requirements. The UCM is a convenient notation to represent use cases. The use cases are selected paths in the system behaviour and they can be related to goals by developers. The goals are used to prioritize some use cases. If a use case presents alternative behaviours or cycles, then the goals prioritize alternatives. The use cases can be simulated. However, use cases do not model data and the state of the system and they present only selected traces. This means that behaviour model as well as the motivation model shown by use cases are incomplete and cannot be used for code generation.

Letier at al. [6] derive event-based transition systems from goal-oriented requirements models. The goal-oriented models are defined in the well known declarative approach KAOS (Knowledge Acquisition in autOmated Specification) [3].

Goals are specified in Linear Temporal Logic and organized using the AND and OR refinement structures. Then the operations are derived from goals as triples of domain pre-conditions, trigger conditions and post-conditions for each state transition. The declarative goal statements are transformed into the operational model. To produce consistent operational models, a required trigger condition on an operation must imply the conjunction of its required preconditions.

Van at al [16] propose goal-oriented requirements animation. The modelling formalism is the UML State Diagrams that are generated from the goal specifications and called Goal State Machines (GSMs). A GSM contains only transitions that are justified by goals. The GSMs are synchronized through event broadcast. A GSM that can't accept an event in its current state keeps it in a queue. These events will be submitted to goal state machines internally. This means that the composition of GSMs contains extra states that cannot be composed from the states of separate GSMs and can prevent achieving of goals.

The problems are mostly caused by different semantics used by process modelling and goal modelling techniques. Letier at al [6] explained that the operational specification and the KAOS goal models use different formalisms. KAOS uses synchronous temporal logics that are interpreted over sequences of states observed at a fixed time rate. The operational models use asynchronous temporal logics that are interpreted over sequences of states observed after each occurrence of an event. Most operational formalisms have the asynchronous semantics. Letier at al. [6] admit that in order to be semantically equivalent to the synchronous KAOS models, the derived event-based models need to refer explicitly to timing events.

6 Conclusion and Future Work

This paper has presented an approach to motivation modelling. The approach is based on an extra binary relation included into the process model. This relation is used to identify the transitions in the process that lead to goal states.

The presented motivation model uses the semantics of Protocol Modelling which combines the CSP parallel synchronous composition and concurrency and therefore avoids the semantic mismatch between process modelling and goal modelling techniques, identified by Letier at al. [6]. Synchronous goal models can be rendered in protocol models. The motivation model relates the processes to system goals and transforms them into objectives in terms of goal states in the business process of services. Goal states as objectives are specific and measurable and motivation models can make services more effective by motivating actions leading to the goal states.

Reflecting the objectives in the models is important for requirements engineering. New goals can challenge the business process. Such questions as, if the process model supports some needed `Means` or if an `End` is no longer relevant to the enterprise, are the elements of business process analysis [15].

The most interesting direction for future work is connecting web services on the basis of matching motivation models. Motivation model can be used to direct communication of collaborative services and to verify the realizability of service collaboration.

Acknowledgement. The author thanks A.McNeile for sharing ideas and fruitful discussions.

References

1. A. Alsumait, A. Seffah, and T. Radhakrishnan. Use Case Maps: A Visual Notation for Scenario-Based Requirements. *10th International Conference on Human - Computer Interaction, http://wwwswt.informatik.uni-rostock.de/deutsch/Veranstaltungen/HCI2003/*, 2003.
2. A.Tversky and I. Simonson. Context-Dependent Preferences. *Management Science*, 39(10):1179–1189, 1993.
3. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993.
4. C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
5. ITU. Formal description techniques (FDT). User Requirements Notation Recommendation Z.151 (11/08). http://www.itu.int/rec/T-REC-Z.151-200811-I/en.
6. E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transition systems from goal-oriented requirements models . *Automated Software Engineering archiveD*, 15(2):1–22, 2008.
7. A. McNeile and E. Roubtsova. Composition Semantics for Executable and Evolvable Behavioural Modeling in MDA. *BM-MDA'09*, pages 1–8, 2009.
8. A. McNeile and E. Roubtsova. Motivation and Guaranteed Completion in Workflow. *submittered to SOSYM*, 2011.
9. A. McNeile and N. Simons. http://www.metamaxim.com/.
10. A. McNeile and N. Simons. Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling*, 5(1):91–107, 2006.
11. R. Milner. *A Calculus of Communicating Systems.* volume 92 of Lecture Notes in Computer Science. Springer, 1980.
12. OMG. *Unified Modeling Language: Superstructure version 2.1.1 formal/2007-02-03.* 2003.
13. OMG. Business Motivation Model. Version 1.1.formal/2010-05-01. 2010.
14. K. Pohl and C. Rupp. *Requirements Engineering Fundamentals.* Rocky Nook, 2011.
15. The Business Rules Group. The Business Motivation Model. Business Governance in a Volatile World. 2010.
16. H. T. Van, A. van Lamsweerde, and C. P. Philippe Massonet. Goal-oriented requirements animation. In *RE*, pages 218–228, 2004.
17. E. Yu. Modelling Strategic Relationships for Process Reengineering. *Ph.D. Thesis. Dept. of Computer Science, University of Toronto*, 1995.
18. E. Yu, L. Liu, and Y. Li. Modelling Strategic Actor Relationships to Support Intellectual Property Management. *LNCS 2224 Spring Verlag. 20th International Conference on Conceptual Modeling Yokohama, Japan*, pages 164–178, 2001.

A Metamodelling Approach to Behavioural Modelling

Adrian Rutle[1], Wendy MacCaull[1], Hao Wang[1], and Yngve Lamo[2]*

[1] Centre for Logic and Information, St. Francis Xavier University, Canada
{arutle, wmaccaul, hwang}@stfx.ca
[2] Bergen University College, Norway yla@hib.no

Abstract In this paper we propose a metamodelling approach to behavioural modelling. The approach combines diagrammatic modelling with formal foundations based on category theory and graph transformations. The static semantics of behavioural models is represented by instances of (meta)models, while their dynamic semantics is represented by transition systems. Transitions are described by coupled model transformations. To illustrate the approach, we present a running example of a workflow model for health services delivery.

1 Introduction

Model-driven engineering (MDE) promotes models as the primary artefact of the software development process. Unlike traditional approaches where models are used merely for documentation purposes, the relation between the final results of the development process – the executable software systems – and the models remains synchronized during the software development process (design, maintenance, deployment, testing, etc.). A software model is an abstract representation of some aspect of a software system, such as the system's structure, design, behaviour, etc. In MDE, the software models are automatically transformed to program code. Currently, many different MDE technologies automatically generate code from models. These technologies are particularly suited to specifying the structural aspects of software systems. Generally, the actual behaviour is still programmed manually. Some technologies for behavioural modelling in MDE exist, e.g., [23,12]. However, there are still some challenges to overcome before these technologies fully benefit from MDE. Current approaches are often at a low level of abstraction and lack domain concepts for specifying behaviour [17]. To overcome this issue, the use of domain specific modelling languages (DSMLs) is proposed. However, existing DSMLs for behavioural modelling have limited support for metamodelling and it is difficult to reuse the models and their transformations.

This paper presents a metamodelling approach to behavioural modelling, allowing us to reason about behavioural models at different levels of abstraction. Following the MDE methodology, each state of the software system corresponds to an instance of the model; and, each execution path allowed by the software system corresponds to a sequence of instance transitions. The dynamic semantics of behavioural models is described using a transition system.

To describe the transition system, we employ *coupled model transformations* [27,4], an adaptation of *coupled software transformations* [20]. Coupled transformations are useful in several areas of computer science such as schema evolution, grammar evolution, format evolution, etc. The idea is to couple a model with each of its instances

* Currently W.F. James Chair Professor at St. Francis Xavier University

and define coupled transformation rules which simultaneously transform the model and its instances. We use a particular kind of coupled transformations where only instances evolve (the models stay unchanged). The transitions begin with the *start instance*, representing the start state of the system. By applying the rules, we get reachable instances that are on at least one execution path from the start instance. In this way, we obtain the set of reachable instances, a subset of the set of instances of the model.

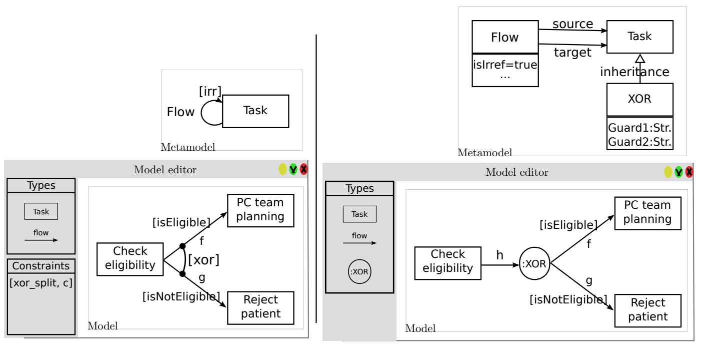
To illustrate our approach, we present a running example of a workflow model for health services delivery. Workflow models are behavioural models used in the development of software to support complex business processes. A workflow consists of steps (called *tasks*) performed by various participants, e.g., persons, robots or software components, in order to achieve a business goal. Despite their popularity, most workflow modelling languages lack a solid formal foundation; e.g., there is no complete formal semantics for all constructs of the Business Process Modelling Notation (BPMN) [7]. In our approach, static and dynamic semantics are well formalised in a metamodelling hierarchy; thus model transformations and code-generation techniques can be used in refinement processes to obtain executable workflow software systems from workflow models. Although we use workflow models in our example, our approach may be adapted to other kinds of behavioural models. Currently we are considering fault tolerance, especially compensation [6], another aspect of behavioural models. We have already considered behaviour wrt. real time information in [29].

In modeling in general and behavior modeling in particular, correctness of models are usually ensured by translating the models into a formalism that provides language and tool support to express and verify semantic consistency conditions [19]. Such support is complicated by the fact that results of verification are given in terms of the formal language and usually requires complex backtracking methods to be represented in the original model because of the semantic gap between the formal method and the original modelling language. Our approach tackles this challenge by combing formal semantics and diagrammatic syntax in one formalism. In this way, correctness of the models can be ensured without translation to another language; we inspect the models with respect to type and constraint conformance, moreover the model transformation system ensures that the models have the intended dynamic behavior.

Section 2 outlines our metamodelling approach. Section 3 presents the semantics of behavioural modelling. Section 4 presents the transition system and discusses some analysis techniques to check the correctness of the models. Sections 5 and 6 present some related and future work and conclude the paper.

2 Metamodelling

A simple metamodelling hierarchy consists of metamodels, models and instances of models. Modelling languages are represented by metamodels, software systems are represented by models, and the possible states of each software system are represented by the instances of the models. The metamodel defines syntax of the modelling language; i.e., it defines the types and relations between types. Each model must *conform to* the language's metamodel; i.e., it must respect the typing and other constraints of the language; we will explain this below. Instances, in turn, must conform to models.



(a) (b)

Figure 1: Two metamodels and their corresponding model editors

To create and modify models, a model editor is constructed from a metamodel. For example, consider a metamodel where we have the general *concepts* Task and Flow (see Fig. 1a). The associated model editor will allow users to define models consisting of specific tasks and flows, e.g., Check eligibility and f, respectively. In order to understand how the represented software system will behave, it is necessary to inspect the instances of the models; instance editors are used to create these instances. In fact, constructing an instance editor from a model is analogous to constructing a model editor from a metamodel since a metamodel is just a model that has the role of being a metamodel wrt. models defined by the associated model editor [21]. In this way we can construct a metamodelling hierarchy in which a model at any level can be considered a metamodel wrt. models at the level below it, known as multi-level metamodelling.

Model editors must deal with constraints at two meta-levels. First, there are *metamodel constraints*; that is, the editor should not allow definition of models which violate the constraints of the metamodel. For example, if in the metamodel we require that flows are irreflexive (see Fig. 1a) the model editor should not allow users to define loops in models. Second, there are *model constraints*; that is, if users want to define models with a satisfactory degree of precision, they sometimes need to add constraints to the models. Instances of these models should satisfy these constraints. Examples of this kind of constraint are routing constraints such as XOR, AND, OR, etc.

Metamodel constraints are usually enforced by the model editor's validation mechanisms. These mechanisms prevent the definition of models which do not conform to the language's metamodel. In most current modelling techniques, model constraints are coded as types in the metamodel; e.g., an XOR constraint is coded as a type XOR which has two guards as its attributes (see Fig. 1b). A model element :XOR (typed by XOR in the metamodel) describes the property that exactly one of the two flows (e.g., f or g in Fig. 1b) can be followed, based on the guards (e.g., isEligible and isNotEligible).

In our approach, model constraints are coded as diagrammatic predicates over models; e.g., in Fig. 1a, the predicate [xor_split,c] is used to describe the same property as described by :XOR, where c is a parameter for the guard (or condition) [isEligible]. We chose this technique since it clearly distinguishes between types (such as Task, Flow, etc.) and constraints (such as XOR, OR, etc.). Defining all model constraints as types, however, will complicate the metamodel. Another commonly used alternative is, despite the graph-based nature of models, to define model constraints using textual languages such as the Object Constraint Language (OCL) [22]. This introduces several challenges to the maintainance of links between constraints and models, especially wrt. model transformations [26]. Use of diagrammatic predicates avoids these challenges.

No matter which approach is chosen for coding constraints, they must be encoded in the software system which will be generated from the model. Most importantly with regard to MDE, to enable reasoning about models – i.e., before code-generation – the semantics of these constraints must be well-defined already in the modelling language. For models, the semantics is all about which structures are qualified as their instances; that is, in the same way a metamodel defines certain restrictions or language requirements and each model which conforms to the metamodel must satisfy these requirements, a model also defines certain domain requirements and each instance which conforms to the model must satisfy these requirements.

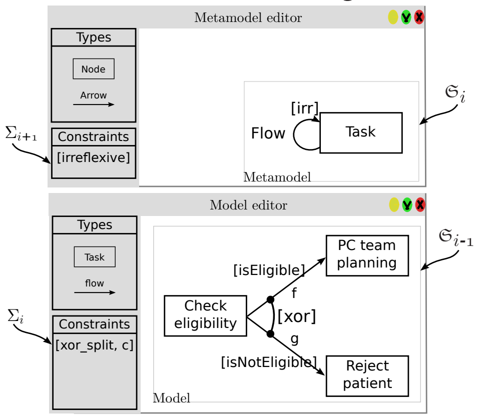
The approach of this paper is based on the Diagram Predicate Framework (DPF), which provides a formalisation of multi-level (meta)modelling and model transformations based on category theory [3] and graph transformations [10]. We briefly review the basic concepts of DPF used for the formalisation of modelling; for details and formal definitions, the interested reader can consult [8,9,25,24,26]. In DPF, a model is represented by a *specification* $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ which consists of a *graph* S and a set of *constraints* $C^{\mathfrak{S}}$ specified by a *predicate signature* Σ. A predicate signature consists of a collection of *predicates*, each having a name and an arity (or shape graph). A constraint consists of a predicate from the signature together with the subgraph of the model's underlying graph which is affected by the constraint; e.g., an `XOR` constraint with a condition c in the model in Fig. 1a consists of `[xor_split,`c`]` and a subgraph of the model which in this case is the whole underlying graph of the model. We use the terms "specification" and "(meta)model" interchangeably.

We define the semantics of a predicate as the set of graphs satisfying the predicate, called the instances of the predicate. For example, for the `[irreflexive]` predicate all graphs which do not include a loop are in the set of its instances. The semantics of a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ is given by the set of its instances. Similar to the semantics of predicates, the set of instances of a specification $\mathfrak{S}$ consists of all graphs which are (i) typed by the underlying graph S and (ii) satisfy the constraints $C^{\mathfrak{S}}$.

To facilitate the discussion of metamodelling hierarchies, we will define the conformance relation between models at adjacent levels of a hierarchy. We distinguish between two kinds of conformance: *typed by* and *conforms to*. A specification $\mathfrak{S}$ is typed by a graph T if there exists a graph homomorphism $\iota : S \to T$, called the *typing morphism*, between S, the underlying graph of $\mathfrak{S}$, and T. A specification $\mathfrak{S}$ is said to conform to a specification $\mathfrak{T}$ if $\mathfrak{S}$ is typed by T and S is an instance of $\mathfrak{T}$; e.g., imagine adding a new flow f' from the task Check eligibility to itself in Fig. 1a; then the model would still be typed by the metamodel, however, it would not conform to it because of the violation of the irreflexivity constraint.

In DPF, a modelling language is described as a modelling formalism $\mathcal{F}_i = (\Sigma_i, \mathfrak{S}_i, \Sigma_{i+1})$. The figure below shows the correspondence between the elements of modelling formalisms and modelling languages as was explained in Fig. 1a. The corresponding metamodel of the modelling language is represented by the specification $\mathfrak{S}_i$ which has its constraints formulated by predicates (e.g., `[irreflexive]`) from the signature Σ_{i+1}.

These constraints should be satisfied by all specifications (e.g., $\mathfrak{S}_{i-1} = (S_{i-1}, C^{\mathfrak{S}_{i-1}} \colon \Sigma_i)$ in the figure) which are specified by $\mathcal{F}_i$. The constructs used for defining constraints at the next level (e.g., [xor_split, c]) which are available for the users of the modelling language are located in the signature Σ_i. As we see from the figure, there is no difference between metamodel editors and model editors. In DPF, a modelling formalism may represent a modelling language at any level of a metamodelling hierarchy (indicated by the use of the subscripts $i-1, i, i+1$) whether it is used for creation of meta-metamodels, metamodels, models, instances, instances of instances, etc.



3 Behavioural Modelling

We now adapt DPF to behavioural modelling. We introduce the needed concepts and illustrate them with a running example of an excerpt of a workflow model for Palliative Care [11]. We start by defining a workflow modelling formalism, then we define the static semantics of workflow models as reachable instances. We will provide a conceptual framework which can be used to describe fine-grained states of software systems, and, may be used to deal with some of the challenges existing in workflow modelling, such as flexibility in the definition of various diagrammatic routing constraints.

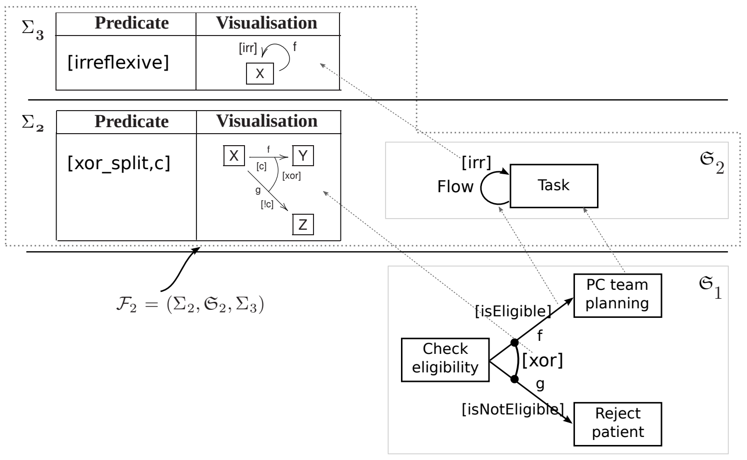


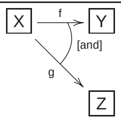
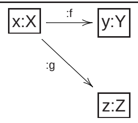
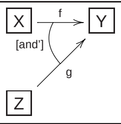
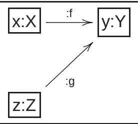
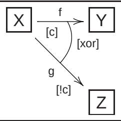
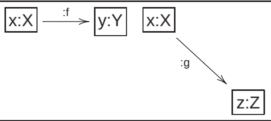
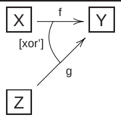
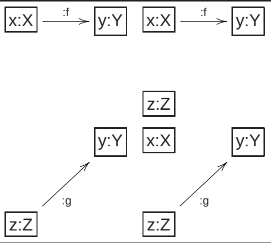
Figure 2: The modelling formalism $\mathcal{F}_2 = (\Sigma_2, \mathfrak{S}_2, \Sigma_3)$ and the workflow model $\mathfrak{S}_1$

Workflow modelling languages provide constructs to define tasks and their routing flows. In this section we introduce the modelling formalism $\mathcal{F}_2 = (\Sigma_2, \mathfrak{S}_2, \Sigma_3)$ used for the specification of workflow models (see Fig. 2). The signature Σ_2 contains predicates for splitting and merging; the metamodel $\mathfrak{S}_2$ includes the types Task and Flow;

the signature Σ_3 constains predicates used to constrain the metamodel. The figure also shows an excerpt of a workflow model $\mathfrak{S}_1$ which is specified by $\mathcal{F}_2$.

Table 1 shows the signature Σ_2 with some predicates useful for workflow modelling. The predicates have spans or sinks of two arrows as arity, and are used to define relations between different flows. In general, these predicates may have spans or sinks of any finite number of arrows as arities, but two arrows suffice to explain the modelling formalism. The predicate `[xor_split,`c`]` indicates that exactly one of the two flows must be followed. The parameter condition c is a proposition that may evaluate to true or false. One of the two flows will have c as a condition, the other one will have the negation of c. The predicate `[and_merge]` is used to indicate that both flows must be followed. To save space, we omit from Table 1 the other usual splitting and merging predicates used in workflow modelling, such as `[or_split,`c_1, c_2`]` and `[or_merge]`.

Table 1: A sample signature Σ_2 used for workflow modelling

p	**Visualisation**	**Semantics (set of instances)**
`[and_split]`		
`[and_merge]`		
`[xor_split,`c`]`		
`[xor_merge]`		

We now illustrate how the modelling formalism can be used to define a workflow model. Fig. 3 shows a simplified version of the team-building workflow model $\mathfrak{S}_1$ used for Palliative Care (PC). The specification $\mathfrak{S}_1$ is compliant with the following:

R1 A patient's eligibility has to be checked *before* any other tasks are performed;
R2 *If* the patient is eligible for PC, then build PC team;
R3 *If* the patient is not eligible for PC, then reject patient;
R4 The patient is *either* eligible *or* not eligible for PC, but not both;
R5 *After* PC team planning, assign *both* a general practitioner (GP) *and* a PC nurse;
R6 *If both* the GP and the PC nurse are assigned, then submit the team information.

In $\mathfrak{S}_1$, R1 is specified by the task Check Eligibility which is the only task with no incoming flows. R2 is specified by the task PC team planning, R3 is specified by the

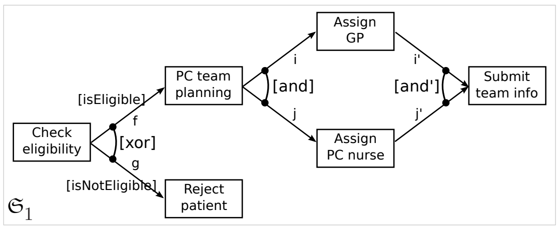


Figure 3: PC team-building workflow represented by the specification $\mathfrak{S}_1$

task Reject patient, and R4 is specified by the two flows f and g, and the constraint `[xor_split,`$isEligible$`]`. R5 is specified by the tasks Assign GP and Assign PC nurse, and the flows i and j with the constraint `[and_split]`. R6 is specified by the task Submit team info and the flows i' and j', and the constraint `[and_merge]`.

In general, the *static semantics* of a model is given by its set of instances, which represent the states of the actual software systems. To analyse a software system which is developed from a model – before the system is implemented – the modelling environment should facilitate instantiation of the model, i.e., creation of instances of the model. Except for EMF [28] which facilitates instantiation of structural models, there is limited support for instance creation in most existing modelling environments.

Recall that the set of instances of a model $\mathfrak{S}$ consists of graphs that are typed by the underlying graph of $\mathfrak{S}$ and satisfy the constraints of $\mathfrak{S}$. To determine whether a graph satisfies the constraints of a model, we need to check the graph against the semantics of the corresponding predicates. The semantics of the predicates in Table 1 is illustrated by listing the set of their valid instances. We use x:X to denote the typing morphism $\iota : x \mapsto X$, and we write :X if x is the only instance of type X. Fig. 4 shows three graphs which are instances of the workflow model $\mathfrak{S}_1$ in Fig. 3.

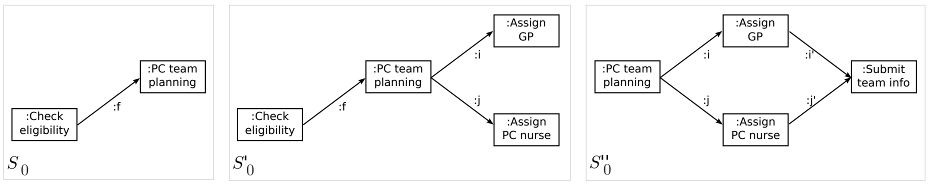


Figure 4: Three sample instances S_0, S'_0 and S''_0 of $\mathfrak{S}_1$ from Fig. 3

As we see from Fig. 4, the graphs alone are not enough to represent the states of the software system. For example, although the graph S''_0 is both well-typed and satisfies the constraints of $\mathfrak{S}_1$, it does not represent a “reachable” state of the system. This is because the flow f from the task Check eligibility to the task PC team building in the workflow model indicates that the task instance :Check eligibility must be “executed” before the task instance :PC team building is “executed”; however, this requirement is not satisfied in S''_0 since the fact that the task instance :Check eligibility does not exist implies that it is not executed yet (see Section 4 for the details). Throughout this paper, we use these conventions: “Task” and “Flow” indicate the concepts at the metamodel level; “task” and “flow” indicate model elements typed by “Task” and “Flow”, respectively; “task instance” and “flow instance” indicate instances of “task” and “flow”, at the instance level, respectively.

Since a behavioural model represents a software system's dynamics, we need to determine the set of "reachable" instances of models; that is, one of the instances (which we call start instance) must represent the start state of the system, and the other instances must be narrowed down to those which are reachable from the start instance. We first extend DPF with the necessary techniques to support the creation of reachable instances. In Section 4 we will give a further explanation of the start instance and the transition rules which are used to generate the reachable instances.

We use a modelling formalism $\mathcal{F}_1 = (\Sigma_1, \mathfrak{S}_1, \Sigma_2)$ to specify the reachable instances of a workflow model $\mathfrak{S}_1$ (see Fig. 5). The signatures Σ_2 and Σ_1 are shown in Tables 1 and 2, respectively. Fig. 5 shows an excerpt of a reachable instance $\mathfrak{S}_0$ of $\mathfrak{S}_1$. $\mathfrak{S}_0$ is again a specification $(S_0, C^{\mathfrak{S}_0} : \Sigma_1)$ consisting of a graph S_0 and a set of constraints $C^{\mathfrak{S}_0}$ formulated by predicates from Σ_1.

The role of these new constraints $C^{\mathfrak{S}_0}$ is to distinguish between reachable and non-reachable instances. At any state of the workflow software system, the task instances, e.g., :Check eligibility in $\mathfrak{S}_0$ in Fig. 3, are either *enabled*, *running or finished*, according to the constraints specified in the workflow model $\mathfrak{S}_1$. We use the signature Σ_1 shown in Table 2 to annotate the task instances in $\mathfrak{S}_0$ accordingly. Moreover, the condition c of the predicate `[xor_split,`c`]`may be evaluated to either true or false; the signature Σ_1 also includes predicates to denote this. As we see from Table 2, the signature Σ_1 has no semantic counterpart since for this modelling environment instances of $\mathfrak{S}_0$ do not have any practical meaning. We call these constraints "annotations" since they are just syntactic markings on the task instances in the specification $\mathfrak{S}_0$.

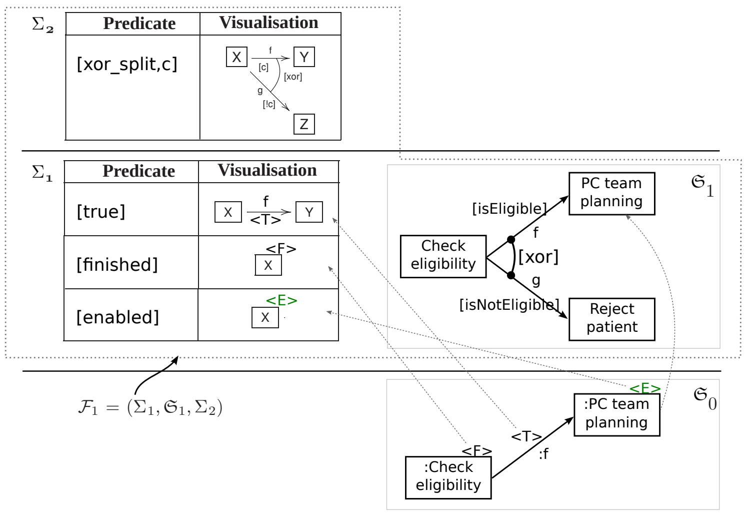


Figure 5: The modelling formalism $\mathcal{F}_1 = (\Sigma_1, \mathfrak{S}_1, \Sigma_2)$ used for creation of reachable instances of the workflow model $\mathfrak{S}_1$, and a reachable instances $\mathfrak{S}_0$

Fig. 6 shows two specifications $\mathfrak{S}_0$ and $\mathfrak{S}'_0$ which are reachable instances of $\mathfrak{S}_1$. In $\mathfrak{S}_0$ both task instances :Check eligibility and :PC team planning are finished (annotated with <F>). In $\mathfrak{S}'_0$ the task instance :Assign GP is running (annotated with <R>), and the task instance :Assign PC nurse is enabled (annotated with <E>).

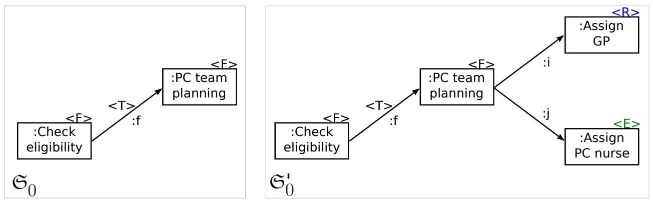


Figure 6: Two reachable instances $\mathfrak{S}_0$ and $\mathfrak{S}_0'$ of $\mathfrak{S}_1$ in Fig. 3

4 Transition System

In this section, we describe the dynamic semantics of worflow models by a transition system. The transition system can be used to calculate the reachable instances of workflow models as well as the transitions between them. The state of a workflow software system is changed according to certain rules during an execution. For example, a task which is enabled may either remain enabled or change to running, a task which is running may either remain running or change to finished, etc. Correspondingly, for the workflow model $\mathfrak{S}_1$, a task instance (in a reachable instance of $\mathfrak{S}_1$) annotated with the predicate `[enabled]` may either remain annotated with `[enabled]` or become annotated with `[running]`; a task instance which is annotated with the predicate `[running]` may either remain annotated with `[running]` or become annotated with `[finished]`, etc. Table 3 shows two rules, t_1 and t_2, which are used to change the annotation of a task instance `x` from `[enabled]` to `[running]` and from `[running]` to `[finished]`, respectively. In rule t_1, the annotation of the task instance `x` is changed in two steps, first the annotation `[enabled]` is deleted, then the annotation `[running]` is added.

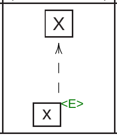
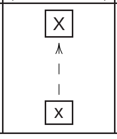
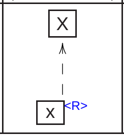
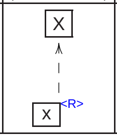
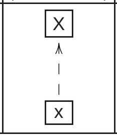
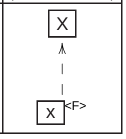
When a task instance is finished, if its type in the model has a consecutive task, we will create a new, enabled task instance which is typed by the consecutive task. In this way, the control flow will be passed from the finished task instance to the next task instance. In our transition system, the addition of an enabled task instance is done by applying rule t_8 in Table 4, which is used to create a task instance `y` with the annotation `[enabled]` (and a flow instance `a`) when the preceding task instance `x` is finished.

Other legal changes which task instances undergo during execution are described by transformation rules shown in Table 4. Rules t_3 and t_4 are used to describe the transitions of spans of flow instances, and t_5, t_6 and t_7 are used to describe the transitions of sinks of flow instances. For rule t_6, we have used $\boxed{\text{x:X}}^{\text{<E|R|F>}}$ to denote the case where we have one of the following: $\boxed{\text{x:X}}^{\text{<E>}}$, $\boxed{\text{x:X}}^{\text{<R>}}$ or $\boxed{\text{x:X}}^{\text{<F>}}$. The rule t_7 expresses that even if the task instance `z` is does not exist yet, the transition can go forward and enable `y`. For the `[xor_split,`c`]`and `[xor_merge]` predicates, there are analogous rules for generation of `z` and `b`, however, we have omitted these rules to save space.

Table 2: A signature Σ_1 used for annotation of workflow instances

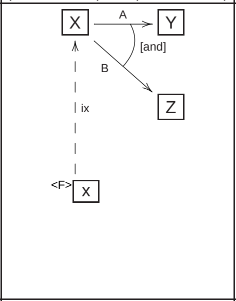
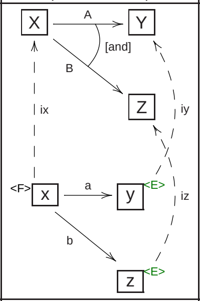
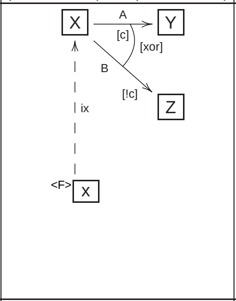
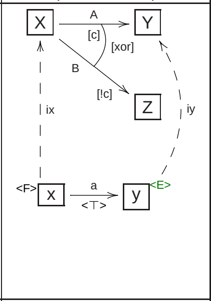
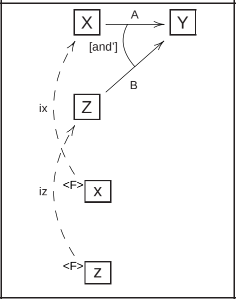
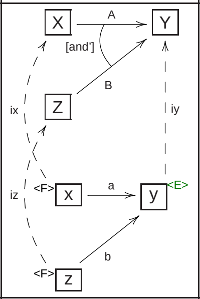
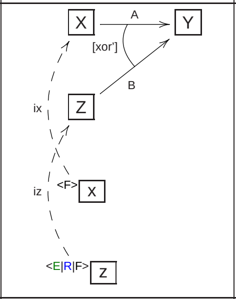
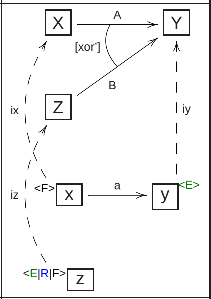
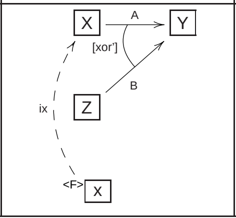
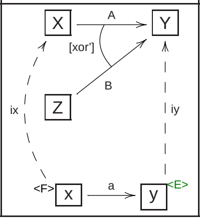
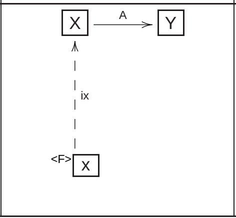
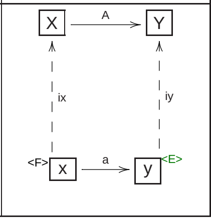
q	**Visualisation**	*q*	**Visualisation**
`[enabled]`	$\boxed{\text{X}}^{\text{<E>}}$	`[true]`	$\boxed{\text{X}} \xrightarrow[\text{<T>}]{\text{f}} \boxed{\text{Y}}$
`[running]`	$\boxed{\text{X}}^{\text{<R>}}$	`[false]`	$\boxed{\text{X}} \xrightarrow[<\perp>]{\text{f}} \boxed{\text{Y}}$
`[finished]`	$\boxed{\text{X}}^{\text{<F>}}$		

Table 3: The coupled transformation rules t_1 and t_2 of our transition system

t	$(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1)$	$(\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$	$(\mathfrak{M}_0 \dashrightarrow \mathfrak{M}_1)$	t	$(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1)$	$(\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$	$(\mathfrak{M}_0 \dashrightarrow \mathfrak{M}_1)$
t_1				t_2			

A sequence of changes represents one execution path of the workflow software system. These changes are formulated by transformation rules (see Tables 3 and 4) which describe the transition system. Given a workflow model $\mathfrak{S}_1$, the start instance $\mathfrak{S}_0^s$ is a specification which conforms to $\mathfrak{S}_1$ and consists of only task instances with no incoming flows, annotated with `[enabled]`. All possible sequences of rule applications starting from the start instance gives the set of reachable instances; these sequences define the dynamic semantics of our behavioural models.

Table 4: Some coupled transformation rules for the transition system

t	$(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1) = (\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$	$(\mathfrak{M}_0 \dashrightarrow \mathfrak{M}_1)$	t	$(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1) = (\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$	$(\mathfrak{M}_0 \dashrightarrow \mathfrak{M}_1)$
t_3			t_4		
t_5			t_6		
t_7			t_8		

Since the routing constraints of the models determine the behaviour of the models, it is necessary to inspect these constraints in order to decide which rules to apply. For example, for `[and_split]`, all of `y`, `a`, `z` and `b` will be created, while for `[xor_split,`c`]`, either `y` and `a`, or `z` and `b` will be created, but not both. To facilitate "generic" rules which apply to any occurences of the `[and_split]` and `[xor_split,`c`]`constraints, we keep track of the (conformance) relation between models and their instances. We do this by employing coupled model transformations to describe transitions between the instances of workflow models. In more detail, for a workflow model $\mathfrak{S}_1$, a transition $\mathfrak{S}_0 \overset{<t>}{\Longrightarrow} \mathfrak{S}'_0$ is given by an application of a coupled transformation rule t, where both specifications $\mathfrak{S}_0, \mathfrak{S}'_0$ are reachable instances of $\mathfrak{S}_1$. We omit the technical details of the application of (coupled) transformation rules (for details see [10,14,27,25,20,15]), and our specific case in which the model part remains unchanged will be elaborated in a future work. Here, we briefly outline the general structure and ingredients of coupled transformation rules and the necessary details to understand our approach to the definition of dynamic semantics for behavioural models.

$$\begin{tikzcd} S_1 \arrow[r, "\phi_1"] & S'_1 \\ S_0 \arrow[u, dashed, "\iota^{S_0}"] \arrow[r, "\phi_0"] & S'_0 \arrow[u, dashed, "\iota^{S'_0}"'] \end{tikzcd}$$

Specification morphisms are used to formaly describe the relation between specifications. A specification morphism [25] is a constraint preserving graph homomorphism [10] between the underlying graphs of the specifications. A coupled specification $(\mathfrak{S}_0 \dashrightarrow \mathfrak{S}_1)$ consists of a specification $\mathfrak{S}_1$ together with one of its instances $\mathfrak{S}_0$; i.e., $\mathfrak{S}_1$, $\mathfrak{S}_0$ and the typing morphism $\iota^{S_0} : S_0 \to S_1$ from the underlying graph S_0 of $\mathfrak{S}_0$ to the underlying graph S_1 of $\mathfrak{S}_1$. A coupled specification morphism is a mapping $\phi : (\mathfrak{S}_0 \dashrightarrow \mathfrak{S}_1) \to (\mathfrak{S}'_0 \dashrightarrow \mathfrak{S}'_1)$ given by two specification morphisms $\phi_1 : \mathfrak{S}_1 \to \mathfrak{S}'_1$ and $\phi_0 : \mathfrak{S}_0 \to \mathfrak{S}'_0$ such that $\phi_0(\mathfrak{S}_0)$ conforms to $\phi_1(\mathfrak{S}_1)$. A coupled transformation rule $t = (\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1) \xhookleftarrow{l} (\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1) \xhookrightarrow{r} (\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$ consists of three coupled specifications and two coupled specification morphisms l, r.

In a coupled transformation rule, $t = (\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1) \xhookleftarrow{l} (\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1) \xhookrightarrow{r} (\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$, $(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1)$ is the *left-hand side* (LHS) and $(\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$ is the *right-hand side* (RHS) of t, and $(\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$ is their interface. $(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1) \setminus l((\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1))$ describes the part of a specification which is to be deleted, $(\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1) \setminus (\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$ describes the part to be added, and $(\mathfrak{K}_0 \dashrightarrow \mathfrak{K}_1)$ describes the overlap between $(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1)$ and $(\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$. Note that the rules in Table 3 are deleting rules, while the rules in Table 4 are non-deleting rules. Non-deleting rules could be seen as special cases of deleting rules where $(\mathfrak{L}_0 \dashrightarrow \mathfrak{L}_1) = (\mathfrak{R}_0 \dashrightarrow \mathfrak{R}_1)$.

As mentioned, a transition from one state $\mathfrak{S}_0$ to another $\mathfrak{S}'_0$ of a workflow specification $\mathfrak{S}_1$ is given by an application of a coupled transformation rule. We remark that after applying a rule, the instances are decoupled from their models, and indicate the transition $\mathfrak{S}_0 \overset{<t>}{\Longrightarrow} \mathfrak{S}'_0$ by a double arrow and the name of the applied rule.

Transition systems in general may be or may not be terminating [10,25]. That is, starting with $\mathfrak{S}^s_0$, it may always be possible to apply more rules. To achieve termination of our transition system, we control the application of transformation rules through (i) the use of *negative application conditions* (NACs) and (ii) priorities [10]. To ensure that the rules can be applied only once via the same match, we require that the RHSs of the rules in Tables 3 and 4 are NACs for themselves. Moreover, we require that the rule t_8 has the lowest priority, which avoids changing the annotation on a single task instance if it is part of a bigger structure. This priority definition is necessary since in a bigger structure there may be dependencies between flow and task instances.

In Fig. 7 we show an execution path of the workflow model $\mathfrak{S}_1$ in Fig. 3. For the sake of illustration, we also show a specification $\mathfrak{S}_0^{\diamond}$ which is not a reachable instance since it violates the semantics of the `[xor_split,`c`]`constraint.

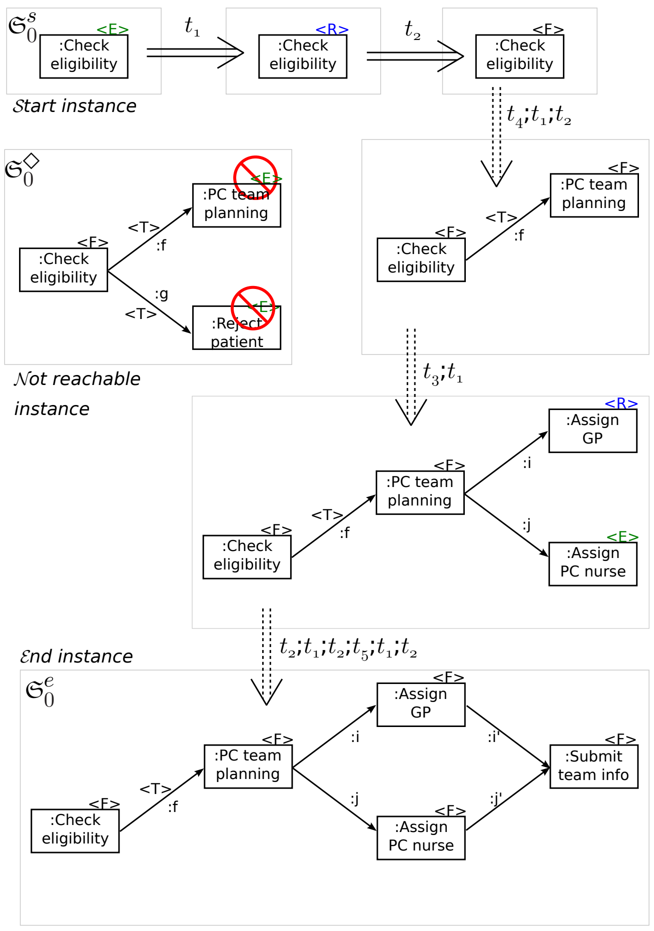


Figure 7: An execution path of the workflow model $\mathfrak{S}_1$, the dashed double arrows represent sequences of transitions, the specification $\mathfrak{S}_0^{\diamond}$ is not a reachable instance

One of the advantages of formalising workflow modelling languages is to facilitate automatic analysis of workflow models. We now outline some properties of workflow models which have to be satisfied in order to make sure that each execution scenario (of workflow software systems developed from the workflow models) *terminates* in an *appropriate* way [1]. Workflow models which have the option to terminate, have proper termination, and, lack dead tasks (i.e., tasks which are not enabled in any execution scenario), are said to be *sound* [2].

Since our transition system is based on graph transformations, we use the termination property from graph transformations [10]. More precisely, we have defined our transformation rules in such a way that for any start state of a workflow model, we can guarantee that the transformation system will eventually terminate and produce an end state; termination in this sense means that no more transformation rules are applicable. Proving that the transformation rules from Tables 3 and 4 together with the control structures are terminating is straightforward, but outside the scope of this paper. In addition to the termination property of the transition system, we need also to require that each task will be annotated with `[enabled]` at least in one state.

We define *end states* as follows (see the last model $\mathfrak{S}_0^e$ in Fig. 7). Given a workflow model $\mathfrak{S}_1$, an end state $\mathfrak{S}_0^e$ is a reachable instance of $\mathfrak{S}_1$ such that no more transformation rules are applicable to $\mathfrak{S}_0^e$, and, at least one task instance with no outgoing flows is annotated with `[finished]`. We use $E^{\mathfrak{S}_1}$ to denote the set of all end states of $\mathfrak{S}_1$.

Now the properties which a workflow model $\mathfrak{S}_1$ must satisfy in order to be sound can be expressed as: (i) *Proper termination*: the transition system terminates always resulting in one of the end states in $E^{\mathfrak{S}_1}$; (ii) *No dead tasks*: for each task X in $\mathfrak{S}_1$, the specification $\mathfrak{S}_0^{\mathsf{X}}$ is one of the reachable instances in the transition system, where $\mathfrak{S}_0^{\mathsf{X}}$ is an instance of $\mathfrak{S}_1$ in which a task instance x:X is annotated with `[enabled]`.

We could prove that for each workflow model $\mathfrak{S}_1$ the transition system (described by the rules in Tables 3 and 4) starting from the start state $\mathfrak{S}_0^s$, will terminate in one of the end states in $E^{\mathfrak{S}_1}$. That is, given the start state and the transition system, we can construct all possible sequences of transformation rule applications and inspect the resulting target specifications – those that cannot be transformed anymore– to check whether they are end states. The non-existence of dead task can be checked analogously.

5 Related Work

Kindler [16] advocated MDE, particularly the *Model-driven Architecture* (MDA) for Process-Aware Information Systems (PAIS). He also argued for the suitability of MDE in PAIS; while many MDE concepts were explained and put in relation to PAIS, a specific modelling language for behavioural and process modelling was not proposed.

Brüning et al. [5] present a strict metamodelling approach to workflow modelling, which makes it possible to easily express semantics of sophisticated transition relationships between activities using UML class diagrams and OCL constraints. Due to shortcomings of UML and OCL wrt. constraint evaluation combined with multi-level metamodelling, this approach flattens the three levels – metamodel, model and instance – into two levels. OCL constraints are defined at the metamodel level and they are forced at the model/instance level. Our approach allows multi-level metamodelling with a unification of structural and OCL constraints in one formalism, and clearly distinguishes between the different levels of the metamodelling hierarchy.

Ghamarian et al. [13] employ a graph transformations-based framework, GROOVE, to provide semantics for behavioural models. Our approach extends graph transformations by using constraint-aware model transformations – i.e., considering diagrammatic constraints in transformation rule definitions and applications – which facilitate the definition of more fine grained rules and better control of their applications [26].

6 Conclusion and Future Work

This paper presents a formal approach to behavioural modelling following MDE methodologies. As a running example, we provide a visually appealing technique for workflow modelling. Two modelling formalisms are used for the specification of workflow models and their instances. Coupled model transformation rules are used to describe a transition system. All possible sequences of rule applications starting from the start instance gives the set of reachable instances; these sequences define the dynamic semantics of our behavioural models. Use of coupled transformation rules avoids the proliferation of rules associated to types; i.e., one rule can be used for each predicate regardless of the types. The approach is illustrated with an application to workflow modelling, but its generalisation to other kinds of behavioural models should be straightforward.

In our coupled model transformations only the instances are changed, while the models remain the same. This category of model transformation falls between traditional model transformations and coupled model transformations. We will explore this new kind of coupled model transformations in future work.

Workflow models often change, even when their corresponding software system is running. Migrating these changes to the software system affects its future and past states. In future work, we will elaborate on this kind of workflow model evolution. We plan to extend our formalism with support for composition of behavioural models with structural models, as found in [18]. As a proof of concept, the proposed approach will be implemented as a plugin to the DPF Workbench [21], a workbench that supports multi-level, diagrammatic (meta)modelling.

Acknowledgments. This research is sponsored by Natural Sciences and Engineering Research Council of Canada and by the Atlantic Canada Opportunities Agency.

References

1. van der Aalst, W.M.P., van Hee, K.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)
2. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
3. Barr, M., Wells, C.: Category Theory for Computing Science (2^{nd} Ed.). Prentice Hall (1995)
4. Becker, S.: Coupled model transformations. In: WOSP 2008: 7^{th} international workshop on Software and performance. pp. 103–114. ACM (2008)
5. Brüning, J., Gogolla, M., Forbrig, P.: Modeling and Formally Checking Workflow Properties Using UML and OCL. In: BIR 2010. LNBIP, vol. 64, pp. 130–145. Springer (2010)
6. Butler, M.J., Hoare, C.A.R., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) 25 Years Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer (2004)
7. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in bpmn. Information & Software Technology 50(12), 1281–1294 (2008)
8. Diskin, Z.: Encyclopedia of Database Technologies and Applications, chap. Mathematics of Generic Specifications for Model Management I and II, pp. 351–366. Information Science Reference (2005)

9. Diskin, Z., Kadish, B., Piessens, F., Johnson, M.: Universal Arrow Foundations for Visual Modeling. In: Anderson, M., Cheng, P., Haarslev, V. (eds.) Diagrams 2000. LNCS, vol. 1889, pp. 345–360. Springer (2000)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (March 2006)
11. Fazle, R., MacCaull, W., Wang, H., Rutle, A.: A Model Slicing Method for Workflow Verification. ENTCS To appear (2012), 9th International Workshop on Formal Engineering approaches to Software Components and Architectures, Satellite event of ETAPS
12. Fujaba Developer Team: The Fujaba Tool Suite, `http://www.fujaba.de/`
13. Ghamarian, A., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. STTT pp. 1–26 (2011)
14. Heckel, R.: Graph Transformation in a Nutshell. ENTCS 148(1), 187–198 (2006)
15. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: Automatability of Coupled Evolution of Metamodels and Models in Practice. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) MoDELS 2008. LNCS, vol. 5301, pp. 645–659. Springer (2008)
16. Kindler, E.: Model-based software engineering and process-aware information systems. Transactions on Petri Nets and Other Models of Concurrency II, Special Issue on Concurrency in Process-Aware Information Systems 2, 27–45 (2009)
17. Kindler, E.: Model-based software engineering: the challenges of modelling behaviour. In: BM-FA 2010. pp. 4:1–4:8. ACM (2010)
18. Kindler, E.: Integrating behaviour in software models: an event coordination notation – concepts and prototype. In: BM-FA 2011. pp. 41–48. ACM (2011)
19. Küster, J.M.: Towards Inconsistency Handling of Object-Oriented Behavioral Models. ENTCS 109, 57 – 69 (2004), proceedings of the Workshop GT-VMT
20. Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: 1st International Workshop on Software Evolution Transformations (November 2004)
21. Lamo, Y., Wang, X., Mantz, F., MacCaull, W., Rutle, A.: DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment. In: Roger, L. (ed.) Computer and Information Science, Studies in Computational Intelligence, vol. 429, pp. 37–52. Springer (2012)
22. Object Management Group: Object Constraint Language Specification (February 2010), `http://www.omg.org/spec/OCL/2.2/`
23. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (FUML) (February 2011), `http://www.omg.org/spec/FUML/1.0/`
24. Rossini, A.: Diagram Predicate Framework meets Model Versioning and Deep Metamodelling. Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2011)
25. Rutle, A.: Diagram Predicate Framework: A Formal Approach to MDE. Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2010)
26. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A formal approach to the specification and transformation of constraints in MDE. JLAP 81/4, 422–457 (2012)
27. Schulz, C., Löwe, M., König, H.: A categorical framework for the transformation of object-oriented systems: Models and data. J. Symb. Comput. 46(3), 316–337 (2011)
28. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0 (2nd Edition). Addison-Wesley Professional (2008)
29. Wang, H., Rutle, A., MacCaull, W.: A formal diagrammatic approach to timed workflow modelling. In: TASE 2012. IEEE Computer Society (2012), to appear

Workshop on

Graphical Modeling Language Development

at ECMFA 2012 Conference
3 July, 2012, Kgs. Lyngby, Denmark

Welcome to the workshop on Graphical Modeling Language Development

Heiko Kern[1], Juha-Pekka Tolvanen[2], Paolo Bottoni[3]

[1]University of Leipzig, Germany
kern@informatik.uni-leipzig.de
[2]MetaCase, Finland
jpt@metacase.com
[3]University of Roma, Italy
bottoni@di.uniroma1.it

Preface

Models play an important role in software development. They not only support communication and understanding, but are increasingly used in automating software development tasks such as code generation, testing, simulation and analysis. While many languages are created for software developers others may be created for business analysts, interaction specialists, test engineers, or persons responsible for product configuration and deployment. Often these languages are domain-specific, created for a narrow application area or for use only inside one company.

