Checking consistency between interaction diagrams and state machines in UML models

Piotr Jacek Puczynski

Kongens Lyngby 2012
IMM-M.Sc.-2012-44

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2012-44

Abstract

The Unified Modeling Language (UML) is the de-facto standard for the object-oriented modeling of (software) systems. It describes a system by modeling different views on the system, e.g. using class and component diagrams to provide a view on the static structure of the system and, e.g., state machines and sequence diagrams to provide a view on the dynamic behavior of the system.

The different views of the system should be consistent, that is, for example, that the class names and methods named in interaction diagrams should correspond to class names and methods used in class diagrams. This can be easily checked syntactically and ensured by the modeler if he uses a modeling tool like Topcased, that allows him to work with a common instance of the meta model for all diagrams. This, e.g., allows using the same class object in a class diagram and in an interaction diagram. However, other connections between the different types of diagrams are not as easily ensured. For example, that an interaction diagram showing the realization of a use case scenario is consistent with the behavior described by, e.g., object life cycle state machines and protocol state machines.

To goal of the thesis is to take the methodology used in the system integration course to describe a system, and to develop a tool that ensures the consistency of the UML model. In the course system integration, a system is described by components which have ports. Ports have required and provided interfaces and protocol state machines describing the possible communication through the ports. Components are implemented by one or several classes which have object state machines to describe their behaviors. Finally, the model of a system contains use cases and use case scenarios as interaction diagrams that describe the interaction between the user and the system.

The task of the thesis is to develop a tool that takes a UML model and performs the following checks:

- Checks that the components are implemented by the classes
- Creates interaction diagrams by extending the interaction diagrams of the use cases to show how the system realizes the use case scenarios given the behavior described by the object state machines
- Alternatively, the user provides the interaction diagrams himself and the tool checks that the interactions are compatible with the classes and their object life cycle state machines
- Checks that the created/provided interaction diagram contains admissible interactions according to the protocol state machines of the ports

The tool should provide sensible hints to help the user to fix the model of the system if problems in the validation occur.

The tool should be implemented preferably as an Eclipse plug-in using EMF to represent the UML model.

Preface

This thesis was prepared at the Software Engineering Section, DTU Informatics, Technical University of Denmark, in partial fulfillment of the requirements for acquiring the Master of Science degree in Computer Science and Engineering. The work on the thesis was carried out in the period from 6th December 2011 to 28th May 2012, having a workload of 30 ECTS credits.

Lyngby, 28-May-2012

Piotr Jacek Puczynski

Acknowledgements

I would like to kindly thank my supervisor Hubert Baumeister for all his support and the patience during the project preparation and execution. His insight into the topic and the personal involvement were the key factors that shaped the project.

I would also like to thank other professors from DTU Informatik for the important discussions about the modeling and the project itself, especially Ekkart Kindler.

Thanks to Kent Inge Simonsen from Bergen University College for important observations about the examples used in this thesis.

Thanks to Aliki Ott from Universität Bremen for providing the important references for the project and the consultation.

I wish to thank OCL Eclipse project lead Edward Willink for answering all of my many questions about the advanced OCL evaluator usage. I admit, I partly repaid his help by finding few important bugs in the OCL evaluator.

I wish to also thank the Topcased developers for giving me the important hints during the project, especially Volker Stolz but also Li Dan and Tristan Faure.

Special thanks to Per Friis for providing the computer hardware that speeded up the development of the project.

Contents

Abstract i

Preface iii

Acknowledgements v

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Structure of the thesis . . . 8
- 1.3 Related work . . . 8

2 Selected elements of the Unified Modeling Language 15
- 2.1 UML as modeling language . . . 15
- 2.2 Semantics Variation Points . . . 17
- 2.3 Classes and instances . . . 17
- 2.4 Components . . . 19
- 2.5 State machines . . . 20
- 2.6 Interactions . . . 24

3 Inconsistencies 27
- 3.1 Sequence diagrams and structural properties of model . . . 27
- 3.2 State machine diagrams and structural properties of model . . . 32
- 3.3 Other structural inconsistencies . . . 33
- 3.4 Behavioral state machines and structural properties of model . . . 34
- 3.5 Sequence diagrams and behavioral state machines . . . 36
- 3.6 Conformance to contract specified by interfaces . . . 40
- 3.7 Related to components . . . 44

4 Case study: Toll System 47
- 4.1 Introduction to the toll system . . . 47
- 4.2 Check-in with toll tag . . . 49
- 4.3 Check-out with toll tag . . . 49
- 4.4 Models of the toll system . . . 50

5 Concepts and approach **51**
5.1 Consistency checking by scenario simulation 51
5.2 Consistency checking algorithm 53
5.3 Direct UML representation 54
5.4 Scenarios 55
5.5 Realizations of scenarios 56
5.6 Extensions of scenarios to realizations of the scenarios 59
5.7 Execution of behavioral state machines 66
5.8 The Simple Action Language 83
5.9 Verification of protocol state machines 93

6 Tool **107**
6.1 General information 107
6.2 Functionalities 109
6.3 Design 116
6.4 Implementation notes 120
6.5 Testing 121

7 Conclusions and Future Work **123**
7.1 Conclusions 123
7.2 Future work 125
7.3 Evaluation 125

A Toll System without components **127**

B Toll System with components **135**

C Scenarios and realizations of scenarios in Toll System **141**

Bibliography **147**

Nomenclature

BES	Behavior execution specification
BSM	Behavioral state machine
EMF	Eclipse Modeling Framework
MOS	Message occurrence specification
OCL	Object Constraint Language
OMG	Object Management Group
PSM	Protocol state machine
SAL	Simple Action Language
UML	Unified Modeling Language

Chapter 1

Introduction

In this chapter, we will see a motivation for finding inconsistencies in UML models and we will discover the project's objectives. We will also get an overview on the structure of the thesis and the related work section.

1.1 Motivation

Modeling plays a central role in the activities that lead up to a deployment of good software. We build the models in order to better understand and communicate the structure and behavior of our software systems.

The Unified Modeling Language (UML) is currently the standard way of modeling object-oriented systems. Different UML diagram types allow different views on the system: *structural* (static) and *behavioral* (dynamic) [Hol04]. Each view can consist of many diagrams of different types. The different diagrams should be consistent with each other since they all represent the same underlying model (see fig. 1.1). Changes in one of the diagrams ultimately affect the underlying model and it may cause (sometimes unexpected) consequences for the other diagrams. Diagrams should, in principle, give a clear view on how the model is structured and how it behaves. Having the different views on the model simplifies the design of the complex systems, but unfortunately, it also makes it easier to introduce inconsistencies [Egy00]. Any inconsistencies can result in serious problems, including the situation where the model may not be able to fulfill the intended functionality.

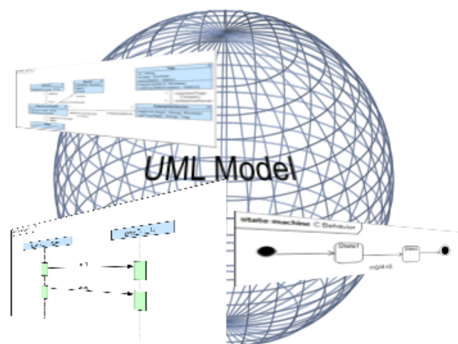

Figure 1.1: The UML model represented as a sphere with different views on it represented as different UML diagrams.

A modeler (a person that models) can easily check some consistency rules using modeling software that allows the modeler to work on instance of the UML meta-model. This approach gives the possibility to use classes from the structure of the model in the elements describing behavior, i.e. to refer from the behavioral diagrams to the structural diagrams. Using modeling software also makes it easier to avoid syntactic consistency problems and guarantees that the model conforms to its meta-model.

A problem arises when we wish to check for consistency between different UML diagrams defining behavior. For instance, a state machine diagram describing the behavior of a class has a relationship to a sequence diagram representing the interaction in a system. The relationship is e.g. the order in which the messages are received and sent on a lifeline (representing an object of the class) in a sequence diagram. This order must correspond to an order of triggers and effects that are placed on transitions that are possible to fire in a state machine diagram during the system execution. The state machine diagram, in this case, defines the behavior of the object represented by the lifeline. The described problem is recognized as the semantic consistency problem and is not trivial to detect (in more complex systems).

Our motivating example will be a naive automated teller machine (ATM) with a single use-case; a client withdraws the money from the ATM (see fig. 1.3). Our ATM system contains a hidden design defect. The ATM system model and its instance (object diagram) are presented in fig. 1.2.

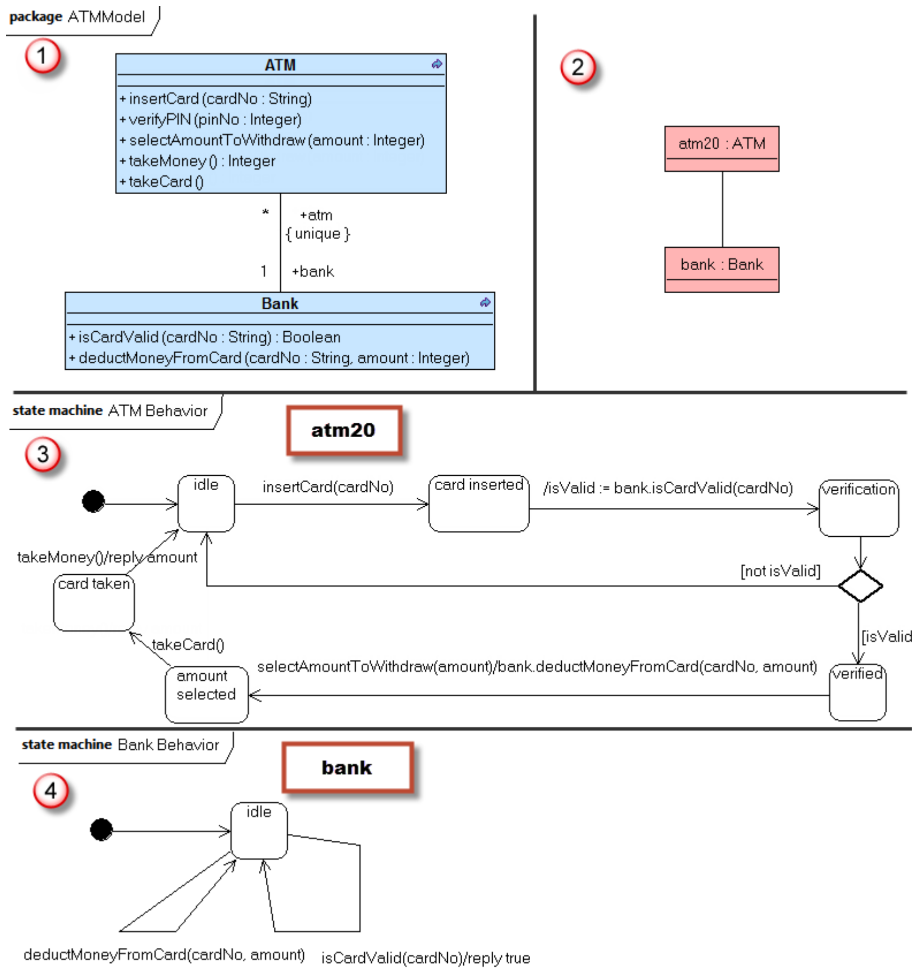

Figure 1.2: The ATM model in UML containing a hidden defect. Marker no. 1: the class diagram. Marker no. 2: the object diagram showing the instance of the system. Marker no. 3: the behavior of the class *ATM* (used by instance *atm20*). Marker no. 4: the behavior of the class *Bank* (used by instance *bank*). The naive behavior of the *Bank* class will always validate every credit-card and deduct the money without validating that the client has the amount on the account.

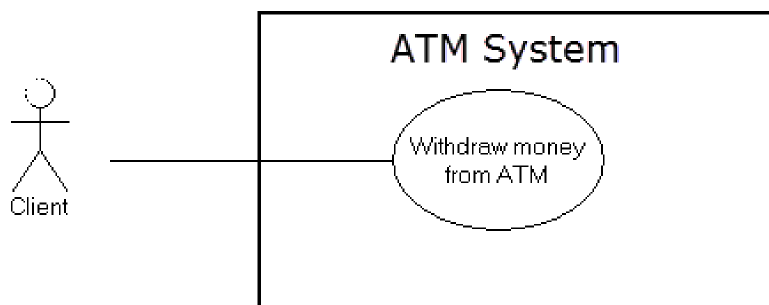

Figure 1.3: The ATM system use-case diagram with one use-case that allows a client to withdraw the money.

Piotr is one of the clients of the ATM system. He doesn't know about the defect and he will try to withdraw 100 DKK from the system (see fig. 1.4).

During the consistency checking between the sequence diagram representing the scenario (in fig. 1.4) and the behavioral state machines (in fig. 1.2) we will create a new sequence diagram of the scenario realization and we will be able to detect the defect (an inconsistency) in the ATM system. The scenario realization is presented in fig. 1.5.

The scenario was simulated by first calling *insertCard* in the *atm20* instance (see fig. 1.3) that was in "idle" state, which resulted in another call to *bank* instance: *isCardValid* that returned `true`. Piotr then tried to call the *verifyPIN* operation in the *atm20* instance. The instance couldn't handle the call at that time because when *Piotr* tries to call *verifyPIN*, the behavioral state machine for the instance *atm20* is in "verified" state and there is no trigger on any outgoing transition for the operation *verifyPIN*

The defect in the ATM system is the wrong behavioral state machine (BSM) for the *ATM* class that does not accept the *verifyPIN* operation. The client could withdraw the money without providing a PIN number (not a good behavior). This is an inconsistency because the trace of the successful scenario from fig. 1.4 cannot be realized by execution of the behavioral state machines. This inconsistency is detected by looking on both the sequence and the state machines diagrams and generating the realization of the scenario. The conclusion is that the given scenario could not be realized in the given model of the ATM system.

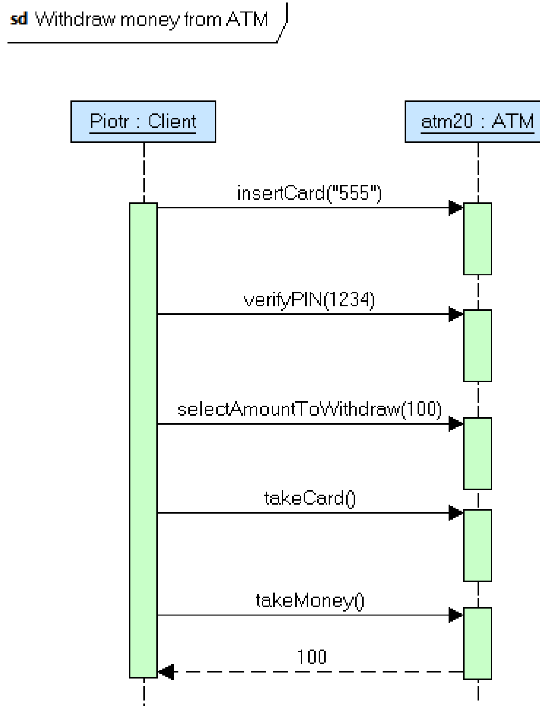

Figure 1.4: The successful scenario of *withdraw money from ATM* use-case. Piotr is one of the clients of the ATM system and the credit-card (with no. 555) holder and he wishes to withdraw 100 DKK.

1.1.1 Aim of the thesis

The aim of this thesis is to describe the approach to the automatic consistency checking between the UML (version 2.2) interactions (understood here as the sequence diagrams) and the state machines diagrams. The presented consistency checking technique includes check of behavioral parts of the model and also structural parts. We aim mostly for detecting semantic inconsistencies. We check model-independent properties (that are defined regardless to a particular model instance) as well as model-dependent properties. The described consistency checking technique can be applied to models of the software systems that are based on the use-cases [Jac92].

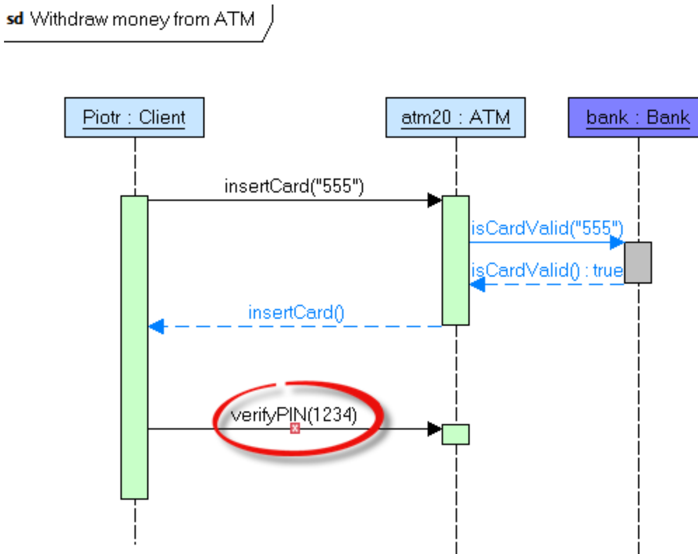

Figure 1.5: The realization of the scenario from fig. 1.4. *Piotr* is not able to send *verifyPIN* message. The elements added during the realization (the actual behavior that was executed internally in the system) have distinct colors.

We divide the interactions into two classes:

1. The sequence diagrams defining *the scenarios of use cases* (see fig. 1.4).
2. The sequence diagrams defining *the realizations* of the scenarios within a system (see fig. 1.5).

We propose a consistency checking approach that simulates the use-case scenarios in the UML models and extend them into realizations of the scenarios by adding missing fragments of the sequence diagram that show the actual internal execution of the system[1]. During the simulation, consistency of the models is checked by examining that a sequence diagram can be correctly extended with respect to an actual state machines execution and the simulated scenario can be correctly realized in a system.

The UML models that we use in this approach must be specific enough for

[1] Please, compare it with the elements that have distinct colors and were added during the realization in fig. 1.5

the purposes of model execution. No implementation of a designed system is needed.

In order to get better idea of what the *extension* means, another example of it is presented in fig. 1.6. This time the scenario is consistent the execution of state machines. In the example, the left sequence diagram is the scenario that was extended by the behavioral state machines execution to the right sequence diagram of the realization of the scenario.

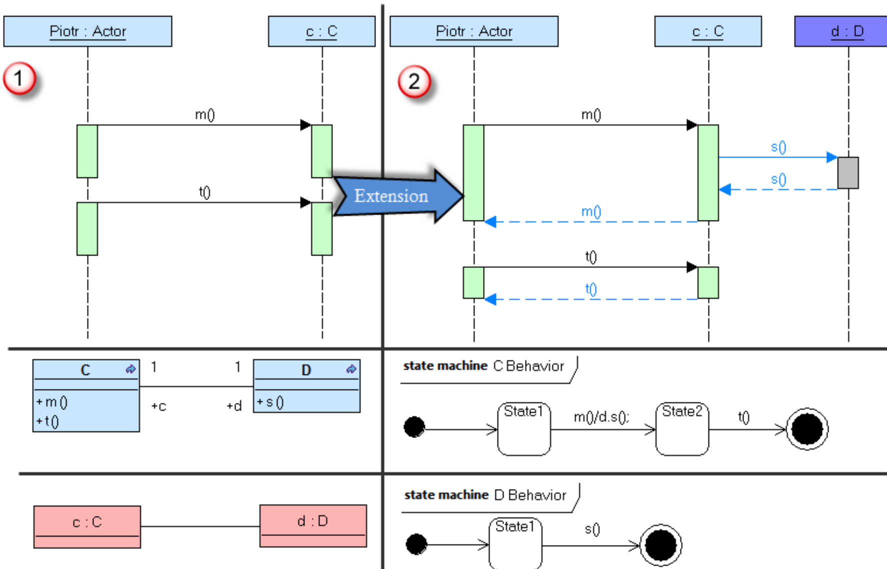

Figure 1.6: The example of the extension of the scenario to realization of this scenario. Marker no. 1: scenario. Marker no. 2: realization of the scenario. The elements added during simulation in the realization have distinct colors.

Our approach addresses the most common inconsistencies that can be found in the model when using the design approaches: Model Driven Architecture (MDA) and Domain-Driven Design [Eva03]. We use methodology used in *System Integration* course at DTU in order to create a tool for consistency checking that facilitates the design of models in graphical framework that helps to find design problems in models.

The target groups that will benefit from our tool are students and teachers of *System Integration* course at DTU. The tool can be also used by other groups of people willing to ensure consistency of their UML models.