The workshop on Graphical Modeling Language Development[1] aims to cover all the phases of language development, including definition, testing, evaluation, and maintenance of modeling languages. Particular attention is given to the principles of modeling language development, especially graphical modeling languages for domain-specific needs. It also includes papers that discuss challenges and new trends.

The workshop does not focus on tools, but recognizes the need for metamodel-based tools, which significantly ease the production of modeling environments. These tools also enable experimentation with the language as it is built, and remove the burden of tool creation and maintenance from the language creator.

In response to the call for papers, 8 submissions were received. Submitted papers were formally peer-reviewed by three referees, and 5 papers were finally accepted for presentation at the workshop and publication at the proceedings.

The workshop program is composed of two parts: paper presentations and group work. Selected papers describe experiences at a practical level, or propose new ideas and approaches. Group work sessions aim at discussing in more detail the topics found most relevant during the paper presentations. Results of the group work will be presented at the end of the workshop.

[1] http://www.dsmforum.org/events/GMLD12/

We would like to thank the ECMFA 2012 organization for giving us the opportunity to organize this workshop. Thanks to those that submitted papers, and particularly to the contributing authors. Our gratitude also goes to the members of the GMLD 2012 Program Committee for their reviews and help in choosing and improving the selected papers.

We hope that you will enjoy the workshop and find the information within the proceedings valuable toward your understanding of the current state-of-the-art in developing graphical modeling languages.

Program committee of the workshop on Graphical Modeling Language Development

Domain-Specific Language Architecture for Automation Systems: An Industrial Case Study

Christopher Preschern, Andrea Leitner, and Christian Kreiner

Institure for Technical Informatics
Graz University of Technology, Austria
christopher.preschern@tugraz.at
andrea.leitner@tugraz.at
christian.kreiner@tugraz.at
`http://www.iti.tugraz.at`

Abstract. This paper presents a domain-specific language (DSL) design for automation systems. We describe basic components of the language, its mapping to automation devices and to automation software elements. The DSL design achieves low domain model complexity and is easy to maintain. Furthermore, it allows easy and intuitive modeling of systems in a domain. We present an industrial case study using the proposed DSL design and evaluate it regarding its maintainability and complexity. For this evaluation we use existing metrics to evaluate the domain model complexity and we introduce novel metrics to evaluate the code generator complexity.

Keywords: domain-specific language, automation system, domain model metrics, code generator metrics

1 Introduction

Domain-specific languages (DSL) allow product modeling on a high level of abstraction and enable structured software reuse through code generation from these models. To develop a DSL, a meta-model has to be constructed for a specific product family. Meta-model development requires careful design of the domain model structure and the mapping of DSL elements to artifacts in the solution space. This is a sophisticated task and several guidelines on how to construct a good domain model exist. Such guidelines can be more detailed if they just address specific domain families, but are rarely present for domain families where DSLs are not often applied. An example for such a domain family where no detailed guideline for DSL development exists is the automation domain.

In this paper we present a flexible design to develop automation system DSLs. We discuss design decisions and their rationale concerning the meta-model and the mapping of DSL elements to automation devices and to generated artifacts like the automation system software. We present and evaluate PISCAS (Pisciculture Automation System), an industrial case study which applies the discussed

DSL design guidelines. For the evaluation of the DSL complexity, we use existing domain model metrics and we introduce novel metrics to measure the code generator complexity. Furthermore, we evaluate the DSL in terms of modeling effort and maintainability.

2 Related Work

Issues regarding the construction of domain models for automation systems are discussed in [9], where experiences with different domain model granularities are presented. Hierarchical, nested domain models are suggested for automation systems to provide different levels of granularity and abstraction. In Leitner's work [8] an evaluation method for the domain model complexity for DSLs and for feature oriented modeling is presented. We use the proposed DSL metrics in our domain model evaluation.

Graphical domain-specific languages are used in [4] to model home automation systems. Eclipse GMF is used to create DSLs where systems can be modeled on different levels of abstraction which is shown on an industrial case study. A graphical DSL for automation systems in the railway domain is presented in [3] where special focus is put on safety constraints of the system. Here, the DSL is used as a formal specification of the system containing system verification functionality. The MetaEdit+ tool suite is used in [2] to model high rack warehouse information systems. These literature examples show case studies for automation system DSLs. None of them, however, handles the topic on a more abstract level and discusses generic design decisions for these DSLs.

Automation system modeling is handled on a more general level by the Christian Doppler Laboratory in Linz, Austria. They developed a tool for variability management and show several case studies in the automation domain [1]. In [11] a textual DSL for general automation systems is suggested. In a more recent work of the Doppler Laboratory, a DSL is applied to the automation domain using hierarchical structuring of the domain model [10]. Compared to our paper they do not focus on specific automation domains, but address generic automation system solutions.

3 Domain-Specific Language Design for Automation Domains

In this section we present general design decisions for the development of automation system DSLs. First we present the required meta-meta-model which we later use to provide guidelines for the development of a DSL for automation domains. We present the DSL mapping to automation devices and to the automation software. Finally, we discuss the rationale of the presented design decisions.

3.1 Meta-Meta-Model

The DSL design suggested in the next section requires the GOPPRR (*Graph-Object-Property-Port-Role-Relationship*) meta-meta-model [6]. This meta-meta-model allows defining the meta-model in form of a DSL. *Objects* as basic DSL elements can be connected with *Relationships* which define a *Role* for the connection to an *Object*. The connection to an *Object* can be further refined by a *Port* to which the connection is attached. The *Port* is attached to the *Object*, while the *Role* is attached to the *Relationship*. *Objects* and their *Relationships* can be gathered in a *Graph*. *Properties* can be added to each of these elements (*Object*, *Relationship*, *Role*, *Port* and *Graph*).

3.2 DSL Design

Physical automation devices connected to the automation hardware (e.g. to the PLC) are represented as basic DSL *Objects*. Variants for device types are defined by *Properties* of an *Object*. Automation I/O modules are also modeled as *Objects*. Wire connections between the automation devices are directly modeled as *Relationships* between *Objects*. The semantics of the *Relationship* is given by the *Port* it is connected to. Each DSL *Object* has a minimum set of basic elements: *Property* 'Name', *Property* 'Voltage', *Port* 'Input', *Port* 'Output'. This set of basic DSL elements represents our meta-model for the automation domain which is shown in Figure 1. Additional *Relationships* or *Ports* can be used to connect concrete objects with additional semantics, but concrete *Objects* still have to adhere to the rules specified for the abstract automation domain object.

All GOPPRR entities apart from *Roles* are used in the mapping of DSL elements to the automation domain. When using the proposed DSL design, *Roles* might still be of interest for some domains which might need additional semantics for their interfaces. The *Port* entity of GOPPRR is especially useful due to its straight semantic mapping to wire connections of automation devices.

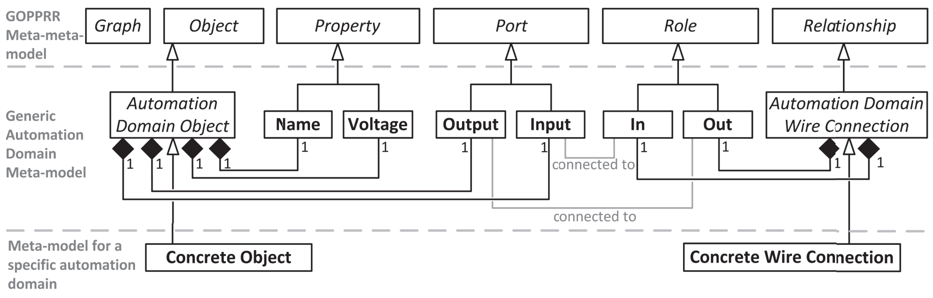


Fig. 1. Suggested meta-model for the automation domain

In the automation software, each *Object* is represented by a function block. Function blocks at least implement the interface variables 'input' and 'output' which represent the corresponding *Ports* in the DSL. Function Block parameters allow configuring the variants modeled with *Object Properties*. *Relationships* between DSL *Objects* are mapped to function block connections (e.g. in a function block diagram). Table 1 contains an overview of the mapping between the DSL, the automation software, and the physical automation devices.

Figure 2 illustrates this mapping and shows how two different aspects of the automation system (physical devices and source code) can be represented by the DSL using the proposed design. This direct mapping between the DSL, the automation software, and the physical devices is possible, because of the nature of the automation domain, which already provides a tight relationship between concepts in the physical world such as physical wires, and corresponding concepts in the automation software such as function block connections representing wires.

Physical system	GOPPRR concepts	Automation software
automation plant	*Graph*	overall software
device	*Object*	function block
wire	*Relationship*	connecting function block interface variables
-	*Role*	-
wire connection	*Port*	function block interface variables
device attribute	*Property*	function block parameters

Table 1. Mapping of the physical system to GOPPRR concepts and to the automation software

3.3 DSL Design Rationale

Graphical DSL - choosing a graphical model representation allows capturing information about the assembly and position of physical objects. This information can be used to generate the system documentation and the visualization, which is an essential part for automation systems.

Directly mapping of physical devices to DSL Objects - Directly mapping physical devices to DSL Objects and physical wires to DSL *Relationships*, makes modeling of the automation system more intuitive for domain experts, because then the DSL is quite similar to function block diagrams which are well known to automation system developers.

Explicit modeling of I/O modules integrates the physical view of the automation system into the DSL and allows capturing all physical wires of the project. Information on the system topology and the electrical wiring is then

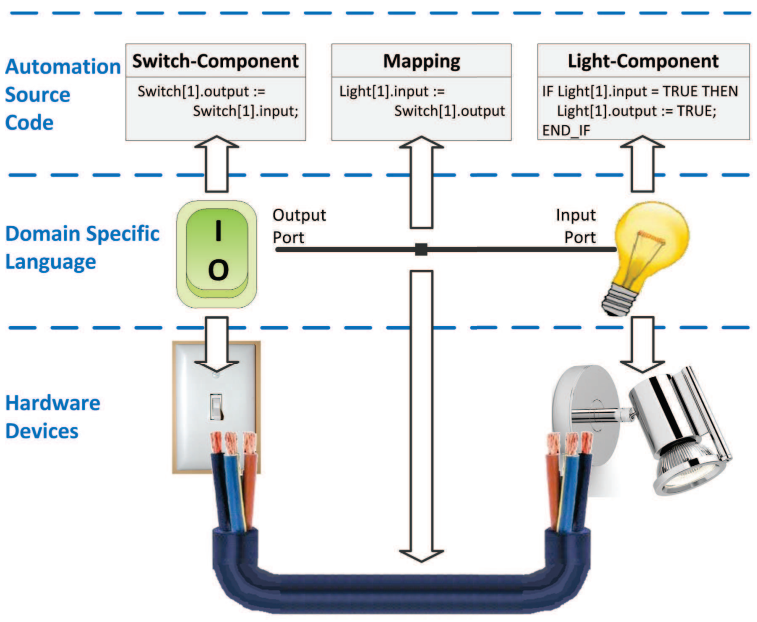

Automation Source Code
Switch-Component
Switch[1].output := Switch[1].input;
Mapping
Light[1].input := Switch[1].output
Light-Component
IF Light[1].input = TRUE THEN Light[1].output := TRUE; END_IF
Domain Specific Language
I
O
Output Port
Input Port
Hardware Devices

Fig. 2. Mapping between the DSL, physical devices, and automation software

present in the model. Explicit modeling of the I/O modules is not absolutely necessary, because the information about the I/O module type to which a device has to be connected to, is implicitly present in the device type. This would allow automatic generation of the wiring connections, which might not always be desired. To model systems with already installed hardware, this automatic wiring plan generation would most likely be inconsistent. The labels for wiring closets can be generated from the information of the module connections. The *Property* 'Voltage' adds the necessary information to a model with explicitly modeled I/O modules for generating the complete wiring plan for the automation system as well as the list of hardware parts required for the automation project.

Abstract DSL Object - The convention that each *Object* has an 'Input' and 'Output' *Port* and a *Property* 'Name', makes the code generators much simpler, because they can use this abstract *Object* interface. The generators for the visualization, the graphical system overview in the documentation, and the mapping between automation system function blocks can be made independent from the *Object* type. Therefore, adding new *Object* types to the DSL or changing existing ones does not affect those code generators. The reason for explicitly making*Object* types rather flexible is that in the automation domain the basic automation elements, which are represented as *Objects* in our meta-model, change rather often [9].

4 Case Study: PISCAS

In this section the industrial case study is presented. The automation domain and the developed DSL are described and advantages regarding the use of the proposed DSL design are evaluated.

PISCAS is a product line for fish farm automation systems. The project was carried out as a master's thesis at the Institute for Technical Informatics at Graz University of Technology [12]. The core functionality of PISCAS includes water oxygen control and fish feeding. Additionally, water level supervision including an alarm system and standard automation system functionality like steering actuators, such as lights, are part of PISCAS. The automation system can be controlled and configured with a visualization integrated into a web portal. Typically fish farms just vary in the amount of ponds and the functionality for ponds like feeding and oxygen supervision. Fish farm automation elements are rather independent from each other and do not interact a lot. Further information on the PISCAS project can be found on the PISCAS website[1].

A graphical DSL was developed to model fish farm projects. The information in the model is used to parametrize a generic fish farm automation software. Fully executable automation code is created including the hardware mapping and the visualization of the automation project. Additionally PISCAS generates a system documentation including an electrical wiring plan and a list of needed hardware components. Configuration files for the web portal and for network devices are generated. For the web portal, SQL configuration files needed for system initialization are generated. The network routers installed for PISCAS automation systems require configuration of a VPN connection which is needed for remote maintenance of the fish farm automation systems. Furthermore, Configuration files for the router to set up this VPN connection are generated. Figure 3 gives an overview of generated artifacts.

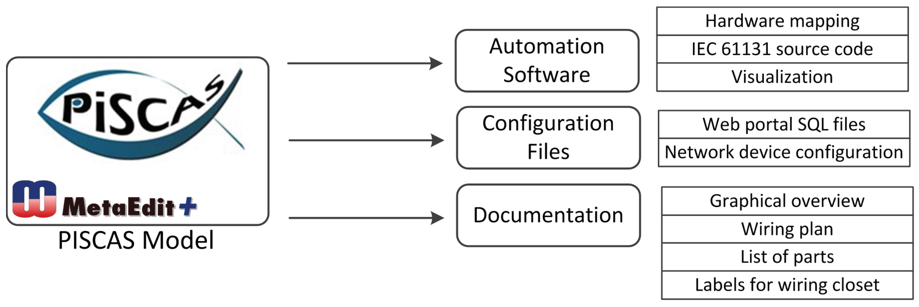


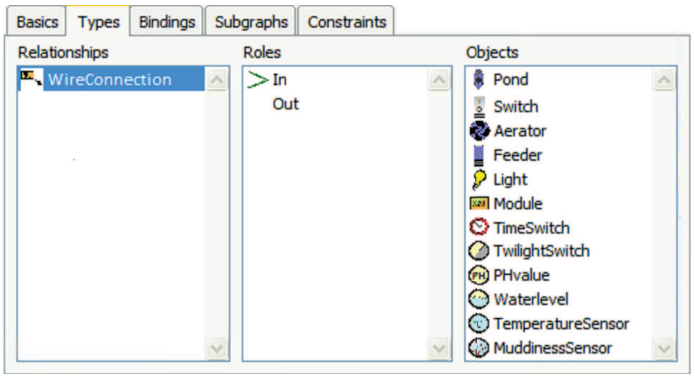
Fig. 3. PISCAS - generated artifacts

[1] http://www.piscas.eu

Bernecker+Rainer[2] (B&R) automation products were used for the PISCAS project. The reason for choosing B&R is that compared to other automation system vendors, the B&R software is easier to generate. All project files including the visualization and hardware mapping files are stored in XML format. Therefore, they are easy to parse and to modify. Metaedit+[3] was used for development of the DSL and for system modeling. Several DSL tools were evaluated according to an evaluation method suggested in [7]. The tools were evaluated regarding technical, management, and product line related criteria. The full evaluation is available in [12]. MetaEdit+ implements the GOPPRR meta-metamodel [5] and is therefore suitable to apply the DSL design proposed in Section 3.2.

4.1 PISCAS DSL

The PISCAS DSL consists of *Objects* representing basic fish farm automation devices, such as a feeder or an oxygen control unit. Each *Object* implements an abstract *Object* definition as presented in Figure 1. The abstract object consists of two *Ports* (Input and Output) and two *Properties* (Name and Voltage). Different variants of a concrete *Object* (e.g. different feeder types) are configured via additional *Properties*. The DSL consists of one *Relationship* type (wire connection) and two *Roles*. A *Relationship* represents an actual physical wire connection.



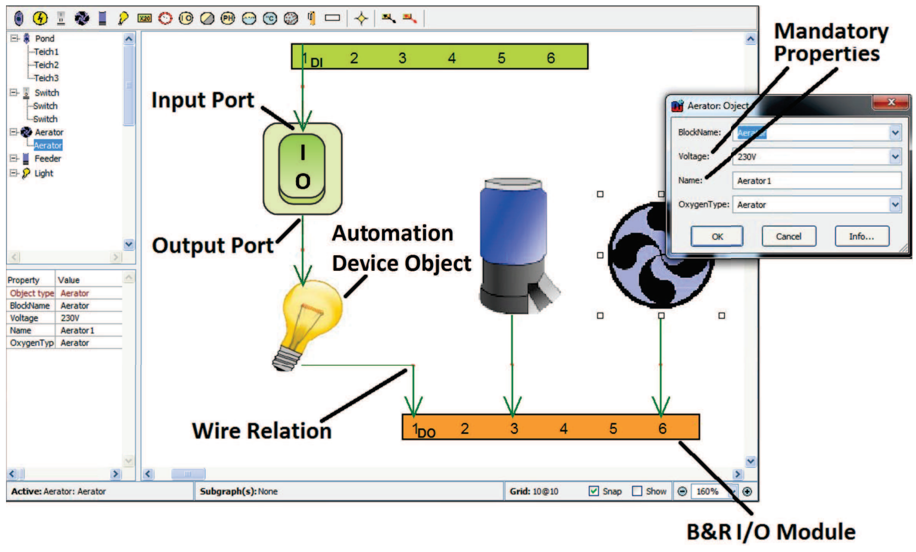
Relationships	Roles	Objects
WireConnection	In	Pond
	Out	Switch
		Aerator
		Feeder
		Light
		Module
		TimeSwitch
		TwilightSwitch
		PHvalue
		Waterlevel
		TemperatureSensor
		MuddinessSensor

Fig. 4. MetaEdit+ DSL elements of the PISCAS language

Figure 4 gives an overview of PISCAS language elements. PISCAS consists of 7 basic DSL elements (2 *Roles*, 1 *Relationship*, 2 *Ports*, (at least) 2 *Properties*). These elements are needed to use the proposed DSL design. Additionally, 18 *Properties* and 12 *Objects*, are used for domain-specific elements. Therefore, the PISCAS DSL consists of overall 37 elements. Code generators are kept as

[2] http://www.br-automation.com
[3] http://www.metacase.com

independent as possible from the *Object* type. This means that each *Object* provides a well defined interface (defined minimum set of *Properties* and *Ports*) which is accessed by the code generators. General code for the generation of the visualization, the wiring plan, the documentation and the automation software function block parametrization and connection can be generated by *Object* independent generators which access this interface. The DSL is independent from basic functional changes or bug fixes in the automation code, because model is just used to configure a generic fish farm automation software. This generic software has a well defined interface to the DSL through the function blocks and their interface variables. Therefore, the generic fish farm automation software can be maintained independently from the DSL as long as the interface between the automation software and the DSL is not affected by a change. Figure 5 shows an example for a fish farm model constructed with the PISCAS DSL.



Property	Value
Object type	Aerator
BlockName	Aerator
Voltage	230V
Name	Aerator1
OxygenTyp	Aerator

BlockName: Aerator
Voltage: 230V
Name: Aerator1
OxygenType: Aerator

Active: Aerator: Aerator   Subgraph(s): None   Grid: 10@10   ☑ Snap   ☐ Show   160%

Fig. 5. Example for a PISCAS fish farm model in MetaEdit+ Modeler

4.2 Evaluation

This section contains experiences from applying the suggested DSL decisions presented in Section 3.2 to the PISCAS project.

Application modeling - Two fish farm systems were modeled with the PISCAS DSL and are currently in operation. Both systems could actually be modeled during meetings with the fish farm owner. This was possible due to the intuitive

	Fish Farm element modeling (ponds, switches, lights, ...)	B&R I/O module modeling
Fish farm A	2h	3h
Fish farm B	1h	1.5h
Add new components to B (model approximately doubled)	1h	2h

Table 2. Time spent on application modeling for the PISCAS systems

system representation and allowed the fish farm owner to directly check the fish farm DSL model.

Explicitly modeling the hardware connections took the most time during the modeling process. Table 2 shows the effort (in hours) required for system modeling for different PISCAS fish farms. The high modeling effort is caused by the high number of connections in the hardware mapping. To reduce this effort, we suggest to generate the hardware mapping in the model automatically the first time. The mapping can still be changed in the model after initial generation.

Bug fixes - Most PISCAS changes were related to bugs in the automation software. Therefore, decoupling automation software maintenance from the DSL maintenance is very important for PISCAS and allows decreasing the overall maintenance effort.

For the generation of a concrete PISCAS system, a generic fish farm automation software is configured with the information in the DSL model. The generic automation software is a complete automation program which compiles and can be used to maintain and debug the automation code without the need to work with the DSL tools. This from the DSL decoupled, generic code allowed easy bug fixes and did not require PISCAS DSL modifications often.

Decoupling the generic automation software allows to test new features in the generic automation system without the need work with MetaEdit+. Therefore, MetaEdit+ did not have to be installed on the computer which was used to develop the fish farm automation software. New automation code can easily be integrated later on into the DSL as long as the interface constraints regarding the automation software (physical automation elements are mapped to configurable function blocks with input and output interfaces representing wire connections) are met. Taking this thought one step further, the generic automation software could even be programmed by someone who does not construct the DSL, as long as the constraints regarding the automation code interfaces where the DSL is mapped to, are met.

DSL Complexity - To assess the complexity of the DSL, two different metrics were used. One metric describes the domain model complexity and one describes the code generator complexity.

The domain model was assessed with the metrics suggested by Leitner [8]. The domain model complexity consists of values describing the complexity of interfaces, elements, and properties. These complexities are defined in general and explicitly for the GOPPRR meta-meta-model used in MetaEdit+. The metrics are shown in Equation 1 where C stands for the complexity and n is the number of items. We modified the element complexity metric, by taking the number of *Ports* (n_{Port}) into account. This number is added to the element complexity, because *Ports* are attached to *Objects*. The reason for preferring these metrics to other domain model metrics such as presented in [13] is that they separately handle the complexity of DSL interfaces and DSL elements which allows us to reason about the element complexity, which is especially interesting in the automation domain due to the high semantics these elements usually carry [9].

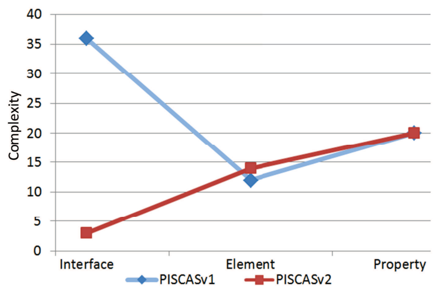
$$\begin{aligned} C_{interface} &= n_{Relationships} + n_{Roles} + n_{Constraints} \\ C_{element} &= n_{Objects} + n_{Ports} \\ C_{properties} &= n_{Properties} \end{aligned} \tag{1}$$

To assess the code generator complexity, Leitner's metrics [8] have been adopted. The interface complexity of the code generators consists of the number of lines of code ($\#LOC$) where any *Role* or *Relationship* type is explicitly used in the generator source code. The element complexity describes the same for the number of *Objects* and *Ports* and the properties complexity handles the occurrence of *Property* types in the code generators. Equation 2 shows these metrics which describe the dependence (D) of the code generators on DSL items. The code generator dependence is a metric describing the affect of domain model changes on the code generator. A lower code generator dependence value suggests less necessary changes in the code generators if DSL items are changed.

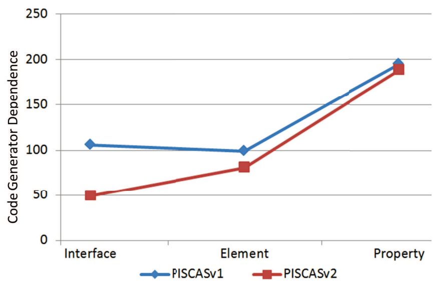
$$\begin{aligned} D_{interface} &= \#LOC_{Relationships} + \#LOC_{Roles} + \#LOC_{Constraints} \\ D_{element} &= \#LOC_{Objects} + \#LOC_{Ports} \\ D_{properties} &= \#LOC_{Properties} \end{aligned} \tag{2}$$

We calculated the two metrics for two versions of the PISCAS DSL. The first version (PISCASv1) did not follow the DSL design guidelines given in Section 3.2. The interface *Roles* carried semantic information which was in some cases redundant. In some other cases this semantic information was later, in the second version, put into the simple 'input' and 'output' *Ports*. The second version (PISCASv2) is a refactored version of PISCASv1 and adheres to the guidelines given in this paper. *Objects* follow the specified interface conventions (input *Port*, output *Port*, name *Property*, voltage *Property*) and, therefore, most *Relationships* became unnecessary for the PISCAS DSL. This makes the domain model a lot simpler, because many *Roles* could be deleted. For the transition from PISCASv1 to PISCASv2, the two *Port* types 'input' and 'output' had to be added. The *Properties* and the number of *Objects* did not change. Most of the *Roles* were removed leading to a much lower interface complexity (see Figure 6(a)). This reduced domain model interface complexity, obviously, lead to a decrease of the code generator interface dependence (see Figure 6(b)).

The code generator element dependence also decreased even though the element complexity of the domain model increased. The lower code generator dependence suggests, that the DSL can easier be modified, because less changes to the code generators are required if domain model items are changed. In particular changes of *Objects* in the domain model seem to have lower affect on the code generators in PISCASv2 due to the decreased code generator object dependence.



(a) Complexity of the PISCAS domain models



(b) Dependence of the code generators from the domain model

Fig. 6. Complexity of the PISCAS domain models

5 Conclusion

In this paper, we presented design decisions to develop a flexible DSL for automation systems. We presented the domain model design and the mapping of automation elements to the DSL. The presented design decisions can be taken as a guideline for automation system DSL developers and can help to develop a flexible and easily maintainable automation DSL.

The proposed DSL design was applied to a fish farm automation system DSL (PISCAS) which was evaluated in terms of modeling effort and maintainability. The maintainability is evaluated by measuring the DSL complexity with domain model and code generator complexity metrics. The introduced code generator dependence metric used for this evaluation works very well to describe the PISCAS code generator complexity. For future work it would be very interesting to evaluate the maturity of the proposed code generator metric by applying it to other code generator based systems. It would also be of high interest to evaluate the proposed DSL design, by developing automation DSLs in other domains by following the proposed DSL guidelines.

Acknowledgments. We would like to thank the company HOFERNET IT Solutions and the FFG for financing the PISCAS project with an 'FFG Innovationsscheck'.

References

1. Dhungana, D., Grünbacher, P., Rabiser, R.: The dopler meta-tool for decision-oriented variability modeling: a multiple case study. Automated Software Engineering 18 (2011)
2. Haselsberger, A.: Design and implementation of a domain specific architecture for programmable logic controllers. Master's thesis, Graz University of Technology, Institute for Technical Informatics (2009)
3. Haxthausen, A.E., Peleska, J.: A domain specific language for railway control systems. In: Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology. ACM (2002)
4. Jiménez, M., Rosique, F., Sánchez, P., Álvarez, B., Iborra, A.: Habitation: A Domain-Specific Language for Home Automation. IEEE Software 26 (Jul 2009)
5. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment. In: Proceedings of CAiSE'96, 8th Intl. Conference on Advanced Information Systems Engineering. Springer (1996)
6. Kern, H., Hummel, A., Kühne, S.: Towards a Comparative Analysis of Meta-Metamodels. In: 11th Workshop on Domain-Specific Modeling. ACM (2011)
7. Leitner, A.: A software product line for a business process oriented IT landscape. Master's thesis, Graz University of Technology, Institute for Technical Informatics (2009)
8. Leitner, A., Kreiner, C., Weiß, R.: Analyzing the complexity of domain models. In: IEEE International Conference and Workshop on the Engineering of Computer Based Systems (2012)
9. Maga, C., Nasser, J., Göhner, P.: Reusable Models in Industrial Automation: Experiences in Defining Appropriate Levels of Granularity. In: 18th World Congress of the International Federation of Automatic Control (IFAC). vol. 18 (Aug 2011)
10. Prähofer, H., Hurnaus, D.: Monaco - a domain-specific language supporting hierarchical abstraction and verification of reactive control programs. In: 8th IEEE International Conference on Industrial Informatics (2010)
11. Prähofer, H., Hurnaus, D., Wirth, C., Mössenböck, H.: The Domain-Specific Language Monaco and its Visual Interactive Programming Environment. In: IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE (2007)
12. Preschern, C.: PISCAS - A Pisciculture Automation System Product Line. Master's thesis, Graz University of Technology, Institute for Technical Informatics (2011)
13. Rossi, M., Brinkkemper, S.: Complexity metrics for systems development methods and techniques. Information Systems 21(2), 209–227 (Apr 1996)

Domain-specific front-end for virtual system modeling

Janne Vatjus-Anttila, Jari Kreku, Kari Tiensyrjä

VTT, Technical Research Centre of Finland, Oulu Finland
{janne.vatjus-anttila, jari.kreku, kari.tiensyrja}@vtt.fi

Abstract. The complexity of software and hardware in embedded systems has risen rapidly due to convergence of diverse applications and adoption of multi-core technologies. Consequently, the abstraction level of system design, modeling and exploration needs to be raised to manage the complexity. The Y-chart approach, typically applied in the system-level performance evaluation, allocates/maps a model of application on a model of execution platform and the resulting system model is simulated to obtain performance data. In this work, Domain-Specific Modeling (DSM) has been adopted as means of raising the abstraction level for building, composing, configuring and checking of high-level models in the virtual system performance modeling and simulation approach, called ABSOLUT. Domain-Specific Languages (DSL) were defined to serve as front-ends for application workload, platform and allocation modeling using the MetaEdit+ tool. The results are demonstrated with a video player case example. First experiences indicate that in performance evaluation related tasks the modeling productivity, model management and ease of learning have improved.

Keywords: DSM, DSL, embedded system, virtual system, SystemC, performance exploration

1 Introduction

The complexity of hardware and software has risen rapidly particularly in advanced embedded system domains, like communication systems, during the recent years due to extensive adoption of multi-core technologies. Real-time embedded systems are often computationally intensive and constrained with limited resources, e.g. power/energy, size and cost. The systems accommodate a large number of on terminal and/or downloadable applications offering the users with numerous services related to telecommunication, video, digital television, internet etc. More flexibility, scalability and modularity are expected from the execution platforms to support the applications. The digital processing architectures will evolve from current system-on-chips to massively parallel computers consisting of heterogeneous subsystems connected by a network-on-chip.

The design complexity requires the elevation of the design process to a higher level of abstraction. At the system level, models of entire platforms can be built that enable hardware-software co-development and rapid, early design space exploration. Such models provide quick feedback for the designers about the effect of their design decisions to critical system metrics like performance. Complex interactions and the highly dynamic nature of systems make their static analysis difficult, which is why such executable models are indispensable [1].

In this work we use ABSOLUT [2] methodology and toolset as a backend for early phase system level performance simulation of embedded systems. The main modeling phases in the virtual system modeling include specifying computing platform's capacity model, application workload and allocation of workload to computing resources of the platform. The result of an allocation is a virtual system model, which can be simulated using the OSCI SystemC simulator to measure performance data, e.g. utilization of platform resources.

Domain-Specific Modeling (DSM) raises the level of abstraction beyond programming by specifying the solution directly using domain concepts. It is used in many application domains and in particular embedded application development with domain specific language (DSL) has been adopted in many companies [3].The final products are generated from these high-level specifications [4]. The benefits of DSM come in many forms like easier modeling/programming, productivity increase, better code quality and maintenance ability [5].

In this paper we propose applying DSM in virtual system modeling domain and present a MetaEdit+[6] based prototype DSL that can be used as a front-end to the ABSOLUT performance modeling and simulation approach. It resembles a traditional Graphical User Interface (GUI) of modeling tool and it can be used like one. We applied the DSL in an example case study and present the enhanced performance evaluation workflow using DSM.

The rest of the paper is structured as follows. In Chapter 2 system-level performance modeling and evaluation and used ABSOLUT approach are described. Chapter 3 discusses Domain-Specific Modeling. Chapter 4 presents the DSL front-end for ABSOLUT approach through applying the DSM method for virtual system model development phases. In Chapter 5 the ABSOLUT workflow and the use of the DSL in it is presented and results of the case study are shown. Chapter 6 gives conclusions.

2 System-level performance modeling and evaluation

Performance evaluation approaches can be divided into three categories: analytical approach, simulations and measurements [7]. The analytical approach is suitable for early performance evaluation, but the accuracy of results is low because it requires many simplifications and assumptions. In simulations, the execution of an application is simulated using a computer program. Simulation provides more accurate results than analysis since it is possible to incorporate more details of the system in the models. Simulations are suitable for early evaluation, since they can be performed before implementations of hardware and/or software are completed. Measurements can be done with real applications, prototypes of applications or benchmark programs that mimic real software. An implementation of the execution platform is, however, required in all cases and therefore measurements are not suitable for early evaluation.

Performance simulation approaches are categorized in the European EDA Roadmap [8] into virtual systems, virtual platforms and virtual prototypes:

- Virtual system approaches combine abstract application models with an abstract execution platform model. The applications are represented using e.g. workload models, traces or task graphs but not as real instructions of processors. The platform model typically has a high abstraction level and capacity models of components instead of instruction set simulators.
- Virtual platform approaches use real application software compiled to binary form in simulations. The execution of the applications is simulated on top of a virtual platform model, which contains one or more instruction set simulators. The platform models need to be functionally complete and use accurate memory maps to be able to execute the application binaries.
- Virtual prototype approaches also use real application binaries with instruction set simulators. The virtual prototype approaches model full functionality of the execution platform using hardware description languages like VHDL or Verilog.

ABSOLUT [2] is a virtual system performance simulation approach intended for early evaluation of embedded computer systems and for exploring the design space. It is also a set of tools, which assist the designer with application and platform modeling, allocation and configuration, simulation and result visualization. It has been applied to several case studies ranging from contemporary mobile phone platforms to future high-performance systems consisting of hundreds of components. Fig. 1 presents the main phases of ABSOLUT performance modeling method.

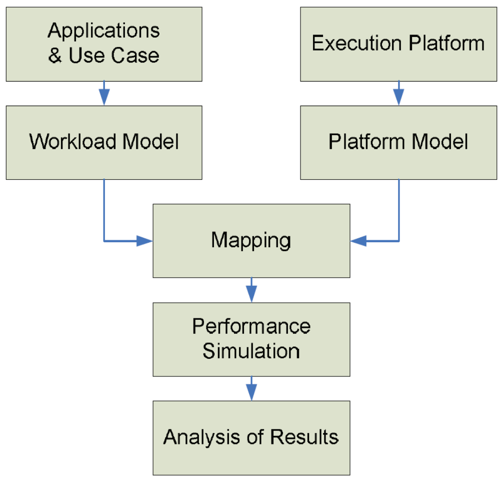


Fig. 1. Y-chart model of ABSOLUT performance modeling

Applications are modeled as layered workload models, which ultimately consist of abstract, instruction-like workload primitives. Several techniques have been developed to create workload models from information sources such as application specifications or execution traces. A compiler-based tool exists to create workload models automatically from application source code [9].

As application functionality is abstracted, the complexity of execution platform models is also reduced especially with respect to the processing elements. The data paths of processing elements need not be modeled in detail and data transfers and storage are simulated only from the performance point of view. A capacity model of platform can be rapidly constructed from components in a model library with the help of a platform generation tool.

The virtual system model, constructed by allocating workload models on top of the platform components, is simulated using the IEEE standard SystemC [10] simulation kernel and models based on the TLM standard [11]. Performance, power and energy consumption is obtained from simulation by instrumenting the workload and/or platform models with custom performance probes. Designer can freely set the probes and extract e.g. resource utilization, execution latencies or interconnection traffic.

3 Domain-Specific Modeling

Basically, Domain-Specific Modeling is creating and using modeling languages for specific purposes. Domain-specificity of modeling means that key concepts of each domain can be used as the modeling elements of the language. Modeling elements can have graphical symbols and the use of DSL is actually placing these elements on diagram according language modeling rules. Diagrams made with DSL are formal descriptions of systems and applications and as such suitable for documentation, analysis and transforming them to other forms like source code or other artifacts. In particular, the generation features complete the benefits that DSM and DSLs provide.

Tool support for applying DSM exists in commercial, academic and open source tools [12]. True DSM tools enable developing of new DSL, which can be run on top of the tool or as a standalone program depending of the tool. Increasing interest towards DSM has brought DSM features also to IDEs [13].

DSLs are often made to very specific domains inside companies, which are not in public use. However, DSLs are applied in the graphical user interfaces of some embedded system development tools, too. For example, CoFluent Studio [14] and Simics [15] include DSL that can be used for modeling of explored system. These apply for somewhat similar purposes as ABSOLUT and have therefore similar modeling phases. However, the modeling notions differ due methodology differences. In addition, it is important to notice the fundamental difference between the ABSOLUT DSL front-end and the DSLs of mentioned tools. Neither GUIs nor DSLs of the mentioned tools are implemented on an actual DSM tool as the front-end presented in this paper is.

4 DSL front-end for virtual system modeling

Virtual systems for performance simulations are often made from existing components and in an ideal case, the modeling should not require much coding. However, the platform model needs to be composed and configured. The application modeling requires making a high-level model of application. The allocation of application elements to the processing elements of platform model is one of the modeling phases.

In this work, the DSM approach is applied to the modeling phases. The same DSM environment that consists of a DSM tool, a few DSLs and ABSOLUT tools applies also for building simulateable models, running simulations and analyzing results.

4.1 DSM workflow and environment

Although every domain has its special features, a workflow consisting of the next four phases for developing DSLs for different domains can be used [16]:

1. Identifying abstractions and how they work together
2. Specifying the language concepts and their rules (meta-model)
3. Creating the visual representation of the language (notation)
4. Defining the generators for model checking, code, documentation, etc.

In our case, the Y-chart model shows clearly the modeling phases and their interrelations. Consequently, it was natural to proceed towards modeling phase specific abstractions. The existing ABSOLUT model library and experiences of tools of the domain were the basis according which the language concepts, properties, rules and notations were defined. The notation design in prototype development phase had naturally low priority. In the generator definitions, the goal was to enable the usage of the existing ABSOLUT tools and generation of compatible mid- representations.

Proceeding according to the workflow requires expertise of both the problem domain and the DSM. Additionally an appropriate DSM tool is needed. In this work, the DSM tool MetaEdit+ 4.5 Workbench [6] developed by MetaCase has been used and the solution presented here contains some tool specific notions. The tool provides different diagram editors for modeling with the DSM languages, support for the DSL and generator definition and is upgraded with new versions by the tool vendor.

4.2 DSL for workload modeling

The workload modeling DSL is developed for the compiler based workload generator of ABSOLUT, which produces workload models from normal C/C++ code. Any application modeling DSL that can generate C code can therefore be used in producing the ABSOLUT compatible workload model. Existing C code is also used for workload generation. The workload modeling DSL of ABSOLUT front-end is targeted for workload model and workload trace configuration and generation.

The workload modeling is made with the diagram presentations of *Workload Modeling Graph*, on which the DSL objects and connections are placed. Fig. 2 presents an example of such a diagram. It contains two *Workload Model* objects, which both are linked to two *Workload Trace* objects. The object symbols contain symbol names and information about the status of the workload models and traces. The MetaEdit+ toolbar contains the modeling elements of the graph type and the generator buttons.

The workload modeling scheme consists of two phases and both have an own object notation in this solution. The *Workload Model* object is used for defining and generating the workload model according to the object properties. Properties are used to select the valid external workload model generator, the generation script and the source code folder. The MERL workload model generator uses the property information and generates the workload model. Several *Workload Model* objects can be used in a single diagram to generate different workload models from the same source code or from many different source codes.

The *Workload Trace* object is used to make the workload trace that can be allocated to the platform model. It needs a link to the *Workload Model* object defining the workload model that is executed in trace generation. The properties of the *Workload Trace* object define all the parameters that are needed to execute the workload model. The object has also a property, which defines path used to store the generated trace.

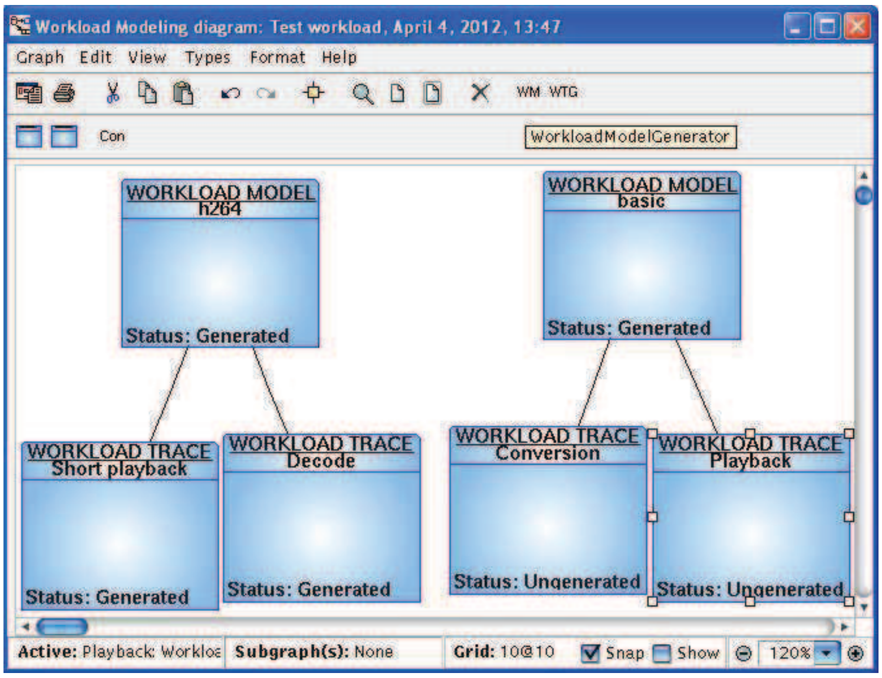


Fig. 2. Diagram presentation of *Workload Modeling Graph*

The MERL generator first runs the workload model that produces the workload trace and then stores the trace files to defined path. Generated workload trace consists of files of which each contains trace of one workload model thread. Several *Workload Trace* objects can be linked to single *Workload Model* object, which enables producing different workload traces from the workload model based on the parameters.

4.3 DSL for platform modeling

The key concepts that are needed in the platform modeling are the various hardware elements (see Table 1). The element types and their properties are specified with different property sets. In addition to the objects, some relationship types and role types are needed for linkage between hardware elements. If large platforms are modeled, the sub-system object could be applied in the DSL too. Modeling rules can be included in the DSL to prevent the designer from making impossible or erroneous connections between elements or other kind of modeling mistakes.

A diagram editor is the only alternative for a block based platform modeling. Modeling work convenience depends of the modeling elements. By using different symbols for the language concepts, perceiving of the platform model is easier. Different shapes, colors and text of the component objects symbols enable this. The relationship and role symbols can also have tuned symbols, if there are many of them or if they have properties.

Generators can have more than one purpose in the platform modeling DSL. Checkup generators are another way to confirm the platform validity in addition to the rules set in the DSL definition. The main purpose of the generator in the platform modeling is to produce a description of a platform that can be used in the simulation phase. Generators can also be used to update the modeling element list, which is important when the components are modeled elsewhere.

Our DSL for platform modeling consists of platform objects, their relationships, roles, modeling rules and the platform model to XML generator. The objects and other elements of the platform modeling DSL are listed in Table 1.

Table 1. Modeling elements of platform modeling DSL.

Element	**Description**
Processor	Object is used to model processor and accelerator components.
Memory	Object is used to model memory components.
Bus	Object is used to model bus components.
Interface	Object is used to model interface of subsystem.
Subsystem	Object is used to model subsystems of platform.
Router	Object is used to model connections between subsystems.
Connection	Relationship used to model all connections in the platform diagram.
Master	Role that connects master component to connection.
Slave	Role that connects slave component to connection.

The integration of the DSL with ABSOLUT and its component library is established by importing the component library description to the DSL. The import updates the pull down list of each hardware element. For example, a new processor type can be selected from the pull down list of *Processor* object types when an updated ABSOLUT component library definition has been imported.

An example ARM processor platform, modeled in a diagram presentation of the *Platform Modeling Graph* is presented in Fig. 3. Using of the platform modeling DSL resembles the use of many other platform-modeling tools. Components are picked from toolbar, placed, connected to other components and configured from the properties of the particular component type.

Subsystem objects are also defined in the DSL to help the modeling of large platforms in smaller parts. When they are used, the top level of platform is composed from *Subsystem* and *Router* objects. The architecture of each subsystem can be modeled with a separate platform diagram as Subsystem object decomposition. The i*nterface* objects are used inside them to define how they are connected to other subsystems.

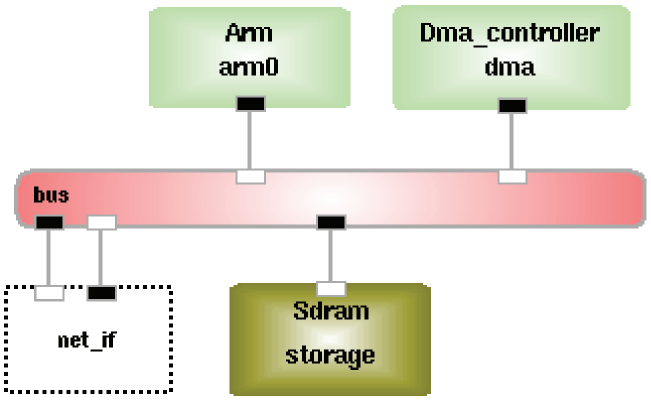


Fig. 3. Platform model made with platform modeling DSL

The MERL generator is used to generate XML descriptions of the modeled platforms. The generator goes through all the platform diagrams that are related to the platform model and includes their connections, components and the whole platform structure into the XML file.

4.4 DSL for allocation

An allocation DSL should contain objects presenting application model elements and computing resources of platform. Some way to link application model elements to selected platform model elements is also required. A generator for generating allocation file belongs also to allocation DSL.

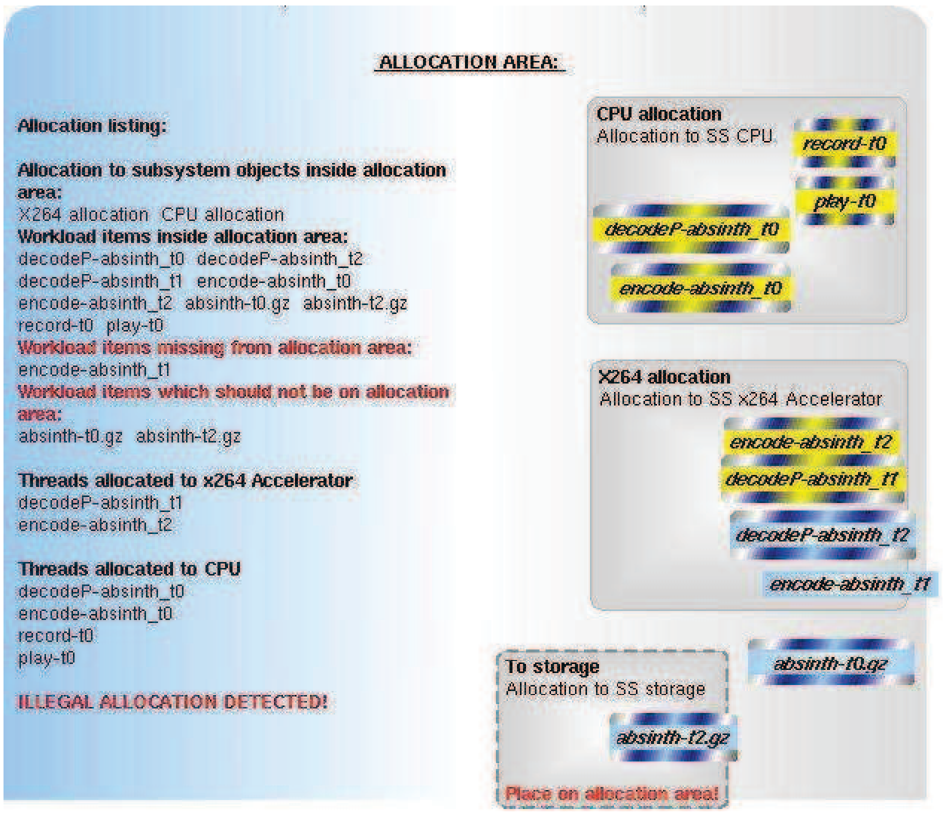
Object locations on a diagram can be utilized with the used DSM tool for building the allocation DSL. The links between application and platform elements can be based on object locations. The allocation DSL uses the *Allocation Area* and the *Allocation to Resource* allocation objects. The application elements are items of the workload, and the *Workload Thread* objects and the *Workload Thread Group* objects are used in the allocation diagram in addition to the allocation objects. The location of workload objects with respect to the allocation objects defines the allocation.

There are several ways to bring workload item objects to the diagram. They can be picked from the object list, which shows existing objects, or they can be created manually from the scratch. They can also be generated according to corresponding workload traces or according prompt input.

The MERL generator producing the allocation file detects all the *Allocation to Resource* objects that are placed on the *Allocation Area* object. The generator also detects items of the workload on top of the *Allocation to Resource* objects and forms the allocation file accordingly. When a workload item is on the *Allocation to Resource* object, which has the valid resource property and the *Allocation to Resource* object is on the *Allocation Area* object, the workload item is correctly placed. Warnings are generated on the allocation objects when the objects are placed wrongly. The *Allocation Area* object contains also a listing of the allocation, which changes according to the locations of the other objects in the diagram. The generator producing the listing can also print warnings, because the *Allocation Area* object has information of the workload items and the platform resources, which can be used in composing the allocation.

Dynamic symbols are used as guidance for the designer towards valid allocations. The symbols of workload objects are made dynamic and the symbols are changing during the allocation work depending how they are placed. A correctly placed workload item has symbol, which is emphasized with yellow. Wrongly placed workload item has two different symbols.

Fig. 4 shows an example allocation, modeled in a diagram presentation of the *Allocation Modeling Graph*. It contains the *Allocation Area object*, *Allocation to Resource* objects and *Workload Thread objects*. The allocation in it contains errors, which are reported in the allocation listing.


ALLOCATION AREA:

Allocation listing:

Allocation to subsystem objects inside allocation area:
X264 allocation CPU allocation
Workload items inside allocation area:
decodeP-absinth_t0 decodeP-absinth_t2
decodeP-absinth_t1 encode-absinth_t0
encode-absinth_t2 absinth-t0.gz absinth-t2.gz
record-t0 play-t0
Workload items missing from allocation area:
encode-absinth_t1
Workload items which should not be on allocation area:
absinth-t0.gz absinth-t2.gz

Threads allocated to x264 Accelerator
decodeP-absinth_t1
encode-absinth_t2

Threads allocated to CPU
decodeP-absinth_t0
encode-absinth_t0
record-t0
play-t0

ILLEGAL ALLOCATION DETECTED!

Fig. 4. Allocation is illegal because two workload items, irrelevant to selected workload, are placed on the allocation area and one relevant item of workload is not inside allocation objects.

5 Modeling with ABSOLUT DSL

The updated workflow of the ABSOLUT modeling enhanced by the developed DSLs is described using a video player as a demonstration case.

5.1 ABSOLUT workflow with DSL

The ABSOLUT workflow presented earlier in Fig. 1 does not change dramatically because of DSLs. The Y-chart flow is still the basis of the modeling method. The main changes are related to the modeling phases, which are enhanced with the DSLs. The developed front-end can also be used for performance simulations and simulation result analysis but this does not change the overall workflow.

The workload modeling DSL gathers all workload modeling information to one diagram. The workload modeling can be therefore managed more efficiently. Different versions of workload models and workload traces can be generated in a controlled way and stored in an appropriate location.

The platform modeling DSL speeds up the composition of platform models from the ABSOLUT model library components and makes it less error prone by avoiding manual editing of XML documents. The platform description generator produces an XML file, which ABSOLUT tools need to produce the platform source code.

Allocation DSL enables drag and drop like method for allocation of workload items to platform resources. Workload items can be imported which reduces effort. It guides the designer and alerts from illegal allocations. In addition to the graphical allocation, a textual allocation listing is visible during the allocation phase. Allocation description files are generated in a format suitable for the ABSOLUT tools.

The generators are used to start ABSOLUT tools and compilers, which produce the simulation model. There is also a generator, which runs the performance simulation.

5.2 Case study

The video player case study has been made with the described front-end. The player uses H264 high definition video coding which is used in high quality mobile devices. The ABSOLUT workload model was generated from the open source FFMPEG [17] source code. The workload generation DSL was used to set the *Workload Model* configuration and single and two thread configurations of the *Workload Trace* object. Workload traces were generated by the generator that is used to execute the workload models.

Our test case used an OMAP4 like platform [18]. The platform model was composed from platform modeling elements with our platform modeling DSL. The platform model is a simplified version of the OMAP4 but e.g. includes the dual-core CPU for testing of different allocations. The XML description of the modeled platform was produced with the platform description generator.

Two different allocations were made with the allocation DSL so that the effect of changing the workload allocation could be detected. The workload items were imported and allocation objects instantiated. Then the allocation was composed and the allocation description files generated. The same was done for both of the workloads.

SystemC simulations were performed with the OMAP4 platform model and two pairs of allocation files and workload models. The utilization of the ARM cores was the property, which was measured from the simulations. The single-threaded version of the application utilizes the Core 0 100 %, but the Core 1 is not used at all. In the dual-threaded version, the load is evenly divided to both cores, which shows that platform capacity is nicely harnessed. The observed utilization rates for the dual-core CPU of the multi-threaded case are presented in Figure 5.

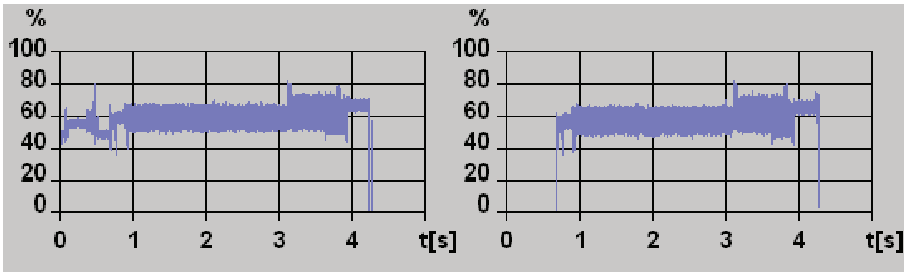


Fig. 5. Data processing load of both cores in the two thread video coding case.

6 Conclusions

This paper describes a way of utilizing the DSM method on the domain of virtual system modeling. The resulting prototype DSL set can be used as a modeling and simulation front-end to the ABSOLUT virtual system modeling tools.

Our experience of DSM was near to zero when the work began and notion of DSM has become clearer in the course of the work. The experiences of the DSM and from the used DSM tool are positive. The DSLs were developed incrementally in smalls steps and they were tested with example data. All three DSLs have evolved steadily without major tool or DSM approach related problems.

The developed DSLs improve the usability of the ABSOLUT, which so far has been used without a graphical front-end. Especially the learning curve shortens due more user-friendly modeling. In particular, the DSL front-end makes modeling easier for designers who are not experienced with SystemC [10] and TLM [11].

The phases of the ABSOLUT virtual system modeling - workload modeling, platform modeling and allocation modeling - were carried out with the developed DSL in a video player case study. The performance simulation was carried out for two system models from which the performance data was recorded. According to our experiences, the DSL-aided workload, platform and allocation modeling is a workable idea.

Our work continues with the refinement of the ABSOLUT DSL. The usage of a DSM tool for simulation observation and simulation result analysis front-end is also an interesting research direction that has already been pretested. There are also possibilities to explore how this sort of DSM approach suits to other embedded system modeling phases.

Acknowledgements. This work is supported by the European Commission and Tekes – the Finnish Funding Agency for Technology and Innovation - under the grant agreement ARTEMIS-2010-1-269362 PRESTO.

The adoption of DSM principles and environment was guided by experts of MetaCase. Especially Mr. Janne Luoma contributed in solving issues faced during the work. Dr. Juha-Pekka Tolvanen receives also our gratitude for professional comments.

7 References

1. Gerstlauer, A., Chakravarty, S., Kathuria, M., Razaghi, P.: Abstract System-Level Models for Early Performance and Power Exploration. In 17th Asia and South Pacific Design Automation Conference, pp. 213-218. IEEE (2012).
2. Kreku J., Hoppari M., Kestilä T., Qu Y., Soininen J.-P., Andersson P., Tiensyrjä K. Combining UML2 Application and SystemC Platform Modelling for Performance Evaluation of Real-Time Embedded Systems, 18p. EURASIP Journal on Embedded Systems. Hindawi Publishing Corporation (2008).
3. Sprinkle, J., Mernik, M., Tolvanen, J-P., Spinellis, D., What Kinds of Nails Need a Domain-Specific Hammer?, IEEE Software, July/Aug, 2009.
4. DSM Forum, http://www.dsmforum.org/ (2012).
5. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling full code generation, Wiley-IEEE Computer Society Press (2008).
6. Domain-Specific Modeling with MetaEdit+, http://www.metacase.com (2012).
7. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling, 685 p.. John Wiley & Sons, Inc. (1991).
8. European EDA Roadmap. Technical report, 352 p. CATRENE (2009).
9. Kreku, J., Tiensyrjä, K., Vanmeerbeek, G.: Automatic workload generation for system-level exploration based on a modified GCC compiler. In Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 369-374. IEEE (2010).
10. Grötker, T. and Liao, S. and Martin, G. and Swan, S.: System design with SystemC. Springer, 2002.
11. SystemC Transaction-level Modeling Standard, TLM-2.0, http://www.accelera.org (2012).
12. DSM Tools, http://www.dsmforum.org/tools.html (2012)
13. Eclipse Modeling Project, http://www.eclipse.org/modeling/ (2012).
14. CoFluent Studio, http://www.cofluentdesign.com (2012).
15. Wind River Simics, http://www.windriver.com/products/simics/ (2012).
16. Tolvanen, J.-P.: Domain-Specific Modeling: How to Start Defining Your Own Language, http://www.devx.com/enterprise/Article/30550 (2006).
17. FFmpeg, http://www.ffmpeg.org/ (2012).
18. OMAP Mobile Processors, http://www.ti.com/ (2012).

A Comparison of Ecore and GOPPRR through an Information System Meta Modeling Approach

Vladimir Dimitrieski, Milan Čeliković, Vladimir Ivančević and Ivan Luković

University of Novi Sad, Faculty of Technical Sciences
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
{dimmy, milancel, dragoman, ivan}@uns.ac.rs

Abstract. In this paper we present a comparison between the main concepts of the Ecore meta-meta-model and the GOPPRR meta-language. Through our previous research we have specified the PIM concepts of our model driven software development tool for information system design IIS*Case using Ecore implementation of Meta object Facility 2.0 in Eclipse Modeling Framework (EMF). We have also modeled the same concepts by Graph-Object-Property-Port-Role-Relationship (GOPPRR) meta-modeling language provided by the MetaEdit+ meta-modeling environment. Both MetaEdit+ and EMF provide the environment for the meta-models specification. In this paper we give a brief overview of MetaEdit+'s and EMF's main concepts and syntax, as well as an example of IIS*Case PIM concepts modeling using Ecore and GOPPRR meta-meta-models. We also present the main differences between two meta-modeling environments, EMF and MetaEdit+, from the PIM concept modeling point of view.

Keywords: Model Driven Approaches, Domain Specific Languages, Domain Specific Modeling, EMF, MetaEdit+, Information System Modeling.

1 Introduction

Domain-specific languages (DSLs) are special-purpose languages designed to solve a particular range of problems. DSLs are tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use in their domain of application, compared with general-purpose programming languages.

Nowadays, DSLs are of increasing importance for the development of software and other systems. In specifications of DSLs, visual notations are often used. They are to be supported by a tool environment consisting of visual editors, simulators and model transformers. Existing approaches for generating or adjusting the desired tool environments rely on meta-modeling concepts, grammars, or some kind of logics. Dependent on the underlying concepts, different kinds of editors are generated.

Several Eclipse projects are heading towards meta-technology to define DSLs. Eclipse Modeling Framework (EMF) [1] mainly generates the underlying models of visual and textual editors that may be extended by additional syntax checks, implementing certain rules, using Object Constraint Language [2]. A graphical view of visual editors can be hand coded on the basis of Eclipse Graphical Editor Framework (GEF) [3] or generated using Graphical Modeling Framework (GMF) project [4].

MetaEdit+ [5] is the metaCase tool for development of DSLs. MetaEdit+ is an integrated, repository-based tool aimed at creating and using modeling languages and code generators. It provides a tool support for different modeling languages by configuring the generic tool set with meta-models.

DSLs may take a distinguished role in the modern information system (IS) development process. In IS development DSLs may be used for various purposes, such as: conceptual modeling, specification of rules and constraints, code generation, generation of test cases, specification of transformations between models etc. A detailed overview of DSL usage in the context of Model Driven Software Development (MDSD) and IS development may be found in [6]. Through our research, we are developing a textual DSL, named IIS*CDesLang. It is aimed at modeling PIM specifications of an IS. Our research goals are to couple it with our model driven software development tool, named Integrated Information Systems CASE Tool (IIS*Case). IIS*Case provides IS modeling and prototype generation. At the level of PIM specifications, IIS*Case provides conceptual modeling of database schemas and business applications. Performing a chain of model-to model and model-to-code transformations of PIM models, we obtain executable program code of software applications and database scripts for a selected platform.

In order to provide design of various platform independent models (PIM) by IIS*Case, we have created a number of modeling, meta-level concepts and formal rules that are used in the design process. Our experience from previous research [6, 8, 10], leads to the conclusion that there was a strong need to have PIM concepts specified formally in a platform independent way, i.e. to be fully independent of repository based specifications that typically may include some implementation details.

Our current research is based on three related approaches to formally describe IIS*Case PIM Concepts. The first one is based on the attribute grammars through which we are developing the textual DSL, named IIS*CDesLang. In [8], we present IIS*CDesLang. It formalizes IIS*Case PIM concepts and provides modeling in a formal way. IIS*CDesLang meta-model is developed under a visual programming environment for attribute grammar specifications named VisualLISA [9].

The second approach is based on Meta Object Facility (MOF) [7]. MOF 2.0 is a common meta-meta-model proposed by Object Management Group (OMG) where a meta-model is created by means of UML class diagrams and Object Constraint Language (OCL). This approach is presented in [10]. As we could not find standardized implementation of MOF, we selected Ecore meta-meta-model to implement PIM model, since MOF 2.0 is widely used meta-modeling framework. Ecore is the Eclipse implementation of MOF 2.0 in Java programming language which is provided by EMF. We deploy it to implement a meta-model as a basis for textual and graphical DSL we plan to build in the future.

In the last approach, we deploy MetaEdit+'s GOPPRR [5, 14, 15] as a meta-modeling framework to describe our PIM concepts. MetaEdit+ provides an integrated environment for definition of PIM concepts as well as the definition of their graphical representation using graphical symbol editor. After the definition of meta-model, specified concepts are to be loaded into the MetaEdit+'s repository and then used to define IS models through graphical representation of these concepts. Through our previous research, we have gained a valuable experience in the practical application of Domain Specific Modeling (DSM) for creating our PIM meta-model using different environments and paradigms.

In this paper we present a comparison of EMF and MetaEdit+'s meta-modeling environments and their respective frameworks through modeling of the same PIM concepts. This comparison comprises our previous practical experiences. It is based on the evaluation of environments' meta-language concepts through the comparison of their ease of use. Our goal is to identify the advantages and disadvantages of each environment as both of them are used in implementation of meta-models as a basis for further textual and graphical DSL development. Although we have modeled all IIS*Case PIM concepts using both environments, in this paper we chose to present most representative parts of our PIM concepts, on which we are able to see the true difference between EMF and MetaEdit+.

Apart from Introduction and Conclusion, the paper is organized in three sections. In Section 2 we present related works. In Section 3 we present a comparison of EMF and MetaEdit+'s basic concepts, while in Section 4 we give a presentation of IIS*Case's *FormType* and *FormTypeUsage* PIM concepts specified through the meta-models implemented in MetaEdit+ and EMF.

2 Related Work

Nowadays, meta-modeling is widely spread area of research and there is a huge number of references covering this area. However we could not find a lot of papers, relevant to the comparison of DSM tools. We have found only three of them, presenting EMF and MetaEdit+ concepts and providing their comparison.

In [11] , the author presents the comparison between GEF workbench environment and MetaEdit+. The author also proposes a DSL named Logic Gate Language, for the description of logic circuits models. The language was developed both under MetaEdit+ and GEF. The author reported that, in general, the process of the implementation was much faster in MetaEdit+ than in GEF. GEF also required much more Java language coding, than MetaEdit+.

In [12], the author presents a mapping between MetaEdit+ and EMF meta-meta-level concepts. He proposes the M3-Level-Based Bridges solution that provides interoperability between MetaEdit+ and EMF, i.e. an interface for the exchange of meta-models and models between the two tools. Transformations between models at the M2-level and M1-level have been implemented as the Eclipse plug-in. In this way, the bridge may be used for the model re-usage.

In [13] the authors analyzed a set of meta-modeling languages including Ecore and GOPPRR. To compare the selected meta-meta-models, they defined criteria for their comparison and proposed a comparison framework consisting of abstractions of meta-modeling concepts available in each meta-modeling language. As the last step, the authors evaluated obtained results according to the three aspects: availability of the meta-modeling concepts, definition of relationships and the concepts of structuring, reuse and modularization in meta-modeling.

In [11], development of a graphical DSL is considered. However, we base our comparison on the modeling of PIM concepts that can be later used for development of either textual or graphical DSLs. In [12] the focus is set to mappings between concepts of two meta-languages, only. In [13], the authors compared meta-modeling languages with respect to diversity of the meta-modeling concepts. However, we focus not only to the similarities and differences of the concepts, but also considerations regarding their practical usage.

3 The Main Concepts of EMF and MetaEdit+

Here we give a brief overview of main Ecore and GOPPRR concepts used in the specification of IIS*Case PIM concepts. We make a comparison between the concepts existing in both meta-meta languages that are used in conceptual specification of meta-models. Although the Ecore provides only conceptual structures, GOPPRR concepts also include information about graphical representation of elements. Concepts used in our PIM specification that are specific to one of the meta-meta languages only, are also described. A presentation of basic concepts of both meta-languages is based on a comparison of the corresponding meta-modeling concept classes.

A *grouping concept* allows to structure meta-model elements in defined parts or modules. Regarding their grouping characteristics, Ecore and GOPPRR contain similar concepts. *EPackage* is the Ecore concept used for the model organization. It groups the instances of all Ecore concepts into one logical unit. EPackage's name need not be generally unique. Instead, a URI is used to uniquely identify the package. GOPPRR’s concept for grouping elements is the *Graph* concept. Graph is a collection of objects, relationships, roles and bindings. Graph contains all elements and their explosions to other graphs. Explosion allows each object, relationship or role in a graph to be linked to other graphs. Additionally, Graph can contain properties that describe it further.

A *class concept* defines a class of objects with the same characteristics. A class is the blueprint from which the individual objects are created. *EClass* is an Ecore concept used to define set of model entities. The corresponding concept used in GOPPRR is the *Object*.

A *relationship concept* describes a connection between elements of the model and is a subset of the Cartesian product over the participating object types. *EReference* is an Ecore concept that defines a set of relations between objects. It establishes the link from one EClass instance to another. As the EReference instance links at most two objects, it represents the binary relationship. GOPPRR owns a similar concept named *Relationship*. An instance of Relationship differs from EReference instance, as it may have own properties describing the relationship. It can also link more than two Object instances, of the same or different Object concept. GOPPRR Relationship instances attach to objects via roles and they can define properties for the objects’ connections. They are used to form bindings with objects and roles. *Role* concept exists only in GOPPRR and no direct equivalent concept exists in Ecore. This concept specifies the lines and end-points of relationships and describes how an object participates in a relationship. *Object Set* and *Bindings* are also the concepts existing in GOPPRR only. An object set describes a collection of objects with the same role in a binding. Bindings contain the information about how the objects, ports, roles and relationships in a Graph are connected. *Inheritance* is a special kind of a relationship that allows creating subtypes of other language concepts. Both Ecore and GOPPRR provide this concept. In Ecore only the class concept can be inherited, whereas in GOPPRR all metatypes can be inherited.

An *attribute concept* is a property of a meta-model element. At a model level an attribute can hold concrete values. *EAttribute* is the Ecore concept used to define the characteristics of the EClass instances. *EDataType* is another Ecore concept used for the specification of the EAttribute instances type. The similar concept to EAttribute in GOPPRR is the *Property* concept. Both of them have the same behavior in the usage for primitive attribute data types. Additionally, Property concept may represent the link to another object, specifying it as an object member. In Ecore this is accomplished using the EReference concept.

4 IIS*Case Meta-model

Through our previous research we have formally described IIS*Case PIM models using Ecore and GOPPRR meta-modeling languages and attribute grammars. As we used Ecore and GOPPPR to describe same PIM concepts we found some similarities and some differences in these approaches. In this chapter we describe those findings through detailed description of *FormType* and *FormTypeUsage* PIM concepts. We also give a brief overview of other main PIM concepts: *Project*, *ApplicationSystem*, *ApplicationType*, *BusinessApplication* and *Fundamentals*. Modeling of the IIS*Case PIM concepts is organized through the package concept in EMF. In order to provide grouping of elements inside a graph in MetaEdit+, we added a ***GraphGroup*** concept to the graphical meta-model of MetaEdit+. It is represented in form of dotted rectangle surrounding concepts that need to be grouped as one logical unit. *GraphGroup* element contains a name of the group shown in the top left corner of the graphical representation.

4.1 A Brief Overview of Main PIM Concepts

Everything that exists in IIS*Case's repository is always stored in a context of a project. Therefore, the central concept of the meta-model illustrated in Figures 1 and 2 is the concept of a *Project*. In MetaEdit+ model we restricted the number of *Project* instances to one instance per graph. As one project is one IS specification, a designer may have only one instance of *Project* in a single MetaEdit+'s graph. Also, a designer may not create two projects with the same name as the project's name must have globally unique value.

As it is shown in Figures 1 and 2, *ApplicationSystems* and *Fundamentals* are subunits of a *Project*. Every instance of a *Project* may be connected to zero or more instances of the *ApplicationSystem* and zero or more instances of any descendant of

Fundamentals. *ApplicationSystems* are organizational parts, i.e. segments of a project. Designers of an IS may create application systems of various types. By the *ApplicationType* concept, designers introduce various application system types and then associate each application system instance with one application type.

Fig. 1. A Meta-Model of the Main IIS*Case PIM Concepts in MetaEdit+

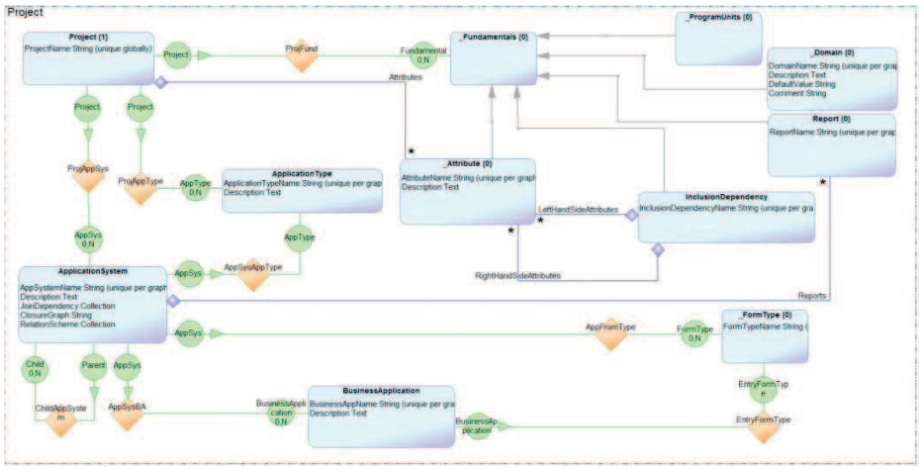
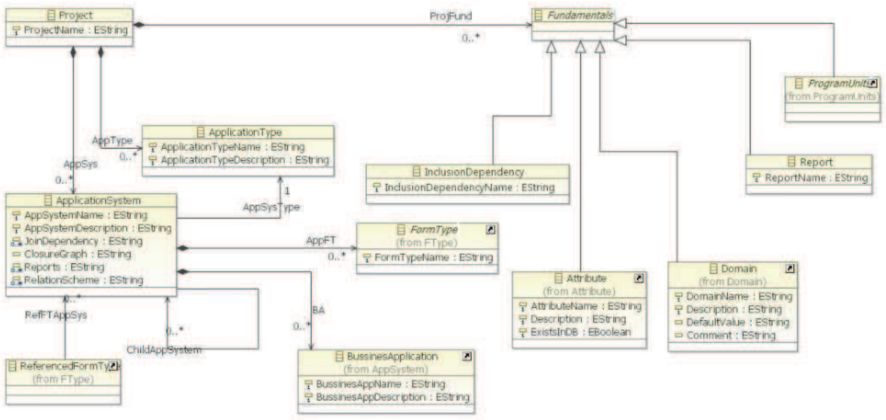


Fig.2. A Meta-Model of the Main IIS*Case PIM Concepts in EMF



Fundamentals (Fundamental concepts) are formally independent of any application system. They are created at the level of a project and may be used in various application systems latter on. Fundamentals comprise zero or more: *Attributes*, *Domains*, *ProgramUnits*, *Reports* and *InclusionDependencies*.

BusinessApplication represents an IS functionality and is organized through a structure of form types. Each business application has a mandatory name and description. One of the form types included into the application system structure must be declared as the entry form type of the business application. It represents the first transaction program invoked upon the start of the business application.

4.2 *FormType*

Form type is the main concept in IIS*Case. The meta-models of this concept are presented in Figures 3 and 4. It abstracts document types, screen forms, or reports that end users of an IS may use in a daily job. By means of the Form type concept, a designer indirectly specifies at the level of PIMs a model of a database schema with attributes and constraints included, as well as a model of transaction programs and applications of an IS.

Each form type has a name that identifies it in the scope of a project, a title, frequency of usage, response time and usage type. All these properties are mandatory. In MetaEdit+ there is no built-in mechanism to declare mandatory properties. Instead, a user is to specify a regular expression option for every property. In EMF this kind of constraint is easier to specify by setting the lower and upper bounds of cardinality. In MetaEdit+, the regular expression option provides a more powerful mechanism for specifying properties' value characteristics.

Frequency is an optional property that represents the number of executions of a corresponding transaction program per time unit. Response time is also an optional property specifying expected response time of a program execution. By the usage type property, we classify form types as: a) menus and b) programs.

Menu form types are used to model menus without data items. Program form types model transaction programs providing data operations over a database. They may represent either screen forms for data retrievals and updates, or just reports for data retrievals. As a rule, a user interface of such programs is rather complex.

Apart from creating form types in an application system, a designer may include form types created in other application systems. Therefore, we classify form types as: a) owned and b) referenced. A form type is owned if it is created in an application system. It may be modified later on through the same application system without any restrictions. A referenced form type is created in another application system and then included into the application system being considered. All the referenced form types in an application system are read-only. In EMF we have modeled the Form Type concept by the Inheritance rule. We have the abstract class named *FormType.* It is superordinated to the classes: *OwnedFormType* and *ReferencedFormType*.

A main advantage of GOPPRR's relationship properties may be used in the modeling of referenced form type concept. Through GOPPRR's concepts we have modeled referenced form type as the relationship named *CallFT*. This relationship is illustrated in Figure 3. The relationship has a property named "Options", which is an instance of *Options* object. Similarly, property named "CalledFTParameters" is a collection of *CalledFormTypeParameter* object instances.

Fig.3. A Meta-Model of *FormType* concept in MetaEdit+

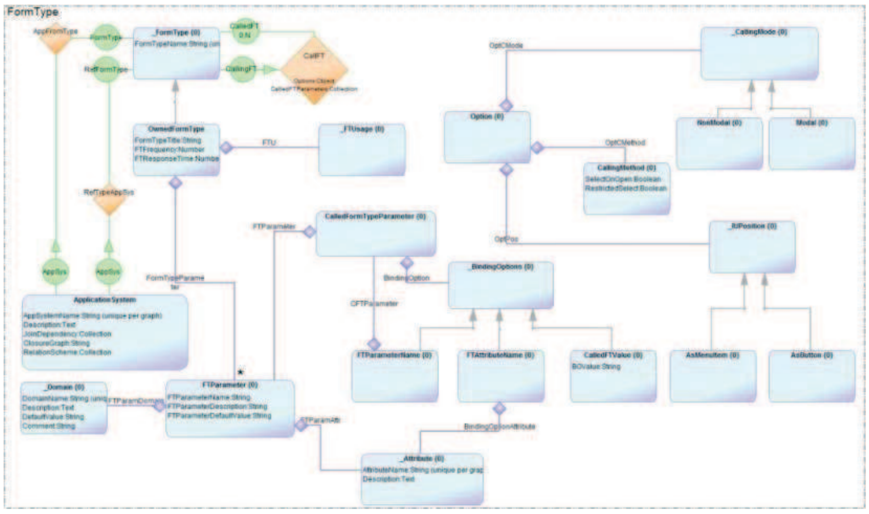
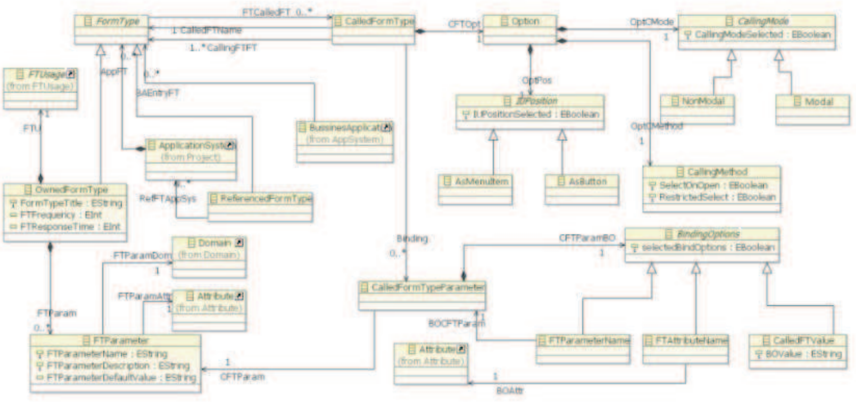


Fig.4. A Meta-Model of *FormType* concept in EMF



Calling referenced form requires some options and form parameters to be set. *Options* contain *CallingMode*, *CallingMethod* and *UIPosition* of the element calling the form. Every form can be called in a modal or non-modal mode. Modal called form must be closed before a user can continue to work in a calling form. Non-modal form specifies that a calling and the called form may exist simultaneously opened on the screen. *CallingMethod* specifies two parameters “Select on open” and “Restricted select”. Select on open specifies if a called program, generated from the called form type, will be opened with an automatic data selection, during the call execution, or not. Restricted select parameter specifies if a called program, generated from the called form type, will be opened in a way to allow only selection of data restricted to the values of passed parameters. *UIPosition* specifies if a called form is shown as a menu item or button in the calling form.

CalledFormTypeParameter in MetaEdit+'s meta-model or *Binding* in EMF's meta-model, is a definition of parameter that is being passed in a form type call. For each form type call parameter we may select this parameter for binding and, if it is selected, to define how a real argument value will be passed to that parameter.

4.3 *FormTypeUsage*

All *FormTypeUsage* concepts are presented in Figures 5 and 6. Each program form type is a tree of component types. A component type has a name, title, number of occurrences, allowed operations and a reference to the parent component type, if it is not a root component type. Name is the component type identifier. All the subordinated component types of the same parent must have different names. Each instance of the superordinated component type in a tree may have more than one related instance of the corresponding subordinated component type. The number of occurrences constrains the allowed minimal number of instances of a subordinated component type related to the same instance of a superordinated component type. It may have one of two values: 0-N or 1-N. The 0-N value means that an instance of a superordinated component type may exist while not having any related instance of the corresponding subordinated component type. The 1-N value means that each instance of a superordinated component type must have at least one related instance of the subordinated component type. The allowed operations of a component type denote database operations that can be performed on instances of the component type. They are selected from the set {query, insert, update, delete}.

A designer can also define component type display properties that are used by the program generator. The concept of component type display is defined by properties: window layout, data layout, relative order, layout relative position, window relative position, search functionality, massive delete functionality and retain last inserted record.

Each component type attribute provides defining a "List of values" (LOV) functionality. To do that, a designer needs to reference a form type that will serve as a LOV form type. He or she should also define how an end user can edit attributes: "Only via LOV" or "Directly & via LOV". Each component type has one or more keys and uniqueness constraints. Both elements comprise one or more component type attributes. Component type keys and unique constraints with non-null values represent the unique identification of a component type instance but only in the scope of its superordinated component instance.

Due to limited space we omit descriptions of many other properties concerning FormTypeUsage concept. Their detailed descriptions may be found in [10].

Through the specification of *FormTypeUsage* concepts we have modeled the same concepts in EMF and MetaEdit+. Those concepts are modeled in the same way and using similar constructs in both environments. Our goal in this subsection was to show that despite all the differences between environments, both of them can be used equally to model the same concepts.

Fig. 5. A Meta-Model of *FormTypeUsage* concept in MetaEdit+

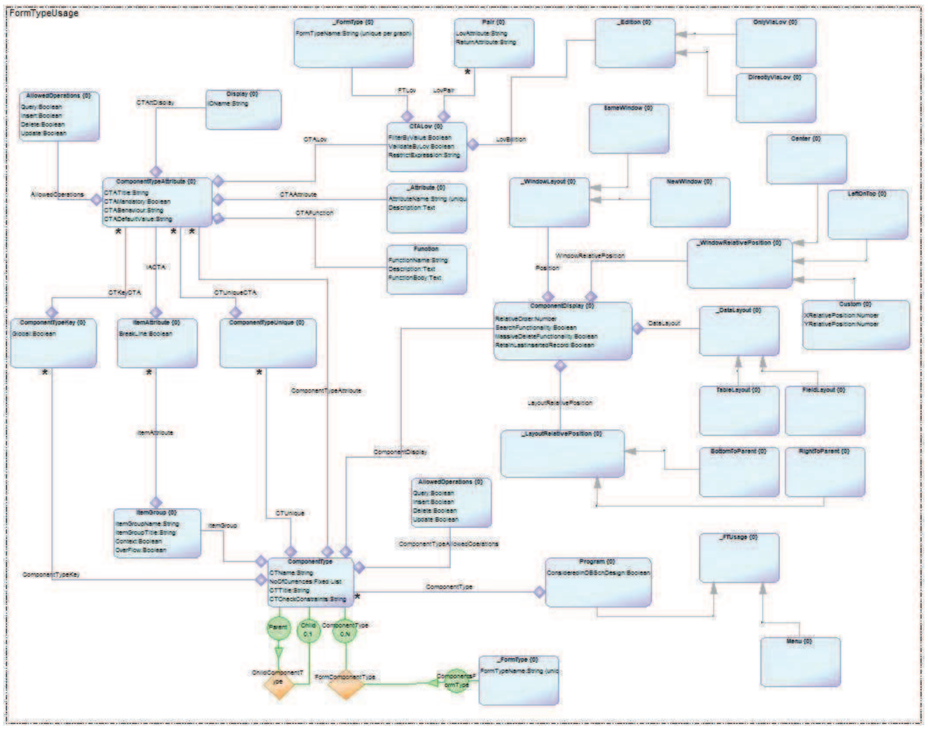
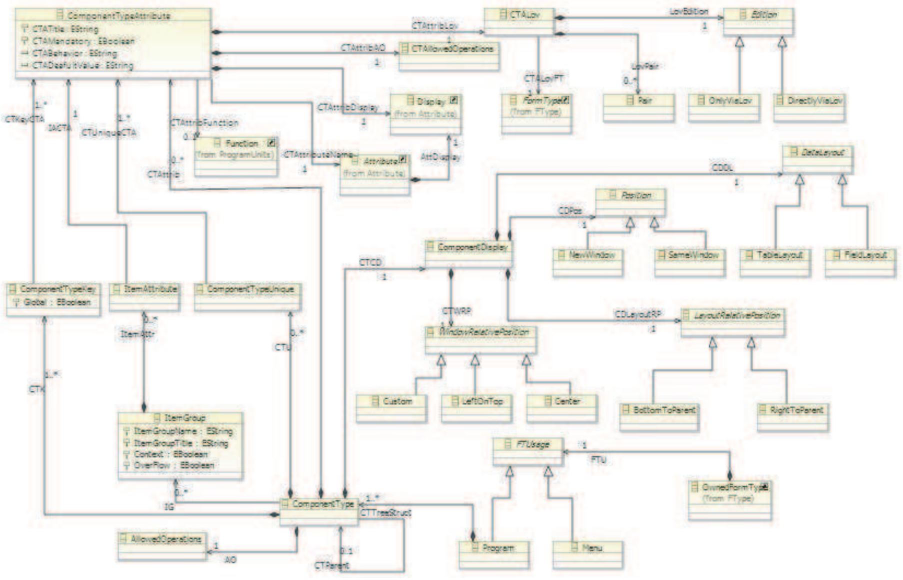


Fig.6. A Meta-Model of *FormTypeUsage* concept in EMF



5 Conclusion

In this paper we presented a comparison between two DSM tools: EMF and MetaEdit+. For this purpose, we explored the MetaEdit+ language definition concepts and Ecore meta-meta-model. The concepts used by MetaEdit+ are described by GOPPRR meta-language that actually represents the meta-meta-model language definition. Ecore is MOF 2.0 implementation used by EMF meta-modeling environment.

Unlike EMF modeling environments that we have worked in, MetaEdit+ allows us to easily generate meta-objects in the repository from our meta-model. Further we can use MetaEdit+'s environment to produce graphical DSL by importing the meta-model. Unlike MetaEdit+'s environment, EMF modeling environment provides only the abstract syntax development. The concrete syntax of some DSLs may be developed under some other Eclipse frameworks, such as Xtext, EMFText or GMF Tooling. These frameworks rely on EMF, and they also use Ecore meta-meta-model. EMF modeling environment is widely used. Using the Ecore meta-meta-model, users have the opportunity to model using the MOF 2.0 concepts as a de facto standard. The meta-model specified under the EMF environment can be further used in the development of some textual or graphical DSLs, using some other Eclipse tools.

Both MetaEdit+'s and EMF's workbenches may be deployed to make an IS model containing this meta-objects. MetaEdit+'s symbol editor also allows instances of meta-objects to have distinctive graphical representation. Therefore designers can easily read and specify models in a graphical way. By specifying IS models, designers have better opportunities for mental testing the ideas and checking validity of their models.

We conclude that MetaEdit+ is an environment well suited with concepts to develop fully functional graphical DSLs. It provides all-in-one environment for defining meta-models as well as the representations of graphical DSLs. On the other hand side, EMF provides developing meta-models only. MetaEdit+ is extensible at the level of graphical meta-language, with new concepts that can make modeling more precise and easier. However, the resulting set of concepts still needs to be mapped onto the actual GOPPRR concepts in MetaEdit+. Unlike the MetaEdit+ tool, EMF does not allow introducing new concepts at the meta-meta-level. EMF is the modeling environment, and majority of its concepts are not used when developing a modeling language. Other tools, like GMF, are also needed when specifying the modeling language. On the other hand all GOPPRR is built for the definition of graphical modeling languages and all of its meta-concepts are well suited for fulfilling its purpose. Ecore concepts are supported by many other tools and environments through standard import and export mechanisms, such as XML Metadata Interchange (XMI). This standardization makes a usage of an environment easier for users already familiar with MOF 2.0 concepts.

Our further research will be directed towards the implementation of mapping between the meta-models using GOPPRR and Ecore specification. One goal is to provide a bridge that will support the transformation from the model specified by one meta-model to the other. One of the goals is to deploy MetaEdit+-EMF-Bridge [12] to import IIS*Case GOPPRR meta-model into EMF and then use a transformation engine like Epsilon or XPand, to define a model-to-model transformation. It should provide the users with the ability to deploy both modeling environments utilizing their advantages. This mapping will also allow exchanging models between coworkers using different environments and thus make them easier work on the same problem with tools they already have.

6 Acknowledgements

The research presented in this paper was supported by Ministry of Education and Science of Republic of Serbia, Grant III-44010.

7 References

1. Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/.
2. Object Management Group (OMG), OCL Specification Version 2.0. http://www.omg.org/docs/ptc/05-06-06.pdf, 2005.
3. Graphical Editing Framework. http://www.eclipse.org/gef/.
4. Graphical Modeling Framework. http://www.eclipse.org/modeling/gmp/.
5. Kelly, S., Lyytinen, K., Rossi, M., MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. CAiSE 1996: pp. 1-21
6. Luković, I., Ivančević, V., Čeliković, M., Aleksić, S.: DSLs in Action with Model Based Approaches to Information System Development. In the book: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 2012. (Accepted for publication, Chapter 17)
7. Meta-Object Facility. http://www.omg.org/mof/
8. Luković, I., Varanda Pereira, M. J., Oliveira, N., Cruz, D., Henriques, P. R.: A DSL for PIM Specifications: Design and Attribute Grammar based Implementation. ComSIS, ISSN: 1820-0214, DOI: 10.2298/CSIS101229018L, Vol. 8, No. 2, 2011, pp. 379-403.
9. Oliveira, N., Varanda Pereira, M. J., Henriques, P. R., Cruz, D., Cramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. ComSIS, ISSN:1820-0214, Vol. 7, No. 2, 2010, pp. 265-289.
10. Čeliković, M., Luković, I., Aleksić, S., Ivančević, V.: A MOF based Meta-Model of IIS*Case PIM Concepts. Proceedings, IEEE Computer Society Press & Polish Information Processing Society, ISBN: 978-83-60810-22-4, Szczecin, Poland, 2011, pp. 833-840.
11. Kelly, S.: Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM, Proceedings of the OOPSLA & GPCE Workshop on Best Practices for Model Driven Software Development at OOPSLA'04, 2004.
12. Kern, H.: The Interchange of (Meta) Models between MetaEdit+ and Eclipse EMF Using M3-Level-Based Bridges, 8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA, 2008.
13. Kern, H., Hummel, A., Hühne, S.: Towards a Comparative Analysis of Meta-Metamodels, 11th Workshop in Domain-Specific Modeling, 2011.
14. Welke, R.J.: CASE Repositories: More than another DBMS Application, Challenges and Strategies for Research in Systems Development, Cotterman, W. and J. Senn (eds.), J. Wiley, Chichester, UK, 1992, pp. 181-214.
15. GOPPRR: MetaEdit+ Workbench User's Guide, Version 4.5, MetaCase, [Online] http://www.metacase.com/support/45/manuals/mwb/Mw-1_1.html

SeMFIS: A Tool for Managing Semantic Conceptual Models

Hans-Georg Fill[1]

University of Vienna
Research Group Knowledge Engineering,
1210 Vienna, Austria,
hans-georg.fill@univie.ac.at,
WWW home page: http://homepage.dke.univie.ac.at/fill

Abstract. Several approaches have been discussed in the past to manage semantic aspects of semi-formal conceptual models based on mappings of their elements to ontologies. In the paper at hand we describe the foundations for these approaches, derive requirements for an according tool support and present the design and implementation of the SeMFIS toolkit together with use cases where it has been successfully applied. In contrast to other approaches, SeMFIS is based on a meta modeling approach that can be easily adapted and extended to support arbitrary conceptual modeling languages. In addition it will be made freely available for the scientific community in the context of the Open Models Initiative.

Keywords: Conceptual modeling, Semantics, Meta modeling, Open Models

1 Introduction

In the last years several approaches have been discussed in the literature that focus on the enrichment of semi-formal conceptual models about information systems with semantic aspects, e.g. [27]. Thereby, the elements of a modeling language or of models whose labels are given in natural language are mapped to a semantic schema. Typically, the semantic schema comes in the form of an ontology, i.e. a computer-usable definition of basic concepts of a domain and the relationships among them [26]. In this way, additional semantic information can be made explicit and processed by machines. In comparison to approaches that are targeted towards an a-priori description of the semantics, these mappings can also be added ex-post, i.e. after the creation of a modeling language or the instantiation of models. This not only leads to enhanced flexibility in terms of processing because the semantic mappings and according processing functionalities are not tightly coupled to a particular model or modeling language. It also enables a stepwise semantic enrichment of models, where the degree of formality of the underlying ontology can be chosen according to a user's needs [13]: for some applications it may be sufficient to use vocabularies or thesauri as an ontology, whereas for other scenarios the use of logic-based languages may be necessary to conduct inferencing [26,32].

Based on these approaches several tools have been developed that support the handling of such aspects. However, in practice most of them have two major shortcomings: Firstly, they are often tied to one particular type of modeling language and/or one particular type of ontology. Although this may be acceptable for realizing a concrete usage scenario, the scientific community would greatly benefit from an approach that can be applied to arbitrary modeling and ontology languages without the need for a complete re-implementation of similar concepts. Secondly, only some of the tools are available on an open source basis or in some other way open to the further development by the scientific community. Therefore, we will describe in the following the necessary foundations and considerations for realizing a flexible, open accessible solution to address these issues. Subsequently, we will present SeMFIS, a tool based implementation that has been realized using concepts from meta modeling and that will be shared using the Open Models Initiative. The remainder of the paper is structured as follows: in section 2 we will clarify some terms and briefly describe the foundations for our approach. Section 3 will discuss the requirements for managing semantic aspects and review existing approaches in this area. Section 4 will present the approach of SeMFIS including its goals, implementation and use cases. The paper will be concluded with an outlook on the next steps in the development in section 5.

2 Foundations

In order to clarify our understanding of the terms modeling method, modeling language, modeling procedure and algorithms in this context, we will revert to a framework proposed by Karagiannis and Kühn [20] - see also figure 1. In their view a modeling method is composed of a modeling technique and mechanisms and algorithms. The modeling technique is further split into a modeling language and a modeling procedure that defines the application of the modeling language by defining steps and delivered results. The modeling language is composed of syntax, semantics, and a notation. Thereby, the notation part is used to explicitly define the visualization of the syntax while obeying the meaning of the syntax elements as defined by the semantics. The semantics itself consists of a semantic mapping and a semantic schema. The mapping connects the elements of the syntax, i.e. the grammar, to the elements of the semantic schema through a reference relationship.

The mechanisms and algorithms are used in the modeling procedure and are applied to the modeling language. They can either be generic, i.e. applicable to arbitrary modeling languages, specific, i.e. applicable only to a particular modeling language or hybrid, i.e. configurable for multiple modeling languages. As our approach builds upon concepts of meta modeling, we will also explain our notion of a meta model. Therefore we revert to the definition given in [28] who consider a meta model to be a model of the abstract syntax of a modeling lan-

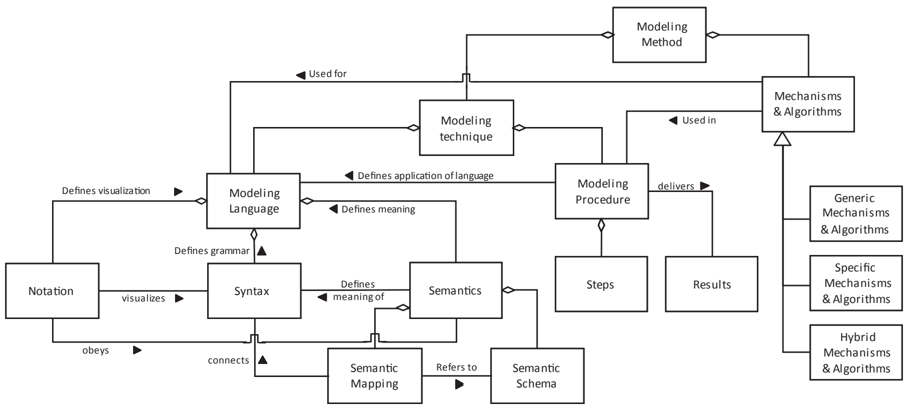


Fig. 1. Components of Modeling Methods [20]

guage. However, as our focus is on *conceptual* models, we also need to take into account the specificities of these types of models here. In contrast to other views on models such as in software engineering or knowledge representation, conceptual models are primarily intended to be used by humans for communication and understanding and not machines [25].

This in turn also affects how semantics is viewed: whereas for models that are intended for machine usage, the addition of some kind of formal semantics is obligatory, conceptual models often revert to natural language descriptions for explaining their use and behavior, cf. [21,25]. The combination of a formal syntax and a natural language description of its use, which is also denoted as a *semiformal* specification, directly affects the application of algorithms [14]: in contrast to formal specifications with a rigourously defined syntax and semantics that offers a theoretical model against which descriptions can be verified, semiformal specifications have only limited checking facilities. Based on the specifications in the meta model, the types of the elements in the model can be identified and accordingly processed based on the instantiation relationship defined by the syntax. However, when it comes to the meaning/behavior of the types and the meaning that is assigned to their instances during modeling, only natural language descriptions in the form of labels that are attached to the elements are available.

To enable the processing of such models, additional semantic specifications are required. These can be added on the level of the meta models and/or the level of the models and may be described using different degrees of formality. A common approach, in particular for conceptual modeling languages in the area of process and workflow modeling, is to map the elements of a meta model to formal semantic schemata. These can be formalisms such as Petri nets [1] or also appropriately represented system runs that are linked to the models via algorithms [15]. Thereby, the behavior of a modeling language can be unambiguously defined and the resulting models can be checked for conformance to these formalisms.

Another direction is to use computer-usable definitions of a domain vocabulary, i.e. ontologies, as a semantic schema [24,2]. This permits e.g. to analyze the structure of a modeling language in terms of semantic phenomena such as synonymity or similarity. It also provides a basis for the application of algorithmic analyses and logic-based inference mechanisms. Furthermore, the use of standardized languages such as OWL or RDF for describing the domain vocabulary allows to exchange the semantic specifications with other tools and services [30,24]. However, when a user instantiates a meta model and adds semantic information in the form of natural language descriptions by using labels for the elements in the models, this information is not known at the design-time of the modeling language as the user does not face any constraints which information to assign. A solution to this is to map the labels to ontologies that contain machine-processable entities of natural language [3,17]. In this way for example semantic similarities between model instances can be determined [7] as well as integration points for services [16] and other modeling languages can be discovered [17].

3 Requirements for Semantic Conceptual Models and Related Approaches

With these foundations we can now derive some basic requirements for tools to handle semantic aspects in conceptual models. Subsequently we will review related approaches in this area and then discuss our approach and implementation of the SeMFIS toolkit that is based upon semantic conceptual models. For the derivation of the requirements we took into account the work by Uren et al. that dealt with similar issues in the context of knowledge management [31].

3.1 Requirements

Regarding *functional requirements*, an according tool ideally has the ability to deal with arbitrary conceptual modeling languages, because we would like to address semantic aspects of conceptual modeling languages and models from a general perspective. It should thus be possible to map elements of a meta model or a model of any type to various types of semantic schemata. In this way, the approach would be highly re-usable for a large range of application scenarios and domains. This also includes that the content of models, the semantic mappings and the semantic schemata should be exchangeable, i.e. that interfaces for accessing their content are available.

To detail requirements concerning *interfaces and the exchange of information*, a tool needs to support widely accepted IT-standards to reduce the effort of learning new methods and simplify the re-use of the contained information. In the area of ontologies based on description logics for example, the web ontology language OWL is one of the most widely used standards. Therefore, any tool dealing with such types of ontologies should be able to support OWL.

From the persepctive of *user interaction requirements*, a tool has to focus on a user-centered design and meet the intended users' abilities and thus ease the handling of semantic aspects. This concerns in particular the effort for dealing with formal issues of the definition of mappings and the use of the underlying semantic schemata. Due to the large effort that may be involved in defining the mappings, a tool should permit the collaboration of multiple users, ideally also in distributed environments. Furthermore, the tool should support the handling of the evolution of the modeling languages, the models, the semantic mappings and the semantic schemata so that the consistency between all these parts can be ensured. As already mentioned in the introduction, the tool should be open for the further development by the scientific community. From an implementation perspective it should also be easily adaptable and extensible so that researchers can implement new functionality and re-use existing ones without much effort.

3.2 Related Approaches

When investigating existing approaches for handling semantic aspects in the ways mentioned above, a large number of tools can be found that have been developed in the context of semantic web. For realizing the vision of semantic web, a core feature is to define mappings between textual resources and machine understandable semantic schemata - for a comprehensive overview of approaches in this field we refer to [31]. Although some of the concepts developed for semantic web can be re-used, these approaches and tools do not focus on the specific properties of conceptual modeling languages or models.

Regarding approaches that do focus on conceptual models, several contributions have been made in the area of semantic business process management. However, only very few publications can be found that deal with these issues from a modeling language independent view, e.g. [5]. Apart from business process modeling also the field of software engineering and service modeling have discussed these aspects [34,18] - but also these approaches are tied to particular modeling languages, e.g. UML class diagrams and SoaML. In semantic business process modeling five tools can be directly related to the above mentioned requirements: the SemPeT tool by the University of Karlsruhe [7], Maestro for BPMN by SAP Research [4], an extension for the ARIS toolkit [29], WSMO Studio [6], and Pro-SEAT [23]. Maestro, WSMO Studio and PRO-SEAT support the BPMN notation for defining process models, SemPeT supports Petri nets and ARIS event driven process chains. SemPet, Pro-SEAT and WSMO Studio support the web ontology language OWL whereas Maestro, ARIS and also WSMO Studio revert to ontologies expressed in the WSML/WSMO format. To the best of our knowledge none of these tools currently explicitly supports the handling of evolutions of semantic aspects. Concerning the licensing strategies only WSMO Studio is explicitly available under an open source license. Each of these tools has been developed for a particular use case: SemPeT has been applied for determining the semantic similarity of process models described by

Petri nets, Maestro for BPMN, the ARIS extension and WSMO studio target the automatic discovery and composition of web services during process execution based on annotations of BPMN process models. PRO-SEAT focuses on enabling the semantic interoperability of process models between different enterprise information systems.

4 The Approach of SeMFIS

In this section we will present the approach of the SeMFIS[1] tool for managing semantic conceptual models. The core parts of SeMFIS have been developed in the course of a research project conducted at Stanford University and are today being further developed by the author at the University of Vienna. The implementation of SeMFIS is provided via the Open Models Initiative [19,22][2]. We will first describe the goals and concepts of the SeMFIS approach and then the concrete implementation and use cases.

4.1 Goals and Concepts

The main goal of the SeMFIS approach is to provide an open platform for describing the semantic aspects of multiple conceptual modeling languages and models. Besides this, SeMFIS also aims at establishing a community for the exchange of know-how on handling these semantic aspects and according implementations. For this purpose it provides a set of *semantic conceptual model types* that are described using meta models, a set of *algorithms and support tools* and a set of *web services*. In the basic configuration these semantic conceptual models comprise a *semantic annotation model type*, an *OWL ontology model type*, a *frames model type*, and a *term model type* - for an excerpt of the meta models see figure 4 in the appendix. The meta models underlying these model types can then be added to other existing meta models as required. The OWL ontology model type is used to represent ontologies based on the OWL specification by W3C in the form of visual models. Similarly, the frames ontology model type represents frames ontologies based on the Protégé frames ontology implementation of the OKBC Knowledge Model as described in [33]. Whereas the support of OWL ontologies originates from the wide spread use of this type of ontologies, frames ontologies were chosen because of advantages in certain scenarios: as they are based on the closed-world assumption where everything is prohibited until it is permitted, they sometimes require less effort for their specification and are easier to handle than OWL ontologies. In addition, for both types of ontologies powerful programming libraries and additional tools such as reasoners and rule engines are available. In addition to these ontology types, the term model type provides a way to represent an extended form of controlled vocabularies. Thereby, terms, their synonyms and a simple generalization/specialization hierarchy of the terms can be defined. Although similar results can be achieved by

[1] SeMFIS stands for Semantic based Modeling Framework for Information Systems

[2] See the project website at `http://www.openmodels.at/web/semfis/`

using one of the two full-fledged ontology types, in industry scenarios where only limited knowledge about ontologies is available, such a 'reduced ontology type' may better meet the users' abilities [13].