1.2 Structure of the thesis

The positioning of this work together with the related work is described in section 1.3. Chapter 2 describes selected elements of UML that are used later in the thesis. Chapter 3 contains the inconsistencies in the UML models that we will address in the presented approach.

A case-study of the toll system is used for the presentation of the approach. The case-study is presented in chapter 4 while the approach itself is described in chapter 5. The presented consistency checking algorithm is divided to three sub-algorithms: the sequence diagrams extension algorithm described in section 5.6, the execution of behavioral state machines algorithm described in section 5.7 and the verification of the protocol state machines (PSM) algorithm described in section 5.9.

The tool that implements the consistency checking algorithm is presented in chapter 6. Finally, chapter 7 contains conclusions and possible future work.

It's important for the reader to read this thesis from the beginning to the end. The thesis is written in a way that the subsequent sections build on each other by adding new information to a reader's vision of a problem and a solution. In the end of the thesis, the reader should have the complete vision of the problem and the solution. The reader is also encouraged to look at appendices at any moment during the reading.

1.3 Related work

Consistency checking of UML models has a long history of research. There are many different types of consistency checking techniques described, however, we can categorize them into two groups.

The techniques in one group use a transformation of the UML models (that are considered high-level specifications) [VVP00] to an intermediate representation (based on UML extension or other representation) before the actual checking. These approaches use algebraic approaches [Tsi00] or model checking [TMH08, KTM08] as verification techniques.

The second group uses simulations [Ger05] or other algorithmic approaches to check the models [LTY03]. These approaches work directly on UML models needing no intermediate representation. The approach presented in this thesis

belongs to this group.

More sophisticated schema of categorizing the consistency checking techniques is described in survey by Usman [UNKC08] and splits the group using the intermediate representations into two sub-groups based on the type of the intermediate representation: one group defining the intermediate representation in a formal language and other defining it as an extension of the UML itself.

A more recent systematic literature review in this research area is presented in [LMT09]. It defines more detailed criteria of evaluating the consistency checking with respect to the UML version supported, the possibility of an extension by a modeler, the integration with the CASE tool, the types of the diagrams supported, the types of the consistency supported, the type of a (formal or informal) technique used and the paradigm used.

It is significant to note that only 1.6% of the papers reviewed in [LMT09] used the UML version 2.0 which was adopted by the Object Management Group (OMG) in 2005 (and the review is from May 2009). The rest of the papers use the UML 1.x. There is also the same low rate of the approaches supporting CASE tools. In comparison, the approach described in this thesis is based on the UML 2.2 meta-model and supports CASE tools.

Following are some examples of consistency checking techniques that help to position this work. We will see how this approach differs from other approaches and why the presented approach is better for specific applications.

The technique presented in this thesis differs from other presented approaches by extending the sequence diagrams representing the scenarios to the realizations of scenarios during the consistency checking.

Attributed Typed Graphs and their Transformation

In the approach by A. Tsiolakis presented in [Tsi00] UML class diagrams are transformed into the attributed typed graphs and sequence diagrams into the attributed and typed graph grammars. The algorithm checks the existence, the visibility and the multiplicity of the classes used in the sequence diagrams. The checking algorithm uses the graph morphisms to check the compatibility of the graphs and the grammars.

This approach does not take into account that a modeler typically works with an instance of a meta-model. The number of checks is quite limited and the

checks are related to the differences between the static (classes) and dynamic (interactions) model elements.

Appending Constraints Information to Sequence Diagrams

In another approach by A. Tsiolakis presented in [Tsi01] UML class diagrams and state machines diagrams are analyzed with respect to a particular sequence diagram. The constraints (representing the properties specified in the analyzed diagrams, e.g. data invariants and multiplicities) are attached to the lifelines in certain positions between the sending and receiving message points. Inconsistencies are identified by formally checking in *enriched sequence diagram* if some of the locations in between such points describe the system states that do not conform to the resolved constraints.

In this approach it is possible to generate pre- and post-conditions for a sequence diagram. This approach takes into the consideration that sequence diagrams can be incomplete; however it treats it equally with inconsistent specifications and does not try to complete the diagram. This approach was not implemented according to [Tsi01].

Instantiable Petri Nets

In an approach by Y. Thierry-Mieg et al. [TMH08, KTM08] behavioral parts of the model are checked. This includes activity diagrams and state machines. The model is, first, transformed to an Instantiable Petri Net (IPN) and then checked by model checking techniques. Only model-independent properties are examined. The reachability of states and unbounded behavior are the examples of the checks that can be performed in this technique. This approach is implemented in the tool *Behavioral Consistency Checker*[2] which is based on Eclipse Modeling Framework (EMF). In the approach, the sequence diagrams are not regarded.

Colored Petri Nets

In a similar approach to consistency checking by Y. Shinkawa [Shi06] Colored Petri Nets (CPNs) are used. This approach is defined based on use-case driven development of the models. The use-case, class, sequence, activity and state

[2] http://move.lip6.fr/software/BCC/

machine diagrams are regarded for checking. The sequence diagrams are checked but the functionality present in our approach of the extension of use-cases to the realizations of scenarios is not in place in this approach.

Finite State Processes and Messages Traces

In an approach by H. Wang et al. [WFZZ05] the dynamic parts of a model are checked. The sequence diagrams are validated against state machines. Intermediate representations are used: Finite State Processes for the state machine diagrams and the messages trace for the sequence diagrams. The intermediate representations are checked by the LTSA model checker. The checks validate the order of messages being sent and received. This approach does not produce the realizations of scenarios from the sequence diagrams representing the scenarios.

`Vooduu` tool and approach using UPPAAL

An approach by K. Diethers [DH04] provides consistency checking between UML state machines and sequence diagrams. The verification uses an intermediate representation of timed automata that is then analyzed in the UPPAAL model checker. The results are transformed back into a sequence diagram. This approach focuses on checking of violation of timing conditions of timed events. Other checks are available, e.g. incorrect message detection based on a sender and a receiver and violation of loop conditions detection. A tool was implemented as a plug-in for the *Poseidon for UML*[3] (the name of the tool is *Vooduu*).

This approach is promising although not well suited for non-deterministic examples of models (with many possible traces of the execution, the inter-dependencies of the clocks and the high number of the variables) due to the state explosion problem.

Mapping to CSP domain

In an approach by G. Engels [EKHG01] UML models (behavioral and protocol state machines diagrams) are checked by first mapping it to the *CSP semantic domain*. Then, using the FDR model checker[4], consistency rules are evaluated.

[3] `http://www.gentleware.com/`

[4] `http://www.fsel.com/`

This approach in the current version is limited to the state machines (although, as authors claim, it could be defined for more diagram types). This approach does not extend sequence diagrams.

OCL Rules Approach

An approach by R. Dubauskaite et al. in [DV10] is a consistency checking technique based on OCL rules. The authors discuss the openness of the approach for a modeler who can easily add the new rules to the checker (by writing them in OCL). The rules check for the specific elements that are defined in the model, e.g. if a representant of a lifeline is defined in the model. The tool is implemented as the MagicDraw plug-in.

This approach mostly regards consistency between behavioral and structural model parts and cannot be applied to inter-behavioral analysis.

The `ViewIntegra` Approach

An approach by A. Egyed [Egy01] uses a transformation between different diagram types for consistency comparisons (no use of intermediate representations). Moreover, 10 different transformations are defined between 11 diagram types and as a result some comparisons require the series of transformations to be executed. By comparing always the same representations, consistency checking and the consistency rules are simplified.

In this approach, sequence diagrams are ultimately interpreted as class diagrams for comparison. It suggests that consistency rules mostly regard the structural features and the behavior is lost during the transformations. The tool has been implemented for selected transformations.

Behavioral Validator of UML

In the approach by B. Litvak et al. in [LTY03] simulations are conducted to ensure the consistency of the sequence and state machines diagrams. The messages order is checked and a type of sent messages is checked. The simulation supports the advanced structures in the state machines diagram, e.g. the forks, the joins, the concurrent states and the composite states. If none of the state machines have defined the trigger for a message in the sequence diagram, it

is considered an error in this approach. The tool is implemented based on ArgoUML[5].

The approach is promising and similar to the approach presented in this thesis in the aspect of using simulations to ensure consistency and in the way the tool can help designers to find problems in the UML models. The tool, however, does not add realizations of the scenarios to the sequence diagrams.

TestConductor and Live Sequence Charts

It is also important to mention an approach for building the systems using the *play-in, play-out* algorithm described in [HM03]. This approach enables a user to design the systems based on the behavioral parts of the model by declaring the interactions in the Live Sequence Charts (LSCs) notation. The *TestConductor* is a tool developed by Rhapsody [Ger05] that checks the interactions in the LSCs by running the simulation and sending the messages on behalf of the environment, when required. After the simulation, the resulting interaction is compared with the original use-case. This approach is partially similar the one presented in this thesis but the extension of the use-cases sequence diagrams to the realizations of the scenarios is not realized here.

Automated Translation from Sequence to State Machines Diagrams

In an approach by S. Sengupta et al. [SKB05] consistency is preserved by automatic generation of the state machines diagrams from the set of sequence diagrams. This approach includes the analysis of OCL constraints in the sequence diagrams (the pre- and post-conditions).

This work is related to this thesis because it uses the opposite approach for generation of the new fragments of the model (from interactions to state machines). In the approach presented in this thesis, the sequence diagrams are completed based on the state machine diagrams.

[5] `http://argouml.tigris.org/`

Chapter 2

Selected elements of the Unified Modeling Language

The consistency checking approach presented in this thesis is based on input of a UML model. In the following chapter we will introduce a description of selected UML elements and notations that are used in the rest of the thesis. Firstly, we introduce the UML as a modeling language, then we describe classes and components, (behavioral and protocol) state machines and interactions. We do not only describe the corresponding UML diagrams but also on the meta-model of the corresponding language units[1]. It is important to note that the presented description is not a complete description of the UML but rather a short description of the selected elements and features that are used later in this thesis. For more information on the usage of the UML please consult the book by Rumbaugh et al. [RJB04]. For in-depth treatment of the meta-model issues, please see the UML 2.2 Superstructure Specification by the Object Management Group (OMG) [Obj09] .

2.1 UML as modeling language

The Unified Modeling Language is a graphical language for creating graphical visualizations in a standard way that helps to understand a modeled system. The UML may be applied to many types of systems, e.g. software applications,

[1] The language unit is a collection of highly coupled concepts in UML.

distributed web applications, databases, business processes, real time systems. The modeling in the UML is important because it helps communicating ideas to other people: these people are other engineers or clients of a development company.

The UML is a very expressive language allowing different views on a system. Different types of UML diagrams are used to show these different views. These views are together enough to capture all types of the systems mentioned before. In the UML 2.2, a system can be modeled with fifteen types of diagrams. The diagrams' hierarchy is presented in fig. 2.1.

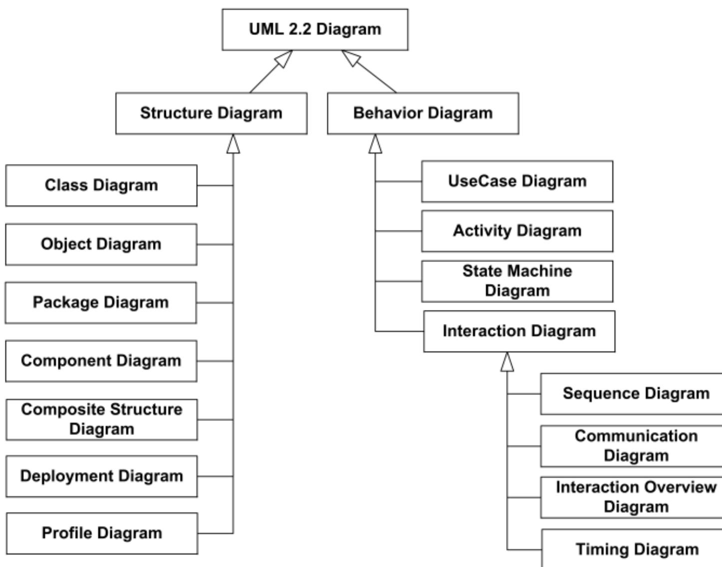

Figure 2.1: Hierarchical view on the types of diagrams in the UML version 2.2.

The UML language creates a basic vocabulary that can be used and shared between clients and developers, students and teachers, etc. The vocabulary is defined in terms of names of UML elements.

The UML models are also used for the forward engineering (source codes can be generated from them). The UML model can be represented in a programming language such as Java, C++, and Visual Basic.

2.2 Semantics Variation Points

A wide range of applications of the UML is caused by the UML being flexible notation, but, on the other hand, being also not precise about some of the semantics and, in particular, some of the run-time semantics[2]: it is manifested by a presence of number of *variation points*. The variation points are explicitly identified in the UML specification to provide a leeway for domain-specific purposes. The variation points become a more important issue when subjected to model-to-code transformations or, is our case, direct model's executions.

2.3 Classes and instances

Class describes the "set of objects that share the same attributes, constraints, relationships, operations and semantics" [RJB04]. Classes are used to show the classification of objects in a modeled system. A class shall have a unique name. The uniqueness of the name must hold within a package in which the class in placed (the enclosing package).

2.3.1 Properties

Classes may have any number of properties. A *property* is a structural feature related to a classifier (and, therefore, to a class)[3].

There are two possible types of properties:

attribute is a named property of a class that describes the possible values that instances of the property may hold;

association's end is one end of an *association* and describes a semantic relationship that can occur between typed elements (e.g. between two classes).

[2]The run-time semantics is a mapping of the modeling concepts into the program execution phenomena.

[3]In the UML meta-model, `Class` inherits from `Classifier`.

2.3.2 Operations

Classes may also have operations. An *operation* is a behavioral feature for invoking a behavior associated to the owning class. The name *operation* is mostly used in the UML in a context of an interface (see section 2.3.5) while the word *method* is used for operations in classes implementing an interface. A *signature* of an operation can include its name, visibility, names and types of parameters, default values of the parameters, and its return type.

2.3.3 Super-class

Classes may have unlimited number of super-classes. Super-classes are the more general classes (parents) connected to a more specific class (child) with the *generalization* relationship. A child may add a new structure and a behavior or to redefine a behavior of its parent.

2.3.4 Abstract class

Classes can be *abstract*, i.e. they do "not provide a complete declaration and cannot be typically instantiated" [Obj09]. Abstract classes are normally parents of other classes.

2.3.5 Interface

An *interface* is a "kind of classifier that represents a declaration of a set of coherent public features and obligations" [Obj09]. Interfaces can own a protocol state machine (see section 2.5.2) that "may impose ordering restrictions on interactions through the interface" [Obj09].

Classifiers can have *provided* and *required* interfaces. Provided interfaces are the ones that a classifier realizes (they are connected to a classifier with *interface realization*). Required interfaces are used by a classifier, i.e. there is a *usage* relationship between a classifier and an interface.

2.3.6 Instance Specification

Instance specifications represent a particular instance of a classifier. Instance specifications can contain slots that represent the instances of attributes (or, more generally, features) and can hold values. The instance specification of an association is called *link*.

Instance specifications can be presented in an object diagram that shows the run-time state of a system's instance.

2.3.7 Class and object diagrams

A graphical representation of a class is a rectangle with the class's name shown. Optionally, inside the rectangle, compartments with the class's attributes and operations can be shown. Abstract classes are recognized by italics used in their names. The example of class and object diagrams are presented in fig. 2.2.

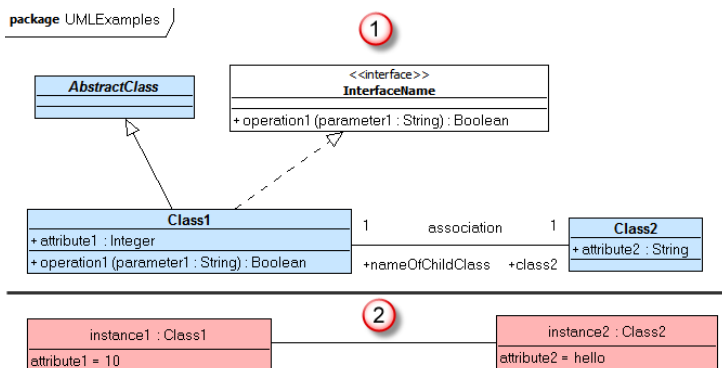

Figure 2.2: Marker no. 1: the example class diagram. Marker no. 2: the example object diagram with instances of classes.

2.4 Components

A component is a concept that describes a "modular unit with well-defined interfaces that is replaceable within its environment" [Obj09]. An important aspect of the component is self-containment, i.e. the component "encapsulates the state and behavior of a number of classifiers" [Obj09].

2.4.1 Ports

Interfaces of components can be exposed via ports (optionally). A port is a "property of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts" [Obj09]. A port is like a "bridge" between internal parts of a component and its external environment.

2.4.2 Component diagram

Components are represented as rectangles containing a characteristic symbol in an upper right corner. Ports are represented as small squares on a border of a component. Required and provided interfaces are presented in a lollipop notation. See fig. 2.3 for example of a component diagram.

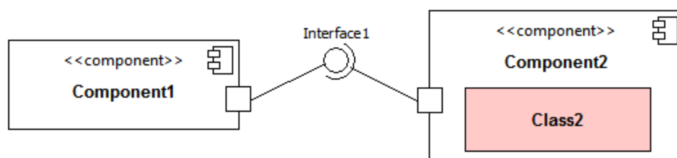

Figure 2.3: The UML component diagram with two components. *Component1* provides an interface to *Component2*. *Class2* is inside *Component2*.

2.5 State machines

There are two types of state machines in the UML: behavioral and protocol state machines. In this section, we will shortly describe both of the types.

2.5.1 Behavioral state machines

Behavioral state machines (BSM) are "used for modeling discrete behavior through finite state-transition systems" [Obj09]. A BSM can be defined in context of a class and, by that, define the class's behavior.

> The behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences. During this traversal, the state machine executes a series of activities associated with various elements of the state machine. [Obj09]

2.5.1.1 Region

Every state machine should have one or more regions in which states and transitions are defined. Region(s) can also be defined in a state.

2.5.1.2 Transition

Transitions are directed relationships between two vertices: a target and a source vertex[4].

A transition can have:

- A number of *triggers* that fire the transition (a trigger is associated to an event, e.g. a call event of an operation).
- An *effect* that is executed during a traversal (firing) of the transition. The UML does not specify a language in which effects can be expressed textually; it allows a designer to choose this language.
- A *guard* that "provides a fine-grained control over the firing of the transition. The guard is evaluated when an event occurrence is dispatched by the state machine. If the guard is true at that time, the transition may be enabled; otherwise, it is disabled." [Obj09]

A special type of transitions is *completion transitions*. Completion transitions do not have any triggers and are enabled during a *completion event* and after entry actions and internal activities of a state are completed. The completion event is "generated upon entering the state." [Obj09]

[4] Vertex is a common super-class for all states and pseudo-states in UML meta-model.

2.5.1.3 State

A state "models a situation during which some (usually implicit) invariant condition holds." [Obj09]

States are divided into two main types:

- *Simple states* that do not have sub-states
- *Composite states* that "either contain one region or are decomposed into two or more orthogonal regions." [Obj09] That implies that a composite state can have many sub-states in its regions.

2.5.1.4 Pseudostate

Pseudostates represent different type of vertices in state machines. Selected types of pseudostates are described below:

initial pseudostate "represents a default vertex that is the source for a single transition to the default state of a composite state" [Obj09] or a state machine;

junction pseudostate is used to chain together multiple transitions;

choice pseudostate, "when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions." [Obj09]

2.5.1.5 Active states configuration

Active states configuration of a state machine is a set of states that are active (i.e. "entered as a result of some transition" [Obj09]) in the state machine. If a state that is active is placed in a composite state, the composite state must also be active (and must be placed in the active states configuration). That's why we can see the active states configuration as a set of trees of states "starting with the top-most states of the root regions down to the innermost active sub-state." [Obj09]

2.5.1.6 State machines diagram

The example state machine diagram is presented in fig. 2.4. Guards are placed inside brackets and effects are always preceded by character "/".

When the trigger for *operation1* is firing the transition, there is a choice based on the *parameter1* value. If the value of *parameter1* is equal to "hello", then the effect on the transition assigns *attribute1* value 15 and replies `true` to a caller, otherwise no assignment is made and replied value is `false`.

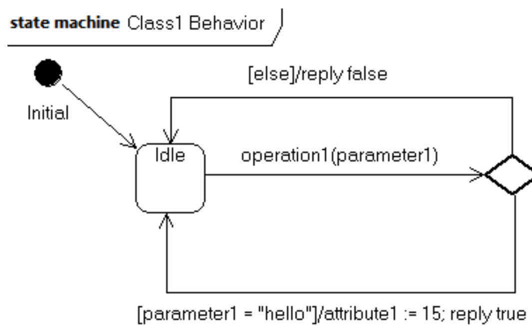

Figure 2.4: The example of a behavioral state machine diagram.

2.5.2 Protocol state machines

Protocol state machines (PSMs) are always defined in a context of a classifier. It includes the situation when a PSM is defined in context of an interface. PSMs "specify which operations of the classifier can be called in which state and under which condition, thus specifying the allowed call sequences on the classifier's operations." [Obj09]

States of PSMs are the same states used in BSMs (see section 2.5.1.3) but additionally can have state's invariants that "specify conditions that are always true when this state is the current state." [Obj09]

2.5.2.1 Protocol transition

Transitions in PSMs can have a pre-condition, a post-condition and a referred operation. No effects are allowed on protocol transitions.

For a protocol transition:

Pre-condition is a "condition that should be verified before triggering the transition." [Obj09]

Post-condition is a "condition that should be obtained once the transition is triggered." [Obj09]

Referred operation is an operation that, when called, triggers the transition.

All unreferred operations (for which there are no protocol transitions) in PSMs "can be called for any state of the protocol state machine, and do not change the current state." [Obj09]

2.5.3 Protocol state machine diagram

A PSM diagram is visible in fig. 2.5. Pre-conditions are visible in brackets, post-conditions in brackets preceded by character "/".

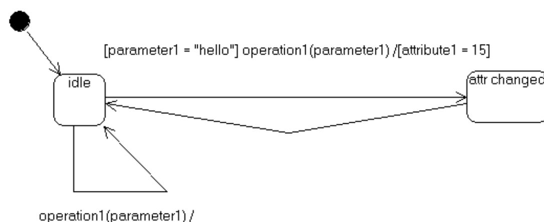

Figure 2.5: The example of a protocol state machine diagram.

2.6 Interactions

Interaction can be seen as a possible valid (or invalid) trace in a system. The trace is described as a "sequence of event occurrences." [Obj09]

Specializing an interaction is "to add more traces to those of the original. The traces defined by the specialization are combined with those of the inherited interaction with a union." [Obj09]

2.6.1 Lifeline

Lifelines represent "an individual participant in an interaction." [Obj09] E.g. a lifeline can represent an actor or an instance specification. The order of occurrences on a lifeline is significant and denotes the order in which these occurrences will occur.

2.6.2 Message

Messages are used to define a "particular communication between lifelines of an interaction." [Obj09] This communication can be, e.g. calling an operation on an object represented by a lifeline. Messages can have arguments that can be used as arguments of a call of an operation.

There are few messages sorts:

synchCall that represents a synchronous call to an operation;

asynchCall that represents an asynchronous call to an operation;

asynchSignal that represents sending of a signal;

deleteMessage that represents termination of a target lifeline;

reply that represents reply for an operation call.

Found message is a message of which sending event is not known (i.e. it does not come from any lifeline). *Lost* message is a message of which receiving event is not known (i.e. it is not targeted to any lifeline).

2.6.3 Behavior execution specification

Behavior execution specifications are kind of execution specifications "representing the execution of a behavior" [Obj09] within a lifeline. An execution specification has the start and finish events occurrences that designates when the execution starts and finishes.

2.6.4 Sequence diagram

An example sequence diagram showing an interaction is presented in fig. 2.6. The interaction consists of two lifelines and two messages. One of the messages is synchCall message to *operation1*. Reply for this operation is `true` (it is visible on the reply message). The two green boxes on the lifelines are the behavior execution specifications.

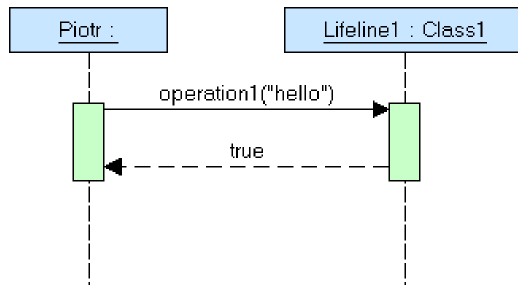

Figure 2.6: The example of a sequence diagram.

Chapter 3

Inconsistencies

In this chapter we will describe inconsistencies that we would like to check in our approach. The inconsistencies are divided into groups to help to understand what they are mostly related to.

3.1 Sequence diagrams and structural properties of model

Inconsistencies described in this section can be detected by an analysis of a structure of a model. They must be enforced before some of semantical checks can be done. They are also relatively easy to detect in the model.

3.1.1 Lifeline representant

Lifelines represent, in this approach, instances of actors, classes or components[1]. If a lifeline representant's type is a class, the class should have defined behavior

[1] The type of lifeline's representant has changed in UML 2.0 to be of type *connectable element*. This implies that instance specifications cannot be directly cross-referenced as representants. The solution is to set a lifeline's representant to type of an instance specification and to use the lifeline name identical to the name of the instance specification. We have chosen this way to identify which lifelines in sequence diagrams represent which instances specifications in a model.

as a behavioral state machine. The representant's type must not be abstract class (it could not be instantiated). Example of this type of inconsistency where abstract class is used as a lifeline's representant is presented in fig. 3.1.

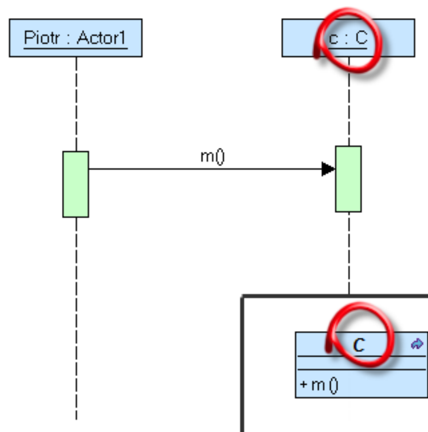

Figure 3.1: The example of call to an abstract class. The abstract class can be recognized by name in italics.

3.1.2 Message occurrence specifications

In our approach, there is a restriction that each message shall define send and receive message occurrence specifications (MOSs). By requiring that, we do not allow found and lost messages (see section 2.6.2) to occur in scenarios. A reason underlying this decision is connected to what an idea of use case is. In use cases, an environment's behavior is typically represented by an actor's behavior in a scenario of a use case; found messages coming from no lifeline or lost messages targeting no lifeline do not make sense if we want to see the interaction of the actors with the system. An example of an erroneous scenario with found and lost messages is presented in fig. 3.2.

Another restriction is for each MOS that is in *covered by* collection of a lifeline[2] in a scenario there must be the corresponding message.

[2] All events occurring on a lifeline are stored in this collection.

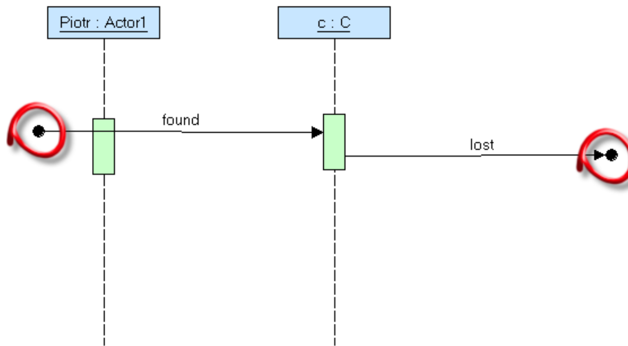

Figure 3.2: The example scenario containing the found and lost messages.

3.1.3 Behavior execution specifications

A consistency check of behavior execution specifications (BESs) is based on their meta-model constructs.

- BES must have *start* and *finish* elements specified.
- If *start* and *finish* elements are not the same fragment, start must occur before *finish* on the lifeline of BES.

It is difficult to show an example diagram with the inconsistencies violating these rules because the constrained elements are not directly visible in diagrams. Even though the elements cannot be directly visualized, the enforcement of these rules is crucial to ensure consistency of order of interaction fragments on lifelines in a model representation.

3.1.4 Call message and target class

If a message representing an operation call is present in a scenario, a type that represents a target lifeline has the operation that is used in the call. The type may declare or inherit the operation from a parent. An example of calling an operation not declared in a target class is presented in fig. 3.3.

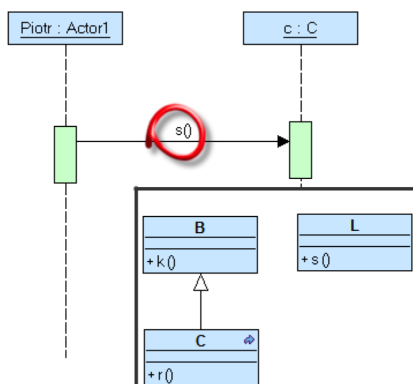

Figure 3.3: The example of call of an operation that is not declared in a target class.

3.1.5 Call message arguments

Arguments of a call message must conform to parameters of a called operation, i.e. types of the arguments must be identical or can be implicitly converted to the parameters' types (e.g. child classes can be used as parents' classes). Lower and upper multiplicities of the parameters must conform to a number of values in the arguments. An example of an inconsistency where a parameter of a call of an operation is expected to be a *String* type but an actual argument is *Integer* is shown in fig. 3.4.

3.1.6 Visibility of operations

Called operations must be visible to callers. *Private* and *protected* operations in a class shall not be called by instances of different classes. An example of this type of inconsistency is shown in fig. 3.5 where an actor calls a *private* operation.

In case of *package private* visibilities, callers should be in the same package with called types.

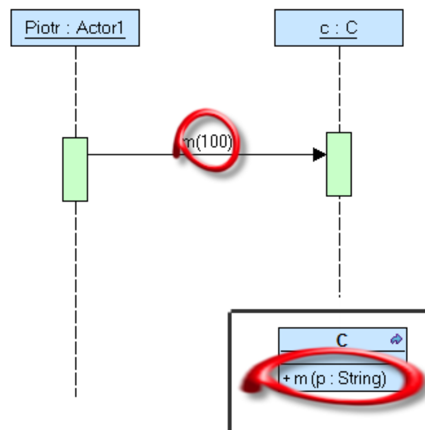

Figure 3.4: The example of an inconsistency between a type of an argument and a type of a parameter.

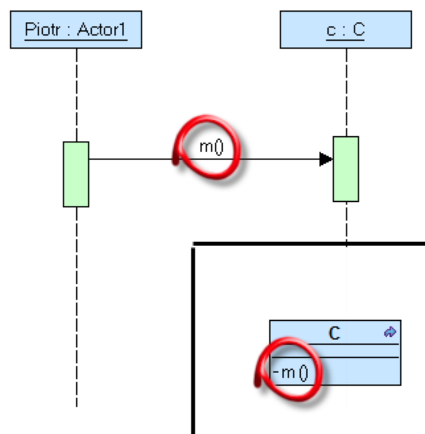

Figure 3.5: The call to a private operation. Private operations are recognized by the minus sign.

3.1.7 Multiplicity

A multiplicity[3] check must ensure that number of instances connected with links in object diagrams (and represented by lifelines in sequence diagrams) conform

[3]The multiplicity used to be also called *cardinality* in the older UML versions.

to associations' ends multiplicities. An example of an inconsistency between a number of links and a multiplicity of an association's end is presented in fig. 3.6. In the example, class C is connected to class D with the association. The multiplicity of the association's end for D constraints a size of the collection from zero to two elements, but in the scenario and in the object diagram there are three instances of D connected to the instance of C.

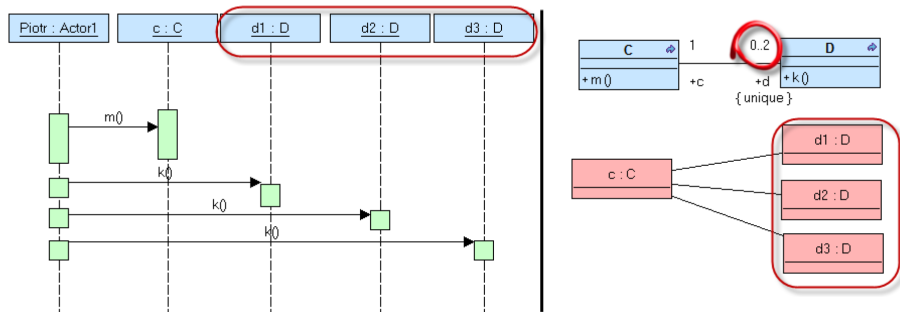

Figure 3.6: An inconsistent number of links with respect to an association's end multiplicity.

3.2 State machine diagrams and structural properties of model

This section describes inconsistencies that must be checked in order to ensure a well-formed state machine. We must have the well-formed state machine if we want to conduct semantical checks.

3.2.1 Region existence

A state machine shall have at least one region. This restriction is implied by the fact that a state machine without regions cannot execute.

3.2.2 Initial pseudo-state

In each region (including regions of state machines and regions of composite states), exactly one initial pseudo-state must be defined. Moreover, there must

be exactly one transition outgoing from the initial pseudo-state. The transition does not have any triggers. An example of an inconsistent region with a transition having a trigger defined and outgoing from an initial pseudo-state is presented in fig. 3.7.

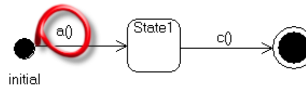

Figure 3.7: A transition having a trigger defined and outgoing from an initial pseudo-state.

3.2.3 Final state

Final states shall have no outgoing transitions. The outgoing transitions would never be taken because, upon entering a final state, a state machine (or an orthogonal region) terminates.

3.2.4 Miracle state

Miracle states have at least one outgoing transition and no incoming transitions. There is no way to enter a miracle state during a state machine execution and thus an existence of it may be sign of a design error. A presence of the miracle states is one of the classical inconsistency types in state machines. An example of a diagram with a miracle state is presented in fig. 3.8.

3.3 Other structural inconsistencies

3.3.1 Defining features of slots in instance specifications

Each slot in an instance specification must be defined by feature that is set to an attribute of the class that is a type for this instance specification. This rule must be enforced in order to obtain a well-structured model of instances that represent objects of their classes.

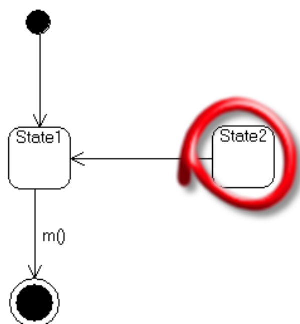

Figure 3.8: The miracle state, i.e. a state that has some outgoings transitions and no incoming transitions.

3.3.2 Defining values in slots

Types and multiplicities of values in slots should conform to the slots' types and multiplicities. If a value is an instance value, i.e. it represents object of a class and not a primitive type, then an instance specification's classifier referred by the instance value should conform to the slot's type. In a special case in which the slot's type is an interface, the referred instance specification must be of type of a classifier that realizes the interface.

3.4 Behavioral state machines and structural properties of model

Inconsistencies described in this section are connected to a behavior of state machines and its conformance to structural parts of a model.

3.4.1 Missing association / link

If a BSM (representing a behavior of one object) call other object, there must be an association between the types of these two objects (usually classes). There must be also links between these two objects' instances. The called object must be accessible from the caller object.

An example presented in fig. 3.9 shows a scenario in which object c is called by an actor. Object c then tries to call an instance of class D that exists in a system. However, there is no link between objects c and d that would allow this call to be executed. Moreover, there is no association between class C and class D. This is clearly an inconsistency.

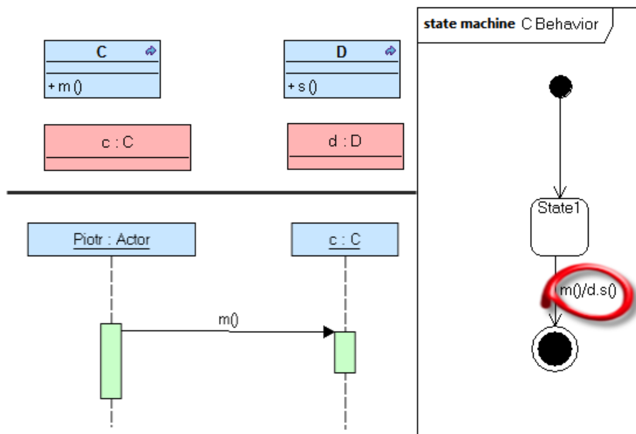

Figure 3.9: The object of class C tries to call the object of class D. The objects are not connected to each other with a link.

3.4.2 Missing reply

Operations with a declared result must return the result. When such operation is synchronously called in a BSM, it initiates run-to-completion step (see section 5.7.1). A path in the BSM that was taken during that step must send back reply with the result of the operation to a caller.

An example of this inconsistency is presented in fig. 3.10 where a state machine does not have reply for a trigger of *m()* despite m is an operation that returns `Boolean` result.

3.4.3 Reply conformance

Operations that are declared to return results of given types and multiplicities must return values that conform to the declarations.

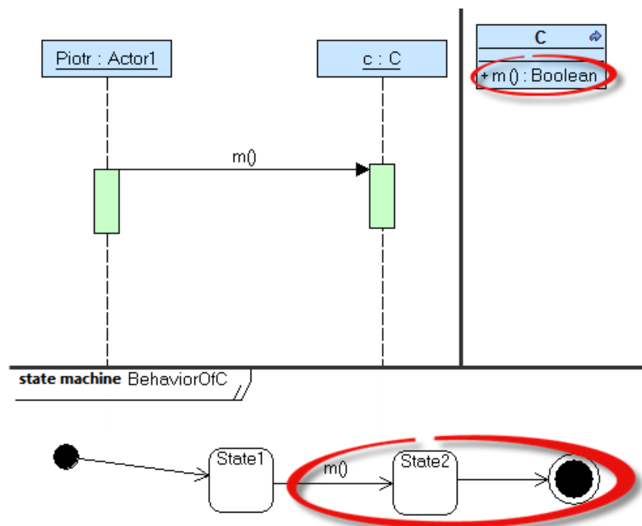

Figure 3.10: A reply for the call of operation *m* is missing in the behavior state machine of *C* despite the `Boolean` result that is declared in *C* class for this operation.

An example is presented in fig. 3.11 where a value returned as a reply for *m()* is of a wrong type.

3.5 Sequence diagrams and behavioral state machines

The inconsistencies in this section are typical semantic inconsistencies detected between sequence diagrams and BSMs.

3.5.1 Order of called operations

An order of messages in scenarios given as sequence diagrams must be realizable in a system. BSMs must be able to handle call events produced by a scenario. An example of an order inconsistency is presented in fig. 3.12.

In the example, we have a synchronous call message *m()* that cannot be accepted

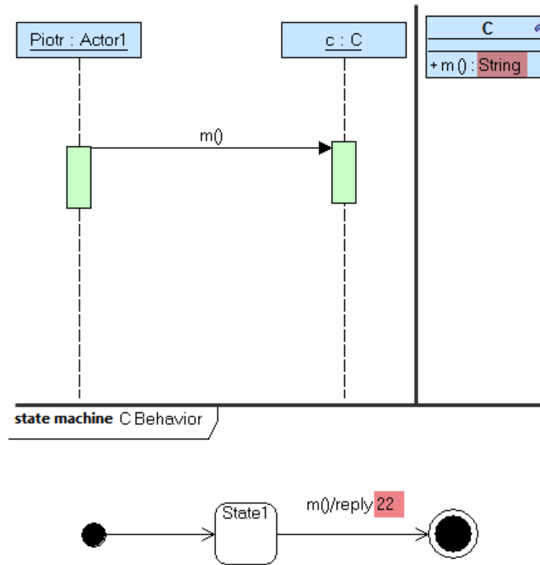

Figure 3.11: The reply for the call of operation *m()* is declared to have the type `String` but the behavior state machine of C returns `Integer` as the actual reply for the call of *m()*.

by a behavioral state machine of instance of class C (the trigger for the call event *b()* is expected) and thus the scenario is not realizable.

3.5.2 Result of external event visible to actor

If a designer constructs a scenario with an actor calling an operation in a system and he will also specify explicitly a result for the operation, then the provided result should be identical with result returned from the system during the realization.

In an example presented in fig. 3.13 we can see a scenario where a designer expects `Integer` result *777* of operation *m()* but a behavioral state machine of C replies with a constant value of *304* so this is an inconsistency. The effects on the transitions in the behavioral state machine are expressed in Simple Action Language described in section 5.8.