For defining the mappings between conceptual modeling languages and models and the different types of ontology models, the semantic annotation model type has been defined based on a previous concept for linking models and ontologies in [10]. It provides constructs for expressing triple statements that contain a reference to a particular meta model or model element, the type of annotation, and a reference to an ontology element. Currently ten types of annotations are pre-defined, however these may extended based on particular needs: 'is equal to', 'is broader than', 'is narrower than', 'is instance of', 'is subclass of', 'is superclass of', 'is instance using fromClass', 'is instance using toClass', 'transfers Value to Slot', and 'is annotated with'. Thereby, the two types referencing 'fromClass' and 'toClass' are used to map from the endpoints of a relation or relationclass.

In addition to the model types, several algorithms were specified to handle the exchange of model information and provide certain processing functionalities required by various use cases as will be described below. These currently include algorithms for: exchanging the models in XML format; exporting frames ontology models in the Protégé frames project format; transferring information from models that are mapped to a frames ontology into instances of that ontology; and obfuscating model information based on mapppings to a subsumption hierarchy expressed in an OWL ontology [12,11]. In order to easily support interaction in distributed web environments, a number of web services were specified. These include functionalities such as the access to the contents of the models and the generation of various graphical formats of the model representation.

4.2 Implementation and Use Cases

The meta models described above were implemented using the ADOxx meta modeling platform[3] that is provided by BOC AG through an open access licence for projects of the Open Models Initiative - see figure 2 for a screenshot of the model editors. From the functionality provided by ADOxx, several components were re-used for the realization of SeMFIS - see figure 3. These encompass not only the modeling component for the automatic generation of model editors from the meta models but also the analysis, simulation and evaluation components as well as the HTML generation and import/export component for exchanging model information. The algorithms for SeMFIS were implemented in the ADOxx scripting language ADOscript. For the implementation of the web services, the ADOxx web service component was used that provides a WSDL interface for the remote execution of ADOscript code. This provided the basis for the implementation of the SeMFIS REST web services. To interact with these services a web based user interface was implemented using the Google web toolkit (GWT) and the LGPL SmartGWT library[4]. Currently this user interface does not provide

[3] For a detailed discussion of the formal aspects of the ADOxx meta modeling approach see [9].

[4] See `http://www.smartclient.com/`

all functionalities of the desktop application, however it is planned to add these in the future.

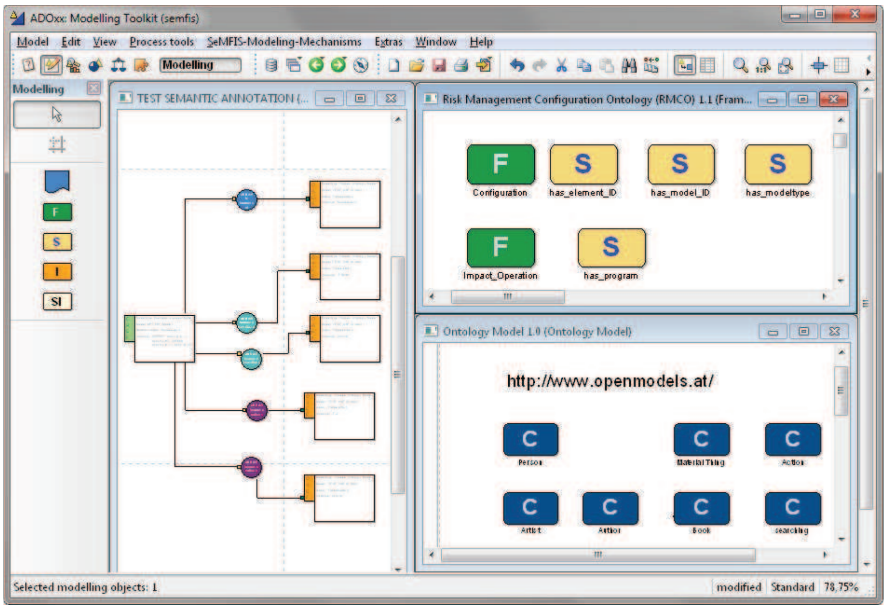

ADOxx Modelling Toolkit (semfis)
Model Edit View Process tools SeMFIS-Modeling-Mechanisms Extras Window Help
Modelling
TEST SEMANTIC ANNOTATION (...
Risk Management Configuration Ontology (RMCO) 1.1 (Fram...
Configuration
has_element_ID
has_model_ID
has_modeltype
Impact_Operation
has_program
Ontology Model 1.0 (Ontology Model)
http://www.openmodels.at/
Selected modelling objects: 1
modified
Standard
78,75%

Fig. 2. Model Editors for Semantic Conceptual Models

To ease the handling of OWL ontologies the Protégé[5] ontology management toolkit was integrated in the architecture through a plug-in. With this plugin parts of OWL ontologies can either be exported in an XML file format and imported in ADOxx or directly submitted to a SeMFIS web service. In this way the vast range of functionalities provided by Protégé and its plug-ins can be re-used for managing ontologies. Although the SeMFIS implementation currently does not provide a specific semantic aspect evolution mechanism, the generic ADOxx functionality for managing model changes and the functionalities of Protégé for handling the evolution of ontologies is available. As all relevant information for expressing semantic aspects is stored in ADOxx, also the generic ADOxx consistency functions, e.g. for ensuring that only existing elements and concepts can be linked, can be re-used.

The SeMFIS tool has already been successfully applied for several use cases in the research on semantic aspects of conceptual models. In [12] the tool has been applied to support tasks in business process benchmarking. Thereby, semantic analyses of business processes could be conducted for the purpose of performance management and confidential information could be obfuscated based on semantic

[5] See `http://protege.stanford.edu`

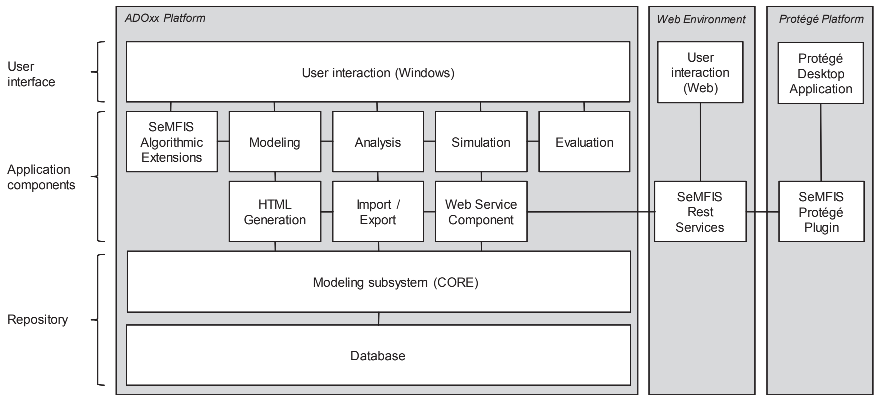


Fig. 3. Architecture of SeMFIS

annotations with concepts from an OWL ontology. For the approach described in [13], annotation and term models were used to provide input for a visualization algorithm that creates user-specific views on process models. Finally, in [8] a mapping between process models and concepts from a frames ontology are used to make risks and their impact on business processes explicit and thus serve as input for simulations.

5 Conclusion and Outlook

In this paper we have described the foundations and conceptions for the development of the SeMFIS tool. Based on the provided meta models, algorithms and services this tool can be linked to arbitrary types of modeling languages for realizing semantic conceptual models. The next steps in the development will be the further development of the web interaction functionalities. These are currently being designed and implemented in several students' projects. Furthermore, also the provision of specific evolution and change handling functionalities will be investigated and integrated in the implementation. In parallel, it is planned to evaluate the practical application of the tool in research and industrial projects.

Acknowledgement

Parts of the work on this paper have been funded by the Austrian Science Fund (FWF) in the course of an Erwin-Schrödinger scholarship grant number J3028-N23.

References

1. Van der Aalst, W.M.P.: Formalization and Verification of Event-driven Process Chains. Information and Software Technology 41(10), 639–650 (1999)

2. Abramowicz, W., Filipowska, A., Kaczmarek, M., Kaczmarek, T.: Semantically enhanced business process modelling notation. In: Hepp, M., Hinkelmann, K., Karagiannis, D., Klein, R., Stojanovic, N. (eds.) Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management. vol. 251. CEUR Workshop Proceedings (2007)
3. Boegl, A., Karlinger, M., M., S., Pomberger, G.: EPCs Annotated with Lexical and Semantic Labels to Bridge the Gap between Human Understandability and Machine Interpretability. In: Smolnik, S., Teuteberg, F., Thomas, O. (eds.) Semantic Technologies for Business and Information Systems Engineering. pp. 214–241. IGI Press (2012)
4. Born, M., Hoffmann, J., Kaczmarek, T., Kowalkiewicz, M., Markovic, I., Scicluna, J., Weber, I., Zhou, X.: Semantic Annotation and Composition of Business Processes with Maestro. In: ESWC 2008. Springer (2008)
5. Diamantini, C., Boudjlida, N.: About semantic enrichment of strategic data models as part of enterprise models
6. Dimitrov, M., Simov, A., Stein, S., Konstantinov, M.: A BPMO based Semantic Business Process Modelling Environment. In: Hepp, M., Hinkelmann, K., Karagiannis, D., Klein, R., Stojanovic, N. (eds.) SBPM 2007. CEUR Workshop Proceedings (2007)
7. Ehrig, M., Koschmider, A., Oberweis, A.: Measuring Similarity between Semantic Business Process Models. In: Roddick, J., Hinze, A. (eds.) APCCM 2007. Australian Computer Science Communications, vol. 67, pp. 71–80. ACM (2007)
8. Fill, H.G.: An Approach for Analyzing the Effects of Risks on Business Processes Using Semantic Annotations. In: accepted for ECIS'2012 (2012)
9. Fill, H.G., Redmond, T., Karagiannis, D.: FDMM: A Formalism for Describing ADOxx Meta Models and Models. In: Maciaszek, L., Cuzzocrea, A., Cordeiro, J. (eds.) to appear in: ICEIS'2012, Wroclaw, Poland (2012)
10. Fill, H.-G.: On the Conceptualization of a Modeling Language for Semantic Model Annotations. In: Salinesi, C., Pastor, O. (eds.) Advanced Information Systems Engineering Workshops, CAiSE 2011. pp. 134–148. Springer (2011)
11. Fill, H.-G.: Using Obfuscating Transformations for Supporting the Sharing and Analysis of Conceptual Models. In: Robra-Bissantz, S., Mattfeld, D. (eds.) MKWI 2012. GITO Verlag (2012)
12. Fill, H.-G.: Using Semantically Annotated Models for Supporting Business Process Benchmarking. In: Grabis, J., Kirikova, M. (eds.) 10th International Conference on Perspectives in Business Informatics Research. pp. 29–43. Springer (2012)
13. Fill, H.-G., Reischl, I.: Stepwise Semantic Enrichment in Health-related Public Management by Using Semantic Information Models. In: Smolnik, S., Teuteberg, F., Thomas, O. (eds.) Semantic Technologies for Business and Information Systems Engineering: Concepts and Applications. vol. 195–212. IGI Press (2012)
14. Fraser, M., Kumar, K., Vaishnavi, V.: Strategies for incorporating formal specifications in software development. Communications of the ACM 37(10), 74–86 (1994)
15. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer October 2004, 64–72 (2004)
16. Hepp, M., Leymann, F., Domingue, J., Wahler, A., Fensel, D.: Semantic business process management: a vision towards using semantic web services for business process management. In: ICEBE 2005. pp. 535–540 (2005)
17. Höfferer, P.: Achieving Business Process Model Interoperability Using Metamodels and Ontologies. In: Oesterle, H., Schelp, J., Winter, R. (eds.) 15th European Conference on Information Systems. pp. 1620–1631. University of St. Gallen (2007)

18. Juicheng, X., Zhaoyang, B., Berre, A., Brovig, O.: Model Driven Interoperability through Semantic Annotations using SoaML and ODM. Information Control Problems in Manufacturing 13(0314) (2009)
19. Karagiannis, D., Grossmann, W., Höfferer, P.: Open Model Initiative - A Feasibility Study (04-04-2010 2008), `http://cms.dke.univie.ac.at/uploads/media/Open_Models_Feasibility_Study_SEPT_2008.pdf`
20. Karagiannis, D., Kühn, H.: Metamodeling platforms. In: Bauknecht, K., Min Tjoa, A., Quirchmayr, G. (eds.) Third International Conference EC-Web 2002 Dexa 2002. p. 182. LNCS2455, Springer, Aix-en-Provence, France (2002)
21. Kaschek, R.: On the evolution of conceptual modeling. In: Dagstuhl Seminar Proceedings. vol. 08181 (2008)
22. Koch, S., Strecker, S., Frank, U.: Conceptual Modelling as a New Entry in the Bazaar: The Open Model Approach. In: Open Source Systems. vol. 203/2006, pp. 9–20. IFIP (2006)
23. Lin, Y.: Semantic Annotation for Process Models: Facilitating Process Knowledge Management via Semantic Interoperability. Ph.D. thesis (2008)
24. Lin, Y., Strasunskas, D.: Semantic annotation of business process templates. In: Smolnik, S., Teuteberg, F., Thomas, O. (eds.) Semantic Technologies for Business and Information Systems Engineering. IGI Press (2012)
25. Mylopoulos, J.: Conceptual Modeling and Telos. In: Loucopoulos, P., Zicari, R. (eds.) Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development. pp. 49–68. Wiley (1992)
26. Obrst, L.: Ontologies for semantically interoperable systems. In: Proceedings of the 12th International Conference on Information and Knowledge Management. ACM Press (2003)
27. Smolnik, S., Teuteberg, F., Thomas, O.: Semantic Technologies for Business and Information Systems Engineering: Concepts and Applications. IGI Press (2012)
28. Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G.: Metamodelling - state of the art and research challenges. In: Giese, H. et al. (ed.) MBEERTS. vol. LNCS 6100, pp. 57–76. Springer (2010)
29. Stein, S., Stamber, C., El Kharbili, M.: ARIS for Semantic Business Process Management. In: Business Process Management Workshops. vol. 17, pp. 498–509. Springer (2009)
30. Thomas, O., Fellmann, M.: Semantic Business Process Management: Ontology-based Process Modeling Using Event-Driven Process Chains. IBIS 2(1), 29–44 (2007)
31. Uren, V., Cimiano, P., Iria, J., Handschuh, S., Vargas-Vera, M., Motta, E., Ciravegna, F.: Semantic annotation for knowledge management: Requirements and a survey of the state of the art. Web Semantics: Science, Services and Agents on the World Wide Web 4, 14–28 (2006)
32. Uschold, M.: Where Are the Semantics in the Semantic Web? AI Magazine 24(3), 25–36 (2003)
33. Wang, H., Noy, F.N., Rector, A., Musen, M.A., Redmond, R., Rubin, D., Tu, S., Tudorache, T., Drummond, N., Horridge, M., Seidenberg, J.: Frames and OWL Side by Side. In: 9th International Protege Conference. Stanford University (2006)
34. Yuxin, W., Hongyu, L.: Adding Semantic Annotation to UML Class Diagram. International Conference on Computer Application and System Modeling (2010)

Appendix A

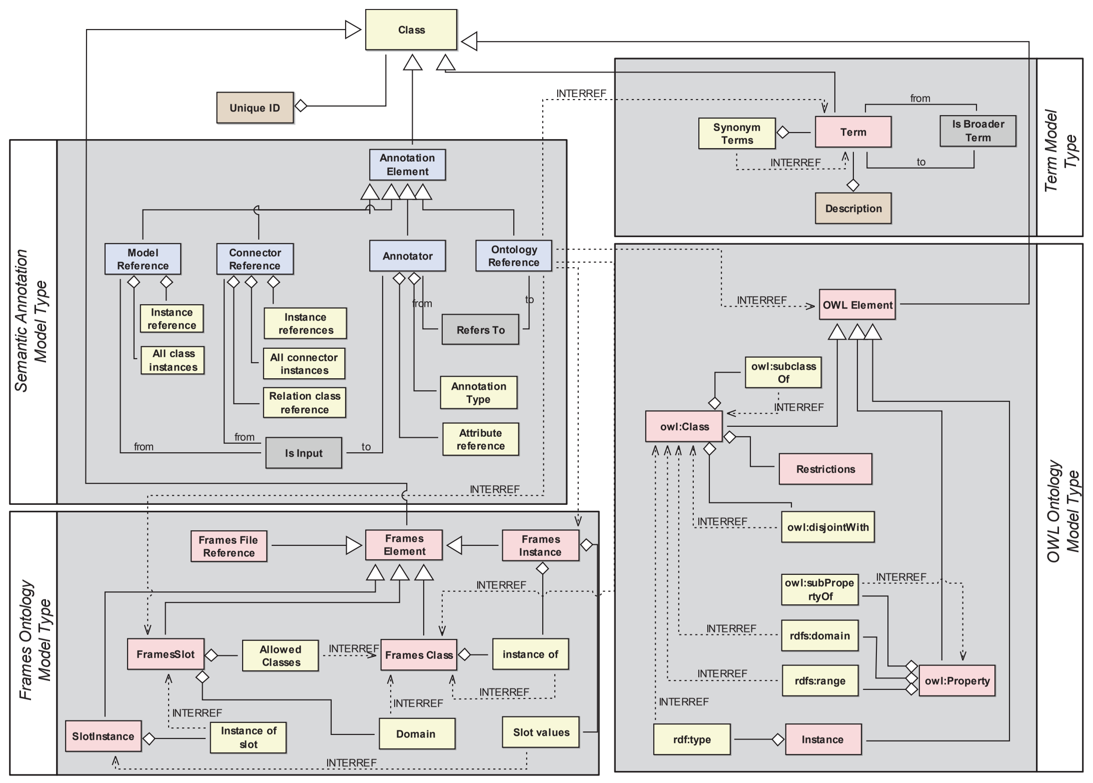


Fig. 4. Excerpt of the SeMFIS Meta Models

Towards collaboration between sighted and visually impaired developers in the context of Model-Driven Engineering

Filipe Del Nero Grillo, Renata Pontin de Mattos Fortes, and Daniel Lucrédio**

Computer Science Department, Institute of Mathematics and Computer Sciences at University of São Paulo, São Carlos, Brazil. Av. Trabalhador São-carlense 400 - Centro, P.O.Box 668. 13560-970 - São Carlos/SP, Brazil
`{grillo,renata}@icmc.usp.br,daniel@dc.ufscar.br`
`http://www.icmc.usp.br`

Abstract. Model-Driven Engineering is rapidly emerging as a powerful way to increase quality and productivity in software development projects. However, its focus on modeling, specially with graphical notations, makes its adoption very difficult to blind and visually impaired users, who have always been able to program with the help of assistive technologies such as screen readers. Without a comprehensive and updated alternative text, this type of software artifact is of little use to a developer with visual impairment. In this paper we present ongoing research and the proposal of a tool to enable the collaboration between sighted and blind/visually impaired software developers. The tool will provide alternative textual representation to models in a web environment, so that collaboration can effectively occur. Details on the technical viability and basic functionality of the tool are presented. We believe these are of great interest to the MDE community, as other researchers and practitioners may build upon our initial ideas to develop their work. We also discuss future investigation possibilities, and the expected contributions of our research.

Keywords: MDE, Graphical, Textual, Accessibility

1 Introduction

Computer programming has historically been a field in which the visually impaired were able to work and teach, because programs are essentially text and, therefore, accessible by the use of assistive technologies such as screen readers. However, the use of visual models became more popular with the growth of the software engineering discipline and visual languages such as the Unified Modeling Language (UML). Visual languages were designed to capture and structure complex problems such as architectural design and requirement specification [1],

** Computing Department at Federal University of São Carlos, Rod. Washington Luís, Km 235 - 13565-905 São Carlos-SP but as they rely on complex visual-dependent software to be developed and read, the blind and visually impaired have many restrictions or do not have access to them at all [9, 13].

As a workaround, blind developers often depend on others to read and explain the concepts to them. This is not a serious problem if we consider traditional software development, where models are used mainly as support and documentation that help programmers to understand what needs to be done in terms of a software solution. The actual software, i.e. the code, is still accessible by the blind and visually impaired. But with the rise of Model-Driven Engineering (MDE), many types of models and their corresponding visual notations gained a new importance in the software project life cycle. On the model-driven approach, models are the main artifact of the development process [7], and are actually used as input to automatic software transformation and code generation. As a consequence, direct access to them is essential, which poses a serious issue to people with visual disabilities.

In theory, every visual model can be translated into a corresponding textual model. For instance, most UML tools are capable of importing/exporting models to the XMI format (XML Metadata Interchange) [15], which is a textual representation. Of course, XMI has many readability issues, but we believe a similar translation process can be applied to produce a much more readable representation, thus making visual models accessible through screen readers. In a model-driven scenario, blind and visually impaired developers could use this alternative representation to read and modify models directly, thus actively collaborating in the development process.

This paper presents an ongoing work towards a multiple representation of models using graphical and textual notations and proposes the Accessible Web Modeler (AWMo), a web-based tool that aims to leverage collaboration with visually impaired users by using multiple presentations for models. Figure 1 demonstrates how the collaboration will happen: with the use of a screen reader, a blind or visually impaired software developer will be able to interact with the textual model while other developers can interact with a traditional graphical model they are used to, when working on the same model. Changes made to one of the views will be shown on the other view and vice-versa.

The remainder of this paper is organized as follows: Section 2 introduces Model-Driven Engineering concepts. Section 3 shows some of the related work found in the literature. Section 4 presents more details about the ongoing research and proposal of the AWMo tool. In Section 5 we present a discussion on the future possibilities of how valuable research data can be extracted from evaluations involving AWMo. Finally, in Section 6 we conclude the paper with some final remarks.

2 Model-Driven Engineering

Model-Driven Engineering (MDE) is the combination of generative programming, domain-specific languages and software transformations, concepts that

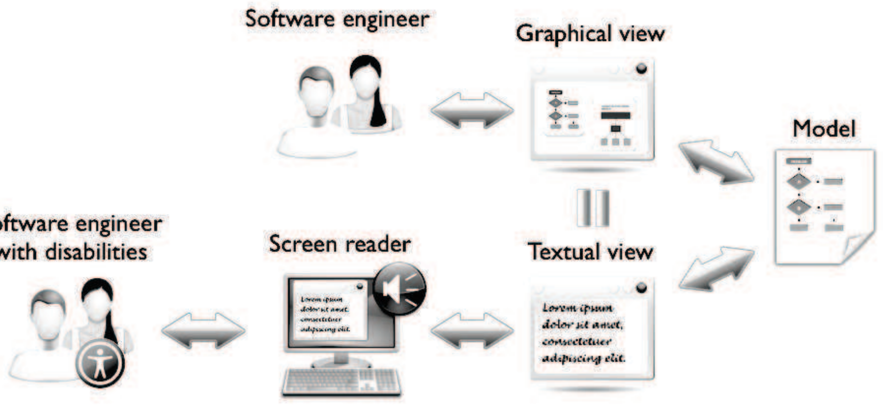

Software engineer
Graphical view
Model
Software engineer with disabilities
Screen reader
Textual view
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Fig. 1. Schematic of how the AWMo tool will work. Developers should be able to edit the model on both graphical and textual views and achieve the same results.

have been explored since 1980 [14, 11]. Its purpose is to reduce the semantic gap between the problem and the solution/implementation, through high level models that shield developers from the complexities of the underlying platform [7]. In MDE, models are used to express domain concepts more effectively, while transformations automatically generate the artifacts that reflect the solutions contained in the models [19].

As a result, the task of the developer becomes simpler and less repetitive. With the use of automations and the higher level of abstraction, the developer can work on a much more conceptual level, leaving implementation details to the responsibility of the code generators.

A particularly effective way of specifying models in the context of MDE is Domain-Specific Modeling (DSM). In contrast with generic modeling approaches like those using UML, DSM uses a Domain-Specific Language (DSL), a smaller, more focused language, designed specifically to provide maximum expressiveness in a particular area or domain [3]. DSLs are normally used in MDE due to some problems with general-purpose languages: UML, for instance, requires that special markings are inserted in order to facilitate transformations, which ends up polluting the models [10]. Even extension mechanisms, which are used in practice to solve part of the problems, are insufficient when MDE is being adopted [10], given its low expressiveness, flexibility and inadequate notation [22].

In MDE, models with different levels of abstraction may be used, from models that have no relation with any computation platform to models that are highly specialized to a specific platform such as Web or mobile development.

Transformations are procedures used to transform a model into another artifact during the software development life cycle. With transformations it is possible to transform a model into a different model, documentation artifacts or even executable code.

3 Related work

In [8], Guerra et al. explore multiple graphical views for the same model, maintaining the consistency between them by the use of a global model and triple grammars. In a later evolution of their work, the authors explore the use of multiple notation on the metamodeling tool AToM^3 with the goal of allowing users to use either visual and textual notations on what they called *Multi-View* DSLs (Domain-Specific Languages). The main goal of their project was to allow the user to model on the notation that is better suited for different perspectives or viewpoints [16].

The TeDUB project (Technical Diagram Understanding for the Blind) tried to address some of the issues of access of blind users to UML by creating a UML reader for the blind, where the users were able to open common UML diagram files like XMI (XML Metadata Interchange), navigate and access its contents with the use of a joystick, sound cues and text to speech [9].

The work of Metatla et al. [12] and Bryan-Kinns et al. [1] explore the use of cross-modality to make diagrams more accessible to workers with visual disabilities. In this context, cross-modality means using more than one sensorial channel to convey or acquire information. A workshop was organized to help the researchers better understand the issues they were dealing with and the needs of their target users. As a result of the workshop they identified that the two limitations that all current aproaches share are the inability to create and edit the diagrams without the assistance of a sighted person and the inefficiency on use of collaborative interaction.

Our approach is closer to the work of Guerra et al. [8, 16], however we intend that the two views represent the whole model, instead of submodels or projections of the model, creating different perspectives. This is required mainly because the different views in our approach are not meant to be used together, in fact, they should be able to completely replace each other in terms of model understanding. We believe a consistent mapping between visual and textual notations can bring better results, given the familiarity that the blind and visually impaired users have with text and screen readers.

The literature also has some reports about Web modeling environments and applications. One of them is SLiM (Synchronous Lightweight Modeling). SLiM is an environment that allows users to collaborate on modeling activities in a synchronized way. It uses techniques such as COMET [2] for server communication and Scalable Vector Graphics [6] for the diagram visual representation over the Web. The main goal of SLiM is to allow the collaboration of users that are geographically apart [20].

Another example is GEMSjax [5], a Web implementation of GEMS (Generic Eclipse Modeling System [23]). GEMS is a project that was created by the Eclipse Foundation to bring together the experience about visual metamodeling tools of the GME community at Vanderbilt university and Eclipse communities such as EMF and GMF. The GEMSjax uses the Google Web Toolkit framework to create a Web interface for modeling and metamodeling activities. There are some other examples that are not Web applications, but Desktop, such as COMA

(COllaborative Modeling Architecture tool) that focus on collaboration [18] and the Eclipse Foundation GEMS that inspired and was used by the GEMSjax mentioned earlier.

Our approach is also web-based, and aimed at leveraging collaboration, and thus we intend to employ well-known techniques for this kind of tool. However, our focus is on the inclusion of blind and visually impaired users, and therefore the support for synchronization and real-time collaboration will be limited.

4 Proposed tool and methods

In this section we discuss the building blocks of our approach. We discuss the technical viability of AWMo and then present more details on how the tool should work.

4.1 Viability

The web environment of AWMo will be based on JSF (Java Server Faces), which will be used to construct the menus and basic interaction functionality. Accessibility guidelines [21] will be adopted during the construction of this base environment, helping to make the interface accessible.

For the textual representation, we intend to use Xtext [4]. Xtext provides a set of tools that allows the definition of a grammar in EBNF (Extended Backus–Naur Form) notation and generation of a textual editor resources such as a language parser, validators and code generators. Xtext can also generate an Ecore metamodel and EMF (Eclipse Modeling Framework) classes from the language grammar. This makes the programmatic management of the parsed (textual) models and metamodels possible.

One of the advantages of Xtext is that the generated tools are independent from the Eclipse environment and can be used, for instance, in a JSF Web application. This indicates that the integration of the Xtext tools with JSF for the development of the AWMo tool is possible and will happen naturally.

For the visual modeling functionality, such as diagram creation and disposition of the visual elements, we intend to use some pre-existing web-based library for graph rendering and construction. Some examples include Raphaël JS[1], jsUML2[2] and Joint JS[3]. From these, we intend to adopt the jsUML2, which is a JavaScript library that already implements many aspects of UML modeling in a web application.

With Xtext, EMF and a visual library, much of the desired functionality is already available: textual language definition, parsing, model manipulation and the creation of visual diagrams. What is left is the mapping between the parsed (textual) model and the appearance and disposition of the visual elements, which will have to be built by hand. However, the generated EMF classes will greatly facilitate this task.

[1] http://raphaeljs.com/
[2] http://code.google.com/p/jsuml2/
[3] http://www.jointjs.com/

4.2 Our approach

Initially, we intend to build a simplified UML class model, mainly because of jsUML2. However, other than that, we see no reason why the idea could not be later extended to other types of models.

Once the textual grammar is defined using Xtext, and the respective textual language infrastructure is generated, they will be integrated into the AWMo JSF Web application. The language parser will be used to parse the defined textual class model and create an EMF-based representation of the model. This representation will be used, along with external positioning data, to transform the textual model into a notation that is accepted by the jsUML2 JavaScript library on the application view.

The following code shows an example of a typical textual class model in AWMo:

Example of the AWMo class diagram textual model

```
class Person {
    attribute firstName, type string, visibility public
    attribute lastName, type string, visibility public

    method getName, return null
}

class Student {
    inherit Person

    attribute grade, type float, visibility private

    method setGrade, return void, parameters  {
        parameter grade, type float
    }

    method getGrade, return float, no parameters
}
```

One of the issues faced by having both graphical and textual views for the same model is that some of the information about the model will be only present on the graphical view. As an example we can cite the spatial information regarding a class such as its X and Y coordinates on the diagram. We intend to store any spatial information apart from the model itself, so it can be used when displaying the graphical editor but will not be represented or even parsed during the use of the textual editor. If spatial information is not available, for example when a developer creates a new element in the textual view, default coordinates will be assigned to it on the graphical representation. This position may not be adequate, however it could be later changed by another user, without any effect on the textual view.

Figure 2 shows the basic AWMo textual and visual modeling scheme. The AWMo model is made of the combination between the textual model, which contains all the semantic and information about the model that conforms to the Ecore metamodel generated by Xtext, and the spatial information which contains all the data required to display the textual model in a graphical way. Every time the model is saved from one of the views, the entire AWMo model will have to be checked in order to maintain the relation between the textual model and the spatial information. For example, if a class is removed from the textual view, its spatial information must also be removed. In the case a new class is added, its the spatial information must be created with default values, as mentioned earlier.

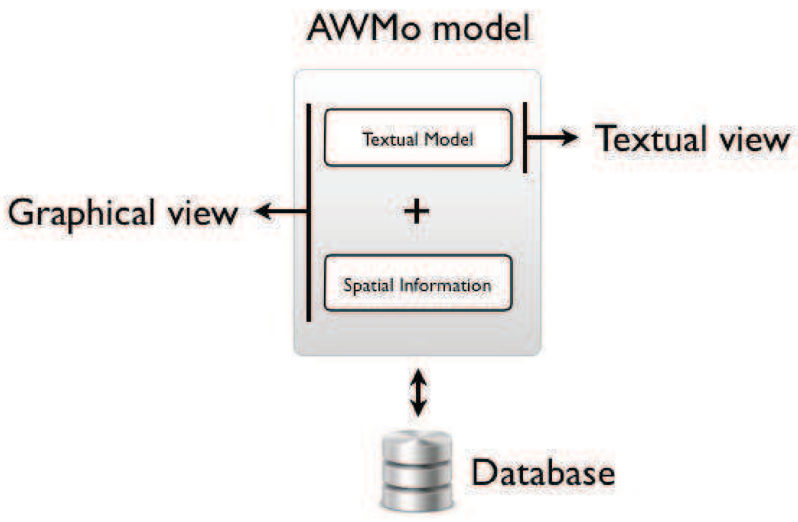
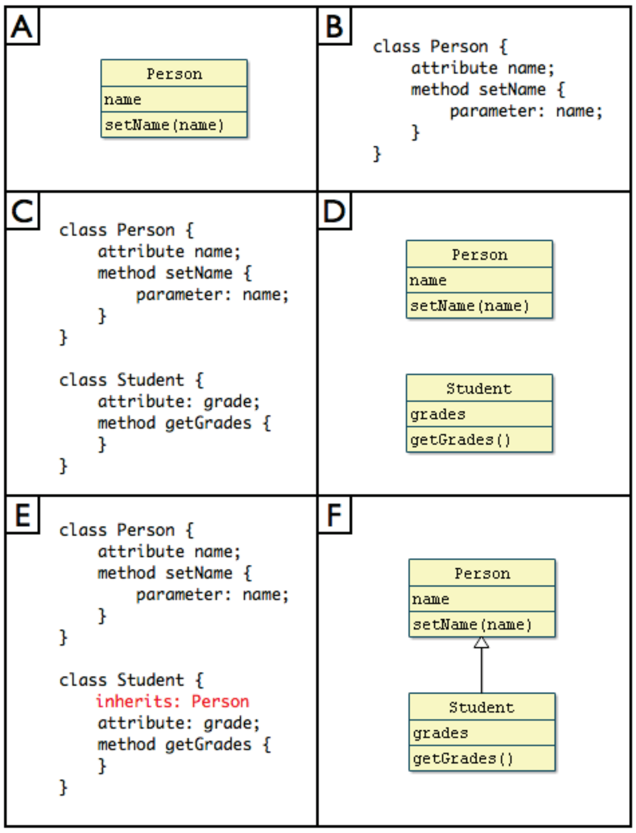


Fig. 2. Representation of the data structure store on database for the AWMo models including both textual and graphical information that are stored apart.

Figure 3 shows an example of use of the proposed tool. At first, on (A), the graphical view is shown and the user added a Person class along with its methods and attributes in the same way he would have done on his preferred UML tool. When the textual view is accessed, the user should see a textual representation of the Person class that is semantically equivalent to the graphical model (B). Then on the same textual view, the user adds a second class, Student (C). Once again, when accessing the graphical view, both Person and Student classes will be visible, reflecting the changes performed by the user on the textual view (D). At last, the developer adds as specialization between the Student and the Person classes (E); This last change is also reflected on the graphical view (F). This simple use case illustrates the bidirectional nature of the approach and gives an idea on how the collaboration between users of the different views will happen.



```
class Person {
    attribute name;
    method setName {
        parameter: name;
    }
}
```

```
class Person {
    attribute name;
    method setName {
        parameter: name;
    }
}

class Student {
    attribute: grade;
    method getGrades {
    }
}
```

```
class Person {
    attribute name;
    method setName {
        parameter: name;
    }
}

class Student {
    inherits: Person
    attribute: grade;
    method getGrades {
    }
}
```

Fig. 3. Illustration of a usage scenario for the AWMo where both graphical and textual views are used to build a simple class model

5 Discussion

With the AWMo tool, there are many evaluation areas to explore. One of the possibilities is to evaluate how it leverages the collaboration between blind or visually impaired and sighted software engineers when modeling with the proposed tool. We expect to completely eliminate the need for an auxiliary person to read diagrams, thus making this collaboration possible in a MDE scenario.

Another common problem we faced while contacting computer programmers with disabilities is that some of them never came to learn UML class diagram because of their special needs and the lack of accessible tools. This makes the process of gathering initial requirements for AWMo a challenge. For that reason we plan on developing AWMo on an interactive way, building a first version that will be incrementally evolved in constant contact with real users, in order to identify their needs with the working tools and then iterate on improvements and validations.

Another – more delicate – issue is the so-called "secondary notation". Usually on a model, there are many ways to convey the same information, however the clarity and readability of the information relies on something that is not always on the language syntax. This is called 'secondary notation', and is defined as additional visual cues that are not part of the language itself but greatly affects the way the information on the model is perceived [17]. An example of such cues is when elements of a graphic that are closely related are represented near each other in a diagram. This information is not part of the language itself and an alternative graphic with the two elements far from each other does not make the graphic wrong according to the visual language syntax, but strongly affects the way the graphic will be read. Another example is inheritance: classes on the top of a diagram are usually higher on the class hierarchy. The same happens with textual models. In most languages, indentation is not part of the syntax, and yet they play an important part on readability. The exact part played by this notation and its importance to blind and visually impaired developers will have to be investigated.

Another possibility of investigation is how these visual cues pointed by Petre (1995) [17] can influence the use of the graphical representation and if, in such cases, the textual representation presents any advantages to help the users understand the models. This comparison transcends the accessibility scenario, and may be useful even without considering the blind and visually impaired developers. We believe there are many interesting issues that could be identified by means of usability and HCI evaluation techniques.

6 Concluding remarks

Visual modeling has been a major problem for blind and visually impaired developers. Without the natural capacity of understanding visual information, these types of model are of little use to these people, who depend on other types of mental abilities to develop software. Most are able to program through the use of screen readers, but in the new MDE scenario the increased importance of models makes this task nearly impossible.

In this paper we present an ongoing work and a proposal for a tool that will allow the collaboration between sighted and blind/visually impaired users in a model-based software development project. We discuss its technical viability and our ideas of how such functionality will be implemented in a web tool. We believe this information can generate healthful discussion and provide some insight for other researchers and practitioners interested in web-based modeling and accessibility.

We also discuss the future possibilities for investigation, highlighting different types of data that could be gathered once the tool is ready. We expect to be able to identify interesting issues and draw important conclusions in the MDE and software engineering areas, helping to advance our knowledge and deliver such inclusion benefits for our society.

References

1. Bryan-Kinns, N., Metatla, O., Stockman, T.: "collaborative cross-modal interfaces". In: Proceedings of Digital Futures '10 RCUK Digital Economy All Hands Meeting (2010)
2. Crane, D., McCarthy, P.: Comet and Reverse Ajax: The Next-Generation Ajax 2.0. Apress, Berkely, CA, USA (2008)
3. Deursen, A.v., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices - ACM Press 35(6), 26–36 (2000)
4. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. pp. 307–309. SPLASH '10, ACM, New York, NY, USA (2010)
5. Farwick, M., Agreiter, B., White, J., Forster, S., Lanzanasto, N., Breu, R.: A web-based collaborative metamodeling environment with secure remote model access. In: Proceedings of the 10th international conference on Web engineering. pp. 278–291. ICWE'10, Springer-Verlag, Berlin, Heidelberg (2010)
6. Ferraiolo, J., Jackson, D.: Scalable vector graphics (SVG) 1.1 specification. W3C recommendation, W3C (Jan 2003), http://www.w3.org/TR/2003/REC-SVG11-20030114/
7. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 29th International Conference on Software Engineering 2007 - Future of Software Engineering. pp. 37–54. IEEE Computer Society, Minneapolis, MN, USA (2007)
8. Guerra, E., Diaz, P., de Lara, J.: A formal approach to the generation of visual language environments supporting multiple views. In: Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on. pp. 284 – 286 (sept 2005)
9. King, A., Blenkhorn, P., Crombie, D., Dijkstra, S., Evans, G., Wood, J.: Presenting UML Software Engineering Diagrams to Blind People. In: Miesenberger, K., Klaus, J., Zagler, W., Burger, D. (eds.) Computers Helping People with Special Needs, Lecture Notes in Computer Science, vol. 3118, pp. 626–626. Springer Berlin / Heidelberg (2004)
10. Kühne, T.: Making modeling languages fit for model-driven development. In: Fourth International Workshop on Software Language Engineering, Nashville, USA (2007)
11. Lucrédio, D., Fortes, R.P.d.M., Almeida, E.S.d., Meira, S.R.d.L.: The Draco approach revisited: Model-driven software reuse. In: VI WDBC - Workshop de Desenvolvimento Baseado em Componentes. pp. 72–79. Recife - PE - Brazil (2006)
12. Metatla, O., BryanKinns, N., Stockman, T., Martin, F.: Designing for collaborative cross-modal interaction. In: Proceedings of Digital Engagement '11: The 2nd Meeting of the RCUK Digital Economy Community (2011)
13. Metatla, O., Bryan-Kinns, N., Stockman, T.: Comparing interaction strategies for constructing diagrams in an audio-only interface. In: Proceedings of the 22nd

British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction - Volume 2. pp. 65–69. BCS-HCI '08, British Computer Society, Swinton, UK, UK (2008)

14. Neighbors, J.M.: Software Construction Using Components. Ph.d. thesis, University of California at Irvine (1980)
15. OMG: MOF 2 XMI Mapping Specification. Disponível em http://www.omg.org/spec/XMI/2.4.1/. Acesso em 15/01/2012 (08 2011)
16. Pérez Andrés, F., de Lara, J., Guerra, E.: Domain specific languages with graphical and textual views. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Applications of Graph Transformations with Industrial Relevance, Lecture Notes in Computer Science, vol. 5088, pp. 82–97. Springer Berlin / Heidelberg (2008)
17. Petre, M.: Why looking isn't always seeing: readership skills and graphical programming. Commun. ACM 38, 33–44 (June 1995)
18. Rittgen, P.: COMA: A tool for collaborative modeling. In: CAiSE Forum - CEUR-WS.org. pp. 61 – 64 (2008)
19. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
20. Thum, C., Schwind, M., Schader, M.: SLIM - A Lightweight Environment for Synchronous Collaborative Modeling. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. pp. 137–151. MODELS '09, Springer-Verlag, Berlin, Heidelberg (2009)
21. W3C: Web Content Accessibility Guidelines (WCAG) 2.0 - W3C Recommendation. Disponível em http://www.w3.org/TR/WCAG20/. Acesso em 23/01/2012 (12 2008)
22. Weisemöller, I., Schürr, A.: A comparison of standard compliant ways to define domain specific languages. In: Fourth International Workshop on Software Language Engineering, Nashville, USA. megaplanet.org, Grenoble, France (October 2007)
23. White, J., Schmidt, D.C., Mulligan, S.: The Generic Eclipse Modeling System. In: Model-Driven Development Tool Implementer's Forum at 45th International Conference on Objects, Models, Components and Patterns (2007)

Second Workshop on Process-based approaches for Model-Driven Engineering (PMDE)

July 3, 2012

Lyngby, Denmark

In conjunction with

ECMFA 2012 conference,

July 4 & 5, 2012, Lyngby, Denmark

Foreword

Welcome to the second edition of the Process-centered approaches for Model-Driven Engineering (PMDE), held in Lyngby, Denmark, on July 3rd , 2012.

The PMDE Workshop aims to gather researchers and industrial practitioners working in the field of Model- Based Engineering, and more particularly on the use of processes to improve software reliability and productivity. Indeed, despite the benefits brought by the Model-Driven Engineering approach, the complexity of today's applications is still hard to master. Building complex and trustworthy software systems in the shortest time-to- market remains the challenging objective that competitive companies are facing constantly. A more challenging objective for these companies is to be able to formalize their development processes in order to analyze them, to simulate and execute them, and to reason about their possible improvement.

The PMDE workshop’s goal in this edition is to present research results or work-in-progress in all areas of process-based approaches for model-driven engineering. The main topics that where targeted this year by accepted papers range from collaborative processes, formal semantics in processes, processes integration to the presentation of new methodologies and visions in the area of process modeling and enactment.

The organization of this second edition of the PMDE workshop would not have been possible without the dedication and professional work of many colleagues. We wish to express our gratitude to all contributors who submitted papers. Their work formed the basis for the success of this year’s edition and we hope, for the upcoming editions of PMDE. We would also like to thank the Program Committee members and reviewers for volunteering their time to help assess the submissions and guarantee the quality of the workshop.
Finally, we are also grateful to the ECMFA 2012 organizing Committee, particularly to Harald Störrle and Ekkart Kindler for their help and valuable advices.

July, 2012, The Organizing Committee.

Reda Bendraou
Redouane Lbath
Marie-Pierre Gervais
Bernard Coulette

Organization

Program committee

- Behzad Bordbar (University of Birmingham, UK)
- Combemale Benoît (IRISA, Rennes, France)
- Garousi Vahid (University of Calgary Alberta, Canada)
- Larrucea Uriarte Xabier (TECNALIA - ICT/European Software Institute, Spain)
- Jason Xabier Mansell (TECNALIA - ICT/European Software Institute, Spain)
- Leon J. Osterweil (University of Massachusetts, USA)
- Tran Hanh Nhi (ENSTA, Brest, France)

Organizers

Bendraou Reda (LIP6, France)

Lbath Redouane (IRIT, France)

Coulette Bernard (IRIT, France)

Gervais Marie-Pierre (LIP6, France)

Specifying the Interaction Control Behavior of a Process Model using Hierarchical Petri Net

Fahad R. Golra and Fabien Dagnat

IRISA / Université Européenne de Bretagne
Institut Mines-Télécom / Télécom Bretagne
Brest, France
{fahad.golra,fabien.dagnat}@telecom-bretagne.eu

Abstract. Management of software development processes is indispensable for systematic development of all the artifacts required or provided by the process. The nature of a software development process can vary from manual to automatic, static to dynamic, concrete to abstract and simple to complex. Thus, a process modeling approach should be able to deal with all these variations. CAMA Process Modeling Framework (CPMF) is a process modeling approach that caters the varying requirements of software development processes. It is inspired from the component based paradigm, where each activity is taken as a component, and the process itself is visualized as an architecture. It models the processes in different abstractions in terms of development phases and multiple implementations. Hierarchical Petri Net is mathematical modeling language having proper definitions of its execution semantics and process analysis. This article concerns the mapping of CPMF metamodel constructs onto a Hierarchical Petri Net, so as to formally define the semantics of the interaction control.

Keywords: Process Modeling, Activities, Petri Nets, Connectors

1 Introduction

All the disciplines of engineering tend to achieve their targets in a systematic manner. Likewise, one of the important considerations for software engineering domain is to develop artifacts in the same fashion. An important management goal is to make the development process cost effective and time efficient, without any compromise on the quality of software. This promotes the use of specific process management methodologies that can help to design, model, execute, monitor and optimize the processes involved in software development [16]. The number of process modeling approaches devised in the recent decades reinforce the importance of process modeling. These approaches offer many different features like support for execution and semantics [3, 12, 4]. The influence of business process modeling over domain specific modeling methodologies has lead to a plethora of languages that are based on the workflows or sequencing approaches [15, 2, 5]. Many of the these process modeling languages still seem to lack the support to effectively model the dynamic processes [14]. These dynamic processes may either be automated, having the authority of creating /destroying other processes during execution or updating themselves on the fly. We believe that in specifying a process as a collection of activities, the workflow should not be the principle focus. Workflows and other sequential approaches are well adapted to model and compose existing web services [2], but are insufficient for modeling a much wider range of development processes. We argue that even though effective dataflow and controlflow is indispensable for the process, still the principle focus should remain on the semantics of the activities.

CAMA Process Modeling Framework (CPMF) targets to model processes at different phases on development. Besides this, it also targets to model the activities using multiple implementations for a single definition. It is presented using three metamodels: *Process Specification Metamodel*, *Process Implementation Metamodel* and the *Process Instantiation Metamodel*. The structure of the first two metamodels has been finalized [9, 10], however the *Process Instantiation Metamodel* is still under development.

The implementation specific metamodel has all the implementation details, which are not present in the specification model and at the same time it has all the principal constructs of the process model, apart from instance specific details. PN is a well known formalism for the qualitative analysis of a variety of systems. Various attempts have been made to define the formal semantics of different process modeling languages through Petri Nets. These endeavors have translated BPMN [5], BPEL [6] and EPC [17] to Petri Nets. A complete survey of the business process model mappings towards Petri Nets can be consulted here [13]. This article focuses on the mapping of the implementation specific metamodel to Hierarchical Petri Nets (HPN), because of its ability to formalize abstractions. Our mapping of CPMF process to HPN describes the interaction control behavior at business level logic of component based processes, and is not targeted towards a complete formal specification. We have tried to exploit the refinement mapping presented in the Hierarchical Petri Nets to formalize the mapping between the abstract and concrete level constructs of CPMF. The focus of this paper is to specify the behavior of connectors to control the interactions between the activities at both abstraction levels and to specify the mappings between these levels. The objective of this mapping is to verify the correctness of the process modeling approach in CPMF in future. This would help in the correct implementation of the tool as well.

2 CAMA Process Modeling Framework

The hallmark of MDE is the usage of multiple models along with defined transformations amongst them. Models are created, modified, merged or split, as the software development project advances. We argue that a unique process model cannot capture all the semantics of the processes at different development stages. For this reason, CPMF presents three metamodels: the *Process Specification Metamodel*, the *Process Implementation Metamodel* and the *Process Instantia-*

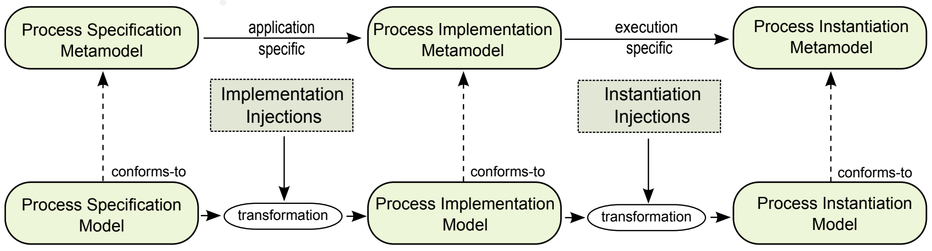

Process Specification Metamodel
application specific
Process Implementation Metamodel
execution specific
Process Instantiation Metamodel
Implementation Injections
Instantiation Injections
conforms-to
conforms-to
conforms-to
Process Specification Model
transformation
Process Implementation Model
transformation
Process Instantiation Model

Fig. 1. Process metamodels for Multi-metamodel development

tion Metamodel, as illustrated in Figure 1. The *Process Specification Metamodel* is used to document the process best practices. It is not specific to the organization or project. When this *Process Specification Metamodel* is applied to a specific project by some organization, it is refined to guide the development process. This refinement is carried out using a transformation that injects the implementation details and provides the *Process Implementation Metamodel*. Another transformation is used in the framework to transform the implementation specific metamodel to instantiation specific metamodel by injecting the instantiation level details to it. The *Process Instantiation Metamodel* is responsible for the execution of the processes for a project, thus it takes into account various details like time plan and status of the project. For example, if we take the ECSS software development standard[7], we can model it as a *Process Specification Model*. A specific implementation model for an application conforming to this standard, can be modeled as *Process Implementation Model*. And finally, for the execution of this application model, we have a *Process Instantiation Model*, which provides support for project management. Model transformations are used between the models conforming to their respective metamodels.

For reasons of brevity we are not discussing any of these metamodels, and the readers are referred to the corresponding articles for the in-depth study [9, 10]. We are looking forward to provide the tool support for modeling the processes using CAMA process framework. In this regard, we are working to extend an existing open source process modeling tool, Openflexo [1]. Openflexo uses BPMN as the underlying process modeling approach. Its execution semantics have been defined though a mapping towards Petri Nets. We want it to offer CPMF as an alternate core process model. In order to validate our process modeling approach, we need to verify the executability of the concrete level process model and the correctness of its mapping towards the abstract level process model. The mapping of the CPMF concrete level process model towards Hierarchical Petri Nets, would allow us in future to verify its executability.

3 Implementation specific process modeling

Process Implementation Metamodel presents the activities in two levels of abstractions: activity types and activity implementations. Activity types at the

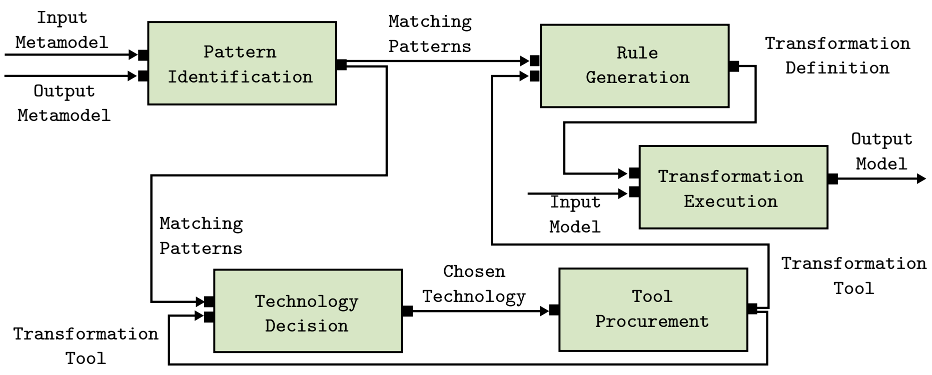


Fig. 2. Abstract level model for transformation process example

abstract level have abstract contracts that deal with dataflow. The dataflow between the activities of a process is modeled through the abstract level process model that deals with the activity definitions and their abstract contracts using artifacts. The notion of connector is hidden within the semantics of the activity definitions. This dataflow is managed through the use of resource pools and appropriate mechanisms of version control. These abstract constructs are shown in the example depicted in figure 2 for a semi-automatic process with five activities supporting model transformation. An activity type can be implemented using multiple activity implementations. Each of these activity implementations at the concrete level have concrete contracts that deal with controlflow. This controlflow is managed using the underlying event management system. Though CPMF does not offer a graphical notation yet. Figures 2 and 3 presents an example with both abstraction levels, for illustration purpose only. The *pattern identification* activity identifies the matching patterns in the input and the output metamodels. It then passes on the matching pattern list to *technology decision* for choosing the implementation technology and to *rule generation* for generating the transformation rules. The *rule generation* activity waits for the confirmation from *tool procurement*. Finally, the developed transformation definition is passed on to *transformation execution* that generates the output model, using the input model.