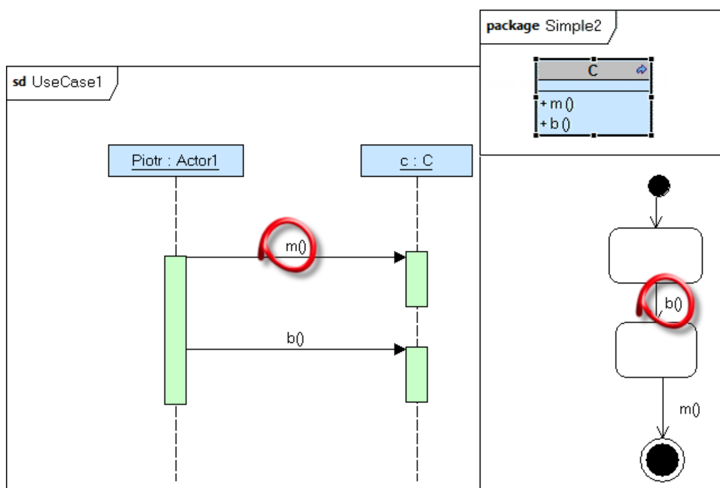

Figure 3.12: The order of the messages in the scenario's sequence diagram cannot be realized by the state machine order of triggers. We assume that the state machine of object c was in the initial state just before the scenario was started.

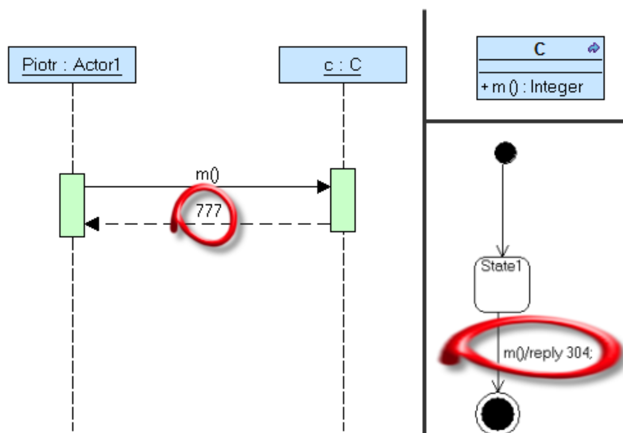

Figure 3.13: The scenario where a designer specifies an expected result of operation *m()*. In the diagram showing the behavioral state machine of C the reply for m is constant integer *304* but the actor expects to see *777*.

3.5.3 Reply for non-existing call

If there is a reply message sent in a sequence diagram but during BSMs execution there is no corresponding call event fired, it's an error.

This type of error can be easily detected by looking only at the sequence diagram, e.g. in fig. 3.14 we can clearly see that there are two replies for only one call.

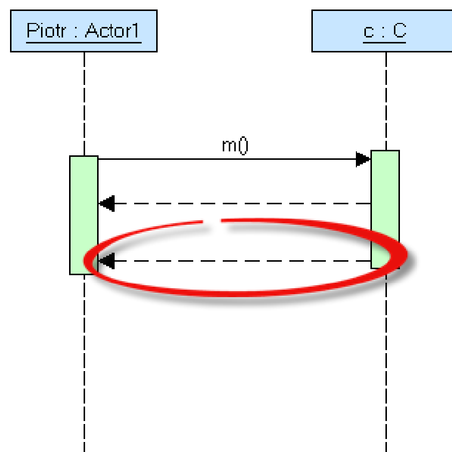

Figure 3.14: Two replies for one call of *m()* operation.

3.5.4 Not realized message

If a synchronous call message is sent according to a sequence diagram but a target BSM does not execute the call event this is an inconsistency.

A sequence diagram in fig. 3.15 shows two messages, one is called by an actor and second is internally sent between two objects c and d. After the message *m()* is called by the actor, the BSM of object c accepts it and then immediately object terminates by entering a final state. In an effect, the message *s()* is never realized by the system during the realization of the scenario. This is the inconsistency.

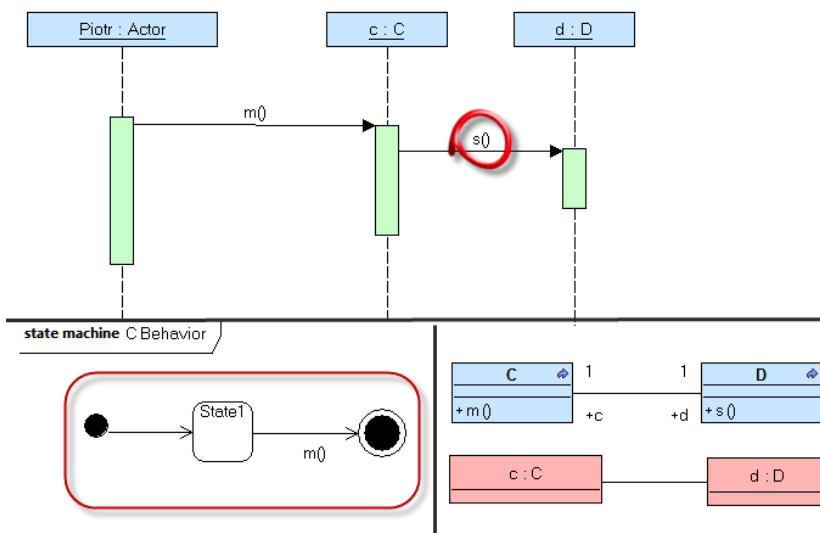

Figure 3.15: The call message in the sequence diagram that was not realized during the realization because the BSM of *c* does not call the function *d.s()* in an effect of *m()* call.

3.6 Conformance to contract specified by interfaces

This section describes inconsistencies that can be detected in a contract conformance specified by interfaces and protocol state machines defined in context of the interfaces.

3.6.1 Realizing class does not implement operation from interface

Classifiers that are connected to interfaces with interface realization relationships must conform to contracts specified by the interfaces. An enforcement of this rule include a check that checks for a set of operations declared in an interface are in fact implemented by a set of methods in realizing classes. An example of a violation of this rule is shown in fig. 3.16.

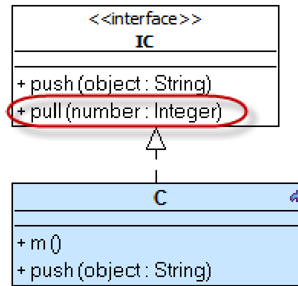

Figure 3.16: Class *C* is realizing interface *IC* but failed to implement operation *pull* from this interface.

3.6.2 An order of methods called does not conform to an order in protocols

If an interface owns a protocol state machine (PSM) then a behavior of a class realizing the interface should conform to the PSM.

One of the rules in PSM's conformance is an order in which methods are called on instances of classes that implement an interface that owns a PSM. It must be possible to realize the order in the PSM.

An example of a violation of this rule is shown in fig. 3.17. In the example scenario, *Piotr* is an actor that calls the methods in *c* that must conform to *ProtocolOfIC* protocol. The first call is to method *m()* that is not referred in the protocol and executes correctly. The next call is *push("test")* that can be accepted by *c* BSM. Unfortunately for this scenario, *push* is also operation referred in the protocol *ProtocolOfIC*. According to the PSM, it is required that *pull* shall be always called before *pull* could be called. This condition is not satisfied in this scenario.

3.6.3 Pre- or post-conditions of operations fail in PSMs

It could be the case that there exist no protocol transitions in a PSM for which pre- or post-conditions for a referred operation are true. The pre-condition must be true before the operation was called; the post-condition must be true after the operation was called. The both (pre- and post-) conditions are specified

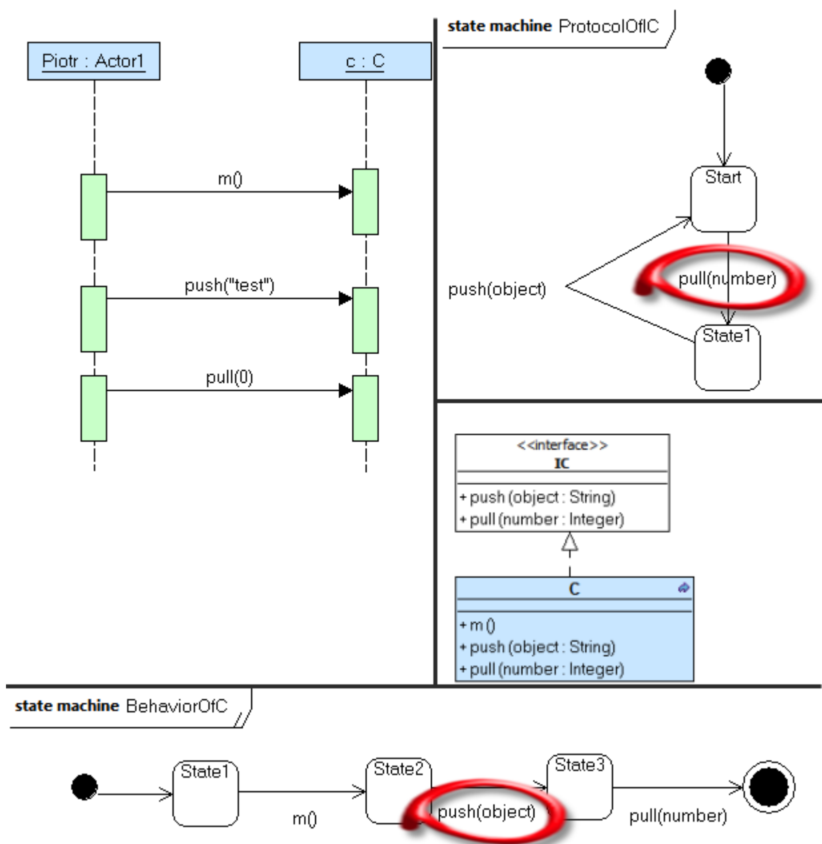

Figure 3.17: Class *C* is realizing interface *IC* that owns the protocol *ProtocolOfIC*. During a simulation of the scenario, a behavior of an instance of the class *C* does not conform to the PSM of the interface: an order of methods called is inconsistent with an order of referred operations in the PSM.

in the Object Constraint Language (OCL) in our examples. In OCL, the precondition is evaluated on a state of a system before a call and the post-condition is evaluated on two states of a system: the state before the call (pre-state) and a state after the call (post-state).

An access to both states in post-conditions gives the possibility to check for differences that were introduced in the post-state during an operation call in relation to the pre-state. If an operation returns a result, the post-state also includes a `result` variable representing a value of the result.

An example of a scenario for which a post-condition failed is presented in fig. 3.18. In the presented scenario, *push()* method in class C always returns a constant value *100*. The PSM expects *99* in the post-condition of the operation *push()*. There are no other protocol transitions in the PSM that could be taken when the result is equal to *100* and therefore the scenario is not realizable.

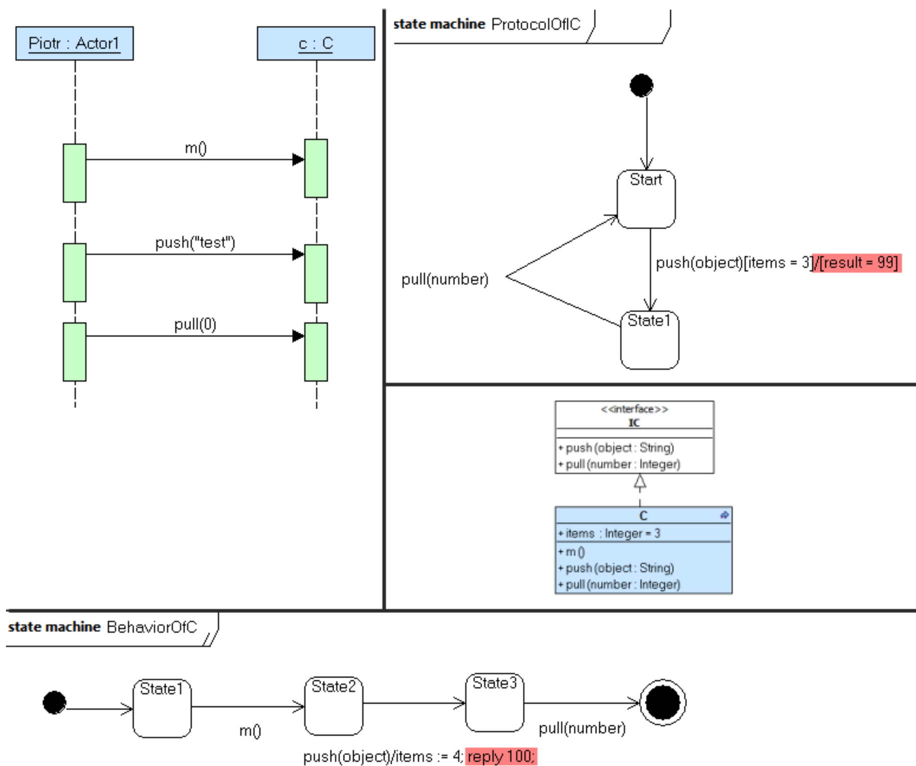

Figure 3.18: Class C is realizing interface *IC* that owns protocol *ProtocolOfIC*. During the simulation of the scenario, behavior of instance of class C does not conform to the PSM of the interface: the post-condition is not satisfied.

3.7 Related to components

3.7.1 Discussion about components

Before we go to the next checks connected to components, we will explain why the checks in components allow to refer from a component to a type outside components (e.g. in a package). In principle, it shall not be possible to refer from a class placed in one component to a typed element outside this component by an association relationship. But what if we have a complex type *Client* that is used in two or more components? In this case, all of the components must know about the type *Client*. We could duplicate the type *Client* in all of the components but then, there will be two identical types *Client* in a system. This will break "don't repeat yourself" (DRY) principle. What to do with this sort of types (like *Client*) without violating the DRY principle and an encapsulation of components?

Currently, there is no known solution to this problem in context of the UML components and the problem is not trivial to solve. For these reasons, in this thesis we will make a simplification and allow having references from contents of components to classes outside any of components. This will allow creating a type of a system presented in fig. 3.19.

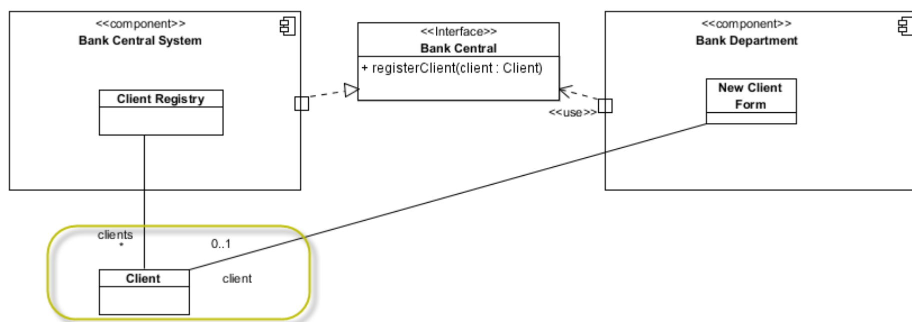

Figure 3.19: Class *Client* is a complex class that is placed outside the components in a package. The components *Bank Central System* and *Bank Department* have the classes that have the associations to *Client* class. The provided interface of the *Bank Central System* component also has an operation with a parameter of the type of *Client* class. This requires the other components that use the interface to know about the type *Client*.

3.7.2 Classes have relationships to other classes in other components

If a class is defined in one component it must not be connected with relationships to a class in another component (though it can be to class outside components). The relationships in this rule are: association, dependency and generalization. This rule supports the semantics of self-contained components, i.e. one component shall not be dependent on another one.

An example of an inconsistency is presented in fig. 3.20. In the example, two components and classes placed inside them have a relationship (generalization) that connects a class in one component to a class in another component.

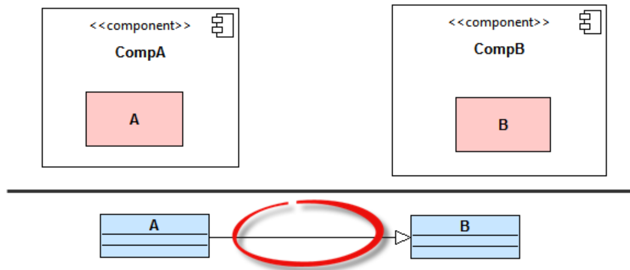

Figure 3.20: Class *A* is in *CompA* component and class *B* is in *CompB* component. The generalization from *A* to *B* that crosses between components is marked as the inconsistency.

3.7.3 Typed elements using types from other components

There shall not be a typed element in a component that uses a type that is located in another component (though it can be a type outside components). The typed elements include attributes, operations' results and operations' parameters in classes.

An example of an inconsistency is presented in fig. 3.21. In the example, class *A* in component *CompA* has an attribute that uses type *B* from other component *CompB*.

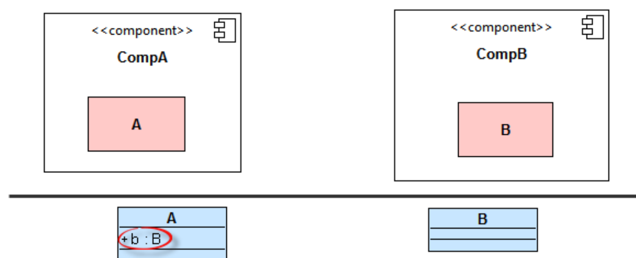

Figure 3.21: Class *A* is in *CompA* component and class *B* is in *CompB* component. Attribute *b* in class *A* is of type *B*. This is the inconsistency.

3.7.4 Provided interfaces not realized by any class in a component

If a component provides an interface, the interface must be realized by a class inside the component (or the component itself). If the interface was not realized in the component, there will be no way of providing this interface to other components that use it.

The example of described situation is given in fig. 3.22 where *CompA* component provides *R1* interface to another component. Problem is that any of the classes inside *CompA* (in this case we have only single class *A*) do not have realization of *R1* interface. In the consequence, *CompA* has no means of providing this interface. This is an inconsistency.

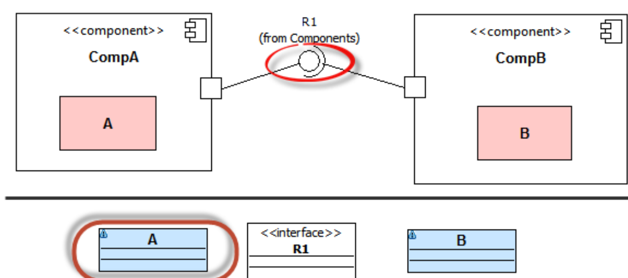

Figure 3.22: Class *A* is in *CompA* component and class *B* is in *CompB* component. *CompA* provides interface *R1* but no class in component *CompA* realizes *R1*.

Chapter 4

Case study: Toll System

In this chapter, we will see a short introduction to a case study of a toll system that will be later used as an example for our consistency checking approach. The case study described here is a (partial) solution for an exam project used in *System Integration* course at DTU in summer semester of 2012 [BK12].

4.1 Introduction to the toll system

The toll system that is modeled is used to manage a motorway company system. The system should, among others, charge owners of vehicles that enter and leave a motorway (see fig. 4.1).

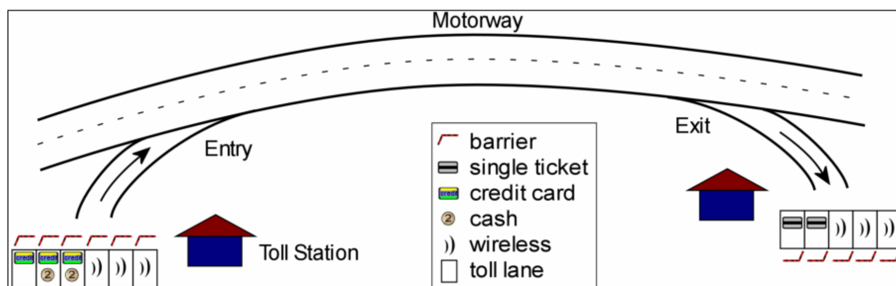

Figure 4.1: A motorway with two toll stations. Source: [BK12].

The system is complex, with many use-cases and actors (see fig. 4.2). In this thesis, we focus on constructing and checking only two use-cases from the system: *check-in with toll tag* and *check out with toll tag*. We choose these two use cases because they provide good examples for our consistency checking technique.