Each activity definition at the abstract level is implemented by several activity implementations at the concrete level. All these activity implementations map to their activity definitions. Activity definitions at the abstract level are bound together through *artifacts*, whereas the activity implementations at the concrete level are bound together using *events*. Events at concrete level map to the artifacts at the abstract level. Connectors at concrete level are defined as separate nodes, outside of activity implementations. These connectors at the concrete level map to the activity definitions at the abstract level, where connector semantics is encoded within activity definition. Inputs to the activity definition are always of OR-kind, whereas output are of AND-kind. There are three type of connectors: *and* ($\wedge$), *or* ($\vee$) and *exclusive or* ($\oplus$). The concrete

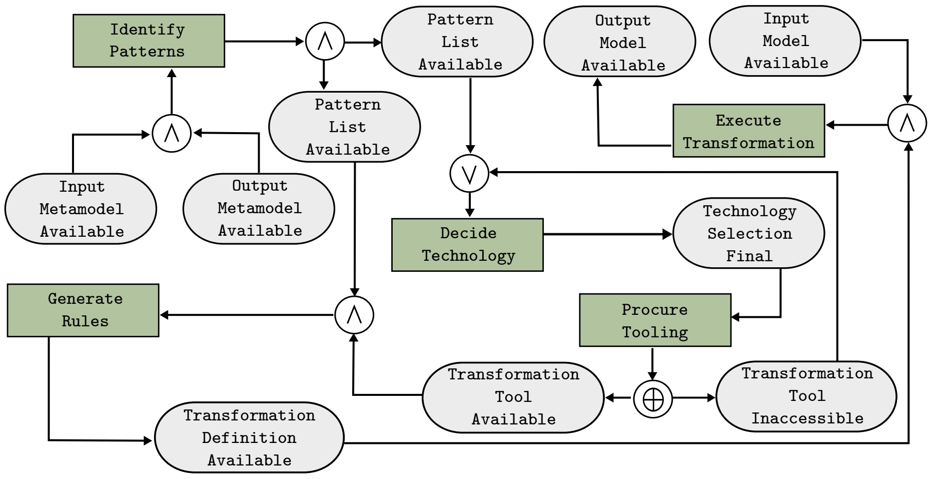

Identify Patterns
Pattern List Available
Output Model Available
Input Model Available
Pattern List Available
Execute Transformation
Input Metamodel Available
Output Metamodel Available
Decide Technology
Technology Selection Final
Generate Rules
Procure Tooling
Transformation Tool Available
Transformation Tool Inaccessible
Transformation Definition Available

Fig. 3. Concrete level model for transformation process example

level constructs of the transformation process example are shown in figure 3. The activity implementations in this figure implement their respective activity definitions in the abstract level, for example *Identify Pattern* is an implementation of *Patten Identification*. Contracts are based on events that map to respective artifacts at the abstract level, for example *Input Metamodel Available* maps to *Input Metamodel*. Connectors at this level are made explicit and they are used for all types of branching or merging of the nodes.

In order to specify the interaction behavior of CPMF, we have chosen to map the constructs of *Process Implementation Metamodel* to Hierarchical Petri Nets. The main constructs of a CPMF process model are *activity definitions* (AD), *artifacts* (W) and *abstract arcs* (Arc^a) at abstract level and *activity implementations* (AI), *events* (E), *connectors* (C) and *concrete arcs* (Arc^c) at the concrete level. All these sets are finite and we will use the term of abstract nodes (set N^a) and concrete nodes (set N^c) for $AD \cup W$ and $AI \cup E \cup C$, respectively. The arcs are directed and map nodes to nodes staying either at abstract level ($Arc^a \subseteq (AD \times W) \cup (W \times AD)$) or at concrete level ($Arc^c \subseteq (N^c \times N^c) \backslash (AI \times AI \cup E \times E)$). Lastly, each connectors is mapped to one of the three connector kinds: $\wedge$, $\vee$ or $\oplus$ by a mapping noted M_C.

A mapping between the concrete level and the abstract level ensures the refinement of process functions from activity definitions to activity implementations. The contracts of the activity implementations need to conform to the contracts of the activity definition. Altogether, this forms a CPMF process as defined below.

Definition 1 (CPMF Process). *Let* $P^a = (AD, W, Arc^a)$ *be an abstract process and* $P^c = (AI, E, C, Arc^c, M_C)$ *be a concrete process. A CPMF process aggregates them as sub-processes in a tuple* $(AD, W, AI, E, C, Arc^a, Arc^c, M_C, f)$*, where* f *defines the refinement mapping between the abstract and concrete level* ($f : P^c \rightarrow P^a$)*:*

- *each event maps to an artifact,* $Im_f(E) \subset W$
- *each activity implementation and each connector maps to an activity definition,* $Im_f(AI \cup C) \subset AD$
- *each arc maps to an arc,* $Im_f(Arc^c) \subset Arc^a$

A process may be viewed as a directed graph (N, Arc), where $N = N^a \cup N^c$ and $Arc = Arc^a \cup Arc^c$. To manipulate a process, we reuse the usual notions of input, output and path.

- for $n \in N$, $in(n) = \{m \mid (m, n) \in Arc\}$,
- for $n \in N$, $out(n) = \{m \mid (n, m) \in Arc\}$,
- a directed path from a node n_1 to a node n_k is a sequence $\langle n_1, n_2, ..., n_k\rangle$ such that $(n_i, n_{i+1}) \in Arc$ for $1 \leq i \leq k - 1$.

As arcs are either at the abstract or at the concrete level, so are the paths.

The connectors can be classified in three sets depending on their semantics defined by the M_C mapping:

- $C_{AND} = \{c \in C \mid M_C(c) = \wedge\}$ is the set of AND kind connectors
- $C_{OR} = \{c \in C \mid M_C(c) = \vee\}$ is the set of OR kind connectors
- $C_{XOR} = \{c \in C \mid M_C(c) = \oplus\}$ is the set of XOR kind connectors

These connectors can also be separated into two sets: *merge connectors* that have more than one input and *fork connectors* that have more than one output:

- $C_M = \{c \in C \mid |in(c)| \geq 2\}$,
- $C_F = \{c \in C \mid |out(c)| \geq 2\}$.

Having these definitions at hand, we can define a well formed CPMF process representing the kind of correct processes that we will be able to translate to hierarchical Petri Nets.

Definition 2 (Well formed CPMF Process). *A well formed concrete process in CPMF is a process* $(AD, W, AI, E, C, Arc^a, Arc^c, M_C, f)$ *that satisfies the following requirements:*

- *On any path in the CPMF process after an AD/AI (respectively W/E), one may not find another AD/AI (respectively W/E) before an W/E (respectively AD/AI).*
- *Events and artifacts can not have an indegree and outdegree of more than one,* $\forall b \in E \cup W \mid \{|in(b)| \leq 1 \wedge |out(b)| \leq 1\}$.
- *There exists a start event with a zero indegree,* $\exists e \in E \mid \{|in(e)| = 0\}$.
- *There exists an end event with a zero outdegree,* $\exists e \in E \mid \{|out(e)| = 0\}$.
- *Activity definitions have an indegree and outdegree of at least one,* $\forall a \in AD \mid \{|in(a)| \geq 1 \wedge |out(a)| \geq 1\}$.
- *Activity implementations have an indegree and outdegree of one,* $\forall a \in AD \cup AI \mid \{|in(a)| = 1 \wedge |out(a)| = 1\}$.
- *A connector has an indegree and outdegree of at least one,* $\forall c \in C \mid \{|in(c)| \geq 1 \wedge |out(c)| \geq 1\}$.

– *Connectors can be partitioned into disjoint sets of merge and fork connectors,* $C_M \cap C_F = \varnothing \wedge C_M \cup C_F = C$.

The first requirement sets the path constraint such that no activity definition/ activity implementation can follow another activity definition/ activity implementation, even if they are connected through a connector (at concrete level). Same ways, no event/artifact can follow another event/ artifact, even if there is connector in between (at concrete level). The next requirement states that an event or artifact can not have more than one input/ output node. The next two requirements precise that the start event has no input node and the end event has no output node. Activity definitions and connectors should also have at least one input and one output nodes, whereas activity implementations should have exactly one input/output nodes. Connectors can be partitioned using two criteria: First by their type, which partitions them into C_{AND}, C_{OR} and C_{XOR} and second by their position, which partitions them into C_M and C_F. Having these two criteria, we can have six different kinds of connectors. A connector that is XOR and merge, means that the process should not proceed, if two synchronized inputs arrive to the node, which does not make any sense in the process modeling, so we do not consider this connector. We have one more connector, *pipe connector*, that is used to connect a single node to another single node when no branching or merging is needed.

4 Mapping to Petri Net

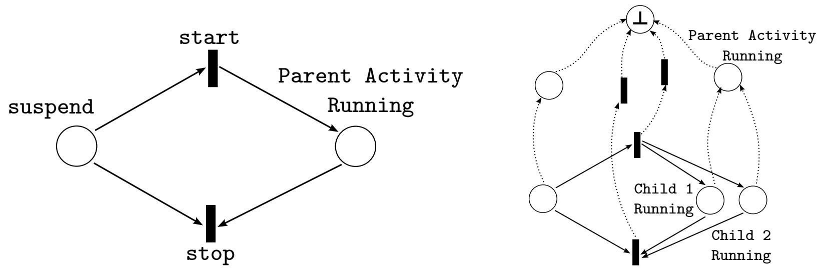
The previous section defined the structure of a CPMF process, whereas in this section we are going to define its interaction behavior by mapping its constructs to Hierarchical Petri Nets. As described before this formalization is targeted towards the business logic level of processes and not towards the formal specification of the processes. Thus we are more concerned about the structural behavior of the processes. As Hierarchical Petri Nets already have a formal specification, we are going to map our process constructs to it.

4.1 Hierarchical Petri Nets

This subsection is a basic presentation of Hierarchical Petri Nets, the interested reader is invited to read [8] to find a complete treatment and all the mathematical definitions. Here, we will only give an intuitive presentation.

First, one has to recall what is a *Petri Net.* It is a bipartite graph N defined as a triple (S, T, F) where S and T are finite sets of *places* and *transitions* respectively and F is a finite set of directed arcs from places to transitions or from transitions to places ($F \subseteq (P \times T) \cup (T \times P)$). It is required that S and T are separated ($S \cap T = \varnothing$) and their union is not empty ($S \cup T \neq \varnothing$). Figure 4(a) contains a simple example of a Petri Net where the places are depicted as circles and transition as black bars.

Then, we are in position to define what is a *Hierarchical Petri Nets (HN).* It is a Petri Net with a *refinement* function. This function links nodes (places or



(a) A simple Petri Net (b) A Hierarchical Petri Net

Fig. 4. Examples of Petri Nets

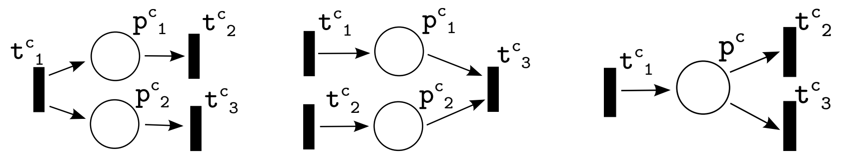
transitions) giving it a structure of tree (a bottom element $\perp$ is added to become the root) as illustrated in the figure 4(b). Formally, a HN is a tuple $(S, T, F, f, \perp)$ where (S, T, F) is a Petri Net and f is the refinement mapping from $S \cup T$ to $S \cup T \cup \{\perp\}$. Moreover f must ensure that (a) it defines a tree structure on $S \cup T \cup \{\perp\}$ where $\perp$ is the root, (b) the arcs (F) only bind leaf nodes (*i.e.* nodes refined by no other nodes) and (c) for any node n, all the leafs refining n that are connected to leafs not refining n are of the same kind of n (both are either places or transitions)[1].

4.2 The translation

We need to map the constructs of CPMF to Hierarchical Petri Nets for this translation. Artifacts/Events are mapped to places, whereas the activity definitions/ implementations are mapped to a small Petri Net Pattern. The corresponding pattern of the activity implementation takes the form of place-transition-place, where the first place corresponds to its pre-conditions and the last to its post-conditions. Activity implementations accepting more than one inputs/outputs can have multiple input/output places. The corresponding pattern of the activity definition is in the form of transition-place-transition, as it has the connector semantics(OR-kind input, AND-kind output) encoded within it. The mapping of connectors also uses sub-Petri Nets, where each type of connector is mapped to a different corresponding Pattern of small Petri Net. Places and transitions in small Petri Nets corresponding to the logical connectors are also part of the Petri Net formed by the process.We are postponing the arcs in between two connectors for later discussion, as they would be resolved later, as complex connectors.

Definition 3 (Corresponding Petri Net). *A well formed process in CPMF is defined as a tuple* $CP = (N^a, N^c, AD, AI, W, E, C, Arc^a, Arc^c)$ *where* $Arc^c \cap (C \times C) = \varnothing$. *Let* $HPN(CP) = (S, T; F, f)$ *be a Hierarchical Petri Net generated by the process* CP*, where:*

[1] This last constraint is a form of well kindness property ensuring that f is really a refinement. Any refinement of a place (resp. transition) is also known by the rest of the system as a place (resp. transition).



(a) $C_{AND} \cup C_F$ (b) $C_{AND} \cup C_M$ (c) $C_{XOR} \cup C_F$

Fig. 5. Petri Nets translations of connectors

- $S = W \cup E \cup (\cup_{x \in AD \cup AI}\ S_x) \cup (\cup_{c \in C}\ S_c)$ *i.e. the places in the generated Hierarchical Petri Net are the artifacts, events, places in the activity definition/implementation translation and connector places (representing connector behavior)*
- $T = (\cup_{x \in AD \cup AI}\ T_x) \cup (\cup_{c \in C}\ T_c)$ *i.e. the transitions in the generated Hierarchical Petri Net are the transitions in the corresponding pattern for activity definitions/ implementations and connector transitions (representing connector behavior)*
- $F = Arc^a \cup Arc^c \cup (\cup_{c \in C}\ F_c)$ *i.e. the arcs in the generated Hierarchical Petri Net are both the abstract and concrete level arcs and the also the ones created for representing connector behavior*
- $(f : X_2 \rightarrow X_1) = f : N^c \rightarrow N^a$ *i.e. the mappings in the generated Hierarchical Petri Net are the ones between the concrete level sub process nodes and the abstract level sub process nodes.*

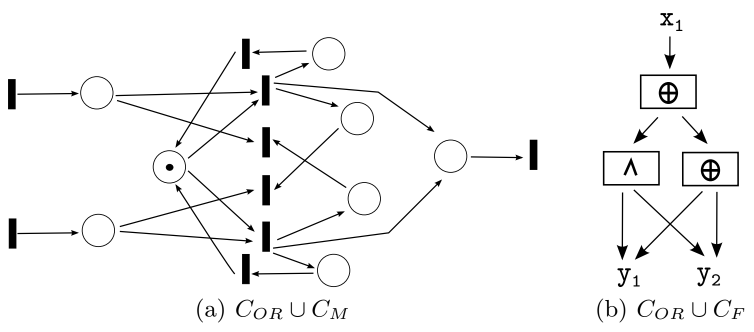
4.3 Connector translation

Connectors are translated to Petri Nets using sub nets *i.e.* each connector corresponds to a small Petri Net in the translation. We were considering six different types of connectors for this process model i.e. a *pipe connector*, two *AND connectors*, a *XOR connector* and two *OR connectors*. Each connector is translated in a manner that the entry node and the exit node of the translated connector is a transition. The places (resp. transitions) created to map the behavior of the connectors, do not correspond to any events or activities.

Pipe Connector: In order to connect an event with an activity directly, where no branching or merging is required, we use a pipe connector. This connector is translated to the Petri Net by a single corresponding transition.

Connectors	**Places** P_c^{PN}	**Transitions** T_c^{PN}	**Arcs** F_c^{PN}
$c \in C_{AND} \cap C_F$	$\{p_x^c \mid x \in out(c)\}$	$\{t^c\} \cup \{t_x^c \mid x \in out(c)\}$	$\{(t^c, p_x^c) \mid x \in out(c)\} \cup$ $\{(p_x^c, t_x^c) \mid x \in out(c)\}$
$c \in C_{AND} \cap C_M$	$\{p_x^c \mid x \in in(c)\}$	$\{t_x^c \mid x \in in(c)\} \cup \{t^c\}$	$\{(t_x^c, p_x^c) \mid x \in in(c)\} \cup$ $\{(p_x^c, t^c) \mid x \in in(c)\}$
$c \in C_{XOR} \cap C_F$	$\{p^c\}$	$\{t^c\} \cup \{t_x^c \mid x \in out(c)\}$	$\{(t^c, p^c)\} \cup$ $\{(p^c, t_x^c) \mid x \in out(c)\} \cup$

Table 1. Corresponding constructs in Petri net for Logical Connectors



(a) $C_{OR} \cup C_M$ (b) $C_{OR} \cup C_F$

Fig. 6. Behavior of OR connectors in Petri nets

AND Connectors: We have two types of AND connectors in CPMF: merge and fork. Like other connectors, this connector also follows the same structure, where the boundary nodes of the translation are transitions that have places in between them. These places and transitions generated for mapping AND connectors are depicted in figure 5(a) and 5(b). Table 1 shows the formal generation of Petri Net constructs, corresponding to each AND connector kind.

XOR Connectors: As discussed earlier, it makes no sense in process modeling that the controlflow should be stopped, if the inputs of the merge connectors are synchronized. Thus we have only one XOR connector that behaves as fork. The generated places and transitions for this connector can be seen in figure 5(c). The formal generation of Petri Net constructs are shown in table 1.

OR Connectors: Just like AND connectors, we have two types of OR connectors. The OR connector that is used for merging is a little more complex than the rest of the connectors. It has to deal with multiple inputs, where the semantics of parallel execution is complex to map to Petri Nets. We have mapped it to a small Petri Net, that is inspired from the discriminator Petri Net [11]. A more comprehensive structure to deal with this problem is presented through this discriminator in synchronizing workflow models [11]. This enhanced discriminator allows the mapping of OR connector to Petri Nets in order to avoid the generation of double tokens during synchronized parallel execution of the inputs, as shown in the figure 6(a). A fork OR connector can be mapped by using a combination of XOR and AND connectors, as shown in figure 6(b).

4.4 Complex Connectors

A fork behavior OR connector can be realized by using AND and XOR connectors but this realization returns some complex connectors in the CPMF process model *i.e.* $(n \times n') \in Arc \mid n, n' \in C$. Such complex connectors can also be intentionally added to meet the semantic requirements of the process model. In order to resolve such complex connectors, we inject a 'dummy' event between the two connectors in the Petri Net i.e.

$$(n \times n') \in Arc \mid n, n' \in C \rightarrow (n \times x) \cup (x \times n') \mid x \in E$$

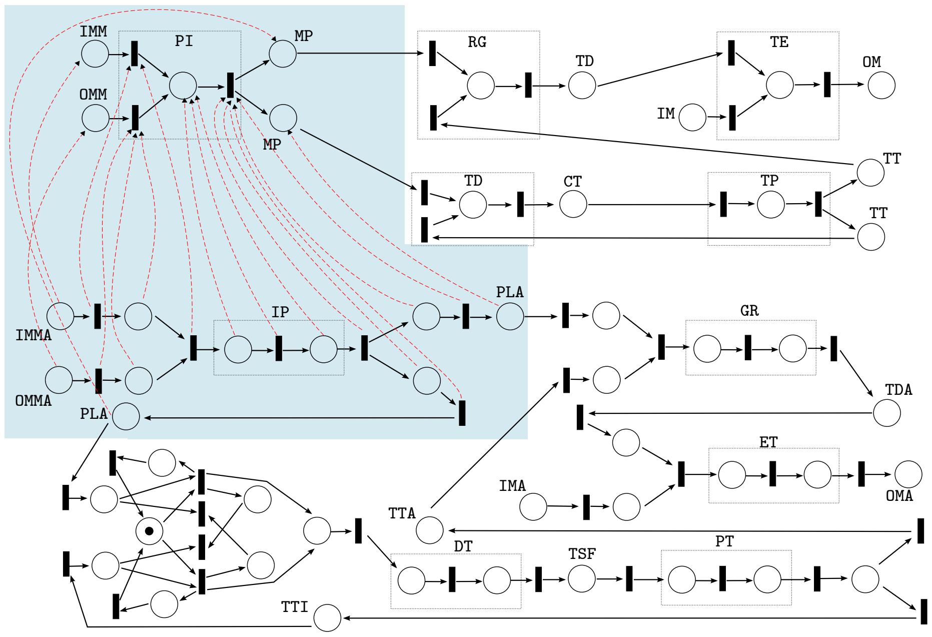


Fig. 7. PN mapping of CPMF process

In order to translate a CPMF process in a corresponding HPN, it is first checked for the complex connectors, which are resolved using these dummy events. Then the rest of the translation is carried out. The translation of the Transformation example of CPMF in corresponding Hierarchical Petri Net can be seen in figure 7. The upper part of the figure depicts the abstract level translation, whereas the lower part depicts the concrete level. For the reasons of clarity, the figure does not show all the mappings, however the portion with highlighted background shows the mapping of constructs between the two levels *i.e.* the *Identify Pattern* activity implementation along with its associated events, maps to the *Pattern Identification* activity definition and its associated artifacts.

5 Conclusion

This paper formally defines the constructs of the CAMA Process Modeling Framework both at the abstract and concrete level. It then takes the definition of Hierarchical Petri Nets and maps CPMF constructs to it. This mapping takes into account the constructs presented in the *Process Implementation Metamodel* of CPMF. Though this approach is not targeting to a formal specification of the approach, still it describes the interaction behavior of the activities in the process model. This interaction behavior is captured by specifying the behavior of each connector at the abstract level and at the concrete level. The dataflow is handled at the abstract level, whereas the controlflow is handled at the concrete level. We are looking forward to use this mapping for the verification of the execution semantics of CPMF. This would also help us in developing the tool support for the CPMF framework, where we are planning to extend Openflexo, an open source tool, whose semantics are already defined using Petri Nets.

References

1. Agile Birds. Openflexo. `http://openflexo.com`, 2011.
2. Boualem Benatallah, Marlon Dumas, Marie-Christine Fauvet, and Fethi A. Rabhi. Patterns and Skeletons for Parallel and Distributed Computing. chapter Towards Patterns of Web Services Composition, pages 265–296. Springer-Verlag, 2003.
3. Réda Bendraou, Benoit Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an eXecutable SPEM2.0. In *14th Asian-Pacific Software Engineering Conference (APSEC)*, pages 390–397, Nagoya, Japan, dec 2007.
4. Shih-Chien Chou. A Process Modeling Language consisting of high level UML-based diagrams and low level Process Language. *Journal of Object Technology*, 1(4):137–163, sep 2002.
5. Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information and Software Technology*, 50(12):1281 – 1294, 2008.
6. Haiqiang Dun, Haiying Xu, and Lifu Wang. Transformation of BPEL Processes to Petri Nets. In *Theoretical Aspects of Software Engineering, 2008. TASE '08. 2nd IFIP/IEEE International Symposium on*, pages 166 –173, june 2008.
7. ECSS. *ECSS-E-ST-40C: Space Engineering - Software.* ECSS (European Cooperation for Space Standardization), Noordwijk, Netherlands, March 2009.
8. Rainer Fehling. A concept of Hierarchical Petri Nets with building blocks. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer Berlin / Heidelberg, 1993.
9. Fahad R. Golra and Fabien Dagnat. Using Component-oriented Process Models for Multi-Metamodel Applications. In *Proc. of the 9th International Conference on Frontiers of Information Technology*. IEEE, Dec 2011.
10. Fahad R. Golra and Fabien Dagnat. Generation of Dynamic Process Models for Multi-metamodel Applications. In *Proc. of the International Conference on Software and System Process, ICSSP*. IEEE, June 2012.
11. B. Kiepuszewski, Arthur H. M. Ter Hofstede, and W. M. P. van der Aalst. Fundamentals of Control Flow in Workflows. *Acta Informatica*, 39:143–209, 2002.
12. Ali Koudri and Joel Champeau. MODAL: A SPEM Extension to improve Co-design Process Models. In *New Modeling Concepts for Today's Software Processes*, volume 6195 of *LNCS*, pages 248–259. Springer Berlin / Heidelberg, 2010.
13. Niels Lohmann, Eric Verbeek, and Remco Dijkman. Petri Net Transformations for Business Processes - A Survey. In *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460 of *LNCS*, pages 46–63. Springer, 2009.
14. Ruopeng Lu and Shazia Sadiq. A Survey of Comparative Business Process Modeling Approaches. In Witold Abramowicz, editor, *Business Information Systems*, volume 4439 of *LNCS*, pages 82–94. Springer Berlin / Heidelberg, 2007.
15. OMG. Software and Systems Process Engineering Metamodel Specification. Version 2.0, April 2008.
16. Peter Rittgen. Paving the road to Business Process Automation. In *Proceedings of the 8th European Conference on Information Systems*, pages 313–319, 2000.
17. W. M. P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639 – 650, 1999.

Formal and Fault Tolerant Design

Ammar Aljer[1], Philippe Devienne[2]

[1]Faculty of Electrical and Electronic Engineering, University of Aleppo, Aleppo, Syria
`Ammar.Aljer@lifl.fr`
[2] Lille's Computer Science Laboratory, University of Lille, Lille, France
`Philippe.Devienne@lifl.fr`

Abstract. Software quality and reliability were verified for a long time at the post-implementation level (test, fault scenario …). The design of embedded systems and digital circuits is more and more complex because of integration density, heterogeneity. Now almost ¾ of the digital circuits contain at least one processor, that is, can execute software code. In other words, co-design is the most usual case and traditional verification by simulation is no more practical. Moreover, the increase in integration density comes with a decrease in the reliability of the components. So fault detection, diagnostics techniques, introspection are essential for defect tolerance, fault tolerance and self repair of safety-critical systems.

The use of a formal specification language is considered as the foundation of a real validation. What we would like to emphasize is that refinement (from an abstract model to the point where the system will be implemented) could be and should be formal too in order to ensure the traceability of requirements, to manage such development projects and so to design fault-tolerant systems correct by proven construction. Such a thorough approach can be achieved by the automation or semi-automation of the refinement process.

We have studied how to ensure the traceability of these requirements in a component-based approach. Reliability, fault tolerance can be seen here as particular refinement steps. For instance, a given formal specification of a system/component may be refined by adding redundancy (data, computation, component) and be verified to be fault-tolerant w.r.t. some given fault scenarios. A self-repair component can be defined as the refinement of its original form enhanced with error detection.

We describe in this paper the PCSI project (Zero Defect Systems) based on B Method, VHDL and PSL. The three modeling approaches can collaborate together and guarantee the codesign of embedded systems for which the requirements and the fault-tolerant aspects are taken into account for the beginning and formally verified all along the implementation process.

1 Introduction

The final decades of the 19th century and the first decades of the 20th century witnessed many efforts to formulate mathematics. Some fruits of these essays are Set theory, Propositional Logic, First Order Logic, etc. Introduced by Alonzo Church in the 1930s, λ-calculus is a primitive method to formalize algorithms where many concepts similar to those of programming languages are well defined such as: Recursion and fixed points, Logic and predicates, Free and bound variables, Substitutions. In the beginning of 1950s Von Neumann described a computer architecture in which the data and the program are both stored in the computer's memory in the same address space. This architecture is to this day the basis of modern computer design. In the beginning programmers wrote their programs as strings of zeros and ones. A work would often be an extremely frustrating activity. Rapidly this task is facilitated depending on Assembly language and OpCode tables. Developed in the mid-1950s, FORTRAN was intended for use in scientific and numerical computing applications. It may be considered as the first high level language. From the outside, it uses formal mathematical-like expressions but actually these expressions and instructions are chosen to abstract the executive machine code. A compiler is written to convert each FORTRAN program code into machine code. Programs were used to partially help client with automatically and rapidly executing an algorithm. Most of later software developments (such as structural programing then OOP) concentrated on the abstraction of the executive machine code. Nowadays writing the implementation is partially automated and designer may give more attention on system structure. Actually with CASE (computer Aided Software Engineering) tools and with techniques such as MDA (Model Driven Architecture), programmer can graphically specify the components of the design, precise the operation of each component and defines the relations between components then executive code is automatically generated. Nowadays computer is used not only to execute a program but to represent a complete system and furthermore to simulate a complex of interacting systems. Verification becomes more and more difficult because its cost increases exponentially with complexity. Reusing is another aspect of complex systems. In most cases programmer reuses ancient classes or libraries (written by him or by others) in new projects. With COSTS (Commercial, off-the-shelf), programmer reuses a complete software system. He ought to adapt them to the novel environment.

Only few efforts are made to formulate the other side of the programming task; that is client requirements. With the increasing machine power and augmenting complexity of computer based systems, Software engineering developed many principles and techniques to formulate client requirements. Comparing to the development of programming language, these efforts rest primitive and a formal gap between what a program do and what a client wants is always exists.

B method (1996) filled partially the gap. It defines what a formal refinement of software is. So it guarantees the complete correctness of software regarding to its formal specification. In our approach this method is generalized to be used in software, hardware and in embedded systems. Proving the correctness of one component is usually expensive comparing to the traditional methods of verification but this is rapidly compensated when the component is reused and when complexity augments; proving the correctness of a system that consists of proven components needs only to prove the correctness of the connections between the components. This approach also facilitates the verification parallelism since each component could be independently proven.

Components in real word (especially hardware ones) do not correspond 100% to their formal specification. This is a cause for many failures in the system even if it was proven to be correct or if its behavior is verified during the simulation. Another important feature of our approach is the possibility to prove the correctness of model even with real failure scenario if it is combined with a suitable correcting treatment.

2 Domain Specific Languages

On the opposite of programming languages who are designed for experimented programmers, a Domain Specific Language (DSL), comes from a domain and is used by users of this domain. Thus, a successful DSL is of course a used language and first intuitively usable by users of the chosen domain.
One of the first Domain Specific Language (DSL) was introduced for children. Its name was Logo and was designed by Seymour Papert at MIT in the sixties. He was been nominated by Marvin Minsky as “the greatest living educator in Mathematics”. In Mindstorms [5], Seymour Papert explained that some children have difficulties in mathematics logic and this new language was specifically create to improve the way that children solve mathematical problems. Excel and MatLab are two well-known examples of DSL in mathematics too. Excel was even described as a killer application because it is so easy and funny to use by anyone.
This is quite opposite to view of universality in general-purpose programming language, such as C or Java, or a general-purpose modeling language such as UML.
Recently, the DSL approach has really been successful in two domains, web applications and cell phones. There are a lot of View/Edit WebDSLs from which we can generate Java or PHP code, web pages and Seam session beans. For instance, SPIP is a publishing system for the Internet in which great importance is attached to collaborative working, to multilingual environments, and to simplicity of use for web authors
Developing a new DSL needs definitively a good understanding of the application domain, then the next usual consist of finding programming patterns, designing a core language, building syntactic abstractions on top of the core language. But this type of design is a real complex activity and must be based on good tools, especially for verification. A lot of researches have to be lead to propose such an appropriate environment with good tools and libraries.
From the other hand, another challenge is appeared with embedded systems where more or more communities (usually hardware and software) with totally different methodologies, terminologies and measurements should design one common component! The DSLs have to be combined and collaborate in the same final objet.

2.1 VHDL

Due to the difference between hardware product and software product, Production of hardware or software component passes through two different sequences. Software engineers concentrate on requirement collection, development, verification, deployment .etc. Hardware engineers emphasis on functional level, logic gate level, RTL (Register Transfer Level) and printed circuit level. The increasing system complexity obligates both communities to develop their tools towards abstract system level. VHDL that is the dominant language in hardware design was the first to take system level in account.

Even if VHDL was designed for electronic design automation to describe VLSI circuits, it argues that it can be used as a general-purpose language and even can handle parallelism. From hardware community point of view, VHDL may be used to describe the structure of the system since any circuit may be defined as a black box (ENTITY) where all the inputs and outputs are defined then by a white box (ARCHITECTURE) where all the components and connections between these components are declared. Components in the architecture are functionally defined and they could be mapped later to the real word components by an additional level (CONFIGURATION). So it is supported with libraries that contain all specifications of electronic units known in the world. These layers permit to simulate the real circuit in order to verify the design. ARCHITECTURE layer in VHDL may define the behavior of the circuit instead of its structure.

2.2 B METHOD, MOCHA, B-Event

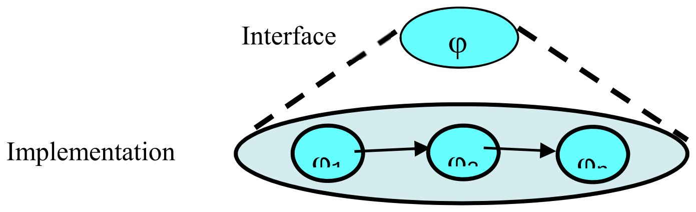
B method [1,2] is known in software engineering as a formal method to specify and to develop finely the specification towards an executable program basing on set theory and first order logic notation. B draws together advances in formal methods that span the last forty years (pre and post notations, guarded commands, stepwise refinement, the refinement calculus and data refinement). During the software development in B method, many versions of the same component may be found. The first and the most abstract one is the abstract machine where client needs are declared. Then the following versions should be more concrete and precise more and more how we obtain the needed specifications. These versions are called refinements except the last one where there is no more possible refinement. This deterministic version is called implementation. B generates the necessary proof obligations to verify the coherence of each component and correctness of the development. Furthermore, B tools help to execute these proofs.

Like B, Mocha [3] is a interactive verification environment for the modular and hierarchical verification of heterogeneous systems. Mocha supports the heterogeneous modeling framework of reactive components and based on Alternating Temporal Logic (ATL), for specifying collaborations and interactions between the components of a system.

Event B is an evolution of B Method. Key features of B Event are the extensions to events for modeling concurrency. The primary concept in doing formal developments in Event-B is that of a model. A model contains the complete mathematical development of a Discrete Transition System. It is made of several components of two kinds: machines and contexts. Machines contain the variables, invariants, theorems, and events (section 2) of a model, whereas contexts contain carrier sets, constants, axioms, and theorems of a mode. The Rodin platform is an open source Eclipse-based IDE for Event B is further extendable with plugins. The overall objective of this open platform is to propose a toll for the cost effective rigorous development of dependable complex systems and services. It focus on tacking complexity (1) caused by the environment in which the software I to operate (2) which comes from poorly conceived architectural structure. Mastering complexity in the shortest time-to-market requires design techniques that support clear thinking and rigorous validation and verification. Coping with complexity also requires architectures that are tolerant of faults and unpredictable changes in environment. This is addressed by fault tolerance design techniques.

3 Formal Hardware Design In BHDL Project

In this section, we are going to focus of the collaboration of two DSL languages, one for mathematics and logic (Event B) and the other to design VLSI (VHDL). Both have been widely validated by complex large industrial applications. Here we try to combine the advanced notion of formal refinement in B with the formal conception of HDL. The core of the work, from which the name of the project BHDL comes, is to create the correspondence between a VHDL design and a B one. In VHDL, the transition from an Entity into a corresponding Architecture is usually performed in one step. In BHDL, this may be performed finely by many steps or levels. We may consider the refinement of a component in BHDL as a replacement by other components. Also we may refine a component by another one which has the same structure and links but with more strict logic property. In all cases the refinement is performed towards lower levels where the behavior of the system becomes more deterministic.
The principal relation between the interface (external view) and its refinement (or between two levels of refinement) is Connection($\varphi1$, $\varphi2$,, …φn) $\Rightarrow \varphi$ which means that the logical connection between the properties of the sub-components should satisfied the properties indicated in the abstract machine that represents the Entity. The property (φ) in the interface is not original in VHDL, it is inserted in special comments in VHDL code so BHDL code does not affect the design portability.
The principle of B refinement permits not only to prove the consistency of Architecture but also to prove the correctness of the design w.r.t. the abstract specification in the external view (VHDL Entity). Furthermore it allows hardware community to built their pattern throw many smooth phases instead of one rough phase from all the components should be specified. The main components of BHDL project are:



$$\varphi_1 \wedge \varphi_2 \wedge \varphi_n \wedge \text{links} \Rightarrow \varphi$$

Fig. 1. Structural refinement and proof obligation.

3.1 A graphical interface for System Entry (VGUI)

It is a Graphical User Interface for Hardware Diagrams. It is an open source tool that may be considered as a simple component description tool. VGUI may be used to create generic interconnected boxes. Each box may be decomposed hierarchically into sub-boxes and so on. The boxes and the connections of VGUI are typed. In cooperation with VGUI developer, we added the possibility to attach logic property to each box and hide data. Eventually, VGUI generates VHDL code annotated with B expressions. This step is optional; designer may use a textual editor to directly write the annotated code to be analyzed by the following step.

3.2 B Model Generator

Here a B model that corresponds to the annotated VHDL model is created. The ANTLR compiler is used to generate B code. From the external view of VHDL or from an entity in VHDL model, it generates the suitable B Abstract Machine that contains the necessary properties of the Entity and traces the structure of VHDL model.
In a similar way, the internal view in VGUI is translated into Architecture in VHDL then into a refinement in B. Because that design in VHDL usually depends of some predefined standard libraries, we created some B components that correspond to some VHDL libraries (such as the Standard logic 1164).

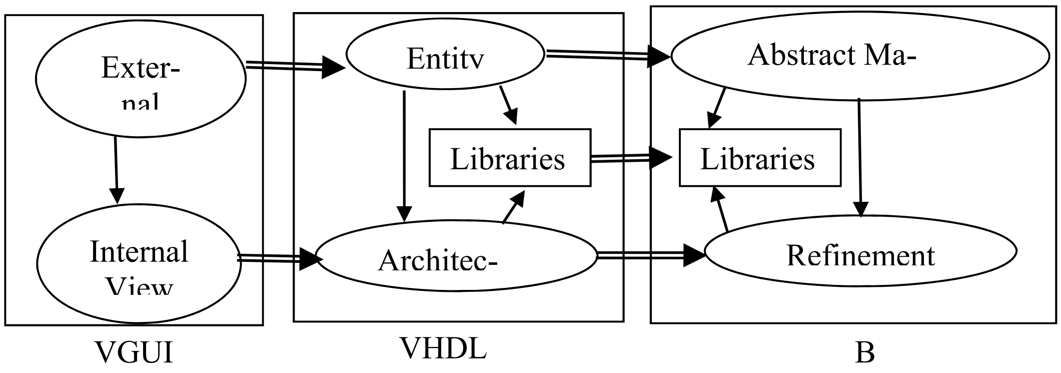


Fig. 2. Main transformations of BHDL.

The compiler is the most important practical part of BHDL project. It is built on ANTLR compiler generator. ANTLR (Another Tool for Language Recognition) is a powerful tool that accepts grammatical language descriptions and generates programs (compliers or translators) that can recognize texts in the described languages, analyzes these texts, constructs trees corresponding to their structure and generates events related to the syntax. These events, written in C++ or in Java, may be used to translate the text into other languages. It can generate AST (Abstract Syntactic Trees) which can stock a lot of information about the analyzed text, provides tree rewriting rules for easily translating these ASTs. The correction of such a translator depends only on the correction of every elementary rewriting rule (declarative semantics). As VGUI, ANTLR is open source software written in Java. The translation from VHDL+ to B in is performed over many steps:

- BHDL Lexer/Parser : which analyses the input VHDL+, verifies the syntax and the semantic of VHDL code, then it generates a pure VHDL tree (AST) with independent branches that contain the B annotations
- TreeWalker: this tree parser parses the previous AST in order to capture the necessary information to construct a new AST that corresponds to B model.
- B-Generator: It traverses the AST produced by the TreeWalker in order to generate B code.

Even if a corresponding B model is automatically created, the design correctness is not automatically proven. The generated B code should be proven to be correct. B tools (AtelierB, Rodin, B4Free, B-Toolkit) render the task easier. It generates the necessary proof obligations (POs), automatically produces an important quantity of these proofs, cooperates with the programmer to prove the remaining POs. Here, if the model is not completely proven, some defects may be detected and the original VHDL design should be modified.

4 PCSI Project

BHDL project is developed in the LIFL (Lille's Computer Science Laboratory). This research first conducted into the AFCIM project (LIFL, INRETS, HEUDIASYC Lab). Eventually the main concepts of BHDL have been extended and implemented with support of PCSI project (Zero Defect Systems) between Lille University, Aleppo University and Annaba University. The main new features of the project are the following:

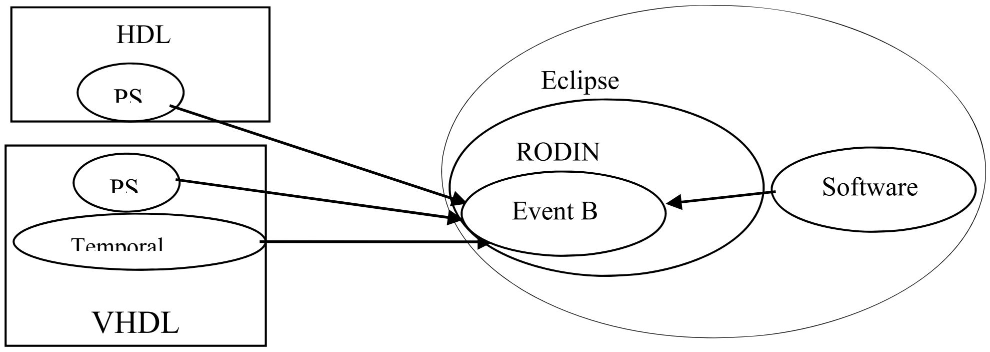


Fig. 3. basic augmentation in the PCSI project (Zero Defect Systems) vs. BHDL.

4.1 Including PSL

Instead of special comments used in the first version of BHDL to represent the logical behavior of VHDL components, we use here a formal language, PSL, standardized in

2005. PSL (Property Specification Language) is a language for the formal specification of hardware. This language is essentially based on Linear Temporal Logic “LTL”. It is used to describe properties that are required to hold in the design under verification. It contains Boolean, Temporal, Verification and modelling layers. The flavour of PSL could be added to many HDL (Hardware Description Language) such as VHDL, Verilog, SystemVerilog. This enlarges the usability of our tool since PSL is expressive and standard.
In this project, we generate a software model which is B representation of a model described by Property Specification Language “PSL” using Event B Systems. Generated model can be proven by using B method techniques; this means a proof of the original PSL model.

4.2 Extending scope of VHDL treated in BHDL

While the first version of BHDL mainly manipulates the design structure decorated with logical properties, here we enlarge the model to accept important concepts of VHDL such as signals where the concept of Time appears.

4.3 Creating the target model using Event-B instead of Classical B

The purpose of Event-B is to model full systems (including hardware, software and environment of operation). Classical B is not suitable to represent temporal properties which are important in hardware design. Furthermore, Event-B facilitates the representation of many subsystems in a global one.
After the creation of a HDL model, it will be traced in B. in order to facilitate the proof of the consistency and the formal refinement of the model; we integrated our work in Eclipse environment. Eclipse is generic platform to develop multi-language software comprising an integrated development environment (IDE) and an extensible plug-in system. The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. The platform is open source, contributes to the Eclipse framework and is further extendable with plugins. Such integration renders the integration between hardware community and software community easy since they work on the same environment. All the tools used in our platform are freely used and distributed (Rodin, Eclipse, Antlr, …).

4.4 Completion wrt robustness

Robustness or Fault-Tolerance is by definition “The ability of a system to respond gracefully to an unexpected hardware or software failure”. There are many levels of fault tolerance, the lowest being the ability to continue operation in the event of a power failure. Many fault-tolerant computer systems mirror all operations -- that is, every operation is performed on two or more duplicate systems, so if one fails the other can take over. Multitolerance refers to the ability of a system to tolerate multiple classes of faults. [19] illustrates a compositional and stepwise method for designing programs and handling Byzantine failures [18]. It proposes also a component based method, starts with a intolerant system and adds a set of components, one for each desired type of tolerant. The complexity of multitolerant design is reduced to that of designing the components and of correctly adding them to the intolerant one.

Indeed, fault tolerance is often based on replication and redundancy. This is involved by the use of hybrid systems with different sources of energy (electric, mechanic). This duplication can be also seen as component refinement or algorithmic refinement. For instance, nowadays, because of the integration of circuits, stuck-at–fault is a more and more frequent fault model. According that the probability that a circuit contains at least k stuck-a-fault is too high, we can generate an equivalent circuit, except that it is k-stuck-at-fault tolerant. We focus here on the problem in evolving a fault-intolerant program to a fault-tolerant one. The question is "Is It possible to add a default scenario to an existing model or program and generate the tolerant model or program?" This problem occurs during program evolution new requirement (fault-tolerance property, timing constraints, and safety property) change. We argue here that refinement can handle this evolution. In others words a fault-tolerant program is a refined form of its intolerant one. We have shown how to apply this formalism to characterize fault-tolerance mechanisms and to then reason about logical and mathematical properties. For instance, the hamming code is a kind of "minimal" + data refinement. By adding data redundancy (extra parity bits), error-detection and even error-correction are possible. This can generalize to handle Byzantine properties. For instance, masking tolerance considers that in the presence of faults each step in the system computation satisfies Validity and Agreement properties. Weakly, Stabilizing tolerance considers that each step in the system computation will satisfy in a near future (reachable state) Validity and Agreement. What we show here is that fault-tolerant design is compositional (component-based method), adaptive (stepwise and hierarchical approach) and formal (these completions are refinements which have to be formally proven). The fault-tolerant design appears as a logico-mathematical completion of an intolerant model in order to tolerant multiple classes of faults.

5 CONCLUSION

The novel aspects of this proposal are the pursuit of a process-based approach, the combination of Formal Methods with Fault Tolerant techniques, the development of FM support for component reuse and composition and the extension of an open and extensible tools platform for formal development. It is clear that the open tools platforms will have a more and more important impact on future research, but they have to be adapted to users and their languages as VHDL. We believe that proposing intelligent interfaces between DSL approaches (as VHDL+PSL) and FM tools (as RODIN) is the shortest way to make these techniques more popular and will encourage greater industrial uptake.

References

1. Abrial J.R. The B-Book: assigning programs to meanings. UK: Cambridge University Press, 1996.
2. Abrial J.R, Modeling in Event-B, System and Software Engineering. UK: Cambridge University Press, 2010.
3. Rajeev Alur, et al. Mocha: Modularity in model checking. In Proceedings of the Tenth International Conference on Computer-aided Verification, Lecture Notes in Computer Science 1427, Springer-Verlag, 1998..
4. Seymour Pappert, Mindstorms: Children, Computers, and Powerful Idea, 1980.
5. Ammar Aljer, Co-design and refinement in B, Ph.D. Thesis, Lille University, 2004.
6. Philippe Devienne , Rabih Oueidat, Formal Tolerant Software/Harware Architecture, In Specification and Verification of Component-Based Systems, Workshop at SIGSOFT 2008/FSE 16 (SAVCBS 08), Atlanta, Nov 2008
7. Philippe Devienne, Ammar Aljer, Extended Model Driven Architecture to B method, Ubiquitous Computing and Communication Journal, 2011
8. Y. Herve, VHDL-AMS – Applications et enjeux industriels (in french), Durand, France, 2002.
9. RODIN : http://www.event-b.org/platform.html
10. Flaviu Cristian, Understanding Fault-Tolerant Distributed Systems, ACM, Feb 1991, 34(2): 56-78
11. Terence Parr, The definitive ANTLR Reference, 384 pages, May 2007
12. DOULOS, PSL Golden Reference guide, Book, 2005.
13. Anish Orora, Gray-Box Component-Based Fault-Tolerance, Logical Aspects of Fault Tolerance (LAFT), a LICS 2009 Workshop.
14. Kenneth E. Kendall et Julie E. Kendall, 2010, Systems Analysis and Design, 8/E, Prentice Hall.
15. Ian Sommerville, 2008, Software Engineering: International Version , 8/E, Addison-Wesley
16. The Rascal Domain Specific Language, INRIA Report, 2009
17. L. Lamport, R. Shostak and M.Pease. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems, 1982.
18. S.S. Kulkarni, A.Arora, Composition Design of Multitolerant Repetitive Byzantine Agreement, 17th Conference on Foundations of Software Technology and Theoretical Computer Science, 1997

Towards an Agile Foundation for the Creation and Enactment of Software Engineering Methods: The SEMAT Approach

Brian Elvesæter[1], Michael Striewe[2], Ashley McNeile[3] and Arne-Jørgen Berre[1]

[1] SINTEF ICT, P. O. Box 124 Blindern, N-0314 Oslo, Norway
{brian.elvesater, arne.j.berre}@sintef.no
[2] University of Duisburg-Essen, Gerlingstrasse 16, D-45127 Essen, Germany
michael.striewe@s3.uni-due.de
[3] Metamaxim, 48 Brunswick Gardens, London W8 4AN, UK
ashley.mcneile@metamaxim.com

Abstract. The Software Engineering Method and Theory (SEMAT) initiative seeks to develop a rigorous, theoretically sound basis for software engineering methods. In contrast to previous software engineering method frameworks that rely on separate method engineers, the primary target of SEMAT are practitioners. The goal is to give software development teams the opportunity to themselves define, refine and customize the methods and processes they use in software development. To achieve this goal SEMAT proposes a new practitioner-oriented language for software engineering methods that is focused, small, extensible and provides formally defined provides formally defined behaviour to support the conduct of a software engineering endeavour. This paper presents and discusses how the proposed language supports an agile creation and enactment of software engineering methods. The SEMAT approach is illustrated by modelling parts of the Scrum project management practice.

Keywords: Method engineering, software engineering methods, SEMAT, SPEM, ISO/IEC 24744, Scrum

1 Introduction

The Software Engineering Method and Theory (SEMAT)[1] initiative seeks to develop a rigorous, theoretically sound basis for software engineering methods. Previous software engineering frameworks and standards such as the Software and Systems Process Engineering Metamodel (SPEM) 2.0 [1] mainly target *method engineers* by providing rich but consequently often complex languages for detailed process definition; but generally not supporting enactment [2]. In contrast, the primary objective of SEMAT is to target *practitioners* (i.e., architects, designers, developers, programmers, testers, analysts, project managers, etc.) by providing a focused, small, extensi-

[1] http://www.semat.org

ble domain-specific language that allows them to create and enact software engineering methods in an agile manner.

An agile approach to software engineering methods is one that supports practitioners in dynamically adapting and customizing their methods during the preparation and execution of a project, controlled through company-specific governance, use of examples and other means. This enables practitioners to accurately document how they work and effectively share their experiences with other teams. To go beyond the current state of the practice a robust and extensible foundation for the agile creation and enactment of software engineering methods is required.

This paper presents and discusses the main new ideas and language concepts from the Essence specification [3] which has been submitted by SEMAT as a response to the Request for Proposal (RFP) "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" [4] issued by the Object Management Group (OMG). The remainder of the paper is structured as follows: In Section 2 we present and summarise the language requirements stated in the OMG RFP. In Section 3 we present the language architecture of the Essence specification and its main language constructs. In Section 4 we illustrate the SEMAT approach by modelling parts of the Scrum project management practice. Section 5 discusses this approach to related work. Finally, Section 6 concludes this paper and describes some future work.

2 Language Requirements and the Meta-Object Facility (MOF)

The OMG RFP "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" [4] solicits proposals for a foundation for the agile creation and enactment of software engineering methods. This foundation is to consist of a *kernel* of software engineering domain concepts and relationships that is extensible (scalable), flexible and easy to use, and a domain-specific modelling *language* that allows software practitioners to describe the essentials of their current and future practices and methods. In this paper we focus on the language. The requirements for the language are summarised in Table 1 below.

Table 1. Language definition (1.x) and language features (2.x) requirements

ID	Name	Description
1.1	MOF metamodel	Abstract syntax defined in MOF.
1.2	Static and operational semantics	Static and operational semantics defined in terms of the abstract syntax.
1.3	Graphical syntax	Graphical syntax that maps to the abstract syntax.
1.4	Textual syntax	Textual syntax that maps to the abstract syntax.
1.5	SPEM 2.0 metamodel reuse	Reuse SPEM 2.0 metamodel where appropriate.
2.1	Ease of use	Easy to use by practitioners.
2.2	Separation of views	Separation of two different views of a method for practitioners and method engineers.
2.3	Specification of kernel elements	Description, relationships, states, instantiation and metrics.
2.4	Specification of practices	Cross-cutting concern, element instantiation, work products, work progress, verification.

ID	Name	Description
2.5	Composition of practices	Overall concerns, merging elements, separating elements, practice substitution.
2.6	Enactment of methods	Tailoring, communication, managing, coordinating, monitoring, tooling.

The language definition (1.x) requirements mandate that the language must be compliant with the MOF metamodel architecture. The Meta-Object Facility (MOF) specification [5] serves as a foundation for the OMG Model-Driven Architecture (MDA) [6] approach and provides us with a formalism to define and integrate modelling languages. The left side of Fig. 1 illustrates the MOF four-layer architecture. MOF defines a meta-metamodel or meta-language at the M3 layer which can be used to define a metamodel or language to support method engineering at the M2 layer. A method engineering modelling language typically defines language constructs to support the definition and composition of methods out of reusable model fragments or templates. These model templates at the M1 level are typically defined by method engineers and are instantiated during a software endeavour, i.e., software development project, and used by the software practitioners (M0 level).

The right side of Fig. 1 illustrates an instance tree. MDA positions MOF as the single meta-language (M3), so there is only one top node for which different languages (M2) can be defined. Each of these modelling languages can be used to define various models (M1) of which different instantiations can be made (M0). In this paper we focus our discussion on a single domain-specific language to support method engineering. Thus in the remainder of this paper we will focus on the M2 layers and below as illustrated by the dashed boxes.

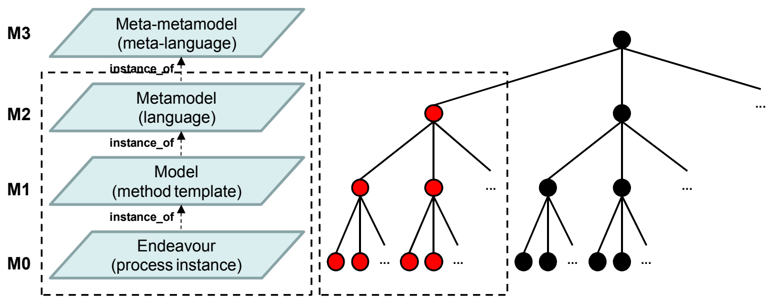


Fig. 1. MOF metamodel architecture and instantiation tree

The language features (2.x) requirements focus in particular on the ability to specify practices, compose practices into methods and enact those methods. The requirements also state that existing foundations such as the SPEM 2.0 [1] should be reused where appropriate.

3 The SEMAT Approach

The Essence specification [3], which is the initial response to the OMG RFP from the SEMAT community, defines a kernel of essentials for software engineering and an associated language for describing practices and methods using the kernel as a standardised baseline. This approach draws inspiration from the ideas presented by Ivar Jacboson et al. in [7] which is supported by the tool EssWork[2]. The SEMAT community is iterating and improving on these ideas with the goal of defining a widely-accepted kernel and language that can be standardised by the OMG.

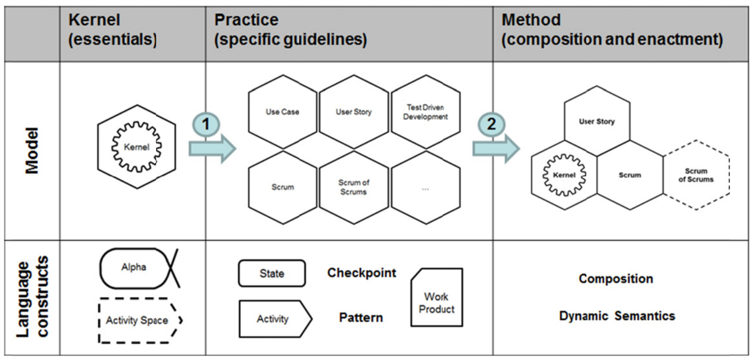


Fig. 2. Overview of the SEMAT approach

Fig. 2 illustrates the SEMAT approach and some of the key language constructs and their proposed graphical notation. A key idea is that in all software endeavours, there is a kernel, i.e., a common ground, which includes a few essential elements of software engineering that can be used as a standardised baseline for describing software engineering methods. The kernel defines a compact set of *Alphas* that represents essential things to work with (e.g., *Requirements, Work, Team, Software System,* etc.) that are universal to software engineering and a small set of *Activity Spaces* that represents the essential things to do (e.g., *Prepare to do the Work, Coordinate the Work, Support the Team,* etc.).

A Practice (e.g., *User Story, Scrum,* etc.) use the kernel as a baseline and provides specific guidelines expressed using language constructs such as *State*, *Activity*, *Checkpoint*, *Pattern* and *Work Product* addressing a particular aspect of the work at hand. The practices are composed to form a method. A definition of a method is given in [8] as "a systematic way of doing things in a particular discipline".

The language contains four parts, 1) abstract syntax, 2) composition algebra, 3) dynamic semantics and 4) graphical syntax. This paper focuses on the static and dynamic semantics which will be elaborated in the following subsections.

3.1 Static Semantics – Creation of Software Engineering Methods

The static semantics of the language specifies a metamodel that contains constructs to describe the kernel, practices and methods. The language has been structured as layers to allow for easier understanding and usage of different subsets. Fig. 3 below shows,

[2] http://www.ivarjacobson.com/process_improvement_technology/esswork

in a slightly simplified form[3], the layers of the language and the key constructs of each layer.

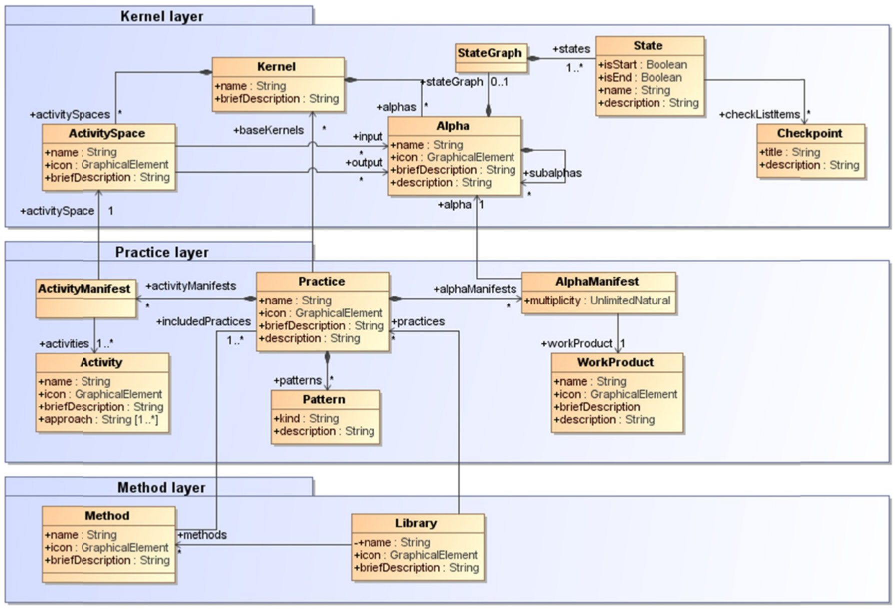


Fig. 3. Language specification – static semantics (modified and simplified metamodel excerpt)

The *Kernel layer* contains the constructs *Kernel*, *Alpha* and *Activity Space*. A *Kernel* is a set of elements used to form a common ground for describing a software engineering endeavour. *Alpha* is short for *Abstract-Level Progress Health Attribute* and represents an essential element that is relevant to the assessment of the progress and health of a software engineering endeavour. Alphas are used to describe subjects whose evolution we want to understand, monitor, direct and control. Each Alpha has a state graph with well-defined states. The states define a controlled evolution throughout its lifecycle. Each state has a collection of checkpoints that describes the criteria that the Alpha must fulfil to be in that particular state. An Alpha may also have subalphas, e.g., for splitting *Software System* into *Components* or *Requirements* into *Requirement Items*. An *Activity Space* provides a placeholder for something to be done in the software engineering endeavour. Activity Spaces frame the activities of a software engineering endeavour at an abstract level, capturing what needs to be done without defining or constraining how it is done. Activity Spaces take Alphas as input to the work. When the work is concluded the Alphas are updated and hence their states may have changed.

The *Practice layer* contains the constructs *Practice*, *Activity*, *Activity Manifest*, *Work Product*, *Alpha Manifest* and *Pattern*. A *Practice* is a general, repeatable approach to doing something with a specific purpose in mind, providing a systematic and teachable way of addressing a particular aspect of the work at hand. An *Activity*

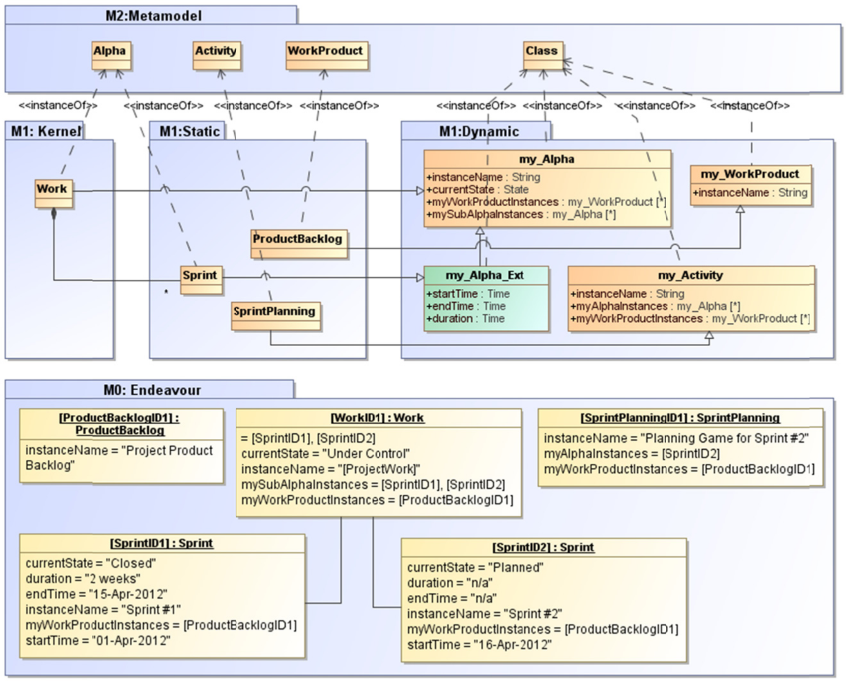
[3] In the Essence specification, the language is actually structured as four layers, as the Practice layer is divided into two: for *simple* and *advanced* practices.

defines one or more kinds of work and gives guidance on how to perform these. An *Activity Manifest* binds a collection of activities to an activity space. A *Work Product* is an artefact of value and relevance for a software engineering endeavour. An *Alpha Manifest* binds a collection of work products to an alpha. A *Pattern* is a general mechanism for defining a structure in a practice. The practice may also specify additional alphas or extend existing alphas with sub-alphas.

The Method layer contains the constructs *Method* and *Library*. A *Method* describes how an endeavour is run. A *Library* includes a collection of practices and methods.

3.2 Dynamic Semantics – Enactment of Software Engineering Methods

In contrast to the static semantics of the language, the dynamic semantics do not specify a metamodel at the M2 layer in the MOF architecture (see Fig. 1), but rather a model at the M1 layer. This model defines abstract superclasses that contain properties for the "run-time" occurrences during the endeavour, i.e., at the M0 layer according to the MOF architecture.



M2:Metamodel
Alpha
Activity
WorkProduct
Class
<<instanceOf>>
<<instanceOf>>
<<instanceOf>>
<<instanceOf>>
<<instanceOf>>
<<instanceOf>>
<<instanceOf>>
<<instanceOf>>
M1: Kernel
Work
M1:Static
ProductBacklog
Sprint
SprintPlanning
M1:Dynamic
my_Alpha
+instanceName : String
+currentState : State
+myWorkProductInstances : my_WorkProduct [*]
+mySubAlphaInstances : my_Alpha [*]
my_WorkProduct
+instanceName : String
my_Alpha_Ext
+startTime : Time
+endTime : Time
+duration : Time
my_Activity
+instanceName : String
+myAlphaInstances : my_Alpha [*]
+myWorkProductInstances : my_WorkProduct [*]
M0: Endeavour
[ProductBacklogID1] : ProductBacklog
instanceName = "Project Product Backlog"
[WorkID1] : Work
= [SprintID1], [SprintID2]
currentState = "Under Control"
instanceName = "[ProjectWork]"
mySubAlphaInstances = [SprintID1], [SprintID2]
myWorkProductInstances = [ProductBacklogID1]
[SprintPlanningID1] : SprintPlanning
instanceName = "Planning Game for Sprint #2"
myAlphaInstances = [SprintID2]
myWorkProductInstances = [ProductBacklogID1]
[SprintID1] : Sprint
currentState = "Closed"
duration = "2 weeks"
endTime = "15-Apr-2012"
instanceName = "Sprint #1"
myWorkProductInstances = [ProductBacklogID1]
startTime = "01-Apr-2012"
[SprintID2] : Sprint
currentState = "Planned"
duration = "n/a"
endTime = "n/a"
instanceName = "Sprint #2"
myWorkProductInstances = [ProductBacklogID1]
startTime = "16-Apr-2012"

Fig. 4. Language specification – dynamic semantics

Fig. 4 illustrates the dualism of the static semantics and the dynamic semantics domain. The *M2:Metamodel*, *M1:Kernel* and a set of *M1:Dynamic* classes are part of the standard proposed by the Essence specification, while the *M1:Static* and the highlighted *_Ext* class in *M1:Dynamic* are extensions added by a practitioner. The static semantic domain defines constructs such as *Alpha*, *Activity* and *Work Product* at M2 and class instances of these metaclasses at M1 such as *Sprint*, *Sprint Planning* and

Product Backlog. The dynamic semantic domain defines superclasses such as *my_Alpha*, *my_Activity* and *my_WorkProduct* for the respective *Alpha*, *Activity* and *Work Product* class instances from the static semantic domain.

The generalization mechanism ensures that the instances on the endeavour M0 layer contain slots that have been defined both from the static semantic and the dynamic semantic domains. Using this mechanism one could also define specific extensions, such as run-time endeavour properties that should only apply to certain subclasses of alphas, e.g., *my_Alpha_Ext* that contains properties such as *stateTime*, *endTime* and *duration* that should apply to *Sprint* instances that are sub-alphas of *Work* that is defined in the kernel. Moreover, dynamic semantics can be formalized with this mechanism by defining operations using the superclasses of the dynamic semantic domain.

The relationship between the static elements and their dynamic counterparts can be supported by tools and services that allow precise definition of these associations. Having such mechanisms in place will provide practitioners the ability to use the methods as described, but also refine and customize the methods according to the needs of the endeavour.

4 Illustrative Approach – Scrum

This section illustrates the SEMAT approach by modelling selected parts of the Scrum project management practice [9] and explores how the Scrum practice may be mapped to the Essence Kernel and Language. A particular *Method* can be composed from a set of well-defined *Practices* that plugs into the *Kernel*. Composition and usage of the method is supported by language's composition and dynamic semantics. The Scrum practice defines specific work products and activities that can be associated with the alphas and activity spaces defined by the kernel.

4.1 Creating the Scrum Practice and Method

In this paper we only present a subset of the Essence Kernel in order to illustrate the principles. Fig. 5 represents a subset of the alphas and activity spaces defined in the Essence specification [3]. The Kernel defines a small set of universal alpha elements such as *Requirements*, *Software System*, *Work* and *Team,* and their relationships, and activity spaces for the endeavour such as *Prepare to do the Work*, *Coordinate the Work*, *Support the Team*, *Track Progress* and *Stop the Work.*

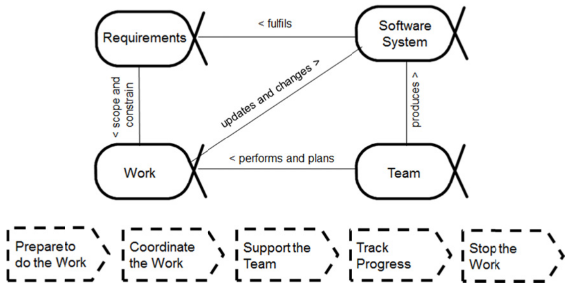
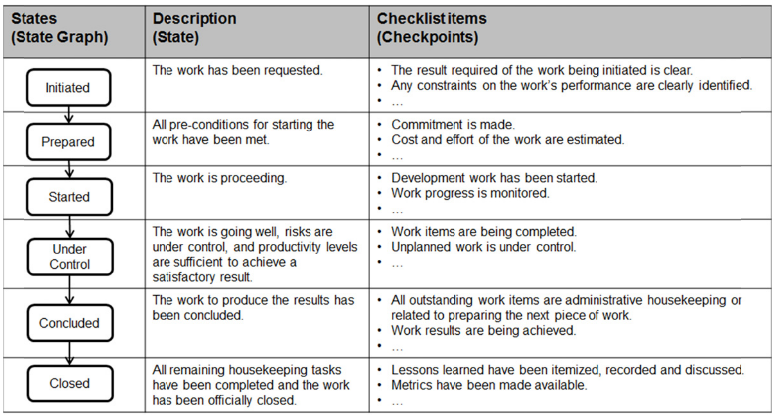


Fig. 5. Alpha and Activity Space subset defined by the Kernel

Since the paper focuses on the enactment part we only elaborate the details of the *Work* alpha. Fig. 6 shows the states of work, the corresponding state descriptions and the checkpoints associated with each state.



States (State Graph)	**Description (State)**	**Checklist items (Checkpoints)**
Initiated	The work has been requested.	• The result required of the work being initiated is clear. • Any constraints on the work's performance are clearly identified. • …
Prepared	All pre-conditions for starting the work have been met.	• Commitment is made. • Cost and effort of the work are estimated. • …
Started	The work is proceeding.	• Development work has been started. • Work progress is monitored. • …
Under Control	The work is going well, risks are under control, and productivity levels are sufficient to achieve a satisfactory result.	• Work items are being completed. • Unplanned work is under control. • …
Concluded	The work to produce the results has been concluded.	• All outstanding work items are administrative housekeeping or related to preparing the next piece of work. • Work results are being achieved. • …
Closed	All remaining housekeeping tasks have been completed and the work has been officially closed.	• Lessons learned have been itemized, recorded and discussed. • Metrics have been made available. • …

Fig. 6. States of work, state descriptions and associated checkpoints

A checkpoint describes the entry criteria for entering a specific state of the alpha and can be used to measure the progress of the alpha in the software endeavour. Checkpoints are typically expressed in natural language and provide a basis for generation of descriptive checklists items that are interpreted by the practitioners.

We extend the Work alpha for Scrum (see left side of Fig. 7). The Work alpha is typically used for the duration of a development project that may cover a number of sprints. Thus we define a new sub-alpha called Sprint. The Sprint Backlog is modelled as a work product that is associated with the Sprint sub-alpha. The Sprint has its own state graph (see middle part of Fig. 7). Scrum comes with its own specific set of rules that should be defined as part of the practice, whereas the Work state machine and its associated checkpoints are more general. The identified Scrum events may be mapped to corresponding activities and associated with the proper activity spaces (see right side of Fig. 7).

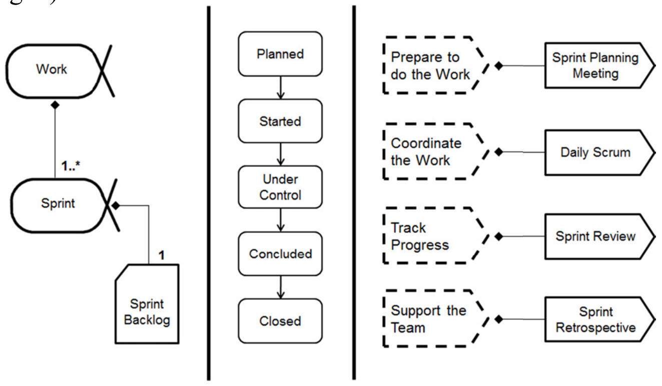
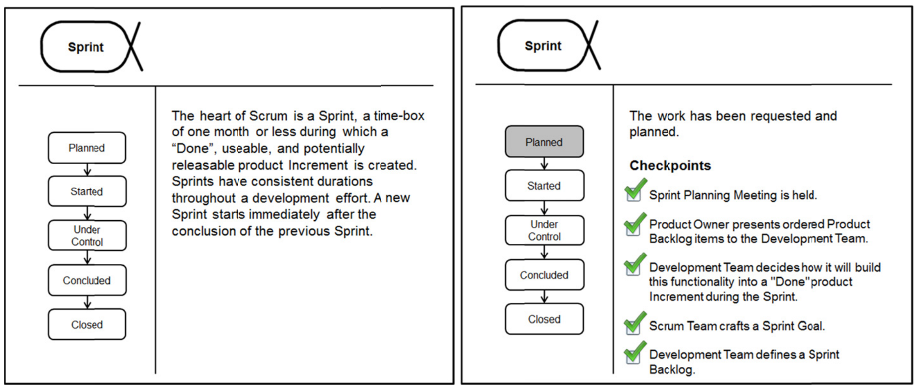


Fig. 7. The Sprint sub-alpha extends the Work alpha (left), the states of the Sprint sub-alpha (middle) and mapping of Scrum activities to the Activity Spaces of the Kernel (right)

4.2 Enacting the Scrum Method

The enactment support in the Essence language recognizes that software development is a creative process and most of the progress within the software development endeavour is done by human agents [10]. Thus, process enactment of software engineering methods and practices must acknowledge the human knowledge worker and provide means of monitoring and progressing the software endeavour through human agents foremost and automation secondly. In enactment, a team of practitioners collaborates on decision-making, planning and execution. Enactment may be partially supported by method repositories and process engines that are linked to tools such as project management and issue tracking systems.

The concrete syntax of the language has been designed so that the usage of the composed practices, i.e., the method, should be easy for the practitioners. For this purpose, the concept of *Alpha state cards* is introduced. Fig. 8 below shows the state card for the Sprint alpha at the *initial* state (left) and in the *Planned* state (right). These cards can be used for reading and understanding the practice, and also how to progress the states of the Sprint according to the checklist defined. This requires that all checkpoints are ticked off.



The heart of Scrum is a Sprint, a time-box of one month or less during which a "Done", useable, and potentially releasable product Increment is created. Sprints have consistent durations throughout a development effort. A new Sprint starts immediately after the conclusion of the previous Sprint.

The work has been requested and planned.

Checkpoints

✓ Sprint Planning Meeting is held.

✓ Product Owner presents ordered Product Backlog items to the Development Team.

✓ Development Team decides how it will build this functionality into a "Done" product Increment during the Sprint.

✓ Scrum Team crafts a Sprint Goal.

✓ Development Team defines a Sprint Backlog.

Fig. 8. Sprint state card (initial state and planned state)

Using the checkpoints of the Alphas it is at the discretion of the team to decide when a state change has occurred. These state cards may also have different views so that instead of the checklist items one could get a list of activities to do in order to progress from one state to another, or which work products to produce. The language supports the definition of different views suitable for different kinds of practitioners.

5 Related Work and Discussions

One main criticism of SPEM 2.0 is the lack of enactment support [8, 11]. Enacting SPEM processes are typically done through mapping, e.g., (1) mapping processes into project plans and enacting these with project planning and enactment systems or (2) mapping processes to a business flow or execution language and executing this with a workflow engine [1]. Designing native dynamic semantics into the language is arguably one of the main requirements that will require a redesign of the SPEM architecture.

The Situational Method Engineering (SME) community [12] has been working on related metamodelling approaches which has resulted in the ISO/IEC 24744 standard [13] that provides an alternative to the OMG SPEM 2.0 specification. Fig. 9 shows the key classes in the ISO 24744 metamodel. The metamodel can be seen as a dual metamodel that defines *Methodology elements* and *Endeavour elements* that are linked together through the concept of powertypes [14]. An explanation of powertypes to model software development methods is described in [11]. There are two types of methodology element classes, namely *Template* and *Resource*. Endeavour elements which includes *Stage*, *WorkUnit*, *WorkProduct* and *Producer,* always have a corresponding methodology element *...Kind* type represented using a powertype relationship. Resource constructs, which include *Language*, *Notation*, *Constraint*, *Outcome* and *Guideline*, represents elements that are directly used without requiring the need for instantiation at the endeavour level.

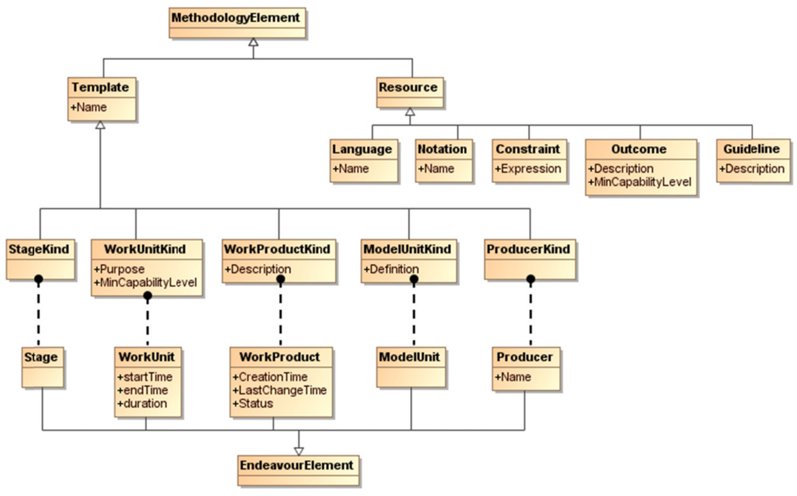


Fig. 9. Overview of the ISO 24744 metamodel

Since advanced metamodelling constructs such as powertypes are not compatible with MOF the ISO 24744 standard uses a different metamodel architecture [11, 13] to support dynamic semantics. This metamodelling issue is not present in SPEM as it leaves out the endeavour M0 layer. In the SEMAT approach we propose a solution that is MOF compliant and define two separate domains, the metamodel of the static semantics and the model of the dynamic semantics. We compose these two domains at the M1 layer of the MOF architecture (see Fig. 4) to support both method and endeavour properties at the endeavour M0 layer. This is very similar to the concept of using powertypes but maintains compatibility with the MOF architecture. Fig. 10 illustrates the difference between the two approaches. The *WorkProduct* and *WorkProductKind* metaclasses from the ISO/IEC 24744 standard are equivalent to the *WorkProduct* (at M2 layer) and *my_WorkProduct* (at M1 layer) in the Essence specification. Standardised endeavour properties on *WorkProduct* (ISO 24744) can be represented as properties on *my_WorkProduct* (Essence). Additional user-specific extensions and properties such as *version* on ProductBacklog (ISO 24744) can be done through extensions of the domain classes as shown using *my_WorkProduct_Ext* (Essence). The resulting slots in the *ProductBacklog instance* at M0 are the same in both approaches.

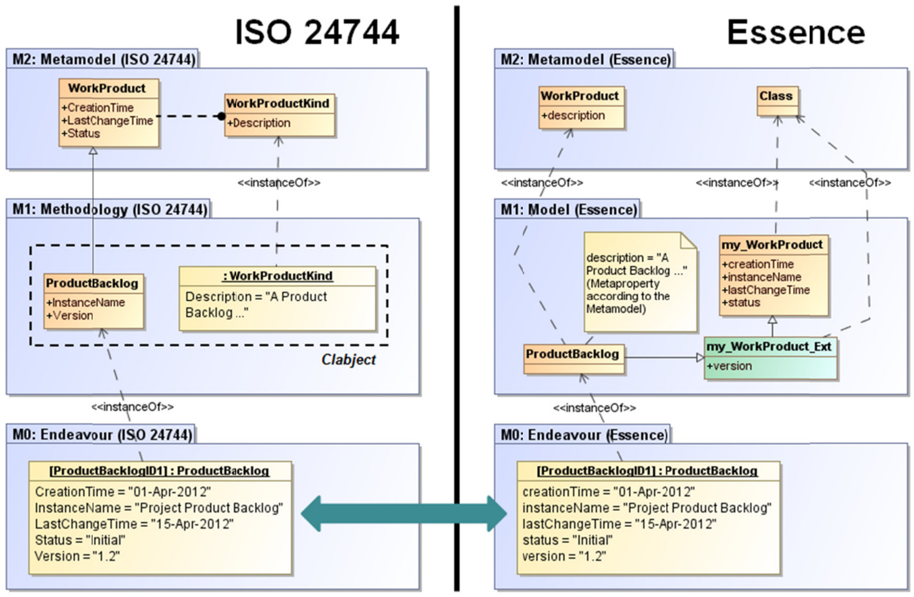


Fig. 10. Comparison of the ISO 24744 and Essence approaches

6 Conclusions and Future Work

In this paper we have presented the main ideas of the Essence language submitted by SEMAT as a response to the OMG RFP "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" [4]. This paper has explained the static and dynamic semantics of the Essence language and illustrated the enactment support by modelling selected parts of the Scrum project management practice.

The proposed language contains a static semantics part defined as a metamodel on the M2 layer in the MOF architecture and a dynamic semantics part defined as a model on the M1 layer in the MOF architecture. The proposed language constructs are intended to simplify software engineering method modelling adaptation for practitioners. The concepts of *Kernel*, *Alpha*, *Activity Space* and *Practice* are introduced as examples of such required language constructs. The kernel elements form a domain model for software engineering and provide a standardised baseline for describing current and future practices that afterwards are adapted and customized for usage, i.e., enactment, in software endeavours. Our discussion has tried to clarify the differences and similarities between the Essence approach and other related standards such as the ISO/IEC 24744.

We are currently progressing this work in the context of SEMAT where we are preparing a revised submission to the OMG RFP. The aim is to take advantage of recent development and experiences from software engineering and method engineering communities in order to give the best possible direct support towards software practitioners.

Acknowledgments. This research was co-funded by the European Union in the frame of the NEFFICS project (FP7-ICT-258076) (www.neffics.eu) and the REMICS project (FP7-ICT-257793) (www.remics.eu), and SINTEF in the frame of the SiSaS project (sisas.modelbased.net). The authors acknowledge and thank collaboration with colleagues within SEMAT (www.semat.org) for foundational input and feedback.

References

[1] OMG, "Software & Systems Process Engineering Meta-Model Specification, Version 2.0", Object Management Group (OMG), Document formal/2008-04-01, April 2008. http://www.omg.org/spec/SPEM/2.0/PDF/

[2] R. Bendraou, B. Combemale, X. Cregut, and M.-P. Gervais, "Definition of an Executable SPEM 2.0", in 14th Asia-Pacific Software Engineering Conference (APSEC '07). Nagoya, Japan, 2007, pp. 390-397. http://dx.doi.org/10.1109/APSEC.2007.38

[3] OMG, "Essence - Kernel and Language for Software Engineering Methods, Initial Submission - Version 1.0", Object Management Group (OMG), OMG Document ad/12-02-04, 20 February 2012. http://www.omg.org/cgi-bin/doc?ad/12-02-04

[4] OMG, "A Foundation for the Agile Creation and Enactment of Software Engineering Methods RFP", Object Management Group (OMG), OMG Document ad/2011-06-26, 23 June 2011. http://www.omg.org/members/cgi-bin/doc?ad/11-06-26.pdf

[5] OMG, "Meta Object Facility (MOF) Core Specification, Version 2.0", Object Management Group (OMG), Document formal/06-01-01, January 2006. http://www.omg.org/spec/MOF/2.0/PDF/

[6] OMG, "MDA Guide Version 1.0.1", Object Management Group (OMG), Document omg/03-06-01, June 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf

[7] I. Jacobson, P. W. Ng, and I. Spence, "Enough of Processes - Lets do Practices", Journal of Object Technology, vol. 6, no. 6, pp. 41-66, 2007. http://www.jot.fm/issues/issue_2007_07/column5

[8] C. Gonzalez-Perez and B. Henderson-Sellers, "Metamodelling for Software Engineering", John Wiley & Sons, Ltd, 2008, ISBN 978-0-470-03036-3.

[9] K. Schwaber and J. Sutherland, "The Scrum Guide", Scrum.org, October 2011. http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf

[10] P. Feiler and W. Humphrey, "Software Process Development and Enactment: Concepts and Definitions", Software Engineering Institute, Technical report CMU/SEI-92-TR-004, September 1992.

[11] B. Henderson-Sellers and C. Gonzalez-Perez, "The Rationale of Powertype-based Metamodelling to Underpin Software Development Methodologies", in Proc. of the 2nd Asia-Pacific conference on Conceptual modelling - Volume 43, 2005, ACM.

[12] M. A. Jeusfeld, M. Jarke, and J. Mylopoulos, "Metamodeling for Method Engineering", The MIT Press, 2009.

[13] ISO/IEC, "Software Engineering – Metamodel for Development Methodologies", International Organization for Standardisation (ISO), ISO/IEC 24744, 15 February 2007.

[14] J. Odell, "Power Types", Journal of Object-Oriented Programming, vol. 7, no. 2, pp. 8-12, 1994.

Model-based Product and Process Integration for Enhanced Collaboration during Mechatronic Design Processes

Holger Seemüller[1] and Holger Voos[2]

[1] University of Applied Sciences Ravensburg-Weingarten
D-88241 Weingarten, Germany
seemueller@hs-weingarten.de
[2] University of Luxembourg
L-1359 Luxembourg
voos@uni.lu

Abstract. The collaborative design of mechatronic systems is still a challenging task as different engineering disciplines have to be considered and coordinated during the design process. Here, an independent and isolated view on discipline-specific tools, model data and activities is not appropriate. So, the integration of these aspects for improved collaboration is still a remaining task in industry and research. Former research have already developed first solutions each directing into the integration of distinct aspects among the involved disciplines. This paper claims that a comprehensive view on the different aspects can bring significant benefits for the design of mechatronic systems. In detail, it presents an approach which combines interdisciplinary system modeling with design activity management by describing and integrating these aspects on metalevel. This integration leads automatically to enhanced possibilities for design activity coordination and monitoring.

1 Introduction

The mechatronics engineering discipline is one of the main innovation leader in industry nowadays. It offers possibilities to achieve highly innovative products offering high performance whilst getting cost related aspects under control [1]. The design of mechatronic systems describes the synergetic creation and integration of mechanical engineering, electrical engineering and information technology for the specification and description of any kind of physical products [2]. However, mechatronic design is a manifold and challenging task as complexity of such systems is increasing and the beneficial application of synergetic interdisciplinary effects has to be considered. This implicates an amount of challenges which have to be met while designing a mechatronic system. The structured handling of the increasing complexity of mechatronic systems is one main challenge. For compliance with market demands, systems have to offer a wider range of functionality with increasing performance. This can only be reached by the combination of mechanical, electrical and control aspects into more and more integrated devices where synergies can be exploited. This causes an increasing amount of mutual interdependencies between different parts of the system. So, mechanical space limitations resulting from the design of a chassis has direct impacts on the assembly of electric components. However, nowadays distributed development environments, discipline-specific terminologies and diverse modeling notations hinder direct communication and agreements. Nevertheless, the needs and requirements of each engineering discipline have to be considered equally. Therefore, systematic approaches for collaborative design even in the early phases of the design process are needed and must be supported methodologically.

The presence and collaboration of diverse engineering disciplines also leads to novel challenges in the management and coordination of the engineering design process. Due to pressure for shorter time to market and so shorter development time, an efficient coordination is crucial in nowadays design processes. It is obvious that a reduction of development time can be achieved through concurrent arrangement of design activities. However, real life projects come along with mutual dependencies between different design steps, especially in cases of strong discipline-spanning cooperation. This inhibits independent and parallel activity processing and requires crucial planning and coordination efforts.

The growing complexity of mechatronic systems as well as the extensive amount of design activities within several engineering domains results in an increasing amount of design model data produced by diverse design tools. However, complex interdependencies between these data exist and have to be considered to allow engineers to work with a common set of information. This problem even exacerbates with enhanced degree of integration and system complexity. The resulting need for consistency management and model integration is therefore evident for successful product design.

This paper intends to make a contribution to the mentioned problems and presents a possibility for collaborative coordination of participated disciplines during mechatronic design processes. This aims for an improved, collaborative system design and an optimized way for design activity coordination. Particularly, the paper claims that a comprehensive view on these aspects will offer extended possibilities for an improved collaboration between involved engineers. Considerations about the relationship between product and process and the different kind of design model dependencies which can be deduced are the foundation of the approach. Metamodels will be used to describe the syntax of a mechatronic system, related design activities and their relations and possibilities will be presented, how this approach can make a contribution to mechatronic product design.

The remainder of the paper is structured as followed: Section 2 shows the current state-of-the-art of mechatronic system modeling and design process management and shows remaining drawbacks. Section 3 presents the approach of an integrated combination of system and process modeling. Section 4 provides a discussion about the approach. Finally, section 5 concludes the paper and gives an outlook for future work.

2 Related Work

The optimization of mechatronic designs has been the topic of a lot of research and industrial projects. However, this field of research is wide and intensive work has already been done in diverse areas. This section details current problems and reviews the related works on which the proposed approach is based.

2.1 System Modeling

The rising complexity of mechatronic systems and their higher degree of discipline-spanning integration comes along with the need for extended systematic approaches and methods. These have to address the needs of all involved engineering domains equally even in early phases of the mechatronic design process. This issue is also discussed in the VDI guideline 2206 [3] which suggests the collaborative creation of an interdisciplinary principle solution during the system design phase. As the guideline lacks of a concrete realization for this principle solution, several attempts have been developed which can be applied for the realization. Whereas [4] proposes with Mechatronic UML a UML profile that extends the common UML2 standard for mechatronic needs, [5] developed a novel specification notation for the description of the principle solution. Modelica [6] offers possibilities for component-based modeling of complex systems with extensive simulation capabilities. Furthermore, SysML is an approach for interdisciplinary system modeling and defined as "a general-purpose graphical modeling language for specifying, analyzing, designing and verifying complex systems that may include hardware, software, information, personnel, procedures and facilities" [7]. As it is defined as a standardized profile for UML2, tool support is widely given and it gains more and more acceptance in industry. Several research projects have already shown the general feasibility of this modeling notation for the creation of an interdisciplinary system model [8–10]. Interviews with industrial partners have shown that the idea of an interdisciplinary system model has not yet been fully accepted in industry. As the creation of an additional model causes further efforts, the concrete benefits must be demonstrated in advance. Furthermore, the introduction of a collaboratively created system model requests novel methodologies and roles within the project. Finally, the continuous usage of a system model during the overall development process of the mechatronic system has to be shown precisely. This is still an ongoing research topic in industry.

2.2 Engineering Design Process Management

The efficient organization and management of engineering design processes is a crucial factor for successful and cost efficient product development. As a consequence, several process models have been developed [3, 11]. They offer a framework to which engineering activities can be aligned and present common methodologies for problem solving and designing engineering systems. However, they are very abstract and give only few information to assist engineers in their daily activities [12].

From business process management (BPM) the idea of modeling a workflow of activities in a formal and computer interpretable way arose. Notations, such as BPMN [13], have been developed to support process designers with standardized modeling languages and, at the same time, workflow management systems opened the possibility for process enactment. Also notations specialized to engineering activities appeared, such as SPEM [14].

However, the established workflow modeling techniques from BPM cannot be applied directly on the modeling of engineering design activities. Deterministic and sequential business process chains are commonly performed several times in the same manner. In contrast, design processes are knowledge intensive and characterized by their weakly-structured and nondeterministic activity order. Additionally, the long duration increases the degree of uncertainty and hinders a complete planning in advance. [12] and [15] have identified that the process depends very much on the product itself and already produced design artifacts. So, results of former design steps influence the further process heavily and lead to an evolutionary process model. Engineering design activities are additionally characterized by very complex interdependencies as engineers have to exchange informations and activities may depend on the results of others. Nevertheless, a maximum degree of concurrency is desirable in order to optimize the process. In industry it is a common practice to work with assumptions to avoid unnecessary delays especially in early phases of the design process. However, this approach leads to high consistency problems, as refined parameters need to be propagated reliably. One approach for successful planning and modeling of design processes is the determination of the right level of granularity of the considered activities. So, Lindemann [16] identifies four levels of hierarchy, separated from micro to macro logic, which can be used for structured planning. This categorization as well the fact that the system and process depend on each other will also be used in the current approach.

2.3 Model Integration

With rising complexity of mechatronic systems, also the amount of model data produced during system design is increasing extensively. These models describe the system from different views on several levels of abstraction, such as models for requirements analysis, CAD models for geometric information or detailed mathematical models for simulation. Whereas the storage and retrieval of physical data can already be managed by established product data management systems, the management of logical dependencies and the consistency between model data is still a challenge. Several approaches exist trying to increase the interoperability and consistency between design tools. Especially the idea of model-based systems engineering lead to novel approaches as design models during the systems engineering process base on formal models. This increases determinism and understandability [17] and offers automated processing by computers. The Modelbus [18] project connects existing design tools with a common bus where data can be exchanged flexibly. So tool chains can be created which automatically transfer model data between the involved tools. Modelisar [19] is a project which offers standardized interfaces between existing design tools which can also be used to exchange model data and create tool chains. Other approaches use the idea of a central system model where important information from several disciplines and design tools are stored and related to design tool specific models [20, 21]. Also the AGENTES project [22] follows the idea of a common system model and applies a multi-agent system for change propagation from system model to design tools and back.

3 Integration of System and Design Activity Modeling

This research continues with ongoing trends within the research field of collaborative mechatronic design. Concretely, it deals with the planning and modelling of design activities to enable concurrent enactment. However, waiting for availability of information hinders the parallel processing and the usage of assumptions leads to intense consistency issues. This approach is founded on the fact that mechatronic design activity planning bases intensively on the characteristics and the structure of the product to be developed. By using information and dependencies of the system, design activities can be coordinated and synchronized. We argue that the consideration of logical dependencies (originating from the product) together with chronological dependencies (originating from the activity sequence) enhances the concurrent processing of design activities. Supported by an IT tool, engineers can be guided through the design process in a pro-active way and inconsistencies can be detected and propagated.

The conceptual background of the approach will be introduced by two technology neutral metamodels that will narrow the terms down to our scope and demonstrate the interrelation between the domains. The first metamodel represents the abstract syntax of a mechatronic product whereas the second metamodel will demonstrate the concepts of mechatronic design activities. These metamodels have been built based on common state-of-the-art insights about mechatronic systems and borrows concepts from common modeling notations. In that way, we make the approach portable to established notations. The mutual integration between product and design activities is represented by cross references between the metamodels. Further, a way will be demonstrated that offers a possible realization of the concepts with common and well-known modeling notations.

The remainder of this chapter shows the abstract syntax and composition of a mechatronic product as well the syntax of mechatronic design activities. Next, a possible application of these concepts is shown. Finally, tool support of the approach is sketched.

3.1 Product Structure

The abstract syntax and composition of a mechatronic product have been defined formally by a metamodel. The design considerations of the metamodel were influenced by current insights about mechatronic systems and their structural composition [22]. The following assumptions were taken and considered:

- An mechatronic system can be described in terms of its requirements, its functional decomposition as well as the product structure. Tracing information can exist to relate information between these different levels of abstraction.
- A mechatronic system is assembled of mechatronic subsystems and components which themselves can be decomposed into an hierarchical structure of mechatronic subcomponents.
- A mechatronic component is an interdisciplinary element which stores information and parameters from diverse engineering disciplines, e.g. from mechanics and electronics [23].
- Mechatronic components are connected via uni- or bidirectional interfaces which offer or use an interface definition
- Dependencies and mathematical relationships between parameters from different levels of abstraction, different components as well different disciplines can exist.

Figure 1 shows an overview of the developed metamodel. The root element of the metamodel is *EngSystem*, which represents the mechatronic system. This system is defined on several levels of abstraction, namely: (1) as the set of requirements (*Requirement*) the system has to fulfill; (2) the set of functions (*Function*) that refine the requirements; and (3) the set of concrete components (*Component*) that realize the functions. A *Requirement* is specified by a textual description. Furthermore the metaclass *Property* can be used to express the requirement more formally with a set of properties. Each requirement can be refined by a function hierarchy represented as *Function* metaclasses. The idea of functional decomposition is common in mechatronic engineering and offers a means to describe a system in a solution independent way [24]. The concrete realization is expressed as a hierarchy of the *Component*s and decomposes the *EngSystem*. The *EngSystem* as well as each *Component* is a subclass of *StructuralElement* and so owns a set of *Property*, *Interface* and *Constraint* metaclasses: A *Property* can be used to express any discrete parameter by stating its Name, the Unit and the respective Value of the Property. Additionally, a PropertyState can be attached which holds information about the maturity of the value. Theses states can be used to support evolutionary development of the product. An *Interface* serves as a connection point between distinct *StructuralElement* instances. The connection between two interfaces is realized via a common *InterfaceDefinition*. It describes an interface in terms of its name and id as well as a set of properties which are again expressed with the metaclass *Property*. An *InterfaceDefinition* is offered by exactly one *Interface* and can be used by several other *Interface*s. The *Constraint* metaclass allows an engineer to express dependencies between any level of hierarchy within the system model. Therefore, a mapping between a *Parameter* of the constraint and an arbitrary *Property* can be defined and the relation between the parameters can be expressed with mathematical expressions. In that way, dependencies within the system can be expressed formally and consistently.

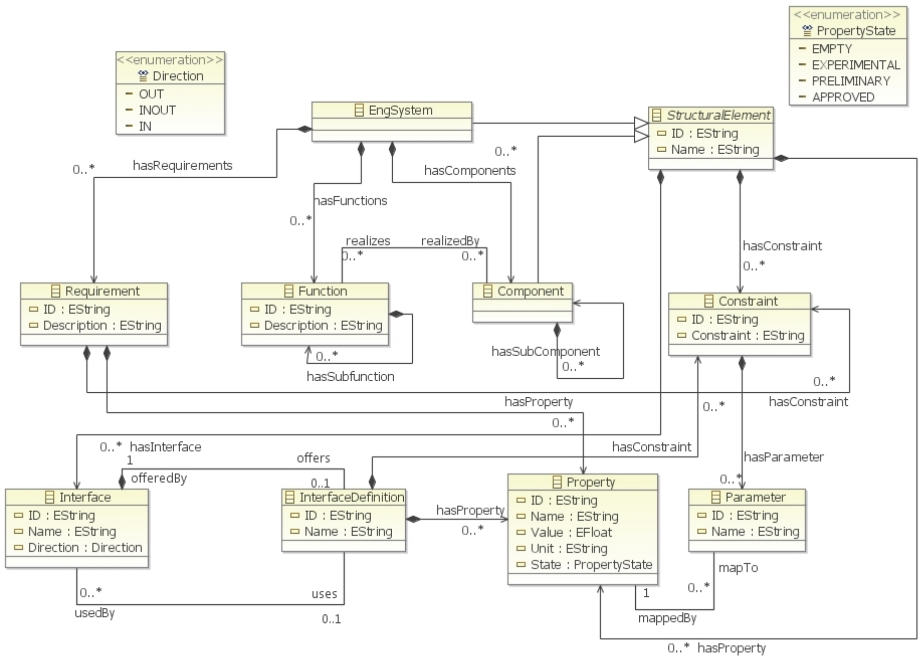


Fig. 1. The engineering metamodel

3.2 Process Modeling

The description of the flow of activities in a formal and computer interpretable way has shown its effectiveness already in business process management. It increases transparency between involved engineers, improves coordination and assistance, allows better planning and management capabilities and leads finally to shorter development cycles [25]. However, as discussed in section 2, engineering design processes have special characteristics that make traditional process modeling not directly applicable to them. According to Lindemann [16] engineering processes can be described on different levels of abstraction. This paper focuses on the operational level "with interrelated activities" [12]. These are characterized by their strong dependencies on mutual exchanged information but weak possibilities for a predictable order. This is the reason why the metamodel contains possibilities to express pre- and postconditions rather than explicit control flow elements.

Principally, it is assumed that engineering design activities consume information about the mechatronic system, e.g. requirements, functionalities or important parameters of previous design decisions. These knowledge is used to produce new design knowledge which can be reflected to the mechatronic product and influences further design steps. The reflection of information may lead to iterations or repetitions of former activities. So, an aligned and synchronous planning of design activities parallel to the evolution of the product is meaningful. This paper performs the integration on metamodel level by adding cross references between the activity and the product metamodel.

For the design of the metamodel, the following assumptions are taken:

– An activity describes a work package with defined input and output information for a single or a group of engineers.
– Activities are coupled via their interdependencies of produced and consumed information.
– The inputs of an activity are specified by a textual description of the work package, related requirements and functions, existing work products such as previously created simulation models as well as a context within the system model. Depending on the level of maturity of the system, some inputs may be optional.
– The output is again a set of properties. These properties can refine existing properties or give new properties an initial value.
– An activity produces or refines work products such as e.g. a simulation model of a specific tool.
– Pre- and post-conditions can specify extended dependencies between activities.

These assumptions lead to the metamodel presented in figure 2. As the process in this paper deals with operational level, the process model shall be declared as an activity model and be represented via the metaclass *ActivityModel*. A model contains a set of independent activities (*Activity*) which are specified by a name, a textual description as well as an optional deadline. One ore more *Roles* which are instances of the metaclass *Role* have to be assigned to an activity indicating the responsibilities. A *Role* is part of a *RoleCollection* storing available roles of the project. Similarly, a *Tool*, which is again part of a *ToolCollection*, can be attached to an *Activity*. Furthermore, produced and consumed *WorkProducts* can be specified by their filenames. With subclasses of the abstract metaclass *Condition*, additional activity flow information can be attached to an *Activity*.
The integration with the system metamodel is realized with the metaclasses *StructuralElement*, *Property*, *Function* and *Requirement*. These classes origin from the system metamodel and specify directly the context as well as attached requirements and functions which deliver additional information for the engineer. Further, the abstract metaclass *ActivityProperty* reflects a concrete *Property* of the system model and offers as optional alias name. Subclasses of an *ActivityProperty* are used to specify input and output properties for an activity.

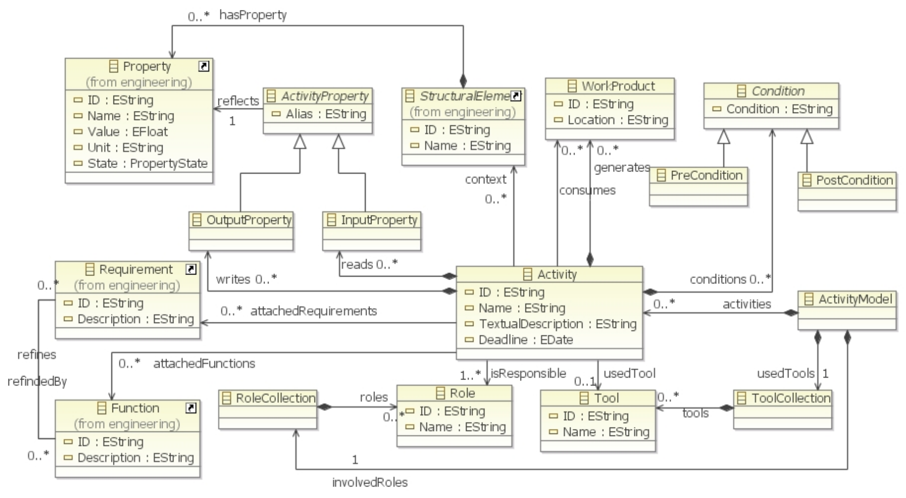


Fig. 2. The process metamodel

3.3 Application

The abstract metamodels can be applied on real life projects in several ways. Several modeling notations have been developed which offer possibilities for the creation of a system model according to the presented metamodels (see section 2). For broader acceptance and generic applicability, some notations have intentionally been designed without clearly defined semantics. So, the project-specific definition of the semantics of syntax elements have been evolved to a common practice at the beginning of a project. By having the structure of a mechatronic product defined formally, the approach can be realized by any modeling notation which allows the modeling of system in these terms. The system modeling language SysML is an example of such a language. In [26] a modeling approach with SysML which can be aligned to the presented metamodel has already been shown. Hence, the authors used the block definition diagram to model the hierarchical structure of the system with the SysML element <block>. The single components have been connected within an internal block diagram and constraints between parameters of the system have been modeled with <constraint>-elements within a parametric diagram. So, that model is expressive enough to describe a system in the presented syntax. Nevertheless, other modeling approaches - also with SysML - are suitable to model such a representation, too.