Figure 4.2: The use-case diagram of the toll system. The selection frame shows the part of the system used as the case study.

In fig. 4.1 each toll station is described to have many toll lanes. For our simplified system that handles the two use-cases, we will provide only one toll lane type: *express lane* (this one is represented in fig. 4.1 as a rectangle with a wireless sign). The express lanes can be used for the toll tag check-in and check-out.

An express lane has:

barrier that opens after a vehicle successfully completed the check-in or check-out;

RFID antenna that reads toll tags which are placed in cars entering the express lane.

Toll tags are used by regular customers for the easier wireless express check-in and check-out. They are placed in vehicles and can be identified by the express lanes. Toll tags are bought in advance.

> Toll tag is an RFID transponder with a range of a few meters which is used to wirelessly identify the vehicle. One toll tag is bound to one vehicle only. The price is not fixed but depends on the distance traveled on the motorway, i.e. by calculating the distance between the check-in and check-out. [BK12]

4.2 Check-in with toll tag

Check in with a toll tag occurs when the vehicle with a toll tag placed inside enters a motorway through an express lane. The toll tag is detected by an antenna in the express lane. If the toll tag is valid, a barrier in the express lane opens. The price for the trip is computed when the vehicle leaves the motorway.

It may happen that the antenna cannot recognize the toll tag or the toll tag is invalid. In this case, it is a cashier who decides how to proceed with the check-in of the toll tag.

4.3 Check-out with toll tag

During the check-out a vehicle with a toll tag inside leaves a motorway through an express lane. The toll tag is detected by an antenna. A distance between entry and exit stations is calculated and a resulting number of kilometers is multiplied by a price for one kilometer. The resultant value is then added to a toll account of an owner of the toll tag. Then, a barrier in the express lane opens.

The sum of tolls incurred in one month is charged in the end of each month.

It may happen that the antenna cannot recognize the toll tag or the toll tag is invalid. In this case, it is a cashier who decides how to proceed with the check-out of the toll tag by charging the owner of the toll tag correctly.

4.4 Models of the toll system

The toll system was modeled in two versions:

- Version without components is presented in appendix A
- Version with components is presented in appendix B
- Use case scenarios and their realizations (the same for both versions) are presented in appendix C

The two versions fulfill the same functionality and are used during the description of the algorithm in chapter 5. A simpler version of the system without components was constructed because some of the sub-algorithms do not require knowledge about components (and protocols). It is therefore easier for the reader to understand these algorithms having the simpler version of the system without components. A proper version of the solution to the toll system design in *System Integration* course would have components.

Chapter 5

Concepts and approach

In this chapter, we will see a technique to detect inconsistencies that were described in chapter 3.

At the beginning, we will introduce a concept of a consistency checking by a simulation of scenarios (see section 5.1), and then we will see overview on the consistency checking algorithm (see section 5.2). A model's representation used during the consistency checking is described in section 5.3.

In the following sections, concepts of scenarios (see section 5.4) and realizations of scenarios (see section 5.5) are introduced.

We will see one consistency checking algorithm divided into three sub-algorithms, i.e. the extension algorithm in section 5.6, the behavioral state machines execution algorithm in section 5.7 and the verification of protocol state machines algorithm in section 5.9.

A language created during this project, Simple Action Language (SAL), that facilitates an execution of BSMs is presented in section 5.8.

5.1 Consistency checking by scenario simulation

A consistency checking technique that is developed in this master's project uses simulation of scenarios provided by a designer to check the consistency of UML

models. In our approach, scenarios that were simulated successfully in a model are realizable the model. We can say that the model's elements that were checked during the simulation of the scenarios are consistent with the scenarios.

If a designer provided a reasonable set of scenarios for a model, i.e. for each use case in the model, there is provided a main scenario and all remaining alternative scenarios; and if all of the scenarios (the main and the alternatives) are simulated successfully by a described algorithm (that will be described in this chapter), then we can say that the model of the system is consistent with its specification by fulfilling the given use cases.

Our approach is designed to be used as a teaching technique for students by helping them to understand how different views on a model relate to each other. The students should also understand how a change in one of the views can (unexpectedly) affects other views on the model. Our approach is also focused on constructing good (verifiable) scenarios that represent use cases from requirements. This approach shows the importance of a good testing as a method for verification of correctness of systems.

In real life, the testing is used for a verification of an implementation; but for our teaching purposes, it is a good practice to also construct tests for models[1]. This gives a unique possibility of an early feedback to students on a design.

We would like to answer following questions:

- Is a design rational?
- Can a design be used for an implementation of a system?
- Will a system implemented based on a design be what we expected it to be?
- Does design fulfill the requirements?
- Can design realize use cases scenarios provided by clients and gathered during requirements collection?

Can this approach be used in the industry? In most of the companies, the models are never developed to the point that they will be suitable for the simulation (or a code generation). Instead, the models are used for a specification of early versions of a system – and this is very reasonable approach, at least when we

[1]Especially if students do not have time to implement a system because the semester is too short – and this happens to be our case.

look at the money costs. The models are sometimes more difficult and time consuming to develop.

There are some companies, however, that use models. In the largest Danish IT company: Danske Bank models are developed in Denmark and then they are sent to India for implementation stage[2]. This type of a development make the modeling more important.

5.2 Consistency checking algorithm

In this section, we will see an overview on a consistency checking algorithm that is developed in this thesis and that is able to detect the inconsistencies described in chapter 3. For simplicity, the description in this section will be given from a high level perspective and then, in the subsequent sections, we will describe separate phases (three sub-algorithms) of the algorithm in more detail.

The main role of the consistency checking algorithm is to check consistency between interactions (sequence diagrams) and state machines (behavioral state machines and protocol state machines diagrams). It is achieved by a simulation of a given scenario in a model.

The algorithm works in two main phases:

1. checking of a static UML model's structure before a simulation is launched;
2. checking of a dynamic UML model's behavior during the simulation.

Consistency checking algorithm is given a scenario to check (scenarios are described in section 5.4). The static properties of the model are checked first. This step is necessary to ensure that the simulation of a scenario could be even started. Most of inconsistencies that we are looking for in this phase are described in sections 3.1, 3.2 and 3.3. Also inconsistencies related to components described in section 3.7 are checked in this phase. If errors are found in this phase, they are reported to a designer and the algorithm terminates without starting a second phase.

The second phase of the algorithm always starts from the extension sub-algorithm (see section 5.6) that triggers an execution of BSMs (see section 5.7). During

[2]This information was informally obtained by me from one of the Danske Bank Group IT section employees, Rune Haxbøl.

the execution of BSMs, the extension sub-algorithm and the verification of PSM algorithm (see section 5.9) become observers of the execution, i.e. they are notified of execution events that occur. They both react on the events, ensuring that the execution is consistent with other fragments of a model.

In the second phase of the algorithm, the test scenario is executed to ensure that BSMs and PSMs are consistent with a scenario (a sequence diagram). Any run-time errors are reported to a designer and, if they occur, the execution of BSMs is stopped. During this phase, elements can be modified in the model or new elements can be added to the model. Any change to the model during a run of the algorithm must be consistent with respect to the structural consistency.

5.3 Direct UML representation

The presented consistency checking algorithm works directly on UML models. There are no transformations to another representation of a model before checking. This approach was taken because it is easier by avoiding the problematic transformations. From another point of view, the UML has really complex meta-model and, therefore, navigating directly in UML models is more difficult during the consistency checking of scenarios in the direct approach. The direct approach is easier to integrate into existing tooling (UML editors) and becomes more important if the approached modify a model during the checking.

Most of the approaches that use transformations do not change anything in UML models. The approach presented in this thesis extends sequence diagrams by adding new UML elements to a model – this would require, if we used transformations, to transform back the changes from an intermediate representation to the UML model.

In the direct approach, it is also easier to show errors immediately during checking. It's easier to indicate in which UML elements an error or a warning occurred: we do not need to translate error information back to see what an error context in the UML model is. Because of that, it is possible to provide much more sensible hints for a designer.

I would also argue that direct approach is more robust if we want to use it continuously during a design of the UML model. E.g. if an error occurred at the very beginning of the simulation, we do not need to wait to first transform the model to an intermediate representation; we can just start the simulation and pinpoint the error.

5.4 Scenarios

Scenarios provided by a designer that were mentioned in the previous sections are, in this approach, UML sequence diagrams that show interactions of actors with a system. Scenario can represent a main scenario or an alternative scenario of a use case. Use cases and their scenarios are developed during requirements stage of a system development.

A scenario is given to the consistency checking algorithm as an input. The algorithm is able to handle scenarios showing messages from or to the actor and scenarios that are partly or completely realized (see section 5.5).

An example of a scenario for the *check-out with toll tag* use-case is shown in fig. 5.1. In the example, an actor *DD12312* is a car that enters an express check-out lane and is interacting with a toll system by calling method *vehicleArrives()* in *antenna* instance. This call represents a detection of the toll tag with id *tag123* in the car by the antenna. The actor then expects to drive through the lane by calling *vehicleDrivesThrough()* in *lane* instance[3].

The scenario does not include any information on how the both methods will be realized: by looking at the diagram, we do not know if *antenna* or *lane* will call other objects in the system and, if they will, in what order will they call them. We do not also know if the toll tag was valid and if a barrier will open.

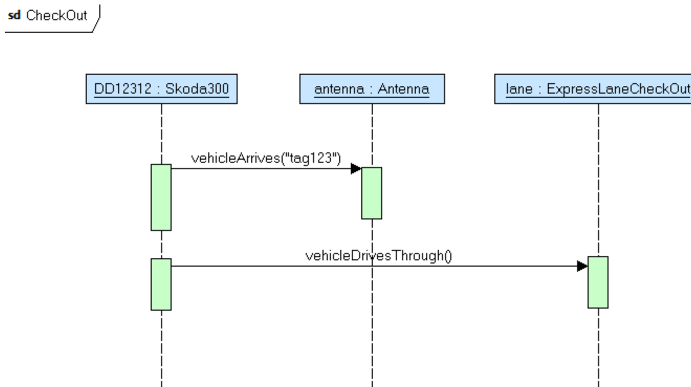

Figure 5.1: Check-out with toll tag successful scenario test.

The scenario must be defined together with a context of a system's state before the simulation was started. To do this in the UML, a designer must prepare an

[3]This represents a possible sensor that is placed in the lane and can detect the car passing.

object diagram specifying a system's state. For our simple scenario, we assume to have the following configuration that is presented in fig. 5.2.

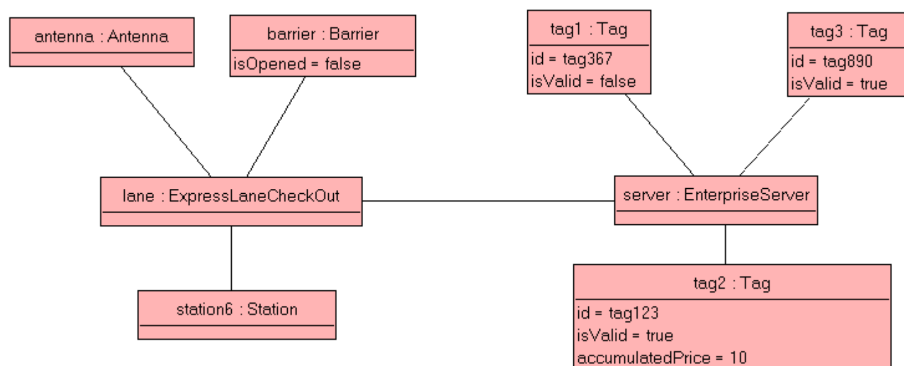

Figure 5.2: The toll system's state before *check-out with toll tag* started simulating.

Three toll tags are registered in the toll system, two of which are valid (*tag2* and *tag3*).*tag2* is the tag with id *tag123* used in our scenario shown in fig. 5.1; an accumulated price for this tag is currently 10 DKK.

The car is not an instance in the system. In approach presented in this thesis, actors are generally considered to be external to a subject system and do not need to be instantiated.

5.5 Realizations of scenarios

Realizations of scenarios are scenarios extended[4] by new elements added during the simulation of scenarios. A realization shows us detailed execution trace of a scenario's simulation:

- what operations were called in the system;
- in what order were they called;
- what were arguments of the operations called;
- what were result values of the operations called.

[4]In the UML terminology we could also use the verb *specialized*. Compare it to the definition of *specializing* from section 2.6.

Additionally, after the simulation, we can see how a system state changed.

In an example presented in fig. 5.3 a realization of the scenario from section 5.4 is visible. In this section, we will focus on describing what we can see in the realization. An extension algorithm is presented later in section 5.6.

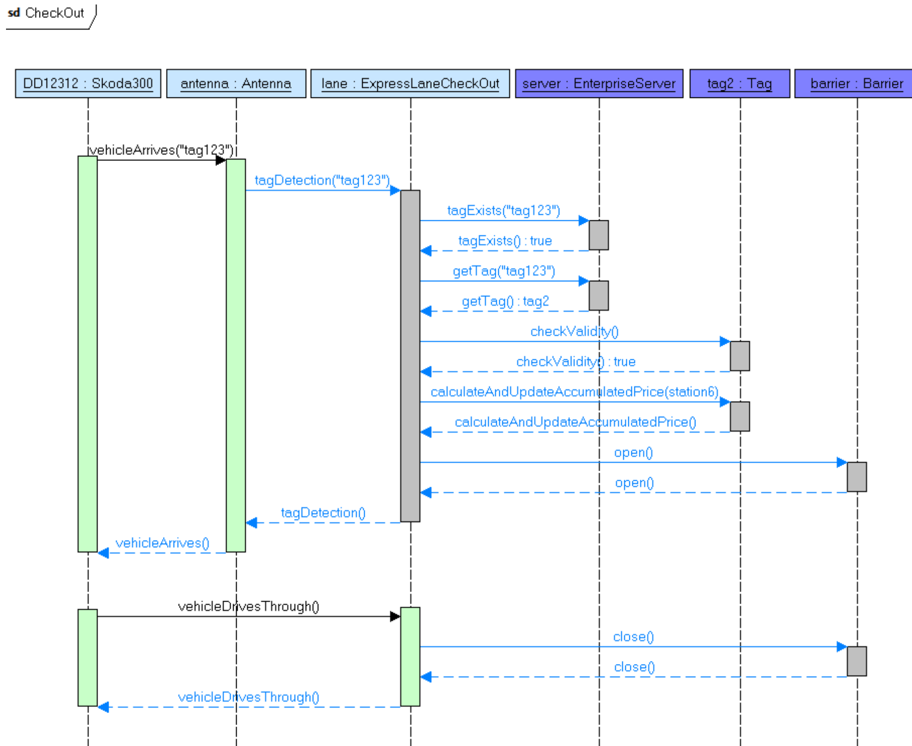

Figure 5.3: The realization of a successful scenario *check-out with toll tag*. Generated elements that were added as an extension with respect to the original scenario in fig. 5.1 have distinct colors.

The diagram in fig. 5.3 is an extension of the scenario in fig. 5.1. It contains the messages that were present in the original scenario; it contains also new, generated elements that are result of the simulation of the original scenario. More specifically, the new elements are the three lifelines: *server*, *tag2* and *barrier*. All the messages presented in blue in the diagram are the new messages that were added during the simulation and were not present in the original scenario. New behavior execution specifications (BESs) are presented in gray.

In the realization of the *check-out with toll tag* scenario we can see how a system was executing the operations that were called by *DD12312*. After the vehicle arrives and the antenna detects the tag with id "tag123", there are two operations called in *server* that check if the tag exists in the system and get the tag object with corresponding id from *server*. Then, validity of the tag is checked and the tag is updated with new accumulated price. *barrier* opens and the car drives through. *barrier* closes after the car left.

The system's state after the simulation of the scenario is presented in fig. 5.4. The value of *accumulatedPrice* in *tag2* increased by 25 DKK. A conclusion: a price for a trip was calculated and the price was added to the toll's account.

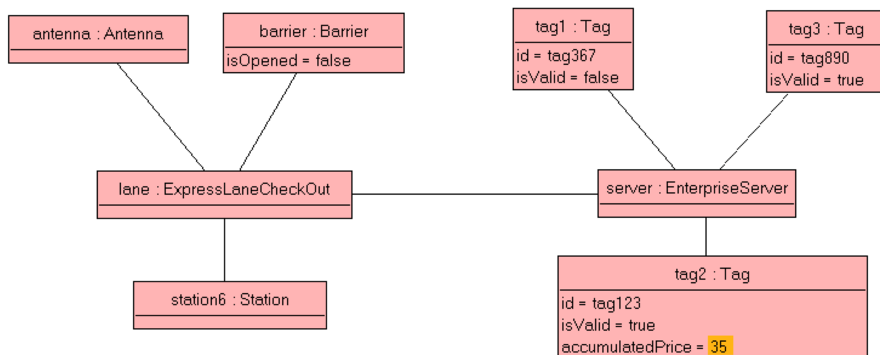

Figure 5.4: The toll system's state after *check-out with toll tag* scenario was simulated. Note, 25 DKK was added to *accumulatedPrice* of *tag2* with respect to the state from before simulation of the scenario (fig. 5.2).

Because the rest of the diagrams in the toll system without components model (including BSMs diagrams) in this example introduce not yet described syntax of SAL, it may be better for the reader not to look at them during the first reading and just trustfully assume that the presented sequence of events really happened during the simulation. For curious reader, all BSMs diagrams and class diagram are available in appendix A.

5.6 Extensions of scenarios to realizations of the scenarios

In this section, we will see how scenarios are extended to realizations of the scenarios. We will introduce an algorithm then describe an example run of the algorithm on *check-out with toll tag* scenario.

5.6.1 Extension algorithm

Scenarios extension algorithm works in a manner presented in a pseudo-code in fig. 5.5 and visually in fig. 5.6.

Figure 5.5: Extension of interactions.

```
1 extendScenario(interaction) =
2    lastSeqMessage ← firstMessageinSeq(interaction)
3    while null ≠ lastSeqMessage
4       if isReply(lastSeqMessage) or not isActor(source(
           lastSeqMessage))
5          error
6       invokeExecution(interaction, target(lastSeqMessage)
           , lastSeqMessage)
7       lastSeqMessage ← nextMessageInSeq(interaction,
           lastSeqMessage)
```

A scenario sequence diagram (interaction) is given as an input for a function *extendScenario*. The input scenario may be one of the possible types:

a scenario described in section 5.4 that shows only the interactions between actors and a system. During an extension, the scenario will be extended by adding new parts of an interaction that occurs in the system internally during simulation;

a scenario realization described in section 5.5 that shows all interactions between actors and a system and between the system's objects internally. The algorithm will validate the scenario realization against the system's actual behavior. If a given realization is not complete, i.e. there are some missing parts of the internal system interactions, they will be completed

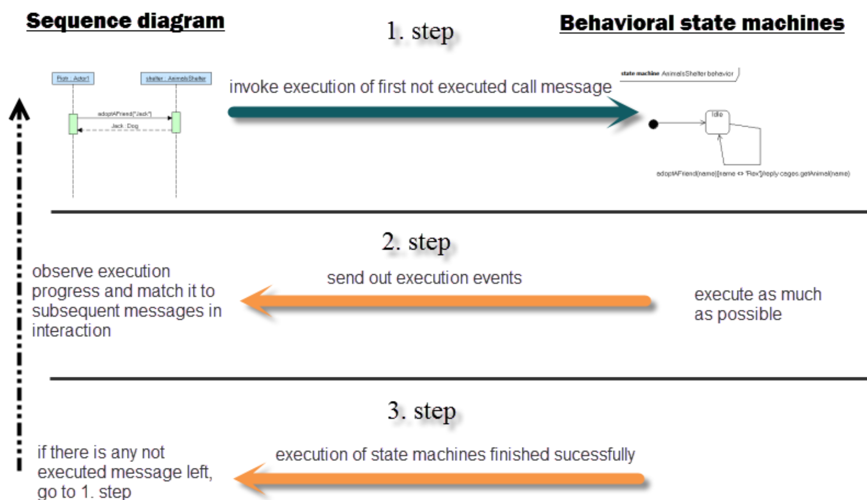

Figure 5.6: Visual description of extension of scenarios algorithm.

during the extension and the already existing parts will be verified[5].

The process of extension can be seen as divided in three steps.

In the first step, the algorithm looks for the first not executed call-message in a sequence diagram and assigns it to *lastSeqMessage* (at the beginning it will be always the first message sent in an interaction). Once it finds the message, it checks if the message is not a reply message (we cannot execute reply messages here and if we found a reply message, it's an indicator that a designer specified more than one reply for the operation[6]).

The message should also be sent from an actor's lifeline – it's because, if the message was sent from an internal system object, it should be the effect of a system's behavior that would trigger the message, not this extension algorithm.

[5]In the thesis, our main focus is a situation where an input sequence diagram describes only an interaction between actors and a system (the first case here); but the consistency checking algorithm can handle also the case where the input sequence diagram contains some parts (or all parts) of an interaction between system objects. That includes the situation where we run the algorithm again on the output from the preceding simulation (on scenario realization) as an input. In this case, algorithm should check if the system will realize this scenario in the same way the second time.

[6]Please, compare it with fig. 3.14.

In line no. 6, starts an execution of an event that the message represents. A source is a checked interaction and a target is BSM for an object that the target lifeline represents. At this moment, algorithm enters second step.

In second step two, the algorithm is still in line no. 6. The algorithm becomes **observer** of the triggered execution of BSMs. The execution produces call and reply events that carry with them following essential information[7]:

- for a call event:
  - what are a source and a target;
  - what method in which object is being executed currently;
  - what are arguments to a call;
- for a reply event:
  - what are a source and a target;
  - what is a triggering call event for this reply;
  - what is a return value.

While the extension algorithm observes new arriving events, it will try to match it to the existing sequence diagram, i.e. it will check the next message candidate that is sent in the checked interaction. If it matches the just arrived event's information then the candidate is accepted. Because there is no substantial difference between matching call and reply events, we can define a general function for matching all types of events and (in case events have no matched candidates) extending interactions. The function is presented in fig. 5.7.

In the function *eventFired*, after event arrives it is filtered by a condition that recognizes events with source in the checked interaction (line no. 9). For such events, the function is not executed. Next, there is a search for a source and a target lifeline of the event in the sequence diagram. We can safely assume that the source lifeline will always be present in the diagram when the event arrives (this is a consequence of step 1 of the extension algorithm and matching next arriving events in the order of their execution). If the target lifeline is not present in the sequence diagram (is null), it will be created; otherwise a search for a message candidate will be conducted. *lastSeqMessage* used in search of the candidate in line no. 17 is declared outside a scope of *eventFired* and initialized (in step 1) to the first not executed message of the interaction.

[7]For details on how the BSM execute, please consult section 5.7.

Figure 5.7: Events matching during extension of interaction.

```
8  eventFired(event) =
9     if source(event) = interaction
10       return
11    sourceLifeline ← getSourceInSeq(interaction, event)
12    targetLifeline ← getTargetInSeq(interaction, event)
13    candidate ← null
14    if null = targetLifeline
15       targetLifeline ← createTargetLifeline(interaction,
            event)
16    else
17       candidate ← nextMessageInSeq(interaction,
            lastSeqMessage)
18    if null ≠ candidate and eventConformsToMessage(event,
         candidate, sourceLifeline, targetLifeline)
19       lastSeqMessage ← candidate
20    else
21       message ← createMessageFrom(event)
22       insertMessageInSeq(interaction, message,
            sourceLifeline, targetLifeline)
23       lastSeqMessage ← message
```

If the candidate is found and it conforms to the event[8], candidate is assigned to *lastSeqMessage*. Otherwise, a new message will be created, inserted in the correct place (after *lastSeqMessage*) in the sequence diagram and assigned to *lastSeqMessage*.

In the third and last step of the extension algorithm, the BSMs execution finishes (there will be no more events arriving). The function in fig. 5.5 moves to line no. 7 and then to a next iteration of a loop where it is checked if there is any message left to be executed. If there is a message to execute, then the algorithm starts from the first step again.

[8]In case of a call message, we can say this message conforms to a call event if an operation specified in the message is the same as an operation of the event and arguments of the message are the same as arguments of the event. In case of a reply event, the result specified in a reply message must be identical with a result carried in the event. In both cases, a source and a target lifelines must be the same.

5.6.2 Found and lost messages

Found and lost messages are the messages in a sequence diagram that, respectively, originate from the environment or are targeted to the environment. In the presented approach, found and lost messages are disallowed in sequence diagrams representing scenarios (compare with inconsistency in section 3.1.2). The environment is represented always by actors. In this context, every message coming from an actor is "found" because it is coming from outside of the system. The messages targeted to an actor ("lost" messages in this approach) may have sense if the actor has its behavior specified (has its own BSM).

5.6.3 Example run of the extension algorithm

In this section, an example run of the algorithm from section 5.6.1 is presented. We will apply the algorithm to the scenario described in section 5.4 and we will see how we construct the realization of the scenario described in section 5.5. For convenience, we present fig. 5.8 in which both the scenario and its realization are presented. The figure contains also additional markers with numbers to which we will refer in a text.

In the first step of the algorithm in fig. 5.5, after *extendScenario* function is called with an argument of a scenario as *interaction*, *lastSeqMessage* is assigned to the first message in the sequence diagram, i.e. *vehicleArrives* (see marker 1). Next, a target of the message is identified to be *antenna* lifeline (see marker 2) that represents an object *antenna* in the object diagram of the toll system (see fig. 5.2). A call event of the *vehicleArrives()* operation with one argument is dispatched to the object and the execution of BSM of *antenna* is started.

The extension algorithm enters the second step at this moment and starts observing execution events from BSMs. The first execution event arriving is:

1. name: vehicleArrives, type: synchCall (from DD12312 to antenna).

Note, it is an event of call of operation that we just invoked in the first step. Upon arrival of the event, *eventFired* operation is called in the events matching algorithm in fig. 5.7 with the event as an argument. The event is filtered out in line no. 9 because its source is the checked interaction (i.e. we invoked this operation from the extension algorithm). The processing of the first event is finished.

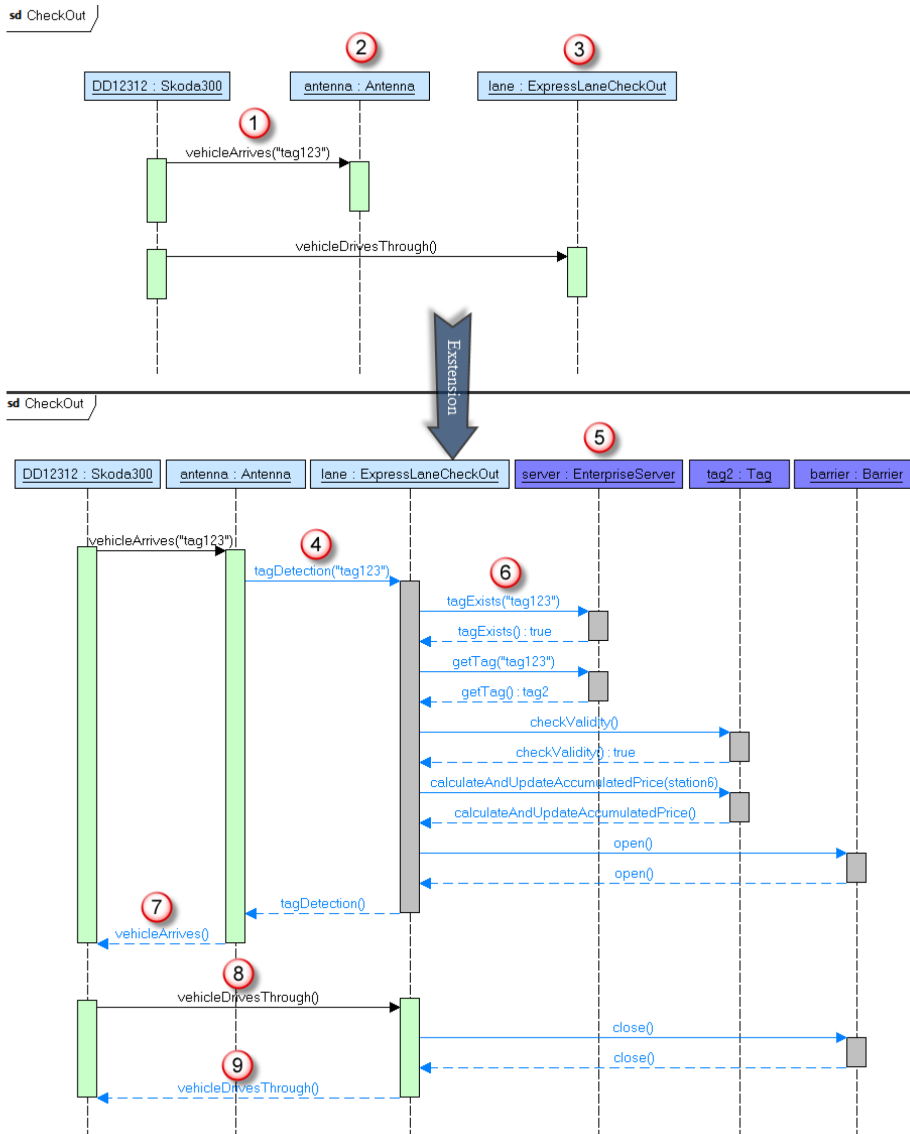

Figure 5.8: The scenario *check-out with toll tag* and its realization. The added elements in the extended realization of scenario have distinct colors. Numbers in circles are markers referred in a text.

The execution of BSMs continues and another object *lane* is called. In the result, the next event that arrives is:

2. name: tagDetection, type: synchCall (from antenna to lane).

Again, *eventFired* operation is called in the events matching function. This time source of the event is not the interaction; instead it is *antenna* object. The function finds both a source and a target lifeline. Note, in this case the target lifeline existed in the scenario (see marker 3) and a candidate message is searched for (in line no. 17) and checked (in line no. 18). The candidate message, in this case, is not found (is `null`) therefore the function generates a new message (line no. 21), insert it into the interaction (line no. 22 and marker 4) and assign it to *lastSeqMessage* (line no. 23).

A next event that arrives is:

3. name: tagExists, type: synchCall (from lane to server).

Handling of it is similar to the previous event; with an exception, that *server* target lifeline is not found and generated in line no. 15 (see marker 5). A new message is generated for *tagExists()* operation (see marker 6).

Next, the following events arrive from BSMs execution:

4. name: ReplyOftagExists, type: reply (from server to lane)
5. name: getTag, type: synchCall (from lane to server)
6. name: ReplyOfgetTag, type: reply (from server to lane)
7. name: checkValidity, type: synchCall (from lane to tag2)
8. name: ReplyOfcheckValidity, type: reply (from tag2 to lane)
9. name: calculateAndUpdateAccumulatedPrice, type: synchCall (from lane to tag2)
10. name: ReplyOfcalculateAndUpdateAccumulatedPrice, type: reply (from tag2 to lane)
11. name: open, type: synchCall (from lane to barrier)
12. name: ReplyOfopen, type: reply (from barrier to lane)

13. name: ReplyOftagDetection, type: reply (from lane to antenna)
14. name: ReplyOfvehicleArrives, type: reply (from antenna to DD12312)

Handling them is similar to the previous two events (some of them generate new lifelines, some of them not) and new messages for all of them are generated. After the last event from this series arrived, *lastSeqMessage* have a value of a reply message for *vehicleArrives()* operation (see marker 7).

At this moment, the BSMs do not have more to execute and the extension function in fig. 5.5 moves to line no. 7, entering the third step of the algorithm.

In the line no. 7, a call message of *vehicleDrivesThrough()* operation (see marker 8) is assigned to *lastSeqMessage*. The while loop continues (the first step again) to a next iteration and dispatch an execution of *vehicleDrivesThrough()* to *lane* object. BSMs start the execution again and produce events (second step):

15. name: vehicleDrivesThrough, type: synchCall (from DD12312 to lane)
16. name: close, type: synchCall (from lane to barrier)
17. name: ReplyOfclose, type: reply (from barrier to lane)
18. name: ReplyOfvehicleDrivesThrough, type: reply (from lane to DD12312)

The first of which is *vehicleDrivesThrough* call event. The call event has a source in the interaction and is skipped by the events matching function. No new lifelines are generated for these events because a lifeline for *barrier* was already created in the previous iteration of the algorithm. After the BSMs do not have more to execute, *lastSeqMessage* is assigned to value of a reply message of *vehicleDrivesThrough()* (see marker 9) and the algorithm enters the third step.

This time, in the line no. 7, `null` is assigned to *lastSeqMessage* (there are no more messages to execute in the interaction). In consequence, this ends the loop and the whole run of the algorithm is finished.

5.7 Execution of behavioral state machines

Since now, we have abstracted out from how BSMs execute. We assumed there were events generated during the execution. In this section, we will look closer

on how BSMs execute, how events are generated and when observers are notified of new events. We will examine what run-to-completion means, and then we will see the execution algorithm, the firing of the transition function and some of interesting problems during the execution. We will also see an example run of BSMs execution. In the end, we will discuss other possible semantics of BSMs execution.

5.7.1 Run-to-completion

A client (i.e. a sequence diagram or BSM) invokes an operation in a specified instance specification of a target object. A call event is then created from the invocation. The event is placed in the events' queue for the target object. The events in the queue are fetched by BSM of the target object, one at a time (the UML standard gives a leeway in defining order of dequeuing). The algorithm to process these events by BSMs is called *run-to-completion* and it's described in the UML specification [Obj09].

The approach presented in this thesis follows the semantics of the run-to-completion from the UML specification, and in [HG97], and in [DDd03].

A run-to-completion step is "an execution of sequence of transitions between two stable state configurations"[DDd03] (stable state configuration is a "configuration in which no further transition is possible without dispatching an event"[DDd03]). The step involves dispatching a completion event and taking completion transitions after entering the stable configuration. In the end of each run-to-completion step an invoked "method terminates and the thread of control returns to the calling object" [HG97]. If the step was triggered by a synchronous call event, the reply is executed at the end or the step.

One step of the run-to-completion is presented in fig. 5.9.

Figure 5.9: Run-to-completion step.

```
1 eventDispatched(event)=
2     transition ← enabledTransition(event)
3     execute(effects(transition))
4     completionTransitions ← completionTransitions()
5     execute(effects(completionTransitions))
6     if isSynchronous(event)
7         send reply
```

An event is dispatched by a dispatcher in line no. 1. An enabled transition for this event is calculated (line no. 2). Then, effects on the fired transition are executed (line no. 3). Next, the completion event is dispatched, i.e. completion transitions are calculated (line no. 4) and effects on them are executed (line no. 5). In the end, a reply for the synchronous event is sent (line no. 7). Note, the execution of effects on both enabled and completion transitions can set the reply.

5.7.1.1 Run-to-completion initialization

During a system initialization, before the run-to-completion is started, all BSMs, representing all instances specifications in the system, receive a completion event: they execute completion transitions and execute effects on them. This is a natural behavior to leave initial pseudo-states and, therefore, prepare for a next run-to-completion step.

5.7.2 Execution algorithm

An algorithm used for the execution of BSMs is presented in fig. 5.10. Events scheduling algorithm is presented in fig. 5.11. They both show the how BSMs execute in our approach for the purpose of consistency checking.

The events used in this section are **not** exactly the same as the UML events. The events used here are part of the consistency checking approach and they only partly overlap and represent the UML events or actions.

Another difference between our approach and the UML specification is that we will use one global queue of events for all object instances, instead of one local queue for each object instance, which is the normal UML semantics. By using only one queue we simplify the processing of events during simulation. Our events contain target and source references so it is possible to derive all the local queues from the global one by looking at where the event is targeted to and assigning it to the local queue of the target object. The events are dequeued from the global queue is first-in first-out (FIFO) order.

A method of firing a transition in this thesis is inspired by work of Alexander Knapp in [Kna04] (especially *forwardTrees* function for the UML 1.x models) and was extended in this work to match UML 2.2 BSMs. The algorithm presented in this thesis models a program flow as events broadcasted from BSMs to observers (i.e. other parts of the consistency algorithm, including the extension

Figure 5.10: Execution of behavioral state machines.

```
1  invokeExecution(source, target, message) =
2     (operation, args, isSynchronous) ← message
3     callOperation(source, target, operation, args,
          isSynchronous)
4     executeFirstFromQueue()
```

```
5  callOperation(source, target, operation, args, isSynchronous)=
6     event ← createCallEvent(source, target, operation, args,
          isSynchronous)
7     addEventToQueue(event)
8     if isSynchronous
9        executeFromQueueAndGetResult(event)
10    else
11       notifyObserversofCallEvent(event)
12       notifyObserversofControlEvent(event)
```

```
13 accept(event) =
14    (source, target, operation, args, isSynchronous) ← event
15    if not argumentsConformToParameters(args, operation)
16       error
17    if isQueryOperation(operation)
18       callQueryOperation(source, operation, args)
19       return
20    if isExecuting(target)
21       if isSynchronous
22          error
23       else
24          eventLost(event)
25    else
26       setExecuting(target, true)
27    transitions ← getTransitionsWithValidGuards(target,
          operation, args)
28    if transitions ≠ ∅
29       toTake ← choose(transitions)
30       if not hasSentNotificationOf(event)
31          notifyObserversofCallEvent(event)
32       reply ← fireTransition(target, toTake) //this can lead
            to a recursive call(s) to callOperation
33       fireCompletionTransitions(target, reply)
34       if not replyConformToResult(reply, operation,
            isSynchronous)
35          error
36       if isSynchronous
37          notifyObserversofReplyEvent(self, event, reply)
38          notifyObserversofCallReturnControlEvent(event)
39    else
40       if isSynchronous
41          error
42       else
43          eventLost(event)
44    setExecuting(target, false)
45    executeFirstFromQueue()
```

Figure 5.11: Scheduling of call events for behavioral state machines.

```
46 executeFirstFromQueue() =
47    if queue ≠ ∅
48       callEvent ← firstElement(queue)
49       removeFromQueue(event)
50       accept(event)
```

```
51 executeFromQueueAndGetResult(callEvent)
52    do
53       executeFirstFromQueue()
54    while callEvent ∈ queue
55    getResultFor(callEvent)
```

sub-algorithm (see section 5.6.1) and the verification of protocol state machines algorithm (see section 5.9.1)). Having an observer pattern in place provides a separation between BSMs execution and other sub-algorithms.

In the algorithm that will be presented, there is a difference between a synchronous and an asynchronous type of calls. The difference lies in the way the SAL defines a reply of a synchronous operation (see section 5.8.2.3 for more details on this topic). The BSM must be able to reply for a synchronous call in the same run-to-completion step it was called to avoid a deadlock (that would be caused by blocking of a caller that will wait for a reply that would never come - compare with inconsistency 3.4.2). Because of that, synchronous calls must be handled when they are called and a returned result must be known in one run-to-completion step.

There are few types of events to communicate the execution state to other sub-algorithms observing the execution:

call event that informs of a received call in BSM;

reply event that informs of a reply for a call from BSM;

completion event that is an internal event used to fire transitions without triggers and with valid guards;

call return control event that helps to model when a program flow returns from a call to a caller (a moment of sending this event is different for synchronous and asynchronous calls).

Following is the description of the algorithm: the first call initiating an execution

is done by a client (in our case, a sequence diagram via the extension sub-algorithm) using *invokeExecution* operation (line no. 1). A message from the sequence diagram is used for *callOperation* arguments (line no. 3).

After *callOperation* is called, a new call event is created (line no. 6). Then, the event is placed in the global events queue (line no. 7). If the call is synchronous, the event queue must be executed immediately until we are able to get a result of the called operation (line no. 9). If the call is asynchronous, the call event and a new call return control event are sent to observers as notifications (lines no. 11 and 12). After the call to *callOperation* returns, it is necessary to call an operation *executeFirstFromQueue()* at least once (line no. 4). This will execute remaining events from the queue, if any (note, an execution of the event will eventually result in *accept* operation called in line no. 50 and in recursive call in line no. 45).

The *accept* operation is called for a scheduled call event in line no. 50: it first checks if arguments of the event conform to parameters of an operation, if not, an error is reported (line no. 16; compare it with an inconsistency in section 3.1.5). If the operation is a query operation (the operation that does not change a state of a system), it is handled immediately (line no. 18) without changing an active state configuration of the BSM (see section 5.7.6). If a target BSM is executing and the call is synchronous, and then it is an error (line no. 22) because of a deadlock that occurs: *isExecuting* means that a BSM is not in stable state configuration (is executing event right now) and cannot fire any transitions. That means, recursive synchronous calls to `self`-instance are never possible and cause an error (see fig. 5.12). This problem was observed first by Harel and Grey in [HG97] and then by Tenzer and Stevens in [TS03][9].