The creation of an activity model is more complex as the cross references to the system model have to be implemented. Actually, in the domain of engineering processes, no standard modeling notation have been revealed so far. Even proprietary models in Microsoft Excel or Powerpoint are still a common means to express design activities in industry. Nevertheless, the approach can be applied as long as the notation is expressive enough to obtain a model that satisfies the syntax of the metamodel. The implementation of the references to the system model can be realized in several ways. So, for prototypical realization, even referencing a property by its string identifier is a valid option. Extended possibilities for cross-referencing include the adaption of the modeling tool for direct access of the system model elements. This extension is currently under development.

3.4 Tool support

Necessarily, the presented approach has to be supported by an IT tool for several reasons: First, change impacts due to modeled dependencies and constraints within the product can only be evaluated reliably by an application that owns this knowledge. Second, the modeled activities need to be enacted in a way similar to traditional business process workflow management systems. Third, timing information of enacted activities have to be traced to take advantage of the chronological dependencies for extended change impact analysis.

The initial architecture of an IT tool has already been presented in [26]. The paper at hand shows the configuration of that system. The presented metamodels are used as a data structure that serves as the interface to the proposed framework. Rather than bounding an existing modeling notation to the semantics of our need, we define a metamodel that specifies our syntax of a mechatronic system as a distinct metamodel and offer transformation possibilities from existing notations. So, established modeling notations can still be used and the approach stays technology neutral.

4 Discussion

The presented approach offers possibilities for activity planning and modeling based on information of the system model. The combination of these two modeling levels possesses two kinds of dependencies and tracing information that can be used for consistency management: First, modeled constraints and associations on system model level define system internal dependencies centrally in one model. These information can be used for change impact analysis after a parameter has been refined by an engineer. So, other elements can be updated based on the new information and affected engineers can be informed. The second kind of dependency are chronological dependencies that come along with the enactment of activities. As each property can serve as an input or be the output of design activities, these properties are related in a chronological order. This enables an extended change impact analysis, as affected design activities can be considered and, if necessary, be restarted with updated information. Further, the coupling of activities by their input/output relationship offers dynamic ordering and automatic enactment of activities by availability of necessary information.

Furthermore, activity modeling bridges the gap between information modeled in the system model and corresponding information with discipline-specific engineering tools. As an activity possesses knowledge about input properties, the used tool as well as the produced work product, implicit mappings between system model properties and tool parameters can be created and used.

Finally, as our description of the system and the activities base on a formal metamodel, they can be used as the input for an IT system that is able to enact the activity model.

5 Conclusion and future work

This paper presented an integrated approach for system and activity modeling in the context of mechatronic design processes. Two metamodels that describe a mechatronic system and related design activities have been developed. It has been highlighted that the combined usage can enhance the coordination of engineers in distributed environments and improves collaboration and consistency management. Furthermore, model data consistency can be ensured by the combined consideration of system model and chronological dependencies.

Future work will deal with the realization of the approach. For beneficial application, IT-based tool support is necessary and the mentioned prototype [26] has to be refined by integrating the demonstrated metamodels and concepts. Next, transformations from established system and process modeling notations to the metamodels will be developed. Furthermore, process management has to be integrated into the existing multi-agent system in a way we already described in [26]. So, chronological dependencies can also be considered. Additionally, specialized tool adapters will be needed that serve as interfaces to discipline specific modeling and simulation tools. So, automatic mappings between properties of the system model and the parameters within an IT tool are possible to automate model consistency. In parallel, the metamodels will be revised and use cases will show the applicability in real life projects.

References

1. Tomizuka, M.: Mechatronics: from the 20th to 21st century. Control Engineering Practice **10**(8) (2002) 877–886
2. Möhringer, S., Stetter, R.: A research framework for mechatronic design. In: International Design Conference - Design 2010, Dubrovnik, Croatia (May 2010)
3. VDI, V.D.I.: Entwicklungsmethodik für mechatronische Systeme. VDI Richtlinie 2206. Beuth Verlag (2004)
4. Schäfer, W., Wehrheim, H.: Model-driven development with Mechatronic UML. In: Graph transformations and model-driven engineering. Springer-Verlag, Berlin, Heidelberg (2010) 533–554
5. Frank, U.: Spezifikationstechnik zur Beschreibung der Prinziplsung selbstoptimierender Systeme. PhD thesis, Universität Paderborn, Heinz Nixdorf Institut, Rechnerintegrierte Produktion (2006)
6. Tiller, M.: Introduction to Physical Modeling with Modelica. 1 edn. Springer US (May 2001)

7. OMG: OMG Systems Modeling Language (OMG SysML) 1.2. available at http://www.omgsysml.org.
8. Follmer, M., Hehenberger, P., Punz, S., Zeman, K.: Using sysml in the product development process of mechatronic systems. In: International Design Conference, Dubrovnik, Croatia (May 2010)
9. Wlkl, S., Shea, K.: A computational product model for conceptual design using sysml. In: ASME 2009 International Design Engineering Technical Conferences Computers and Information in Engineering Conference, San Diego, USA (August 2009)
10. Rosenberg, D., Mancarella, S.: Embedded systems development using sysml. Ieee Transactions On Audio Speech And Language Processing **17**(1) (2009) C1–C4
11. Pahl, G., Beitz, W., Feldhusen, J., Grote, K.H.: Engineering Design: A Systematic Approach. 3rd ed. edn. Springer London, London (2007)
12. Roelofsen, J., Lauer, W., Lindemann, U.: Product model driven development. In: 14th European Concurrent Engineering Conference, Ghent (April 2007)
13. OMG: Business Process Model and Notation (BPMN)) 2.0. available at http://www.omg.org/spec/BPMN/2.0/.
14. OMG: Software & Systems Process Engineering Meta-Model Specification 1.2. (2008) available at http://http://www.omg.org/spec/SPEM/2.0.
15. Schwarz, J., Adameck, N., Frank, D., Siebert, R., Haasis, S.: The use of feature-based workflow techniques in automotive product development. In: 7th International Conference on Concurrent Enterprising, Bremen, Germany (June 2001)
16. Lindemann, U.: Methodische Entwicklung technischer Produkte: Methoden flexibel und situationsgerecht anwenden. Springer (2009)
17. Ogren, I.: On principles for model-based systems engineering. Systems Engineering Journal **3**(1) (2000) 38–49
18. Hein, C., Ritter, T., Wagner, M.: Model-Driven tool integration with ModelBus. In: Workshop Future Trends of Model-Driven Development. (2009)
19. Consortium, M.: Fmi. http://functional-mockup-interface.org/ accessed: 2011/11/23.
20. Thramboulidis, K.: The 3+1 sysml view-model in model integrated mechatronics. JSEA **3**(2) (2010) 109–118
21. Qamar, A., Wikander, J., During, C.: Designing mechatronic systems: A model-integration approach. In: 18th International Conference on Engineering Design, ICED11, Copenhagen, Denmark (August 2011)
22. Stetter, R., Voos, H.: Agentes - agent based engineering of mechatronic products. In: Proceedings of the 8th International Symposium on Tools and Methods of Competitive Engineering (TMCE) 2010, Ancona, Italy (April 2010)
23. Thramboulidis, K.: Challenges in the development of mechatronic systems: The mechatronic component. In: ETFA. (2008) 624–631
24. Lamm, J.G., Weilkiens, T.: Funktionale architekturen in sysml. In Maurer, M., Schulze, S.O., eds.: Tag des Systems Engineering 2010, Munich, Germany (November 2010)
25. Fricke, E., Negele, H., Schrepfer, L., Dick, A., Gebhard, B., Haertlein, N.: Modeling of concurrent engineering processes for integrated systems development. In: Digital Avionics Systems Conference, Bellevue, WA, USA (October 1998)
26. Stetter, R., Seemüller, H., Chami, M., Voos, H.: Interdisciplinary system model for agent-supported mechatronic design. In: 18th International Conference on Engineering Design, ICED11, Copenhagen, Denmark (August 2011)

Design Decisions for UML and MOF based Domain-specific Language Models: Some Lessons Learned*

Bernhard Hoisl[1,2], Stefan Sobernig[1], Sigrid Schefer-Wenzl[1,2], Mark Strembeck[1,2], and Anne Baumgrass[1,2]

[1] Institute for Information Systems, New Media Lab,
Vienna University of Economics and Business (WU Vienna)
[2] Secure Business Austria Research (SBA Research)
`{firstname.lastname}@wu.ac.at`

Abstract. In recent years, the development of domain-specific modeling languages (DSMLs) that are based on the MOF and/or UML has become a popular option in the model-driven development context. As a result, the model-driven software engineering community collected many design and implementation experiences. However, most research contributions on this topic do not aim at supporting the DSML development process as a repetitive decision-making process. In this paper, we document some of our experiences gathered from developing ten MOF/UML-based DSMLs and present our experiences in a reusable manner via decision templates. In particular, this paper focuses on design decisions for the initial phase of the DSML development process, i.e. the definition of the DSML's core language model.

Keywords: Domain-specific modeling, Domain-specific languages, Design decisions, UML, Model-driven development

1 Introduction

In model-driven development (MDD), a domain-specific modeling language (DSML) is a specialized modeling language tailored for a particular application domain (e.g., access control, backup policies, or system auditing) (see, e.g., [1,2,3,4]). Thus, a DSML's abstraction level, its expressiveness, and concrete syntax are customized for software developers and for experts in the DSML's application domain. Often DSMLs are developed based on the Unified Modeling Language (UML) [5]. The UML can leverage industry-grade tool support, scientific evaluations of its semantic foundations, and standardized modeling extensions (e.g., SoaML for service-oriented systems [6]). The UML benefits

*This work has partly been funded by the Austrian Research Promotion Agency (FFG) of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) through the Competence Centers for Excellent Technologies (COMET K1) initiative and the FIT-IT program.

from its organizational maintenance through the Object Management Group (OMG) and builds upon a standardized metamodel: the Meta Object Facility (MOF) [7]. With this, the MOF and the UML provide a rich DSML development toolkit.

In recent years, a number of contributions discussed the development of domain-specific languages (DSLs). Examples include empirical research evidence (e.g., case study research [8,9,10]), DSL development processes [1], development guidelines and patterns [2,3,4,11], or selected facets of UML-based DSMLs [12,13]. Despite the availability of such sources of design knowledge, most contributions fall short in one or several respects: Many experiences lack empirically gathered evidence (e.g., an explicitly documented research design). Many are not specifically tailored toward DSMLs in general, or MOF/UML-based DSMLs in particular, but rather toward textual DSLs. Others reflect design knowledge which is specific to a particular toolkit (e.g., the Eclipse Modeling Framework, EMF). Our work complements the experiences mentioned above by providing reusable design knowledge for designing the core language model of MOF/UML-based DSMLs; i.e. specific options, consequences, and dependencies of decisions in this particular phase of DSML development.

The purpose of this paper is to present our experiences, lessons learned, and some of the challenges we faced while developing ten MOF/UML-based DSMLs over the last years. For an overview of these projects see Table 1 (P1–P10). From these experiences, we extracted two decision points with corresponding decision options for the initial DSML development phase of constructing the *core language model*. The core language model captures all relevant domain abstractions and specifies the relations between these abstractions. Accordingly, we defined a core language model for each of our DSMLs. We document the design decisions in a reusable manner by adopting decision templates inspired by related work on documenting architectural design decisions (see, e.g., [3]). The basic phases of DSML development are adopted from [1].

The remainder of the paper is structured as follows: In Section 2, we introduce the process model of DSML development according to [1]. In Section 3, we describe the relations between the decisions and the respective decision options in a structured manner. Limitations of our contribution are discussed in Section 4. Section 5 provides an overview of related work and Section 6 concludes the paper.

2 Background: DSML Development Phases

Before we outline the lessons learned from our DSML projects (see Table 1), we give an overview of the DSML development process applied in our projects (for a detailed discussion see [1]). The following steps were performed iteratively to build the DSMLs:

Define DSML core language model One first defines an initial core language model and the corresponding language model constraints for the target domain. By following a domain analysis method, such as domain-driven design

#	Objectives	Domain
P1	An approach to model interdependent concern behavior using extended UML activity models [14].	Separation of concerns
P2	An integrated approach for modeling processes and process-related RBAC models (roles, hierarchies, statically and dynamically mutual exclusive tasks etc.) [15].	Business processes, role-based access control (RBAC)
P3	A UML extension for an integrated modeling of business processes and process-related duties; particularly the modeling of duties and associated tasks in business process models [16].	Business processes, process-related duties
P4	An approach to provide modeling support for the delegation of roles, tasks, and duties in the context of process-related RBAC models [17].	Business processes, delegation of roles, tasks, and duties
P5	A UML extension to model confidentiality and integrity of object flows in activity models [18].	Data confidentiality and integrity
P6	UML modeling support for the notion of mutual exclusion and binding constraints for duties in process-related RBAC models [19].	RBAC (consistency checks for duties)
P7	Incorporation of data integrity and confidentiality into the model-driven development of process-driven service-oriented architectures [20].	Integrity and confidentiality for service invocations
P8	Integration of context constraints with process-related RBAC models and thereby supporting context-dependent task execution [21].	Business processes, RBAC, context constraints
P9	A generic UML extension for the definition of audit requirements and specification of audit rules at the modeling-level [22].	Audit rules
P10	An approach based on model transformations between the valid structural and behavioral runtime states that a system can have [23].	Model transformation

Table 1. Overview of conducted DSML development projects.

(see, e.g., [24]), domain abstractions are identified and form the language model of a DSML. Because the language model often cannot capture all restrictions and/or semantic properties of the DSML elements, language model constraints are added, if necessary. This phase results in the *DSML core language model* and a catalog of *DSML language model constraints.*

Define DSML concrete syntax In this phase, graphical or textual notation symbols as well as composition and production rules are defined. The DSML core language model and the DSML language model constraints serve as input to produce the *DSML concrete syntax specification.*

Define DSML behavior The behavior specification of a DSML determines how the DSML elements interact to produce the behavior intended by the DSML designer. Syntax and behavior of a DSML are usually defined in parallel. The *DSML behavior specification* (e.g., control flow models, formal textual specifications) is the output of this phase.

DSML platform integration All artifacts defined for a DSML are mapped to the features of a selected platform, either by extending an existing platform or by developing a new tool set. Platform integration is achieved by defining model transformations (see, e.g., [25]) to convert a model into another platform-specific model (model-to-model transformation, M2M) or into machine-readable software artifacts (model-to-text transformation, M2T).

# Decision/Option	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
D1 Language model formalization										
O1.1 M1 class model										
O1.2 Profile definition	×		×				×		×	×
O1.3 Metamodel extension	×	×	×	×	×	×	×	×	×	×
O1.4 Metamodel modification										
O1.5 Combination of options[1]	⊂		⊂				≍		⊂	⊂
D2 Language model constraints										
O2.1 Explicit constraint expressions	×	×	×	×	×	×	×	×	×	×
O2.2 Code annotations										
O2.3 Constraining M2T transformations							×		×	
O2.4 Textual annotations		×			×			×		×
O2.5 Combination of options[1]		≍			⊂		≍	⊂	⊂	⊂
O2.6 None										

Table 2. Overview of design decision points and options.

3 Collected Decisions on the Core Language Model

Most of our DSMLs (see Table 1) provide modeling support for different types of security aspects in a business process context. P10 [23] is an exception and aims at describing program transformations in dynamic programming environments. Each of the ten DSML projects adopted the development process sketched in Section 2. However, due to differing requirements, we did not always perform all DSML development phases. For example, we do not provide platform integration for P10. Thus, to document our experiences from the ten projects, we focus on a single phase and on two specific decisions that appeared in each of the ten projects. In particular, this paper reports on the core language model definition and on the respective design decisions.

For each case presented in Table 1, we identified different decision options. The development of a DSML core language model requires two important decisions: DSML language model formalization (Section 3.1) and defining language model constraints (Section 3.2). Table 2 summarizes these options for both design decision points and lists the options adopted for each of the ten cases. Fig. 1 depicts an overview of the two decisions, the corresponding options, as well as the interdependencies between the decisions and their options. These relations are then discussed for each decision in the respective *Consequences* sub-section (see below).

3.1 D1 Language Model Formalization

Decision *In which way should the domain concepts be formalized?*
Context Domain abstractions are identified and form the language model of a DSML (i.e., the abstract syntax). This language model definition can be expressed, for instance, in a narrative text form, with mathematical expressions (e.g., set algebra), or via a modeling language (e.g., the UML). The language model definition serves as input for the phase of formalizing the domain constructs into the core language model expressed via the UML.

[1]⊂ options complementary; ≍ options equivalent

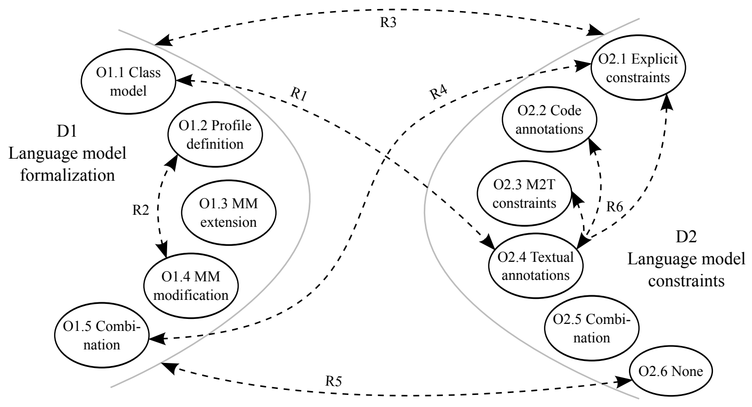


Fig. 1. Relations (R1–R6) between decision options.

Options For UML-based DSMLs, the language model can be defined within the boundaries of the modeling language via dedicated language extension constructs (such as UML profiles) or by extending the modeling language to provide the required semantics (see, e.g., [5,26]).

O1.1 M1 class model: UML class models are an ad-hoc instrument to formalize domain abstractions. Domain concepts can be expressed as classes and relationships as associations.

O1.2 Profile definition: Profiles are a language extension option to tailor the UML for different purposes. A profile consists of a set of stereotypes which define how an *existing* UML metaclass may be extended.

O1.3 Metamodel extension: A metamodel extension introduces, for instance, new metaclasses and/or new associations between metaclasses (MOF-based extension [7]).

O1.4 Metamodel modification: In contrast to a metamodel extension, existing metaclasses of the UML metamodel are modified; e.g., by changing the type of a class property or by deleting existing associations (MOF-based extension [7]).

O1.5 Combination of options: A combination may include the definition of a metamodel extension as well as an equivalent profile definition (e.g., P7). Similarly, stereotype definitions can be provided to accompany a metamodel modification (e.g., P9).

Drivers

Domain space: The degree of overlap between the domain space of the DSML concepts and the general purpose language constructs (i.e., the UML specification) has a direct impact on whether a profile definition is sufficient or on whether a metamodel extension/modification is needed (O1.2–O1.4). In general, a UML extension is reusable if it is compliant with the UML standard.

DSML expressiveness: For instance, a UML profile (O1.2) can only specialize the UML metamodel in such a way that the profile semantics do not conflict with the semantics of the referenced metamodel. Therefore, profile constraints may only define well-formed rules that are more constraining (but consistent with) those specified by the metamodel [5]. In contrast, a metamodel extension/modification (O1.3 and O1.4) is only limited by the constraints imposed by the MOF metamodel.

Portability and evolution: A metamodel extension/modification (O1.3 and O1.4) creates a fork of a certain version of the UML specification. The metamodel does not inherit revisions coming from newly released OMG specifications and can deviate from the UML or MOF standard.

DSML integration: Available DSMLs, software systems, and tool support have a direct impact on the design process of a DSML in terms of integration possibilities. For instance, the UML specification defines a standardized way to use icons and display options for profiles (O1.2). Tool support for authoring UML class models and profiles (O1.1 and O1.2) is widely available.

Consequences (see Fig. 1)

R1 Constraint limitations for class models: A class model defines a language model at the UML instance level (i.e. at the M1 level, see [7]). This means, no metamodel is defined to reflect the domain space and, thus, domain concepts can neither be instantiated nor explicitly constrained for their usage as modeling constructs. Thus, restrictions can only be defined in terms of text annotations attached to the language model.

R2 Profile dependency: Dependencies can occur from combined language model formalizations. For instance, profiles are dependent on the UML metamodel. If a profile is combined with a metamodel modification, changes to the metamodel can lead to implicit and unwanted changes affecting the defined stereotypes (e.g., if a stereotype-extended metaclass is modified).

Examples In all DSML projects, we formalized the language models as metamodel extensions (O1.3). Additionally, profiles (O1.2) were employed in P1, P3, P7, P9, and P10. Therefore, we effectively adopted combined strategies (O1.5). In order to be compliant with the OMG specifications, we did not consider modifying the UML metamodel (O1.4). As an example, Fig. 2 depicts an excerpt from a UML extension (taken from P7). On the left hand side, it shows a UML package definition called `SecureObjectFlows::Services` as an example of a metamodel extension, on the right hand side, it shows a UML profile specification named `SOF::Services`. Mappings between these two language-model representations are provided as M2M transformations. Both UML customizations provide the same modeling capabilities for using one of our UML security extensions (for details see [18,20]) with the SoaML specification [6].

3.2 D2 Language Model Constraints

Decision *Do we have to define constraints over the core language model(s)? If so, how should these constraints be expressed?*

Context A core language model has been formalized in the UML, using either a UML metamodel extension/modification, a UML profile, or a UML class model (see Section 3.1). The resulting language model describes the domain-specific language in terms of its language elements and their interrelations. The definition

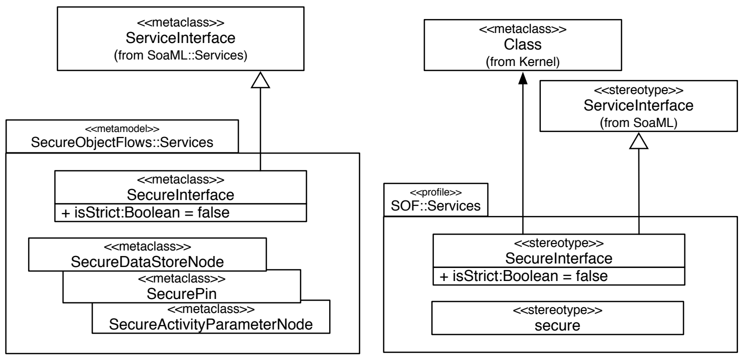


Fig. 2. Exemplary UML metamodel extension and profile definition [20].

of these interrelations is limited through the expressiveness of the MOF and the UML (e.g., part-of relations). A structural UML model, however, cannot capture certain categories of constraints over domain concepts that are relevant for the description of the target domain. Examples are invariants for domain concepts, pre-conditions and post-conditions, as well as guards (referred to as *static* constraints, hereafter). As a result, the language model formalization could be incomplete or ambiguous.

If the language model has been realized by creating multiple formalizations (e.g., multiple profiles), there is an additional risk of introducing inconsistencies provided that the DSML can be used in different configurations (e.g., different profile compositions). Consider, for example, profiles which provide a bridge between two UML extensions.

Options

O2.1 Constraint-language expressions: One can make language model constraints explicit using a constraint-expression language, for instance, via the Object Constraint Language (OCL) or via the Epsilon Validation Language (EVL) in Eclipse.

O2.2 Code annotations: The language model and its elements are enriched through annotations which contain expressions in the host language (or a language embedded within the host language). For example, this can be realized by using model annotations and UML's `OpaqueExpression` [5].

O2.3 Constraining M2T transformations: The constraints over the language model are expressed at the level of transformation templates. That is, template expressions contain checks (e.g., conditional statements based on model navigation expressions) which test model instances for the implicit fit with corresponding domain constraints; e.g., conditional Epsilon Transformation Language (ETL) statements based on Epsilon Object Language (EOL) expressions.

O2.4 Textual annotations: Certain constraints (e.g., temporal bindings) elicited from the target domain cannot be captured sufficiently via evaluable expressions (i.e., constraint language expressions, code annotations) and/or the constraints serve a documentary purpose (to the domain expert). In such cases, unstructured text annotations may capture constraint descriptions meant for the human reader only (e.g., via UML comments).

O2.5 Combination of options: For instance, textual annotations are used as an addition to constraint-language expressions.

O2.6 None: Static constraints over the language model are not made explicit in (or along with) the language model.

Drivers

Constraint formalization: In early iterations (e.g., DSML prototyping), constraints might not be expressed via well-formed, syntactically valid constraint-language expressions, but rather as pseudo-expressions or unstructured text. With the language model maturing during subsequent iterations these annotations can be transformed into evaluable expressions.

Automated language model checking: Depending on whether tool integration for model checking is a requirement, the options O2.1–O2.3 are candidates. A driver toward either option is the intended model-checking time. Relevant points in time follow from the model formalization option adopted (e.g., class model vs. metamodel-based) and the platform-support (model-level or instance-level checks). Language-model checking based on template expressions (O2.3) realizes the latest possible checking point. Therefore, this option does not offer any constraint-based feedback during model development.

Native language model constraints: Constraint-language expressions are developed with the purpose of integrating (i.e., navigating and checking) with the (meta-)model representations. Examples are standard-compliant and vendor-specific OCL expressions for the UML, as well as EVL expressions and Java-coded constraints over secondary Ecore representations of UML models (Eclipse EValidator framework).

Maintainability: Explicitly stating model constraints (O2.1 through O2.3) creates structured text artifacts which must be maintained along with the model artifacts (e.g., the XMI representation). Toolkits and their model representations offer different strategies for this purpose, for instance, embedding constraints into model elements (i.e., model annotations, such as UML comments), maintaining constraint collections as external resources (e.g., separate text files), or editor integration. Each strategy affects the artifact complexity and the effort needed to keep the constraints and the models synchronized.

Portability: If the portability of constraints between different MDD toolkits (e.g., Eclipse MDT, Rational Software Architect, MagicDraw, Dresden OCL) is a mandatory requirement, the platform-dependent options O2.2 and O2.3 can be excluded. However, due to the version incompatibilities and the different vendor-specific constraint-language dialects (e.g., Eclipse MDT OCL), even O2.1 does not guarantee portability for the underspecified sections of the OCL/UML specifications (e.g., navigating stereotypes in model instances or for transitive quantifiers such as `closure` [27]).

Consequences (see Fig. 1)

R3 Conformance between language model and constraints: Constraints on the language model can be defined separately from the referencing metamodel (e.g., using code annotations; O2.2) or at a later stage (e.g., for M2T transformations; O2.3). It must be ensured that language model constraints do not contradict their language model formalization and vice versa. Moreover, constraints may need to be adapted when the corresponding metamodel changes (e.g., OCL navigation expressions).

R4 Constraint inconsistencies: A combination of different language model formalizations (e.g., a UML profile and a metamodel extension; O1.5) may require the duplication and modification of explicit constraint definitions.

R5 Unambiguous language model: If no further constraints to the language model are specified, the language model must be fully and unambiguously defined using the chosen formalization option and their implicitly enforced restrictions (e.g., by using profiles and, thus, inheriting all semantics from the UML metamodel; O1.2).

R6 Impossible constraint evaluation: Some constraints cannot be captured by the means of constraint languages and the underlying language models, code annotations, or model transformation templates (see, e.g., [5]; O2.1–O2.3). Such constraints have to be provided as text annotations in a natural language (O2.4). These constraints either have a documentation purpose only, or they serve for porting the constraints to another environment as they are not bound to a concrete expression form.

Examples In our DSMLs, we encountered all options but code annotations (O2.2) and entirely unconstrained language models (O2.6). So far, we provide constraint-language expressions (O2.1) in the OCL for all of our cases. This is because precise execution semantics were to be expressed in terms of the foundations of UML activities (token flows, e.g., in P1) and of the UML state machines (state/transition; in P10). In eight out of ten DSMLs (P2–P9), these semantics are described by a generic and MOF-compliant metamodel, as well as corresponding metamodel extensions. The generic constraints were then mapped to a UML-based language formalization (i.e. the actual language model and the respective OCL expressions). Code annotations (O2.2) were not considered because the additional model constraints should not be specific to a particular platform (e.g., model representation APIs, generator language). For two DSMLs (P7, P9), we additionally incorporated constraining M2T transformations (O2.3). Textual annotations (O2.4) are either used to complement OCL constraints (P5, P8, P10) or as full substitutes (P2) for otherwise formally expressed constraints.

Constraint 1: The operands specified in a `ContextCondition` are either `ContextAttributes` or `ConstantValues`.

```
context ContextCondition inv:
  self.expression.operand.oclAsType(OperandType)->forAll(o |
    o.oclIsKindOf(ContextAttribute) or
    o.oclIsKindOf(ConstantValue))
```

Constraint 5: The fulfilled$_{\text{CD}}$ Operations must evaluate to true to fulfill the corresponding ContextCondition.

As an example for these two different purposes, consider the above excerpt from P8: For an activity, each action can be guarded by a constraint whose conditions refer to a set of operands and checking operations. At the instance-level (M0), the operations are called to evaluate whether an action should be entered, depending upon some contextual state. Constraint 1 shows a complementary textual annotation. Constraint 5 exemplifies a constraint expressed in natural language due to a model-level mismatch: While the constraint is captured at the language-model level (M2), the operation calls (whose boolean return values are combined to yield the runtime evaluation of the guard) become manifest at the occurrence level of an activity instance (M0) only.

4 Limitations

The most important limitations of the work presented in this paper are that 1) our lessons learned result only from a collective experience and that 2) the underlying decisions were taken by the same group of researchers who developed the ten DSMLs. We reported decisions being characteristic for a single phase (i.e. defining the DSML core language model) and their interdependencies. Documenting the remaining phases (see Section 2) is future work. Moreover, there is the risk of a technology bias given that the ten DSML projects were all performed in a specific technology context (e.g., MOF/UML, OCL, Eclipse modeling tools).

Methodically, this paper presents the results of a narrative synthesis [28] of our DSML development experiences. Therefore, by emphasizing a preselected process model and one of its phases [1], we may have neglected design decisions beyond the scope of this approach. Other risks are the disagreement among the authors during the synthesis process and the dependence of the synthesis results on the review performance of each author (time constraints, level of experience). To mitigate these, we conducted multiple refining iterations over the decision templates and the decision relations, under shifting roles of data checker and data extractor.

5 Related Work

Related work on DSL development [1,2,3,4,8,9,10,11,12,13] was already outlined in Section 1. Below, we review the work relevant for our methodical approach.

For reflecting and synthesizing the decision-related findings from our DSML-development projects, we adapted the guidelines on conducting narrative syntheses proposed by [28]. That is, we selected a process model and its phases as the implicit "theory" underlying our DSML projects. We then collected meta-data about the primary works (e.g., participants, setting, outcomes, target domain, MDD technologies). Based on the selected "theory" (i.e., phases and development artifacts), we then characterized the decisions taken in each development project. In particular, we adopted previously defined decision templates.

The practice of documenting design decisions in a template-based or model-based manner has been proposed for architectural design decisions (see, e.g.,

[29]). In our work, we share the primary motivation of documenting *reusable* design decisions, i.e., decisions and options which are characteristic for every decision-making process in a given technical domain.

6 Concluding Remarks

In this paper, we presented lessons learned from ten DSML development projects in the form of a narrative synthesis. We documented MOF/UML-based decision options and relations between them for the phase of defining the core language model for a DSML in a structured and reusable form. By doing so, we provide decision support for future decision-making processes, facilitate decision documentation, and offer scaffolding for making decisions under incomplete or changing requirements (i.e., in early stages of developing or prototyping). Although we especially focus on design decisions for MOF/UML-based DSMLs, certain decision options do also apply to other modeling languages used in MDD processes. In our future work, we will document additional decision points to cover the remaining phases of the DSML development process.

References

1. Strembeck, M., Zdun, U.: An Approach for the Systematic Development of Domain-Specific Languages. Software: Practice and Experience (SP&E) **39**(15) (2009) 1253–1292
2. Mernik, M., Heering, J., Sloane, A.: When and How to Develop Domain-specific Languages. ACM Computing Surveys (CSUR) **37**(4) (2005) 316–344
3. Zdun, U., Strembeck, M.: Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Projects. In: Proc. of the 14th European Conference on Pattern Languages of Programs (EuroPLoP). (2009)
4. Spinellis, D.: Notable Design Patterns for Domain-specific Languages. Journal of Systems and Software **56**(1) (2001) 91–99
5. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure – Version 2.4.1. Available at: `http://www.omg.org/spec/UML` (2011)
6. Object Management Group: Service oriented architecture Modeling Language (SoaML) – Version 1.0. Available at: `http://www.omg.org/spec/SoaML` (2012)
7. Object Management Group: OMG Meta Object Facility (MOF) Core Specification – Version 2.4.1. Available at: `http://www.omg.org/spec/MOF` (2011)
8. Zdun, U.: A DSL Toolkit for Deferring Architectural Decisions in DSL-based Software Design. Information and Software Technology **52**(9) (2010) 733–748
9. Wile, D.: Lessons Learned from Real DSL Experiments. Science of Computer Programming **51**(3) (2003) 265–290
10. Kelly, S., Pohjonen, R.: Worst Practices for Domain-Specific Modeling. IEEE Software **26**(4) (2009) 22–29
11. Karsai, G., Krahn, H., Pinkernell, C. et al.: Design Guidelines for Domain Specific Languages. In: Proc. of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM). (2009)
12. Selic, B.: A Systematic Approach to Domain-Specific Language Design Using UML. In: Proc. of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), IEEE (2007)

13. Robert, S., Gérard, S., Terrier, F. et al.: A Lightweight Approach for Domain-Specific Modeling Languages Design. In: Proc. of the 35th Euromicro Conference on Software Engineering and Advanced Applications, IEEE (2009)
14. Strembeck, M., Zdun, U.: Modeling Interdependent Concern Behavior using Extended Activity Models. Journal of Object Technology **7**(6) (2008) 143–166
15. Strembeck, M., Mendling, J.: Modeling Process-related RBAC Models with Extended UML Activity Models. Information and Software Technology **53**(5) (2010)
16. Schefer, S., Strembeck, M.: Modeling Process-Related Duties with Extended UML Activity and Interaction Diagrams. In: Proc. of the International Workshop on Flexible Workflows in Distributed Systems. (2011)
17. Schefer, S., Strembeck, M.: Modeling Support for Delegating Roles, Tasks, and Duties in a Process-Related RBAC Context. In: Proc. of the International Workshop on Information Systems Security Engineering (WISSE), Springer, LNBIP (2011)
18. Hoisl, B., Strembeck, M.: Modeling Support for Confidentiality and Integrity of Object Flows in Activity Models. In: Proc. of the 14th International Conference on Business Information Systems (BIS), Springer, LNBIP (2011)
19. Schefer, S.: Consistency Checks for Duties in Extended UML2 Activity Models. In: Proc. of the International Workshop on Security Aspects of Process-aware Information Systems (SAPAIS), IEEE (2011)
20. Hoisl, B., Sobernig, S.: Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models. In: Proc. of the International Workshop on Security Aspects of Process-aware Information Systems (SAPAIS), IEEE (2011)
21. Schefer-Wenzl, S., Strembeck, M.: Modeling Context-Aware RBAC Models for Business Processes in Ubiquitous Computing Environments. In: Proc. of the 3rd International Conference on Mobile, Ubiquitous and Intelligent Computing. (2012)
22. Hoisl, B., Strembeck, M.: A UML Extension for the Model-driven Specification of Audit Rules. In: Proc. of the 2nd International Workshop on Information Systems Security Engineering (WISSE'12), Springer, LNBIP (2012)
23. Zdun, U., Strembeck, M.: Modeling Composition in Dynamic Programming Environments with Model Transformations. In: Proc. of the 5th International Symposium on Software Composition, LNCS, Vol. 4089, Springer (2006)
24. Evans, E.: Domain-driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2004)
25. Mens, T., Gorp, P.v.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science **152** (2006) 125–142
26. Bruck, J., Hussey, K.: Customizing UML: Which Technique is Right for You? Available at: `http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html` (2008)
27. Object Management Group: OMG Object Constraint Language (OCL) – Version 2.3.1. Available at: `http://www.omg.org/spec/OCL` (2012)
28. Cruzes, D., Dybå, T.: Synthesizing Evidence in Software Engineering Research. In: Proc. of the International Symposium on Empirical Software Engineering and Measurement (ESEM). ACM (2010)
29. Obbink, H., Kruchten, P., Kozaczynski, W. et al.: Software Architecture Review and Assessment (SARA) Report, Version 1.0. Available at: `http://kruchten.com/philippe/architecture/SARAv1.pdf` (2002)

Tools and Posters Track

Julia Rubin (ed.)

IBM Research, Haifa, Israel

Preface

ECMFA 2012 holds tool demonstration and poster sessions whose goals are to provide an opportunity for researchers and practitioners to present their most recent experiences in the field of model-based engineering, show-case their tools and have informal discussions about both the latest advances and the challenges ahead. These proceedings provide the tool papers and poster abstracts of those sessions.

This year we accepted 6 out of 9 tool presentations and 4 posters. The submissions cover a spectrum of topics related to model-based engineering, including model-based engineering of embedded systems, architecture design optimization, model query languages, performance modelling and prediction, and model-driven product line engineering.

June 2012

Julia Rubin
Tools and Posters Chair
ECMFA 2012

Tool Papers

CommentTemplate: A Lightweight Code Generator for Java Developers

Jendrik Johannes, Mirko Seifert, Christian Wende, and Florian Heidenreich

DevBoost GmbH
D-10179, Berlin, Germany

Technische Universität Dresden
Chair of Software Technology
D-01062, Dresden, Germany

{firstname.lastname}@devboost.de

Abstract. In this demo we show CommentTemplate, which realises code generation features, as known from model-driven development tools, as a Java language extension. As such, CommentTemplate makes concepts from model-driven development easily accessible to Java programmers. It is realised with Eclipse modelling technology.

1 Introduction

One of the fundamental technologies of model-driven development approaches is code generation. Hence, many code generation technologies emerged over the years (e.g, [1–9] and many more).

An issue model-driven development approaches, and code generation in particular, face when originating from academia, is their adoption by the "common" developer in industry. First, an approach needs adequate tool support above the level of academic prototyping. Second, such tools should be easily accessible for the developer. That is, the developer should be able to immediately start working with the tools without the need to acquire more theoretical knowledge in the beginning (flat learning curve).

We previously developed the Java Model Printer and Parser (JaMoPP) [10], to bring Eclipse modelling and the Java language closer together. JaMoPP consists of a Java metamodel (defined in EMF's Ecore) and the tooling to parse Java source (and byte) code into instances of that metamodel as well as to print instances back to Java source code. This increases integration of modelling and Java both on the modelling and metamodelling level.

CommentTemplate is implemented based on JaMoPP. It takes the important features of existing code generation languages, which are not offered by Java itself already, and implements those as a light-weight extension for Java instead of providing a new language. This demonstrates how fundamental features of model-driven technology can be realised closely to an existing and well-accepted programming language to make these features, which are valuable in their own right, more accessible for programmers.

```
 1 @CommentTemplate
 2 public String helloWorld() {
 3     String greeting = "Hello";
 4     /*<html>
 5         <head><title>greeting world!</title></head>
 6         <body>*/
 7         for (int i = 1; i <= 5; i++) {
 8             String greeted = "World" + i;
 9             /*
10                 greeting greeted!<br/>*/
11             if (greeted.equals("World2")) {
12                 /*
13                     greeted, you are the best!<br/>*/
14             }
15         }
16         /*
17         </body>
18     </html>*/
19     return null;
20 }
```

Listing 1. HelloWorld @CommentTemplate method

2 Demo Description

CommentTemplate is a Java language extension that makes use of and extends the following Java language elements: *multi-line comments*, *methods*, *local variables* and *annotations*. It is realized as an open-source Eclipse plugin that Java developers can install without effort into their Eclipse Java development environment. (cf. Section 3 for installation instructions). In the Demo, we will first explain CommentTemplate on an example—described in the following—and then show different cases of how CommentTemplate is used in customer projects.

A code generation template written in CommentTemplate is shown in Listing 1. A template is defined as a Java method annotated with `@CommentTemplate` that has the return type String. Inside the Method, one can use multi-line comments (`/* */` notation) to define fragments of the template. Around these fragments, arbitrary Java code can be written and used to formulate, for example, loops (Line 7) or conditions (Line 11). Furthermore, one can refer to local variables of type String that are declared before the corresponding template fragment. In the example, the variable `greeting` (declared in Line 3) is referred to two times inside template fragments (Lines 5 and 10).

A `@CommentTemplate` method can be called as any Java method inside arbitrary Java code. However, it will return the expanded template as String. That is, all template fragments are appended and the variables inside the fragments are filled with their values. In the example of Listing 1, the String shown in Listing 2 is produced.

This sums up the basic features of CommentTemplate. However, CommentTemplate offers two additional annotations which help with syntax conflicts between templates and output syntax.

First, one can observe, that no special quotation is used to mark variables in a template fragment in the example (e.g. `greeting` in Line 5). Other code generation tools usually define a fixed symbol for escaping such variables. This

```
<html>
    <head><title>Hello world!</title></head>
    <body>
        Hello World1!<br/>
        Hello World2!<br/>
        World2, you are the best!<br/>
        Hello World3!<br/>
        Hello World4!<br/>
        Hello World5!<br/>
    </body>
</html>
```

Listing 2. Expanded Hello World template of Listing 1

is sometimes problematic, because depending on which output is generated, escape symbols can conflict with the output syntax. CommentTemplate does not define such a symbol itself. However, it allows the user to do so by offering the `@VariableAntiQuotation` annotation which takes a String formatting pattern as argument. In the example, we could add an annotation like `@VariableAntiQuotation("#%s#")` to define that variables should be enclosed in `#` characters. In the example in Line 5, we would then need to write `#greeting#` to a access the *greeting* variable.

Second, CommentTemplate still relies on one problematic fixed symbol, which is `*/` to end a template fragment. To generate this symbol in the output (e.g., when generating Java code with comments), an additional feature is needed. Again, usually, a fixed escape symbol, to escape such symbols which are part of the template language itself, is offered. CommentTemplate makes this configurable by offering the `@ReplacementRule` annotation. This annotation takes two arguments, a *pattern* and a *replacement*, which allows the specification of a replacement for a certain String. For the problem described above, one can use `@ReplacementRule(pattern="#/", replacement="*/")`, which replaces all occurences of `#/` with `*/`. `#/` can then be used as an alternative for `*/`.

Both `@VariableAntiQuotation` and `@ReplacementRule` can be applied on the level of single `@CommentTemplate` methods but also on the level of classes. This allows a fine grained control of which escape characters are used where and helps to avoid syntax conflicts with the output syntax.

CommentTemplate was motivated by the functionalities and ideas behind existing code generation languages and language features such as JET, EGL, Acceleo, Xpand, Xtend2, MOFScript Velocity, StringTemplate or JSP [1–9]. The idea of compiling the templates to Java source code can be found in JET [1], Xtend2 [5] and JSP [9]. The separation of template and output formatting by a smart handling of tab characters was adopted from Xtend2 [5]. A limitation of CommentTemplate is that it does not support expressions inside templates. This can also be seen as a strength which forces the user to separate model and view as publicized by StringTemplate [8]. Unique to CommentTemplate is its closeness to Java and its lightweightness reflected in its low number of new features added to Java and the fact that compiled templates consist of plain Java code without any dependencies despite Java itself.

3 Installation Instructions and Screencast

CommentTemplate can be installed from the DropsBox update-site available at `http://www.dropsbox.org/update_trunk` (category CommentTemplate).
A screencast of the CommentTemplate installation and usage is available at `http://www.dropsbox.org/CommentTemplate`

Acknowledgments

This work is supported by:

References

1. Eclipse Foundation: JET Project. www.eclipse.org/emft/projects/jet (April 2012)
2. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon Generation Language. In: Proc. of ECMDA-FA'08. Volume 5095 of LNCS., Springer (2008)
3. Eclipse Foundation: Acceleo Project. www.eclipse.org/acceleo (April 2012)
4. Eclipse Foundation: Xpand Project. www.eclipse.org/modeling/m2t/?project=xpand (April 2012)
5. Eclipse Foundation: Xtend Project. www.eclipse.org/xtend (April 2012)
6. Eclipse Foundation: MOFScript Project (April 2012)
7. Apache Software Foundation: Apache Velocity Project. velocity.apache.org (April 2012)
8. Parr, T.: StringTemplate. www.stringtemplate.org (April 2012)
9. Oracle: JavaServer Pages Technology. www.oracle.com/technetwork/java/javaee/jsp (April 2012)
10. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. of SLE'09. LNCS, Springer (March 2010)

A Model-based Tool for Automated Quality-driven Design of System Architectures

Ramin Etemaadi and Michel R. V. Chaudron

Leiden Institute of Advanced Computer Science
Leiden University, Netherlands
{etemaadi,chaudron}@liacs.nl

Abstract. For designing a software system architecture, a large number of quality properties needs to be addressed in nowadays complex software systems. These quality properties are mostly conflicting and make the problem very complex. In practice, software architects manually try to come up with a set of different architectural designs and then try to identify the most suitable one among these. This process is time-consuming and may lead the architect to suboptimal designs. We propose a tool which is named AQOSA (Automated Quality-driven Optimization of Software Architecture) to tackle this problem. AQOSA aids architects by automatically synthesizing optimal solutions. To this end, AQOSA uses a model-based approach to evaluate component-based software architecture quality properties. It uses multi-objective evolutionary algorithms to alternate architectural solutions. Finally, it suggests optimal solutions along with the trade-offs between multiple quality objectives.

Keywords: AQOSA Tool ; Architecture Design Optimization ; Software Design Quality ; Model-Driven Software Development (MDSD) ;

1 Introduction

The quality of the architectural design is critical factor for the successful development of a software system. The architecture has deep impact on quality properties such as performance, safety, security, energy consumption and cost. Hence, methods and techniques are needed for designing good architectures to be able to address various quality constraints in the early phases of system development. In this paper the authors introduce an architect assistant tool for automated software architecture design.

The contribution of this paper is introducing a tool for automated software architecture design optimization that supports multiple quality attributes including *response time*, *processor utilization*, *bus utilization*, *safety*, and *cost.* It optimizes multiple quality attributes at once and supports multiple degrees of freedom for varying architectural solutions.

The paper is organized as follows: Section 2 describes AQOSA tool briefly. In Section 3 the demonstration plan for the tool is discussed. Finally, conclusions are given in Section 4.

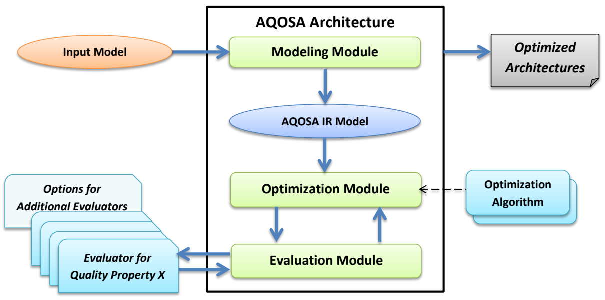


Fig. 1: AQOSA Architecture

2 AQOSA Toolkit

2.1 Process

To describe the process of using the AQOSA tool, let us to start with inputs. The tool takes as input:

– an initial functional-part of the architecture (i.e. components that provide the needed functionality and their connections),
– a repository that contains a set of (specifications of) hardware and software components,
– a set of typical usage scenarios,
– objective functions which are typically defined and normalized as the distance between the ideal performance for non-functional quality properties and the actual performance.

Then, the AQOSA iterates through the following steps as a generation in the evolutionary optimization. It iterates until some stopping criteria holds. This can be number of generations or related to the objective function:

1. generate a new set of candidate architecture solutions. To this end, AQOSA should know the degrees of freedom in the design. So, by using the repository and these degrees of freedom it generates new alternative solutions.
2. evaluate the new set of candidate architecture solutions for multiple quality properties. This works by generating analysis models from the architecture model using model transformations and then analyzing these models.
   Currently, the tool can support performance (response time, processor utilization, bus utilization), safety and cost quality properties.
3. select a set of optimal (i.e. Pareto-front) solutions as parents to generate the next offsprings in next iterations.

2.2 Modules

Figure 1 shows the architecture of the AQOSA toolkit. It contains three main module which are collaborate together based on a central model (AQOSA-IR). It generates the optimized architecture as output which is described in section 2.3.

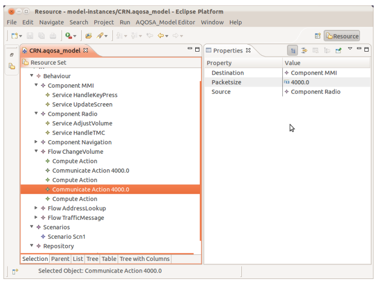
Modeling Module Because AQOSA is designed to optimize architectures in a wide range of domains, it aims to be independent on specific modeling languages. Hence, it uses its own internal architecture representation, AQOSA intermediate representation (AQOSA-IR). Figure 2a shows its editor.

Optimization Module The AQOSA optimization module tries to optimize the architecture with respect to potentially contradicting quality attributes based on evolutionary algorithms. To this end, it automatically generates new architecture designs. This module has been implemented based on the Opt4J optimization framework[1]. It supports Evolutionary Multi-Objective Algorithms (EMOA) to improve the architecture such as NSGA-II, SPEA2, SMS-EMOA. Figure 2b demonstrates the optimizer interface during optimization process.

Evaluation Module The AQOSA evaluation module gets an evaluation model which is transformed from an AQOSA-IR and a decoded genotype for specific evaluation purpose. It feeds these models to each evaluator and returns the results to the optimization module. Rather than having built-in evaluators for quality properties, the AQOSA tool allows state-of-the-art external evaluators to be easily plugged into the framework. Performance attributes are implemented by extending of JINQS[2] Queueing Networks (QN) library and safety by a Fault Tree Analysis (FTA) method inspired by [3]. For instance in safety analysis, the evaluation model contains a fault-tree which is derived from genotype and the failure characteristics of components. This model will be sent to safety evaluator.

2.3 The Results

As output, AQOSA reports a set of optimal architectural solutions. This set is a Pareto-front set of solutions with respect to the trade-off between different quality objectives. For the analysis of the trade-off between solutions we have developed a separate tool which plots solutions in 3D and allows architect to manipulate solutions interactively.

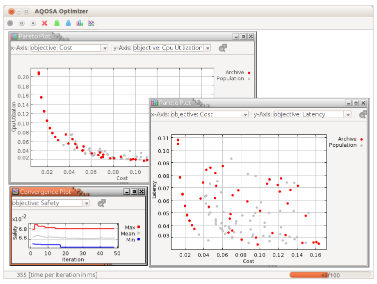

Resource - model-instances/CRN.aqosa_model - Eclipse Platform
File Edit Navigate Search Project Run AQOSA_Model Editor Window Help
Resource
CRN.aqosa_model
Properties

```
Resource Set
  Behaviour
    Component MMI
      Service HandleKeyPress
      Service UpdateScreen
    Component Radio
      Service AdjustVolume
      Service HandleTMC
    Component Navigation
    Flow ChangeVolume
      Compute Action
      Communicate Action 4000.0
      Compute Action
      Communicate Action 4000.0
      Compute Action
    Flow AddressLookup
    Flow TrafficMessage
  Scenarios
    Scenario Scn1
  Repository
```

Property	Value
Destination	Component MMI
Packetsize	4000.0
Source	Component Radio

Selected Object: Communicate Action 4000.0

(a) AQOSA-IR Editor Interface



(b) Optimization Interface

Fig. 2: Tool Screenshots

3 Demonstration

The demonstration of the tool includes three major parts:

Model Inputs: Modeling of the tool input ingredients includes:

– model the system components that provide the needed functionality and their connections,
– define the repository of hardware and software components which gives alternatives to AQOSA for replacements,
– model some typical usage scenarios,
– specify the required objectives.

Optimization Setup: There are various settings which should be configured properly for optimization process. This part shows how to configure optimization algorithm, number of generations, mutation or crossover rates and etc.

Results Analysis: Afterward, a seperate tool could be used by the architect to analyze the results of AQOSA tool interactively. So, he would be able to choose the best suitable solution among the set of optimal candidates.

4 Conclusion

This paper introduces a tool for automated software architecture optimization that supports multiple quality attributes. It offers a new tool to aid architects finding good designs in complex design situations with many potentially conflicting quality requirements. This tool reduces development time and improves the quality of the architecture design. We presented the details of the AQOSA tool which supports *response time*, *processor utilization*, *bus utilization*, *safety*, and *cost*. Future work is defining the knowledge-based methods for implementing software tactics as means to generate alternative architectural solutions.

Acknowledgment

This work has been supported by the Netherlands national project OMECA (Optimization of Modular Embedded Computer-vision Architectures).