In case of an asynchronous event arrival while a target BSM is executing, the event is lost (line no. 24).

Next, transitions outgoing from the active states configuration of a target BSM with valid guards and having triggers for a given operation are calculated (line no. 27). Arguments of the event must be taken into consideration during this process (guards can refer to names of arguments of a called operation).

If no valid transitions are found and if a call is synchronous, it is an error (line

[9] Harel and Grey wrote [HG97]:

> ...when the client's statechart invokes another object's operation, its execution freezes in midtransition, and the thread of control is passed to the called object. Clearly, this might continue, with the called object calling others, and so on. However, a cycle of invocations leading back to the same object instance is illegal, and an attempt to execute it will abort.

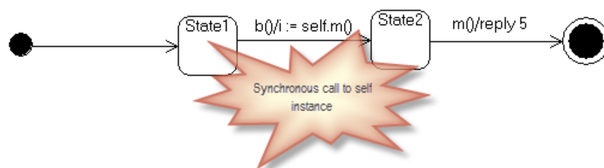

Figure 5.12: The presented BSM cannot execute a recursive synchronous call to `self`. In a moment, *self.m()* is called, a transition with trigger *b()* is still in a process of firing and BSM is not in a stable state configuration. The result of *m()* cannot be determined.

no. 41; compare with an inconsistency in section 3.5.1). For an asynchronous call, the event is lost (line no. 43).

If there is any transition to take, one is chosen (line no. 29). Note, if there is more than one transition (the transitions are *conflicting*), the choice (and the behavior of the BSM) is non-deterministic is flagged as a warning (see fig. 5.13).

Our current approach, in case of internal non-determinism, is to always choose one fixed transition in the set but flag the situation as a warning. However, if the BSM describes behavior of an external component (that is placed outside our system and therefore yields external non-determinism) we would like to give a choice to a user (a designer) at run-time of which transition to take. More on the topic is presented in 5.7.4.

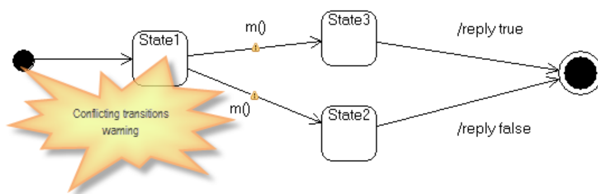

Figure 5.13: There are two conflicting transitions with trigger for *m()*, both outgoing from the *State 1*. In a result there is an internal non-deterministic choice in this BSM. The situation must be detected and flagged as a warning during BSMs execution. One of the conflicting transitions must be chosen during the execution.

If it was not done before (the notification could be sent earlier for asynchronous call in line no. 11.), the observers are notified of the call event (line no. 31).

Next, the chosen transition is fired and an effect on the transition is executed (line no. 32). An action language (SAL) is defined for effects on transitions and is described in section 5.8.

Firing a transition, in most cases, also changes an active states configuration of a target state machine. Firing a transition can also call operations in other BSMs. Calling other BSMs results in a recursive call to *callOperation* (line no. 5). See section 5.7.5 for more details on this topic.

A completion event is dispatched internally to a target BSM (line no. 33), i.e. all transitions that do not have any trigger and have valid guards are fired. During firing of completion transitions, simple infinite loop detection is in place so that a completion self-transition without any guard should not be taken (see fig. 5.14). Replies for synchronous calls can be modified during firing of completion transitions.

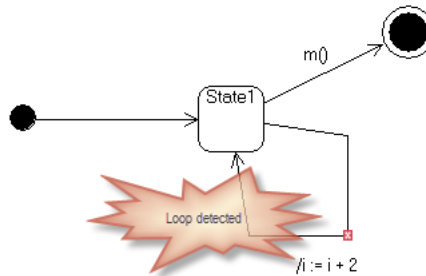

Figure 5.14: The infinite loop detection during the execution of BSMs is able to detect a completion self-transition with no trigger and no guard.

The reply is checked for a type and a multiplicity conformance to the operation result in line no. 34 (compare with an inconsistency in section 3.4.3). Asynchronous calls shall not return results.

If a call is synchronous, the algorithm notifies observers of a new reply event and a new call return control event (lines no. 37 and 38). Replies for synchronous calls are always sent after BSM stabilizes, i.e. after it is not possible to proceed without dispatching the next event from the queue. This semantics follows the semantics of David Harel and Eran Gray in [HG97] and formal UML semantics of state machines in [DDd03].

After all the previous steps were executed, a target BSM is set not to be executing anymore (line no. 44) and a next event (if any of them left or were produced by the last run-to-completion step) can execute (line no. 45).

A scheduling algorithm presented in a pseudo-code in fig. 5.11. The FIFO events' queue is used in functions of the scheduler. Every time *executeFirst-FromQueue()* operation is called, and if events queue is not empty, the first call event is taken out from the queue and *accept()* function is called for a target BSM (line no. 50).

For synchronous calls, *executeFromQueueAndGetResult()* function is executed. The function ensures that a synchronous event will be executed (using *do-while* loop in line no. 52). Then, a result of this event is retrieved (line no. 55).

5.7.3 Guards on transitions

Guards are specified on transitions. In our approach, guards are defined in the Object Constraint Language (OCL) language [Obj06]. A transition can be fired only when a guard on the transition is satisfied. There are no side-effects of evaluation of guards, i.e. evaluating guards must not change a state of a system. A guard on a transition is either "`else`" statement or an OCL query expression returning a Boolean result. A guard is evaluated in a context of an instance specification that corresponds to an object for which a BSM (in which the transition is defined) executes.

Transitions with guards provided as "`else`" statement must be outgoing from *choice* or *junction* pseudo-states that do not have any other outgoings transitions with guards equal to "`else`" statement and have at least one other transition with a guard other than "`else`" statement. The transition with "`else`" statement as a guard is taken only when guards on all other transitions outgoing from a source pseudo-state evaluate to `false`.

In contrast, guards provided as OCL query expressions can be placed on any transition that is not outgoing from *initial* pseudo-state.

If a guard is defined on a transition on which there is a trigger for a call event, arguments' names of the call are available in the guard. An example of this situation is presented in fig. 5.15.

Figure 5.15: The guard using arguments of the call event to evaluate.

5.7.4 Internal vs. external non-determinism

During the run-to-completion, "it is possible for more than one transition to be enabled within a state machine"[Obj09] (see fig.5.13). These transitions are called *conflicting* transitions in the UML specification. If such conflict happens between transitions in the same region of the state machine, only one transition may fire. The choice, which enabled transition to fire, is non-deterministic.

We divide non-determinism into two categories:

1. an internal non-determinism where "the agent makes the choice as to which action to take"[VdH94], and
2. an external non-determinism where the choice is up to the external environment.

A non-deterministic choice can be *angelic*, *demonic* or *erratic* dependent on being the "*correct*, desired action whenever possible"[VdH94] (angelic) or "*incorrect*, undesirable one if given that possibility"[VdH94] (demonic) or unpredictable (erratic).

In classes and components that are parts of a UML model that we design, an existing non-determinism is internal. I.e., we can specify BSMs in the way they will have internal non-determinism. In this approach, a solution for the internal non-deterministic choice is to always take one fixed transition from a set of all possible enabled transitions[10]. An existence of the internal non-determinism is marked as a warning so that a designer knows about a problem.

In the UML models, it is possible to specify components that represent external components that we use but do not model. If we would like to use classes from such components during BSMs execution we would need to model at least one BSM that will model external non-deterministic choices about possible actions in an external component. Such a BSM should be explicitly marked as belonging to an external component by a designer. In this case, the eventual choice of a transition is given to a designer during the simulation (a designer chooses one transition from the list of enabled transitions).

[10]This is neither *angelic*, *demonic* or *erratic* choice type. It can be seen as one version of a non-exhaustive choice that will be consistent between subsequent user runs of the same scenario with the same initial system's state.

5.7.5 Firing a transition

Firing a transition function is presented in fig. 5.16. Firing a transition first removes source states from an active states configuration in an inside-out manner, i.e. in case of composite states the most inner state is removed first. Exit actions in removed states should be executed immediately when a state is removed from an active state configuration.

Figure 5.16: Firing a transition.

```
1 fireTransition(conf, transition) =
2    leaveSourceStatesInsideOut(conf, transition)
3    reply ← getReply(execute(effect(transition)))
4    enterTargetStatesOutsideIn(conf, transition)
5    return reply
```

An effect of a transition is executed. Effects are specified in the SAL (see sec. 5.8). If a `reply` statement (see section 5.8.2.3) is executed, a result is returned and saved in *reply* variable. If an effect contains call expressions (see section 5.8.2.10), it leads to one or more calls to the *callOperation* function from fig. 5.10.

Then, target states are added to active states configuration in an outside-in manner, i.e. the most outer composite state is added first. Entry actions must be executed immediately after addition of the state to an active state configuration. Finally, the variable *reply* is returned.

5.7.6 Query operations

A query operation is an operation that "leaves the state of the system unchanged" [Obj09, p. 104]. Query operations are therefore useful to implement getter operations in classes.

We define body of a query operation as an *OCL body expression* [Obj06]. This allows evaluating a result of a query operation immediately, without executing a complete run-to-completion step in a BSM. This is particularly useful in our approach because it avoids already the described problem of the deadlock in case of a recursive `self`-call depicted in fig. 5.12. This also makes BSMs layout simpler because a designer does not need to define transitions with triggers for getter operations.

There is an important issue with semantics however: unfortunately, the UML 2.2 meta-model has no means to specify a query operation's body. What the UML has is *bodyCondition*, that is "an optional constraint on the result values of an invocation of" [Obj09, p. 104] an operation. That means that it is not used to *return* a result, but to *check* a result. Additionally, a result of *bodyCondition* must be a `Boolean` value.

To resolve the problem, in our approach, we will define semantics of UML *bodyCondition* to be the same as semantics of OCL body expression, i.e. that it is "used to indicate the result of a query operation" [Obj06, p. 9]. Consequently, we allow *bodyCondition* to be specified in a model as an OCL body expression that can return any type.

5.7.7 Example run of the execution algorithm

In this section, we will look at an example run of the execution of BSMs during the *check-out with toll tag* scenario simulation (see fig. 5.17).

Each object instance presented in fig. 5.17 in the object diagram has its own BSM instance. The BSMs of instances that actively take part in the scenario are presented in fig. 5.18.

Before the scenario simulation is started, a system is instantiated, i.e. all the BSM instances presented in fig. 5.18 are created. A completion event is dispatched to each of the new BSM instances (compare with section 5.7.1.1). In the result, all of the BSMs move from initial states to "Idle" states.

Next, the simulation of the scenario starts. The extension function (compare with fig. 5.5) invokes the *invokeExecution* function (see fig. 5.10, line no. 1) with the first message in the scenario (*vehicleArrives*) as a parameter. The properties of the message are extracted in line no. 2 and then, a call to the function *callOperation* in line no. 3 is executed (*source* in this call is an interaction diagram of the scenario and *target* is *antenna* instance).

The *callOperation* function invocation (line no. 5) creates a new call event based on the given parameters (line no. 6). The new event is placed in the event's queue (that was empty before) in line no. 7.

Because the call is synchronous (the message *vehicleArrives* in the sequence diagram is synchronous), a scheduler is given a task to execute all events necessary to get a result for the newly created event (line no. 9).

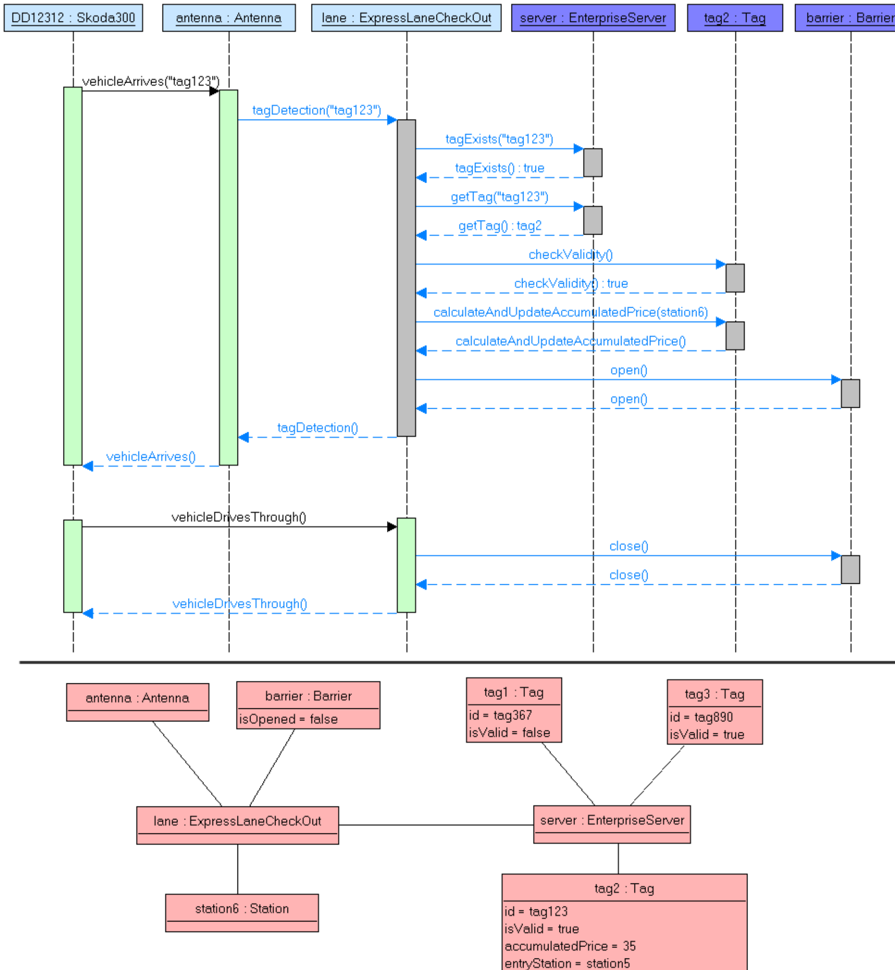

Figure 5.17: The realization of the scenario *check-out with toll tag* and the corresponding object diagram **after** the simulation finished.

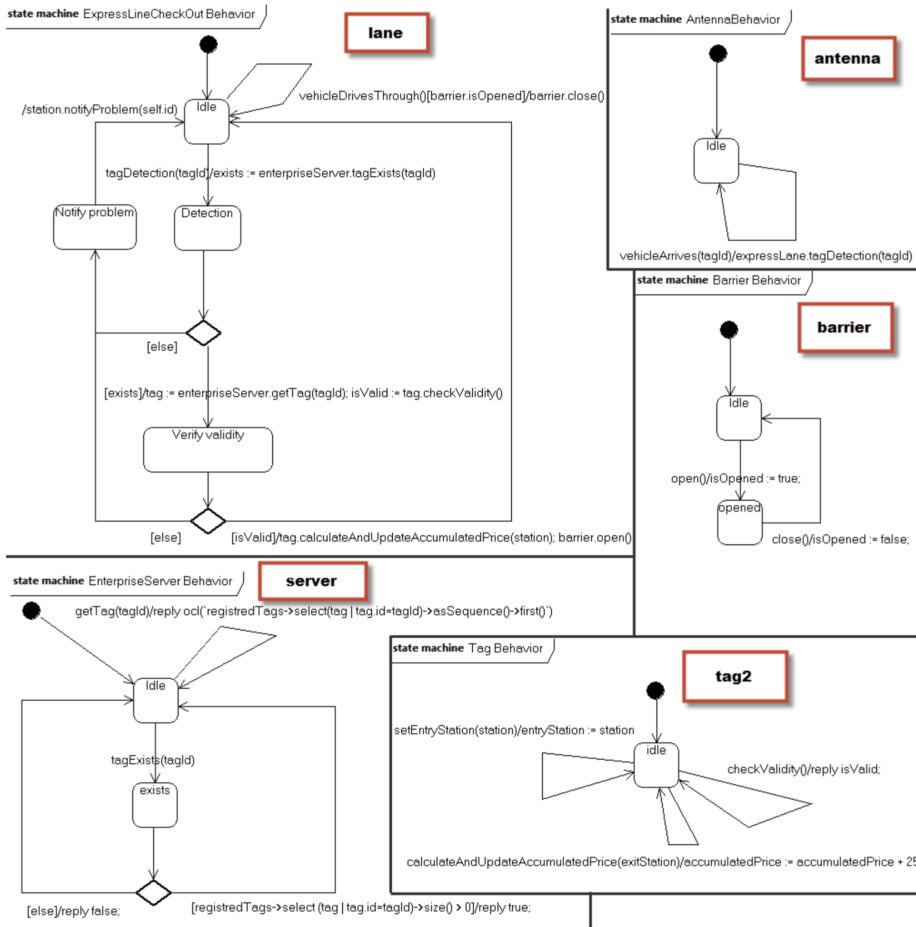

Figure 5.18: The BSMs of instances that actively take part in the *check-out with toll tag* scenario realization. The names of the corresponding object instances are presented in the rectangles.

The loop in line no. 52 in the scheduler calls *executeFirstFromQueue* (line no. 53). Because the newly created event is in the queue, a function *accept* is called in line no. 50.

accept function starts a run-to-completion step in the BSM of the *antenna* instance from checking that an operation is available in the instance (it is declared publicly) and that the arguments ("tag123" of type `String`) conform to the parameters (one `String` parameter) of the called operation (*vehicleArrives*) in line no. 15.

The operation *vehicleArrives* is not a query operation and the *antenna* instance is not executing anything else at that moment so the function moves to line no. 26 where a flag *isExecuting* is set for the *antenna* instance.

The transitions with valid guards for the operation *vehicleArrives* are found in the active state configuration of the *antenna*'s BSM (at that time it contains only "Idle" state). There is only one enabled transition (compare fig. 5.18, *antenna*'s BSM). The enabled transition is chosen deterministically (because it's the choice from a set containing a single element) in line no. 29.

Next, the notification for the observers about the call event is sent in line no. 31 (the observers include *eventFired* function in fig. 5.7):

1. name: vehicleArrives, type: synchCall (from DD12312 to antenna).

The execution of the effect on the chosen transition is performed in line no. 32. The executing effect is specified as "expressLane.tagDetection(tagId)". In the effect, the *lane* object will be selected as a target and the (recursive) invocation to the *callOperation* function (line no. 5). The exact arguments of the call will be:

```
callOperation(antenna, lane, tagDetection, ["tag123"])
```

For a clarity of a description, we will skip the part of the simulation that involves the call of *tagDetection* operation and all the effects (incl. the subsequent calls and replies) that it triggers[11].

After executing, all of the events required to get the result of the synchronous

[11] We can do this because the process of the execution of the subsequent recursive calls and the run-to-completion steps in other BSMs will be similar to the *antenna* instance's BSM execution that we are examining here.

call to *tagDetection*, *reply* is assigned an empty value (compare to `void`[12]) (still in line no. 32). In this case, an empty value is assigned because on the firing transition there is no `reply` statement (see section 5.8.2.3).

Let's look at the active states configuration of *antenna* during the firing of the transition. The "Idle" state is removed from the configuration once the transition started the firing process (see fig. 5.16, line no. 2). After the effect finishes executing, the "Idle" state is added back to the active configuration (see fig. 5.16, line no. 4). The state is added back because, in this case, the firing transition is a self-transition coming back to "Idle" state.

Next, a completion event is dispatched and completion transitions are fired in line no. 33 (in fig. 5.10). We do not have any completion transitions in *antenna*'s BSM outgoing from "Idle" state so we do nothing. *reply* is also not modified in this case, it could be though, if there were any completion transitions with the `reply` statement.

The *reply* variable passes the check in line no. 34 because it has an empty value and the *vehicleArrives* operation should not return any result (see the class diagram in fig. A.1 in the appendix).

Because the call is synchronous, the observers are notified of the new reply event (line no. 37) and of a new call return control event (line no. 38). The *isExecuting* flag is then unset for the *antenna* instance in line no. 44.

Finally, the remaining events are dispatched from the event's queue in line no. 45 and then in line no. 4. In fact, nothing is dispatched because the queue is empty at that time. Dispatching of the remaining events here is useful if we had some asynchronous events waiting in the queue.

At this point, *invokeExecution* function returns to a caller (the extension function). The function *invokeExecution* will be called again for a next message (*vehicleDrivesThrough*) with a target set to *lane* instance. The function will execute similarly to the execution of the message *vehicleArrives* that we have just seen.