References

1. Lukasiewycz, M., Glaß, M., Reimann, F., Teich, J.: Opt4J: a modular framework for meta-heuristic optimization. In Krasnogor, N., Lanzi, P.L., eds.: GECCO, ACM (2011) 1723–1730
2. Field, T.: JINQS: An Extensible Library for Simulating. Multiclass Queueing Networks, http://www.doc.ic.ac.uk/ ajf/Research/manual.pdf
3. Forster, M., Trapp, M.: Fault Tree Analysis of Software-Controlled Component Systems Based on Second-Order Probabilities. In: ISSRE, IEEE Computer Society (2009) 146–154

MQ-2: A Tool for Prolog-based Model Querying

Vlad Acretoaie and Harald Störrle

Department of Informatics and Mathematical Modeling,
Technical University of Denmark
Richard Petersens Plads, 2800 Lyngby, Denmark
s100988@student.dtu.dk,hsto@imm.dtu.dk

Abstract. MQ-2 integrates a Prolog console into the MagicDraw[1] modeling environment and equips this console with features targeted specifically to the task of querying models. The vision of MQ-2 is to make Prolog-based model querying accessible to both student and expert modelers by offering powerful query features and a tight integration with the host modeling environment.

1 Motivation

MQ-2 is designed to support the model querying approach described in [1, 2] and its successor, the Visual Model Query Language (VMQL) [3]. The main impetus behind the development of MQ-2 has been the feedback gathered in follow-up interviews with participants to a paper-based usability study of VMQL [3]. A consensus has emerged among interviewees concerning the high impact of tool support on the usability of any model querying approach. MQ-2 leverages this observation and brings VMQL one step closer to its goal of becoming a fully usable model querying solution targeted at student and expert modelers.

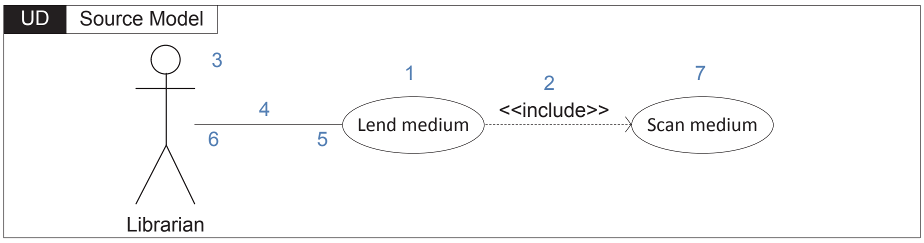
The remainder of this paper is organized as follows. Section 2 introduces the querying approach supported by MQ-2, Section 3 provides an overview of MQ-2's architecture and Section 4 proposes a demonstration plan.

2 Querying

Consider the use case diagram in Fig. 1, inspired by the Library Management System (LMS) test scenario (see Sec. 4). In order to perform queries on this diagram, it must first be transformed from its XMI representation into the Prolog fact database also shown in Fig. 1, with model element IDs highlighted in blue in both the diagram and its Prolog representation. This database consists of facts of the form `me(type-id, [tag-value], ...])`, where `type` is a model element's metaclass, `id` is an arbitrary unique identifier, `tag` is an atom representing one of the model element's properties, and `value` is the value for this property. There is a one-to-one mapping between model elements and Prolog facts.

Once a model's Prolog representation is created, it can be queried from any Prolog console. For instance, the query

[1] https://www.magicdraw.com/



```
:-module('Source Model',[me/2]).
    me(model-0,[annotation-id(1),ownedMember-ids([1,3,4,7]),name 'LMS',
                visibility-public]).
    me(useCase-1,[ownedMember-ids([2]),name-'Lend medium',
                  visibility-public]).
    me(include-2,[visibility-public,addition-id(7)]).
    me(actor-3,[name-'Librarian',visibility-public).
    me(association-4,[ownedMember-ids([5,6]),visibility-public,
                      navigableOwnedEnd-ids([6,5]),memberEnd-ids([6,5])]).
    me(property-5,[visibility-private,type-id(1),association-id(4)]).
    me(property-6,[visibility-private,type-id(3),association-id(4)]).
    me(useCase-7,[name-'Scan medium',visibility-public]).
```

Fig. 1. A use case diagram (top) and its encoding as a Prolog fact database (bottom)

```
me(useCase-Id,Attrs), member(name-'Lend medium',Attrs).
```

returns all model elements of metaclass `useCase` having the value 'Lend medium' for their `name` meta-attribute. It also binds the returned model elements' identifiers to the `Id` variable and their list of meta-attributes to the `Attrs` variable. In short, the query finds the *Lend medium* use case. Its execution is facilitated by the integration of a Prolog console into MagicDraw provided by MQ-2.

The MQ-2 console offers several model querying specific features not available in a generic Prolog console, as specified in Table 1.

Table 1. MQ-2 console features

Transforming models to Prolog fact databases.
Pre-consulted library predicates.
Showing query results sequentially or all at once.
Showing query results in the MagicDraw Search Results Tree.
Highlighting query results in diagrams where they appear.
Highlighting selected console text in relevant diagrams.

However, queries formulated using the `me` predicate directly are cumbersome to formulate. To compensate for this, MQ-2 implements several library predicates introduced in [1, 2] (see Fig. 2). Using library predicates, retrieving the *Lend medium* use case can be accomplished more intuitively via the `get_me` predicate:

```
get_me(model, name-'Lend medium', useCase-Id, _).
get_me(MODEL, TAG-VAL, METACLASS-ID, VAL)
        Matches all elements of MODEL containing the TAG-VAL meta-attribute pair.
match(SOURCE_MODEL, QUERY_MODEL, BINDINGS)
        Returns bindings between QUERY_MODEL and  SOURCE_MODEL.
match(SOURCE_MODEL, QUERY_MODEL, CONSTRAINTS, BINDINGS)
        Returns bindings between QUERY_MODEL and SOURCE_MODEL considering a list of VMQL constraints.
```

Fig. 2. Sample MQ-2 library predicates

The `match` predicate allows formulating queries using the host modeling language. It returns a list of bindings between elements of the query and source models, and optionally accepts a list of VMQL constraints. For instance, the `distinct` constraint specifies that no two query model elements may be bound to the same source model element. A complete list of VMQL constraints is available in [3]. A future goal for MQ-2 is to support the specification of constraints directly on the query model as comments endowed with the `<vmql>` stereotype.

3 Architecture

The proposed framework for Prolog-based model querying consists of a host modeling tool (currently MagicDraw) including the MQ-2 plug-in, an SWI-Prolog[2] installation, and the Java Prolog Bridge (JPL)[3] library (see Fig. 3). The host modeling tool acts as a model repository and a front-end for interacting with MQ-2, while SWI-Prolog acts as a query execution engine. MQ-2 itself is a plug-in extending the UI of the host modeling tool and providing built-in Prolog modules that implement functionality such as model matching. This architecture enables MQ-2 to remain easily portable to other host modeling tools.

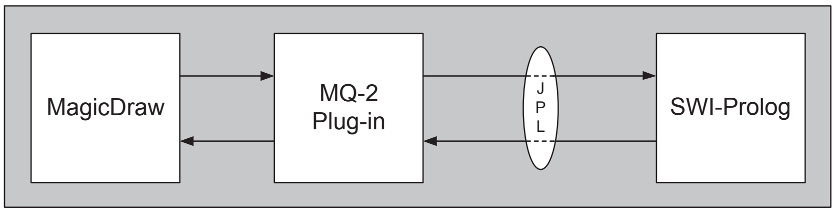
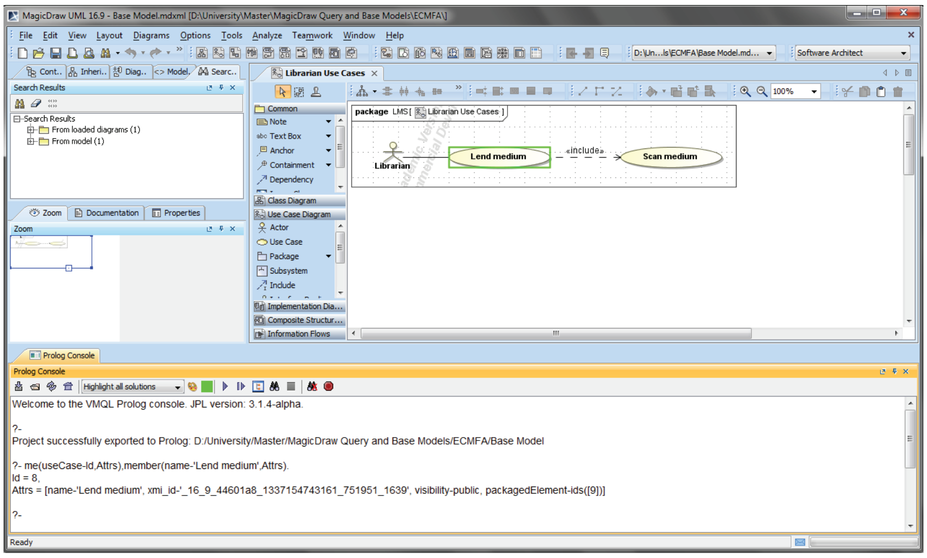


Fig. 3. MQ-2 deployment with MagicDraw as host modeling tool

A screenshot of MQ-2 is presented in Fig. 4. It features the MQ-2 Prolog console and toolbar on the bottom of the screen. The diagram pane shows the diagram introduced in Sec. 2, and the console query retrieves the *Lend medium* use case. As a result, this use case is highlighted in green on the diagram pane.

[2] http://www.swi-prolog.org/
[3] http://www.swi-prolog.org/packages/jpl/



```
Welcome to the VMQL Prolog console. JPL version: 3.1.4-alpha.

?-
Project successfully exported to Prolog: D:/University/Master/MagicDraw Query and Base Models/ECMFA/Base Model

?- me(useCase-Id,Attrs),member(name-'Lend medium',Attrs).
Id = 8,
Attrs = [name-'Lend medium', xmi_id-'_16_9_44601a8_1337154743161_751951_1639', visibility-public, packagedElement-ids([9])]

?-
```

Fig. 4. Screenshot of MagicDraw featuring the MQ-2 plug-in

4 Demonstration Plan

MQ-2 will be showcased on a UML design model created by a group of students in the context of the Requirements Engineering course taught at a Master's level at the Technical University of Denmark. The model specifies the requirements for an LMS used by a local library for the purpose of managing its book inventory, loans, librarians, and readers. This usage scenario has been selected in view of the fact that MQ-2 is envisioned to be used by students taking the same course in the next academic year, providing arguably the best feedback as to whether MQ-2 meets its design goal of acting as a usable model querying tool.

The tool demonstration will include transforming the source model into a Prolog fact database, querying individual model elements, and querying model fragments containing VMQL constraints. The various query result display methods provided by MQ-2 will also be highlighted.

References

1. Störrle, H.: A logical model query interface. In: Intl. Ws. Visual Languages and Logic (VLL'09), pp.18–36. CEUR (2009).
2. Störrle, H.: A PROLOG-based Approach to Representing and Querying UML Models. In: Intl. Ws. Visual Languages and Logic (VLL'07), pp.71–84. CEUR (2007).
3. Störrle, H.: VMQL: A Visual Language for Ad-Hoc Model Querying. J. Visual Languages and Computing 22(1), 3–29 (2011).

Rapid Performance Modeling and Reasoning with UCM2PCM

Christian Vogel[1], Heiko Koziolek[2], Thomas Goldschmidt[2], and Erik Burger[1]

[1] Karlsruhe Institute of Technology (KIT), Germany
[2] ABB Corporate Research Germany, Industrial Software Systems Program

1 Introduction

Industrial software systems, such as distributed control systems, follow complex information flows. In addition, these systems have challenging performance requirements for high throughput and short response times. Thus, early performance modeling is desired to identify performance bottlenecks and design efficient information flows. Performance modeling notations require expertise from the performance domain, and their graphical representations are often difficult to discuss with stakeholders. This complicates reasoning on different alternatives for scalable architectures and leads to a general reluctance to apply performance modeling in this domain.

The notation of use case maps (UCM) has been created by analyzing how software systems are typically sketched in early stages on white boards. UCMs capture high level software components as well as the control flow for specific usage scenarios in an intuitive notation. The UCM2PCM tool transforms UCMs into instances of the Palladio Component Model (PCM) [1], which enables performance predictions for component-based software systems. The transformation bridges semantic gaps between the requirements-oriented UCM notation and the component-oriented PCM notation. Users can create UCMs using existing graphical editors and transparently run performance solvers using the Palladio tool chain. Our tool supports a two-staged software performance engineering process, where domain stakeholders can perform rapid initial modeling based on UCMs, and performance experts and architects can refine the resulting PCM models.

2 Foundations

As a part of the User Requirements Notation (URN) specification [4], *Use Case Maps (UCM)* are used to visualize how a system works and what the requirements and causal responsibilities are. UCMs are behavioral diagrams and use scenarios. This is a similarity to UML sequence diagrams, but UCMs remain on a higher abstraction level. In difference to sequence diagrams, UCMs do not show all messages or signals that are exchanged between components or actors, but only control flows that are important for the behavior of a system. Skipping the details enhances the overview and allows the usage of UCM early in the design process, where not much detail is specified yet [3].

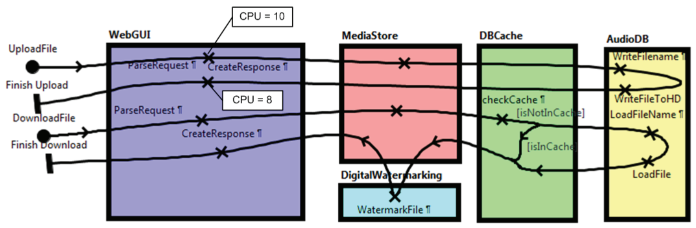
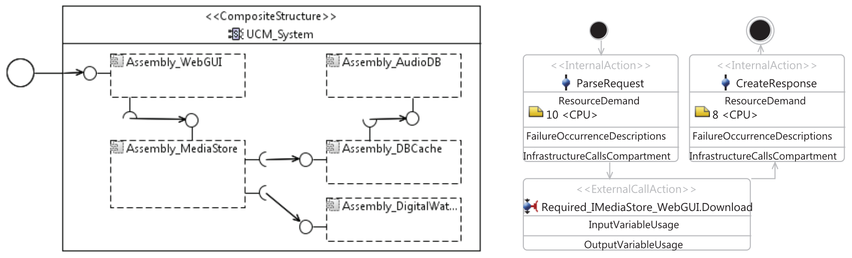


Fig. 1. Use case map of the *MediaStore* components and inner control flow including examples for performance annotations.

A UCM diagram consists of paths that show a possible control flow through a system. Fig. 1 shows an example of an UCM diagram for a *MediaStore* software. Every path has at least one start and one end point. Paths can fork and join. A cross on a path represents so-called *Responsibilities*, which describe the actions that take place on a high abstraction level.

To model the control flow, forks can be added to a path. An *AndFork* splits a path into two branches that are executed in parallel. An *OrFork* offers alternative branches. To decide which branch is taken, conditions or probabilities can be specified. The *OrJoin* joins branches. Additionally, an *AndJoin* contains a barrier that continues the control flow only when all incoming branches finished executing. UCMs also support the modeling of components. With components, the entities involved in a scenario can be specified, and an architectural structure of a system can be defined. Components in UCMs can also be nested. In a diagram, components are represented by a box. All elements inside the box are bound to that component.

The *Palladio Component Model (PCM)* [1] is a domain-specific tool for component based software engineering (CBSE) that is based on the Eclipse Modeling Framework (EMF). Its focus is performance prediction for software models. It offers the comparison and selection of several design alternatives of a software system in early stages of development, where only a model exists. The PCM Workbench provides multiple methods to transform architecture models into performance models. Analytical and simulation models are supported. Fig. 2 shows an excerpt of the MediaStore example modelled in Palladio.



(a) System of the *MediaStore*

(b) Performance abstraction of the *Download* service

Fig. 2. *MediaStore* PCM Example

3 Implementation/Validation

UCM2PCM is implemented as an Eclipse plug-in. The transformation uses the Eclipse Model to Model (M2M) Transformation framework and is implemented in QVT-Operational. For the creation of UCMs, we use the jUCMNav editor[5], which is based on the Graphical Modeling Framework (GMF). UCM2PCM can be invoked from inside Eclipse, accepting a UCM model as input. The resulting PCM model can be further refined by editing it with the PCM Workbench, or analyzed using the existing PCM solvers and simulators.

The transformation maps UCM components to PCM components, then segments each UCM path per component to create *Service Effect Specifications (SEFF)*, which are used as an abstraction of component behaviour in Palladio. Since UCMs do not support the modeling of performance properties natively, textual performance annotations for UCMs are defined in UCM2PCM, which are then transformed into PCM performance abstractions. Fig. 1 includes example annotations specifying the number of CPU units required to process a given action. If the user does not specify performance properties, default values are set. These annotations are then transformed into the corresponding performance annotations in the *Internal Actions* of the SEFF as shown in Fig. 2(b).

To validate the UCM2PCM tool, we need to show that its mapping of model elements is valid and the tool is useful. Thus, we have evaluated (a) the accuracy of the transformation (i.e., the correct mapping) and (b) the usability of UCMs for performance modelling. To address (a), we transformed performance-annotated use case maps of three heterogeneous, mid-sized systems [1, 2, 7] into the respective PCM models. We then ran a series of experiments and compared the simulation results of these models with simulation results from former PCM models of these system. The difference between the original model and the UCM-based models was below 10 percent in most cases. Further details can be found in [6]. Despite some differences in the simulation results, we deem the accuracy of UCM2PCM sufficient to support early design-time performance decisions. The low differences for different kinds of models demonstrate that UCM2PCM was successful in bridging most semantic differences in these models.

To address (b), we let six users model a UCM and apply the UCM2PCM transformation. We then asked them for feedback in a user survey to evaluate the usability of UCM2PCM. While all of the participants were able to quickly produce a UCM, a majority was afraid that UCM would not be suitable for complex models. In future work, we will thus further simplify UCM2PCM and support the creation of large models with the tool. The comprehensibility of the models was deemed good or very good by all participants, especially for non-experts. The results of this user survey are however not statistically significant due to the small sample size. A future empirical study should investigate a larger sample size, analyze different design alternatives, and compare UCM modeling with other methods.

4 Conclusion/Presentation Plan

The tool presented in this paper lowers the entrance barrier for performance engineering by combining an intuitively understandable modeling language (UCM) with a sophisticated performance engineering approach (PCM). Software engineers can create UCMs and use the defaults in the presented approach to transform them into PCM instances. The simulation and analysis tools of the PCM workbench deliver performance values which can be used for the evaluation of design alternatives.

If more detailed performance properties are required, the generated PCM instances can be further adapted. Thus, the approach enables software engineers to model performance properties on different levels of abstraction. Data-flow oriented systems can be described more easily and clearly than with PCM alone, without losing the benefits of the PCM simulation and analysis techniques.

In the tool presentation, we plan to demonstrate the creation of a UCM for the MediaStore example system. After modeling the system components and the control flow, textual performance annotations will then be added to the UCM components. The UCM2PCM transformation will then be invoked, resulting in a PCM instance which is analysed with the simulation engines in the Palladio toolchain. A comparison of the resulting PCM instance and the UCM will demonstrate the easier legibility of the control flow in the UCM notation, while the performance properties can be modelled in more detail with the PCM notation.

References

1. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. JSS 82, 3–22 (2009), `http://dx.doi.org/10.1016/j.jss.2008.03.066`
2. Brosig, F., Huber, N., Kounev, S.: Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In: 26th IEEE/ACM Intl. Conference On Automated Software Engineering (ASE) (November 2011)
3. Buhr, R.: Use case maps: A new model to bridge the gap between requirements and design. In: OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15 (1995)
4. International Telecommunication Union (ITU): User requirements notation (URN) - Language definition, z.151 edn. (11 2008), `http://www.itu.int/rec/T-REC-Z.151/recommendation.asp?lang=en&parent=T-REC-Z.151-200811-I`
5. Mussbacher, G., Amyot, D.: Goal and scenario modeling, analysis, and transformation with jucmnav. In: Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on. pp. 431–432. IEEE (2009)
6. Vogel, C.: A Use Case Map Editor for Rapid Performance Modeling and Reasoning. Master's thesis, Karlsruhe Institute of Technology (KIT) (2012)
7. Wu, X., Woodside, M.: Performance Modeling from Software Components. In: Proc. 4th International Workshop on Software and Performance (WOSP'04). vol. 29, pp. 290–301. ACM Press, New York, NY, USA (2004)

MDE in Practice: Process-centric Performance Prediction via Simulation in Real-time

David Redlich[1], Stephanie Platz[2], Thomas Molka[1], Wasif Gilani[1], and Ulrich Winkler[1]

[1] SAP Research Center Belfast, United Kingdom,
`[david.redlich|thomas.molka|wasif.gilani|ulrich.winkler]@sap.com`
[2] Hasso-Plattner-Institut Potsdam, Germany,
`stephanie.platz@student.hpi.uni-potsdam.de`

Abstract. Today's markets are growing faster, more volatile and competitive. As a consequence businesses are exposed to greater pressure to react fast against changes, preferably even before these changes result in negative effects for the organisation, e.g. violations of service level agreements (SLAs) or legal compliance failures. This paper introduces a model-driven tool that enables pro-active decision making based on performance predictions in real-time to avoid violations.

Key words: models at run-time, event-driven business process management, business process simulation, business activity monitoring

1 Background

The introduced tool offers process-centric decision support in real-time which is an application of Event-Driven Business Process Management (EDBPM). EDBPM emerged from the combination of the two disciplines Business Process Management (BPM) and Complex Event Processing (CEP) [1]. Practically, this is realised by two individual platforms interacting with each other through interfaces or events, one a BPM system, which is to model, manage, and optimise a business, and the other one a CEP engine [4]. In general, CEP deals with the event-driven behaviour of large, distributed enterprise systems [2], i.e. events produced by the system are captured, selected, aggregated, and eventually abstracted to generate complex events representing high-level information about the situational status of the system. In the case of a business process execution environment these events consist of raw data like process instance id, timestamp, and type of the state change, e.g. `2011-05-26 T 10:45 CET: Activity ''Check availability'' completed, pi-id: 253`. A CEP engine can then compute complex events containing high-level information about the performance of a business process, e.g. process instance occurrence, and activity net working time. The extraction of performance parameters from live-events is a common application of real-time monitoring of business processes, which is in general called Business Activity Monitoring (BAM). Usually raw live-events from a business process execution are not of interest in the context of BAM, instead the aggregation of these into performance related parameters is carried out [3].

Furthermore, the tool makes use of a range of model-driven concepts, such as model to model and model to text transformations, model annotations, model management, model weaving, etc. to deal with the complexity of supporting a wide range of business process modelling languages. These model-driven concepts are generally designed to be applied at design time whereas the demoed tool is operating at run-time. Due to special concerns that need to be addressed in the case of a run-time application of model-driven concepts the notion of models@run.time has evolved. In [5] a model at run-time (M@RT) is defined "... a causal connected self-representation of the associated system that emphasises the structure, behaviour, or goals of the system from a problem space perspective".

2 Tool Description

The proposed solution enhances the BAM capabilities of existing EDBPM tools like Slipstream [7], which produce real-time performance parameters related to business processes, with the ability to further predict the future trends of these parameters. Conceptual architecture (see Figure 1) and principle of operation of the model-driven tool have been presented in [6]. Key characteristics of the demo tool, of which some are shown in the architecture figure, are:

– The simulation results in form of future-events are fed back into the CEP engine without extra implementation effort, i.e. prediction results are computed in the same way by the CEP as the real-time events. The benefit of this

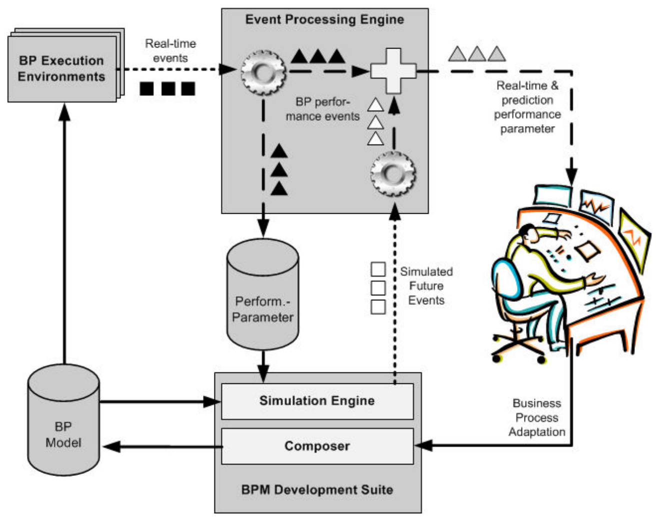


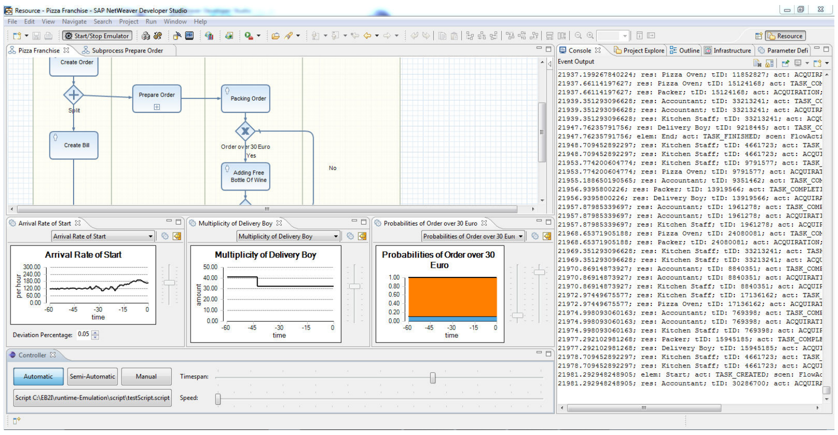
Fig. 1. Data flow of the event-based framework for real-time performance prediction

technique is an easy integration into existing BAM solutions without extra adaptations.
– Contrary to traditional data-centric business intelligence the tool provides improved predictions on the basis of simulation by additionally taking the behavioural information into account, which is captured in the business process model.
– The tool uses a model-driven approach to support the wide range of business process languages applied in practise. Through the utilisation of a decomposed model transformation with an independent simulation model at its core new business process model languages can be easily integrated into our BAM tool.

3 Demonstration Plan

We demonstrate how the manager of a large business can monitor and govern his business performance with the help of the tool. The demonstration is structured as follows:

1. **Use-case introduction**: In the first step we motivate the tool by introducing the use-case: A pizza franchise operating in multiple locations. Charles McGuiness is the manager and is interested in how his business performs. This is captured by Key Performance Indicators(KPIs), such as overall time from pizza order to delivery - if this time exceeds the threshold of 60 minutes, it will have a negative impact on customer satisfaction. The process under study is a modified sales order process which was extended by the following additional behaviour due to a promotional offer: a bottle of wine is added if the order is above 30 Euro. Charles McGuiness wants to make sure that all KPIs are never violated even with this additional behaviour.
2. **Solution introduction**: The solution is proposed, i.e. the benefits and key characteristics of the tool are outlined and explained. A special focus will be the explanation of the model-driven concepts that have been applied in order to make the tool generic.
3. **Showcase the live-demo**: First, the BPM suite is introduced which generates live-events for the pizza franchise while execution (see Figure 2). Then in a second step, the processing of the raw events which is carried out in real-time is explained. In a third and final step the tool's GUI for monitoring KPI's and process performance parameters is shown. Whenever a KPI violation is predicted to happen, the manager Charles McGuiness receives a notification on his mobile device. He is further provided with the functionality to drill-down and identify the root cause of the predicted KPI violation. Since this information is provided before the KPI violation has actually occurred he has time to react in order to prevent this from happening. A recorded demo of the tool can be accessed using the following link: `http://youtu.be/K8NpmK48xB4`.



```
21937.199267840224; res: Pizza Oven; tID: 11852827; act: ACQUIRA
21937.66114197627; res: Pizza Oven; tID: 15124168; act: TASK_COM
21937.66114197627; res: Packer; tID: 15124168; act: ACQUIRATION;
21939.35129309662 8; res: Accountant; tID: 33213241; act: TASK_CC
21939.351293096628; res: Accountant; tID: 33213241; act: ACQUIRA
21939.351293096628; res: Kitchen Staff; tID: 33213241; act: ACQU
21947.76235791756; res: Delivery Boy; tID: 9218445; act: TASK_CC
21947.76235791756; elem: End; act: TASK_FINISHED; scen: FlowActi
21948.709452892297; res: Kitchen Staff; tID: 4661723; act: TASK_
21948.709452892297; res: Kitchen Staff; tID: 4661723; act: ACQUI
21953.774200604774; res: Kitchen Staff; tID: 9791577; act: TASK_
21953.774200604774; res: Pizza Oven; tID: 9791577; act: ACQUIRAT
21955.188650190565; res: Accountant; tID: 9351462; act: TASK_COM
21956.9395800226; res: Packer; tID: 13919566; act: TASK_COMPLETI
21956.9395800226; res: Delivery Boy; tID: 13919566; act: ACQUIRA
21957.87985339697; res: Accountant; tID: 1961278; act: TASK_COME
21957.87985339697; res: Accountant; tID: 1961278; act: ACQUIRATI
21957.87985339697; res: Kitchen Staff; tID: 1961278; act: ACQUIF
21968.65371905188; res: Pizza Oven; tID: 24080081; act: TASK_COM
21968.65371905188; res: Packer; tID: 24080081; act: ACQUIRATION;
21969.351293096628; res: Kitchen Staff; tID: 33213241; act: TASK
21969.351293096628; res: Kitchen Staff; tID: 33213241; act: ACQU
21970.86914873927; res: Accountant; tID: 8840351; act: TASK_COME
21970.86914873927; res: Accountant; tID: 8840351; act: ACQUIRATI
21970.86914873927; res: Kitchen Staff; tID: 8840351; act: ACQUIF
21972.97449675577; res: Kitchen Staff; tID: 17136162; act: TASK_
21972.97449675577; res: Pizza Oven; tID: 17136162; act: ACQUIRAT
21974.998093060163; res: Accountant; tID: 769398; act: TASK_COME
21974.998093060163; res: Accountant; tID: 769398; act: ACQUIRATI
21974.998093060163; res: Kitchen Staff; tID: 769398; act: ACQUIF
21977.29210298126 8; res: Packer; tID: 15945185; act: TASK_COMPLE
21977.292102981268; res: Delivery Boy; tID: 15945185; act: ACQUI
21978.709452892297; res: Kitchen Staff; tID: 4661723; act: TASK_
21978.709452892297; res: Kitchen Staff; tID: 4661723; act: ACQUI
21981.292948248905; elem: Start; act: TASK_CREATED; scen: FlowAc
21981.292948248905; res: Accountant; tID: 30286700; act: ACQUIRA
```

Fig. 2. BPM Suite: Screenshot

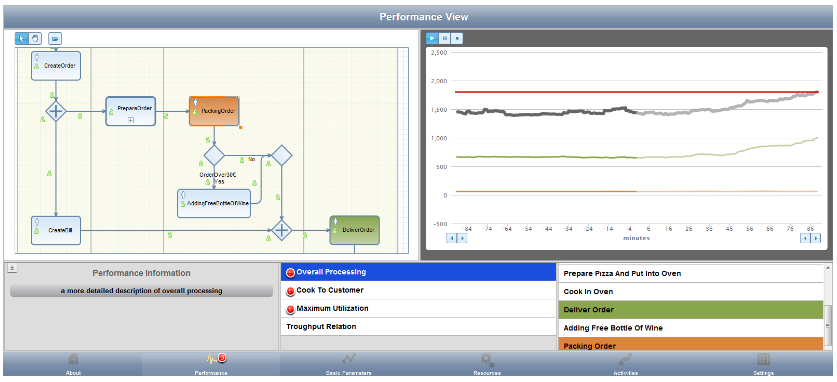


Fig. 3. Process-centric Decision Support in real-time: Web-Application Screenshot

References

1. von Ammon, R., et al.: Integrating Complex Events for Collaborating and Dynamically Changing Business Processes. In: Dan, A., et al.: ICSOC/ServiceWave 2009 Workshops. LNCS, vol. 6275, pp. 370-384. Springer, Heidelberg (2010)
2. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional, Reading (2002)
3. Eckert, M.: Complex Event Processing with XChange EQ: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events. LMU Mnchen (2008)
4. von Ammon, R: Event-Driven Business Process Management. In Proceedings of Encyclopedia of Database Systems, pp. 1068–1071. Springer US (2009)
5. Blair, G., Bencomo, N., France, R. B.: Models@run.time. In Computer, 42(10), 2009.
6. Redlich, D., Gilani, W.: Event-Driven Process-Centric Performance Prediction via Simulation, In: Daniel, et al.: BPM Workshops, LNBIP, vol. 99, pp. 473-478 (2011)
7. Janiesch, et al.: Slipstream: Architecture Options for Real-time Process Analytics. In: Chu, W., et al.: Proceedings ACM Symposium on Applied Computing (2011)

S2T2-Configurator: Interactive Support for Configuration of Large Feature Models

Goetz Botterweck and Andreas Pleuss

Lero-The Irish Software Engineering Research Centre, Univ. of Limerick, Ireland*
{goetz.botterweck, andreas.pleuss}@lero.ie

Abstract. S2T2-Configurator is a visual tool for configuration of feature models. In this tool paper, we focus on interactive techniques that support the configuration of large and complex models.

1 Motivation

Even though the fundamental practices in Product Line Engineering (PLE) are well known and applied in practice [3], it is still a challenge to create and handle product lines of realistic size and complexity. Besides organizational challenges (e.g., how to transition to a PLE approach) a major inhibitor to product line adoption is the complexity of the underlying artifacts [7].

A common way to represent product lines (PL) are *feature models* [5, 4], which describe configuration options like optional features or alternatives to choose from. By configuring a feature model, a user can specify one particular product to be derived from the product line. However, due to the number of features and dependencies between them it is difficult to understand the model as a whole and the consequences of particular configuration decisions during the configuration process.

A potential solution to mitigate this situation are visualization techniques [12, 11], which have been shown to reduce cognitive complexity [2].

In earlier work [1, 9] we introduced *S2T2 Configurator*[1] a tool that demonstrates the use of visualization techniques for common PLE tasks. In this paper we present interactive techniques for the handling of large and complex models, e.g., contextual filters, views for different stakeholders, search-and-highlight, and automated collapse/expand.

2 Interactive support for product configuration

Main structure and layout modes. S2T2 provides various layout styles to show the main hierarchy of the feature model and additional dependencies between

* Supported, in part, by Science Foundation Ireland grant 10/CE/I1855. We are grateful to Mikoláš Janota, Denny Schneeweiss, and Elisabeth Engel who contributed significantly to the concepts and implementation of the S2T2 Configurator.

[1] S2T2 = SPL of SPL Tools and Techniques

features (cross-tree constraints). In Figure 1 we show the "vertical explorer" layout. Other display modes are, e.g., optimized to show the mapping between features and implementation components or to show the effect of configuration decisions on product attributes (see "feature flow maps" in [9]).

Configuration state. Whenever the user decides that a feature should be *selected* or *eliminated* (✔ = selected, ✖ = eliminated), the tool calculates the consequences of that decision, i.e., it tests *all* potential next operations for satisfiability of the model. Consequences are automatically applied to the model (i.e., automatically setting features as selected or deselected if the model would become unsatisfiable otherwise) indicated by gray colored icons. To simplify the understanding of the automatically calculated values, the tool provides an explain function, which on request marks all user decisions and constraints that caused the particular value. This graphical representation is based on a proof generated by the reasoning engine.

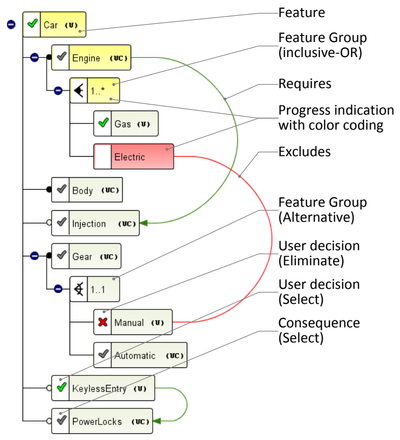


Fig. 1. S2T2 Configurator

Progress indicators. In product configuration all available variability must be resolved until no decision is left open. Hence, in large models it is useful to mark areas that still need decisions. We use color shadings to indicate the configuration progress of subtrees and features. Since these colors can be processed pre-attentively (i.e., without consciously thinking about it), this allows to immediately spot unconfigured areas even in very large models.

Filtering of large models. To support the handling of large models, S2T2 allows to focus on a subset of the model. In order to define what is relevant, users select those features that they are interested in and then activate filtering heuristics (or combinations thereof) that show (i) all directly related nodes, (ii) entire subtrees below, or (iii) all ancestors to indicate relative location in the model.

Furthermore, we provide means to pre-define a list of such views and then step through them in the configuration process. This can be used in several application scenarios: For instance, different "views" can be defined for varying stakeholders and aspects. As an example consider the feature model shown in Figure 2 which shows different aspects of a car, like *CustomerFeatures* ❶, *SoftwareComponents* ❷, *Casing* ❸, and *Electronics* ❹. We can then treat the configuration as a guided process, handling one aspect after another and thus reducing complexity. For instance, by selecting *CustomerFeatures* and then using the function "Focus on subtrees" the user gets the filtered view marked with 🅰. After defining similar views for the features marked with ❶ to ❹ the users can

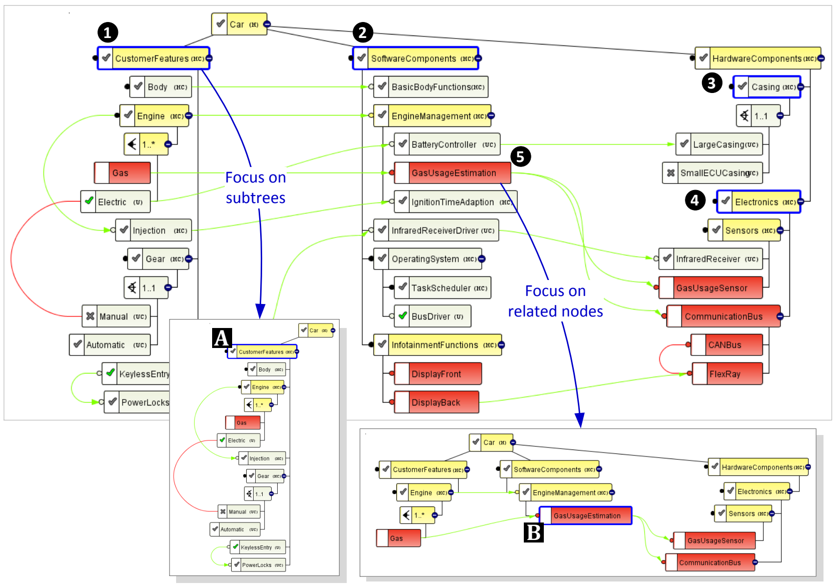


Fig. 2. Focusing on parts of the model

concentrate on one aspects of the model at a time and switch back and forth between these views - similar to a "wizard" interaction pattern.

A similar function allows to focus on a set of features and all related nodes. For instance, by selecting *GasUsageEstimation* ❺ and using "Focus on related features, users get the filtered view marked with **B**. This supports the users in complex configuration decisions, where they have to focus on one feature and consider all consequences in other parts of the model.

The parameters of these views can be adapted during configuration, e.g., the users can add additional features to the set of focused nodes or they can toggle the display of related nodes, which are linked via feature dependencies.

3 Current work

To explore its practical use S2T2 Configurator has been applied to feature models from various domains (online shops, embedded systems [8], component based software architectures [10], software evolution [6]). The largest model used so far for performance tests was a generated test model with 1114 features and 278 cross-tree constraints. The largest models used for configuration by users contained 227 features and 41 cross-tree constraints. In current work, we are evaluating the effects of the suggested interaction patterns on task execution time, error rates, and subjective user preference in a systematic user study.

References

1. Goetz Botterweck, Mikolas Janota, and Denny Schneeweiss. A design of a configurable feature model configurator. In *VAMOS 2009*, 2009.
2. Stuart K. Card, Jock D. MacKinlay, and Ben Shneiderman. *Readings in Information Visualization - Using Vision to Think.* Morgan Kaufmann, 1999.
3. Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.
4. Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming.* Addison Wesley, Reading, MA, USA, 2000.
5. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.
6. Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Model-driven support for product line evolution on feature level (available online, in press). *Journal of Systems and Software (JSS) - Special Issue on Automated Software Evolution*, 2011. http://dx.doi.org/10.1016/j.jss.2011.08.008.
7. Andreas Pleuss, Rick Rabiser, and Goetz Botterweck. Visualization techniques for application in interactive product configuration. In *MAPLE/SCALE 2011*, 2011.
8. Andreas Polzer, Daniel Merschen, Goetz Botterweck, Andreas Pleuss, Jacques Thomas, Bernd Hedenetz, and Stefan Kowalewski. Managing complexity and variability of a model-based embedded software product line. *Innovations in Systems and Software Engineering (ISSE)*, 8(1):35–49, 2011.
9. Denny Schneeweiss and Goetz Botterweck. Using flow maps to visualize product attributes during feature configuration. In *VISPLE 2010*, Jeju Island, Korea, September 2010.
10. Lionel Seinturier, Philippe Merle, Damien Fournier, Valerio Schiavoni, Christophe Demarey, Nicolas Dolet, and Nicolas Petitprez. OW2 FraSCAti user guide. Technical report, OW2 Consortium, 2011.
11. Robert Spence. *Information Visualization.* Addison Wesley, December 2000.
12. Colin Ware. *Information Visualization: Perception for Design.* Morgan Kaufmann, 2nd edition, 2004.

Appendix: Demonstration Plan

- Software architecture, frameworks (Eclipse EMF, Prefuse visualization)
- Changes in the model as the configuration progresses, reasoning engine, consequences, explain, automatic completion
- Configuration with large models
- Progress indicators with color schemes
- Focus: filter view, focus on related features, focus on subtrees, show ancestors, usages of model views
- Feature models with product attributes, visualization with feature flow maps
- Integration with other tools, conversion of DSLs with model transformations.

Poster Abstracts

MADES Language, Methodology and Tools for Avionics and Surveillance Embedded Systems

Alessandra Bagnato[1], Imran Quadri[2] and Andrey Sadovykh[2]
TXT e-solutions S.p.A, 16100 Genoa, Italy
alessandra.bagnato@txtgroup.com,
[2]Softeam, 21 avenue Victor Hugo, 75016 Paris, France
imran.quadri@softeam.fr
andrey.sadovykh@softeam.fr

Abstract. The poster describes the MADES methodology and tools and the reasoning behind their creation within the EU-FP7 project MADES. The two case studies that led to the requirements of the project are also presented.

Keywords: Real-Time Embedded Systems Design, Model Driven Engineering, MARTE, SysML, UML

1 Introduction

The poster presents the MADES FP7 project [1], its novel model driven language, along with the underlying methodology and tools for the design, validation, simulation, and code generation of complex avionics and surveillance based real-time and embedded systems. The poster outlines the defined methodology based on an effective subset of existing standardized UML profiles for embedded systems modeling: SysML and MARTE. The design phases related to the MADES methodology are also illustrated. Additionally, MADES case studies and the developed tools are described in the poster, namely Softeam's open source Modelio environment[2], that is capable of fully supporting SysML and MARTE profiles along with the dedicated MADES diagrams associated with the different modeling design phases; as well as MADES Component Repository and MADES Code Generation and Verification environments [0]. MADES project adds as unique features a specific set of diagrams to help in increasing design productivity, decrease production cycles and promote synergy between the different designers/teams working at different domain aspects of the global system in consideration.

2 References

1. MADES: EU FP7 Project. (2012) , http://www.mades-project.org/.
2. Modelio: The Open Source UML Modeling Environment. (2012) http://www.modelio.org/.

A. Bagnato, A. Sadovykh, R. F. Paige, D. S. Kolovos, L. Baresi, A. Morzenti, and M. Rossi. MADES: Embedded systems engineering approach in the avionics domain. In Proceedings of the Workshop on Hands-on Platforms and tools for model-based engineering of Embedded Systems (HoPES), 2010.

EU FP7 ENOSYS PROJECT: Integrated modeling and synthesis tool flow for embedded systems design

Etienne Brosse and Andrey Sadovykh
Softeam, 21 avenue Victor Hugo, 75016 Paris, France
etienne.brosse@softeam.fr
andrey.sadovykh@softeam.fr

Abstract. The poster describes the methodology and tools developed in the EU FP7 ENOSYS project, providing a complete flow for designing and generating of real-time and embedded systems. Additionally, two case studies illustrating and validating the design methodology are also expressed related to the project.

Keywords: Real-Time Embedded Systems Design, Systems-on-Chips (SoCs), Behavioral Synthesis, Design Space Exploration, Model Driven Engineering, UML, MARTE

1 Introduction

The poster presents the ENOSYS FP7 project [1], which proposes a high abstraction level design methodology for the design and implementation of next-generation Systems-on-Chips (SoCs). The poster outlines the ENOSYS methodology that takes as entry points UML-based models and the MARTE profile, to semi automatically generate a SoC implementation. The poster will illustrate the relevant features of the project, such as SoC Co-Design using UML/MARTE, automated synthesis and code generation of hardware/software from high level UML models, software source code optimization and design space exploration aspects. Finally, the poster will showcase design and implementation of two end-user case studies, namely a JPEG 2000 image compression system and a transmit section of a 802.16m (WiMax) mobile standard.

2 References

1. ENOSYS: EU FP7 Project. (2012), http://www.enosys-project.eu/.

DevBoost Tools for Model-Driven Software Modernisation

Jendrik Johannes, Mirko Seifert, Christian Wende, and Florian Heidenreich

DevBoost GmbH
D-10179, Berlin, Germany

Technische Universität Dresden
Chair of Software Technology
D-01062, Dresden, Germany

`{firstname.lastname}@devboost.de`

This poster shows a set of Eclipse/EMF-based modelling tools and how we use them for software modernisation. These tools include known tools from our toolset, applied in the new context of modernisation, as well as newly developed tools. The poster shows which challenges of software modernisation are addresses by each tool. It also illustrates how the tools interact to perform and automate complete modernisation scenarios. The tools are parts of the DropsBox [1] toolset developed by DevBoost and TU Dresden. They are:

– **EMFText** was originally designed as a tool for DSL development [2]. In the context of software modernisation, it is used to extract information from heterogeneous artefacts into one connected homogeneous EMF model.
– **JaMoPP** transforms Java source code into EMF models and vice versa [3]. It is therefore used for a variety of modernisation tasks (e.g., enforcing architecture changes) in the context of Java application modernisation.
– **CommentTemplate** is a light-weight code generation tool that is used, in the context of software modernisation, to generate parts of a modernised system based on (parts of) the models extracted from the existing system.
– **BuildBoost** is a (meta-)build framework that is highly extensible and focused on minimal-to-zero build configuration. It is used to automate and provide continuous integration for software modernisation processes.

References

1. DevBoost GmbH and Software Technology Group Dresden: The Dresden Open Software Toolbox (DropsBox). www.dropsbox.de (April 2012)
2. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Proc. of ECMDA-FA'09. Volume 5562 of LNCS., Springer (June 2009) 114–129
3. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. of SLE'09. LNCS, Springer (March 2010)

Gefördert durch:
Bundesministerium für Wirtschaft und Technologie
aufgrund eines Beschlusses des Deutschen Bundestages
exist Existenzgründungen aus der Wissenschaft
EUROPÄISCHE UNION
ESF Europäischer Sozialfonds für Deutschland

eMoflon: A Metamodelling and Model Transformation Tool

Anthony Anjorin[⋆], Marius Lauder[⋆], and Andy Schürr

Technische Universität Darmstadt,
Real-Time Systems Lab,
D-64283 Merckstraße 25, Darmstadt, Germany
{anthony.anjorin,marius.lauder,andy.schuerr}@es.tu-darmstadt.de

Model Driven Engineering (MDE) is an established means of dealing with the increasing complexity of modern software systems by providing suitable abstractions and tools specifically tailored for a certain domain. The MDE process involves establishing a *metamodel* to specify the relevant concepts and relationships in a domain, and *model transformations*, to provide the semantics of *models*, i.e., instances of the metamodel. Based on a suitable metamodel and model transformations, a *domain specific language* can be defined to reduce the gap between problem and solution domains, thereby increasing productivity, improving communication with domain experts, increasing software quality, and supporting interoperability. eMoflon[1] supports the MDE process with a metamodelling and model transformation environment offering a unique set of features as it:

– extends the Eclipse Modelling Framework (EMF) in a natural, object-oriented manner by providing the possibility of modelling *behaviour* via model transformations, from which a Java implementation for operations in the metamodel is generated.
– supports *bidirectional* model transformations, generating a pair of unidirectional model transformations automatically from a single specification.
– uses an extension of a professional UML tool *Enterprise Architect*[2] as its frontend.
– provides a backend that is seamlessly integrated into Eclipse/EMF.
– has a solid formal foundation based on algebraic graph transformations, using *programmed graph transformations* for unidirectional model transformation, and *triple graph grammars* for bidirectional model transformation.
– is 100% generative, i.e., all specifications are mapped to standard Java code, which can be extended and mixed with hand-written code and used without any runtime dependencies[3] on eMoflon.
– uses predominantly *visual* modelling languages.
– is 100% Ecore/EMF compatible.
– is developed via a *bootstrap*, i.e, a substantial part of eMoflon is built using eMoflon.

⋆ Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.
[1] www.moflon.org
[2] http://www.sparxsystems.de/uml/
[3] Required libraries are added automatically to the buildpath as a set of jars.

1st Workshop on European Industrial & Academic Collaborations on Real Time & Embedded Systems Modelling and Analysis

Michel Bourdellès, Laurent Rioux, Sébastien Gérard

A lot of European initiatives funded by the European Commission and/or directly by countries lead to experiment on the modelling of RTES from different industrial domains (Communications, Automotive, Space, Railway …), dealing with system modelling declined as formal, Component, Application/Platform allocation modelling, model transformation and analysis, process integration, test, functional & non functional properties verification, methodology adaptation, requirements traceability, real platform results confrontation.

The objective of this workshop is to present ongoing industrial /academic current work on the modelling and analysis of real time and embedded systems. A particular attention will be given on successful stories in the integration and assessment on the exploitation of R&D improvements on industrial designs.

RT-SIMEX : Performance Retro-modelling

Laurent RIOUX Ph.D (THALES Research and Technology)

This presentation presents the final results from the French ANR project RT-Simex. RT-Simex proposes a set of tools to analyze timing of parallel embedded code and trace the simulation results back to the initial models from which the code was generated. The whole tool-set relies on standard formats (UML/MARTE, Open Trace Format) to ensure a perennial use. This presentation will also presents the global RT-SIMEX process from retro-modeling to PSM/PIM model debugging.

Principles and Tool for Time- and Space-Partitioned Systems

José Rufino (FCUL - Faculty of Sciences of University of Lisbon – Lisboa, Portugal)

This presentation highlights the main challenges in the integration of Time- and Space-Partitioned (TSP) principles in mission critical systems, such as autonomous vehicles. The design of a TSP-based safety kernel to guarantee the functional safety of vehicle operation is addressed, together with the evolution of the Cheddar real-time scheduling analysis tool towards TSP-specific scheduling analysis and generation of the corresponding onboard computer configuration parameters.

MADES: An effective UML/SysML/MARTE methodology for Real-Time Embedded Systems design and implementation

Imran Quadri PhD (Softeam)

This presentation presents an overview of the EU FP7 MADES project, that aims to develop novel model-driven techniques for the design, validation, simulation, and code generation of complex real-time and embedded systems for avionics and surveillance embedded systems industries. In this presentation, we will illustrate the MADES language built on an effective SysML/MARTE subset for embedded systems specification, along with verification & validation (V&V) and code generation aspects, related to the MADES case studies dealing with on-board and ground based radar systems.

PRESTO: Results from execution trace analysis

Shuai Li (THALES Communications & Security)

This presentation presents results the global synoptic and results of the ARTEMIS PRESTO project.. This project focuses on RTES design process enriched of : (a) test traces exploitation (generated by test execution in the software integration phase induced by the industrial development process, to validate the requirements of the system) along with (b) platform models and (c) design space exploration techniques.

ENOSYS: Integrated modeling and synthesis tool flow for embedded system design

Etienne Brosse (Softeam)

The presentation will describe the ENOSYS FP7 project, which proposes a high abstraction level design methodology for the design and implementation of next-generation Systems-on-Chips (SoCs) in order to shorten time to market. The presentation outlines the ENOSYS methodology and illustrates some features of the project: such as SoC Co-Design using UML/MARTE, automated synthesis and code generation of hardware/software from high level UML models, software source code optimization and design space exploration

VERDE: Industrial results on component based modelling analysis

Olivier Hachet (THALES Communications & Security)

MAENAD: Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles

Ernest Wozniak, CEA LIST

Fully Electric Vehicles (FEV) promise clear benefits to society. At the same time, the engineering of FEV introduces significant new challenges. MAENAD is refining the EAST-ADL architecture description language for meeting these challenges.