[12]This is one of the places where there should be a clear distinction between `null` and an empty value (i.e. `void`). A function can return `null` but it doesn't mean it returned no result at all. In this case, we have a function *vehicleArrives* that does not have any return value declared (in the class diagram) and it should return an empty value.

5.7.8 Other possible semantics of behavioral state machines execution

In this thesis, we have chosen to use the UML run-to-completion semantics that is described in UML specification [Obj09] and formalized in papers like [HG97] and [DDd03]. However, and this is worth noticing, the semantics is not well suited for executions of some types of scenarios involving the synchronous events that would be otherwise natural scenarios in object-oriented systems.

One of the problematic scenarios that was already described is inability to perform recursive synchronous `self`-calls (including the example presented in fig. 5.12). The inability is caused by the deadlock that occurs when one synchronous call invokes the subsequent synchronous call in another objects that then call-backs synchronously to `self`-object that is frozen (`self` is still executing the initiating synchronous call). It is possible to address the problem by using solution described in [TS03]. The solution introduced special type of BSMs that separate them into two types of state machines: protocol state machines (that are here BSMs with removed effects) and *method state machines* that "allow definition of a state diagram in the context of an operation, but do not provide detail about the particular features and behavior of this kind of state diagram" [TS03]. In this solution, the recursive `self` synchronous callback is possible because for each call event, a new instance of *method state machine* will be executed avoiding the deadlock.

The next problematic scenario is a kind of a callback that would not be able execute during a synchronous call event if executed according to our `reply` semantics (see section 5.8.2.3). The scenario is presented in fig. 5.19.

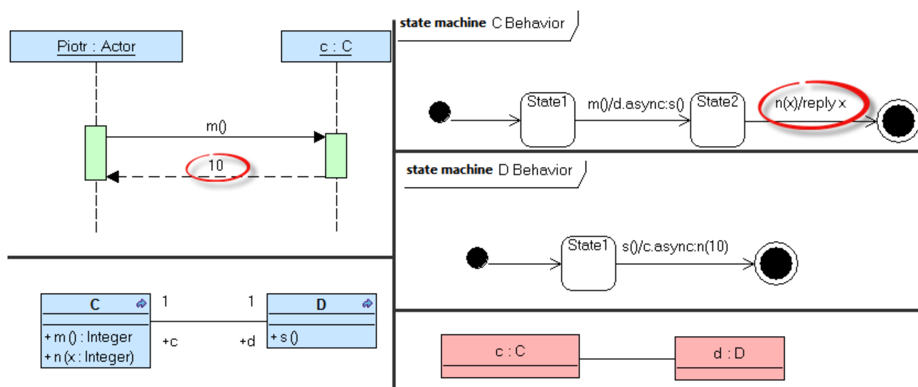

Figure 5.19: The callback scenario that is not possible to execute with our run-to-completion semantics.

After an object *c* receives a synchronous event *m()*, *d* is asynchronously called and in the result, it should call back to *c* value of 10. A reply of *m()* depends on the callback's argument value. This is an error because BSM of *c* will need to reply the result value in one completion step, i.e. it cannot accept the callback event sent by *d* before sending the reply of *m()*.

To resolve the problem, the reply semantics could be changed so that a reply is sent immediately when executed in an effect. This enables delaying a reply in multiple run-to-completion steps. That will, unfortunately, lead to the situation where a reply would be needed for all triggers, even if the triggered operation would not declare to return a result and, therefore, will make BSMs look more complex.

In this project, I have chosen the standard semantics for the run-to-completion, knowing about the possible problems and investigating different options and even implementing some of them in my prototypes. Then, the confrontation of a theory with usability had taken place before a decision to keep the standard semantics. The main reasons for not using other semantics of BSMs execution is to keep close as possible to the original UML semantics.

5.8 The Simple Action Language

Effects on transitions in BSMs can contain behavior expressions. A behavior expression is executed when a transition is fired. The UML does not define any particular language to be used for behavior expressions; instead, to allows choosing an action language for a model.

Simple Action Language (SAL) was defined during this Master's project to specify behavior expressions on transitions in BSMs. The language supports basic Integer and Boolean operations and partially maps to UML Actions. The mapping includes *CallAction* and *ReplyAction* defined in the UML standard.

In this section the SAL language is described. Firstly, the SAL grammar is described, and then examples of usage are presented, and then the semantics of the language is described. In the end of the section, in a short summary, some limitations of the language are pointed out.

5.8.1 SAL grammar

The SAL uses a textual representation. The following grammar in EBNF notation describes the SAL grammar:

⟨*stat-list*⟩ ::= ⟨*statement*⟩ (‘;’ ⟨*statement*⟩)*

⟨*statement*⟩ ::= ⟨*reply*⟩ | ⟨*assignment*⟩ | ⟨*expression*⟩ | ⟨*empty*⟩

⟨*reply*⟩ ::= ‘reply’ ⟨*expression*⟩

⟨*assignment*⟩ ::= ⟨*identifier*⟩ ‘:=’ ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*Boolean expression*⟩
  | ⟨*integer expression*⟩
  | ⟨*Boolean constant*⟩
  | ⟨*integer constant*⟩
  | ⟨*null constant*⟩
  | ⟨*string constant*⟩
  | ⟨*OCL expression*⟩
  | ⟨*collection expression*⟩
  | ⟨*navigation expression*⟩
  | ‘(’ ⟨*expression*⟩ ‘)’

⟨*Boolean expression*⟩ ::= ⟨*expression*⟩ ‘and’ ⟨*expression*⟩
  | ⟨*expression*⟩ ‘or’ ⟨*expression*⟩
  | ‘not’ ⟨*expression*⟩

⟨*integer expression*⟩ ::= ⟨*expression*⟩ ‘+’ ⟨*expression*⟩
  | ⟨*expression*⟩ ‘-’ ⟨*expression*⟩
  | ⟨*expression*⟩ ‘*’ ⟨*expression*⟩
  | ⟨*expression*⟩ ‘/’ ⟨*expression*⟩
  | ⟨*expression*⟩ ‘%’ ⟨*expression*⟩

⟨*Boolean constant*⟩ ::= ‘true’ | ‘false’

⟨*integer constant*⟩ ::= ‘-’? ⟨*digit*⟩+

⟨*null constant*⟩ ::= ‘null’

⟨*string constant*⟩ ::= ‘"’ `string` ‘"’

⟨*OCL expression*⟩ ::= ‘ocl(´’ `OCL query` ‘´)’

⟨*collection expression*⟩ ::= '[' (⟨*expression*⟩ (',' ⟨*expression*⟩)*)? ']'

⟨*navigation expression*⟩ ::= ⟨*identifier*⟩
| ⟨*call expression*⟩
| ⟨*navigation expression*⟩ '.' ⟨*identifier*⟩
| ⟨*navigation expression*⟩ '.' ⟨*call expression*⟩

⟨*call expression*⟩ ::= ⟨*call*⟩ | ⟨*call*⟩ ⟨*selector*⟩

⟨*call*⟩ ::= ⟨*synchronous call*⟩ | ⟨*asynchronous call*⟩

⟨*synchronous call*⟩ ::= ⟨*id*⟩ '(' (⟨*expression*⟩ (',' ⟨*expression*⟩)*)? ')'

⟨*asynchronous call*⟩ ::= 'async:' ⟨*id*⟩ '(' (⟨*expression*⟩ (',' ⟨*expression*⟩)*)? ')'

⟨*identifier*⟩ ::= ⟨*id*⟩ | ⟨*id*⟩ ⟨*selector*⟩

⟨*id*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩)*

⟨*digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*letter*⟩ ::= ['a'-'z', 'A'-'Z', '_']

Operations precedence is (from highest to lowest): `not`, `%`, `/`, `*`, `-`, `+`, `and`, `or`.

Examples of the SAL valid statements are:

- `x := 10`
- `p := isInPState()`
- `obj.get(apple).equals("apple", true)`
- `reply (ocl(´obj.apples->size()´) + 1) * -8`

5.8.2 SAL semantics

5.8.2.1 List of statements

List of statements (stat-list) is list of SAL statements in which each statement is separated by a semicolon. The statements are executed from the leftmost to the rightmost. For example `op1(); op2()` will execute `op1()` first and then `op2()`. The semicolon after the rightmost statement can be skipped.

5.8.2.2 Expression

Every expression is the SAL language is evaluated to a collection of zero or more UML value specifications. If in a resulting collection there is only one element, the collection may be treated as a single value depending on a context of how it is used; the collection can be implicitly converted to the single value.

The UML standard profile itself does not have any value specifications for collections as primitive types. The only way to express that an element specifies a collection of values in UML is to provide lower- and upper-bounds for a multiplicity of the element. The SAL therefore must use its own collection types during evaluation of an expression and then match it to the UML multiplicity of an element when the expression is used in the specific context of this element. E.g. when we assign a value of an expression to a variable x, the SAL context-wise check will assure that the expression's value can be assigned to x by checking the type and the multiplicities of x and then comparing them with the type and the number of elements in a collection that is a result of the expression. If the type or the number of elements in the collection is invalid, it is an error.

5.8.2.3 Reply statement

A reply statement is the SAL representation of the UML *ReplyAction*. The reply statement corresponds to a trigger that was the last trigger used to accept a synchronous call event of an operation that returns a result. The reply statement accepts return values of this operation and completes the execution of the synchronous call in a BSM. After accepting, at the end of a run-to-completion step, values of the reply statement are returned to a caller. An example of the reply statement used in a BSM is presented in fig. 5.20.

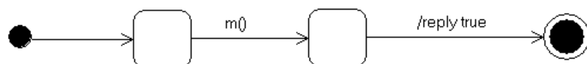

Figure 5.20: The example of the SAL reply statement for a call of operation *m()* in a BSM.

In an unusual situation, there may be more than one reply statements executed during one run-to-completion step of a BSM execution after an operation call. In this case, the value of the last reply statement completely overrides values of

previous reply statements and will be the one that returns a result to a caller. This situation is also flagged as a warning.

If at the end of a run-to-completion step there was no reply statement executed, and the current event processed by a BSM is a call event of an operation that declares to return a result, this situation is considered an error. An error also occurs if a reply statement is executed during initialization of BSMs if there is no triggering event involved (during execution of a completion event during initialization of the run-to-completion). Reply statements must not be executed for asynchronous call events or call events of operations that do not declare any return types.

5.8.2.4 Assignment

Assignment is the SAL representation of a mixture of few UML actions, e.g. *WriteVariableAction*, *WriteLink* and *WriteStructuralFeature*. The assignment can create a new local variable (if it didn't exist before assignment) or change an existing variable.

All variables' names must be accessible from a behavioral classifier that owns a BSM (owner)[13] in which an assignment is declared. In other words, we need to be able to resolve a model context of a variable.

Depends on what is a model context of a variable that we assign to, there may be many different contexts of an assignment:

An assignment to attributes of an owner is made in a context of an instance specification for which a BSM of the owner is executed. After the assignment is successfully executed, there will be a slot in the instance specification with a name corresponding to the attribute's name used.

An assignment to a new local variable is also made in a context of an instance specification for which a BSM of the owner is executed. First, when we create a new local variable, an attribute (a model context) must be created in the owner[14]. The SAL evaluator creates an attribute (with private visibility) in the owner for each created local variable. A type and

[13] We will be later using *owner* in a text that was defined here.

[14] The model context element specifies a type and multiplicities of values in a variable. The OCL evaluators require model context for identifiers to evaluate. If we would like to use a local variable created by an SAL assignment in OCL guards then we must have the model context for this variable.

multiplicities of the new attribute will be inferred from an expression that we assign to. The assignment then behaves like an attribute assignment.

An assignment to associations' role names is made in a context of instance specifications that represent links of an association that we assign to.

There are two types of assignments in the SAL:

1. *replace all* assignment uses syntax without a selector (see section 5.8.2.5), e.g. `x := 2`. This assignment replaces all values in x. If x was a variable with lower- and upper- multiplicities equal to 1, right side of the assignment must evaluate to single valued collection. If x was a collection, all previous values will be removed and then value *2* will be added to the collection x;

2. *replace one* assignment uses syntax with a selector, e.g. `x[1] := 2` can be used for ordered collections to replace or add only one selected value in the collection. In this assignment, the right side of the assignment must evaluate to one valued collection. If the selector index points to out of bounds, it is considered an error with an exception of selector index pointing to "-1", and then a value at the end of the collection is replaced[15].

5.8.2.5 Selector

Selector is used to specify an index of a value in an ordered collection. The selector must evaluate to a single integer value greater or equal to 1 or equal to -1. -1 index value represents the last element of the collection (it represents the UML infinity number). The index of the first value in a collection is 1. An index 0 is undefined and is considered an error.

5.8.2.6 Boolean expression

Defines a set of the standard Boolean operations: *not*, *and*, *or*. The operations can be evaluated on single-valued operands only. E.g. `false or (true and not false)` will evaluate to `true`.

[15]This semantical rule is inspired by the UML *AddStructuralFeatureValueAction* where adding an element at an infinite index results in adding the element to an ordered collection at the end.

5.8.2.7 Integer expression

Defines a set of the standard integer operations: *addition* (+), *subtraction* (-), *multiplication* (*), *division* (/), *modulo* (%). The operations can be evaluated on single-valued operands only. E.g. `2 + 5 * 3` will evaluate to `17`.

5.8.2.8 OCL expression

The SAL gives a possibility to evaluate an OCL query to a result that can be then used as a SAL collection. The link between SAL and OCL is provided by an OCL expression in SAL, e.g. `ocl(´obj.apples->size()´)` returns size of *obj.apples*. The character ´ (acute accent) is used for escaping the OCL expression.

A main reason for introducing the OCL expression in the SAL is a great expressiveness of the OCL that enables evaluating of more advanced queries.

One could argue that it is better to borrow the OCL syntax and use it in the SAL syntax. I agree with this view completely; however a required amount of work to accomplish it would be rather big and could be a foundation of a separate thesis (at least).

5.8.2.9 Collection expression

The SAL gives a possibility to create own collections. At the moment, only sequences can be created by the SAL evaluator (ordered lists of values allowing duplicates). All collections in the SAL are flattened[16]. The SAL automatically flattens nested collections by extracting their values and adding to a resulting collection. E.g. `[a, b, [c, [d]], e]` evaluates to `[a, b, c, d, e]`. Creation of an empty collection is possible by stating `[]`.

The collection expression was introduced to the SAL in order to give a flexibility to call operations that have parameters with upper multiplicities greater than 1 or lower multiplicities equal to 0. A single argument of a call to such operation can therefore accept more than one value or no values, e.g.

- `m([1, 2, 3])` is a call to operation *m* with one argument of a collection containing three elements;

[16] This is similar approach that is used for the OCL collections.

- `m2([], 8)` is a call to operation m with two arguments, the first argument is an empty collection.

5.8.2.10 Call expression

The call expression is SAL representation of UML *CallAction*. The call expression accepts a number of arguments and matches them to parameters of a specified operation. Call expressions may be synchronous or asynchronous[17]. We can call operation asynchronously by adding *async:* before the name of the operation, e.g. `async:op(true)`.

A synchronous call invokes a specified operation with given arguments and waits till an execution of the operation completes. An asynchronous call does not wait for the operation to complete. If a synchronous operation returns result values then the call expression evaluates to these values. It is possible to assign a variable to a call expression that evaluates to a result of an operation called, e.g. `result := m(2, 3)`.

By default, a call expression context is an instance specification of the owner of BSM, i.e. operations called in call expressions must be present in the owner. The context can be changed by using navigation expressions (see section 5.8.2.12).

It is possible to specify a call expression in a context of a collection containing many instance specifications. If all of the instances have an operation with a given name then the call expression calls each instance in the collection and flattens the results of the called operations into one resulting collection.

5.8.2.11 Identifier

Identifier in the SAL can refer to an association end role name or to an attribute of an instance specification for which the identifier is evaluated.

If an identifier expression refers to an attribute, and then the identifier expression returns all values from a slot that represents this attribute. If there is no slot for this attribute in the instance specification (the attribute was not initialized), `null` value is returned.

If an identifier expression refers to an association end role name, and then the

[17] In the UML, beside synchronous and asynchronous calls to operations, there exists a signal element to express asynchronous communication.

identifier expression returns all instance specifications connected to the current instance specification with links representing this association. If there are no links for this association, `null` value is returned.

If there is an ambiguity between names of an association end and an attribute (see fig. 5.21 for an example), the attribute will take precedence during the evaluation of the identifier.

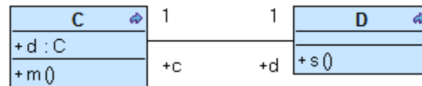

Figure 5.21: The ambiguity between names of the association end *d* and the attribute *d*.

`self` identifier provides self-identity. `self` always returns the current context of the expression.

A selector (see section 5.8.2.5) may follow an identifier to filter a single element from a collection.

5.8.2.12 Navigation expression

Navigation expression is used to change a context of an evaluation for call expressions and identifiers. A navigation expression navigates through instance specifications connected with links by using provided role names of ends of associations connected to the current context. If a role name is missing on an association, the navigation through the association is not possible.

For navigation expressions, the default context is implicitly `self`, e.g. `d.s().p` has the same meaning as `self.d.s().p`.

Navigation expressions are allowed navigating only when the current context (left side of the expression) is single valued. Any other case is an error. To navigate through multiplicity many ordered collections of instance specifications connected by links, it is necessary to use selector to specify a single instance for navigation.

An example of a navigation expression used to change context of a call expression is presented in fig. 5.22. The expression `d.s()` in the example is evaluated in a context of instance *c* and uses role name *d* of an association to navigate

to instance d of class D connected with a link which is an instance of the association.

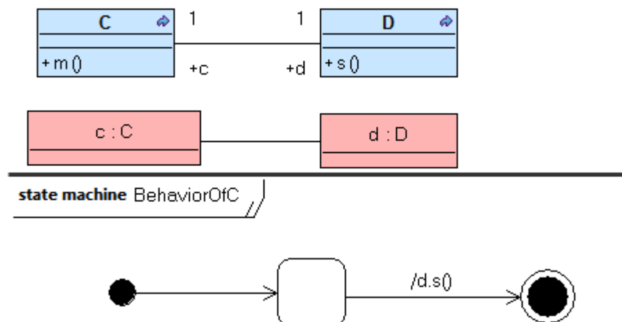

Figure 5.22: The example of the SAL navigation expression used in BSM defining behavior of class C.

5.8.3 SAL summary

The SAL provides basic functionality of calling operations (synchronously and asynchronously) from BSMs and to send and receive results of operations. SAL provides functionality of reading and writing variables by changing slots' values and links. The support of collections is minimal, but it is enough to call operations with parameters that have multiplicity many elements. It is also possible to read and write one specified element of a collection.

The current version of the SAL does not have a constructor and a destructor of instances (with exception of links (that are also instances) that can be created and removed by the assignments). The creation and deletion was out of the scope of the project. Moreover, it was just not necessary to have these functionalists for chosen use cases in the case study (described in section 4). The creation and deletion can be added later without a big effort.

The SAL does not also have advanced filtering mechanisms available in the OCL. Instead, we provide a connection to OCL by introducing the OCL expression in SAL where all those advanced constructs are available for a user. The only reason we propose this solution is tight deadline of the project and the priorities of SAL in the project.

There exist more complex action languages that have a support for all types of

actions, e.g. Jumbala [Dub06].

5.9 Verification of protocol state machines

Protocol state machines (PSMs), in the presented approach, are defined in a context of interfaces. Although the UML semantics allows using protocols in context of any classifier and port, we concentrate on the context of interfaces. The presented technique of checking PSMs can be easily applied to check protocols in a context of any classifier or port but it is out of the scope of this project[18].

In this section, we will see an algorithm to verify PSMs and an example run of the algorithm on a model of the toll with components (see appendix B).

5.9.1 Verification of protocol state machines algorithm

A separate instance of a PSM checking algorithm is created for each object instance of a class that implements an interface (which has a PSM defined). E.g. an object instance x is of class C. C implements two interfaces i1 *and* i2, each with a PSM defined. There will be two instances of the PSM checking algorithm for x, one instance of the algorithm verifies for the PSM of *i1* and another instance of the algorithm verifies the PSM of *i2*.

The PSMs verification algorithm uses *PSM states*. A PSM state represents one possible path in a checked PSM that can be taken. The algorithm constructs all possible PSM states according to events that are generated (notified) during a BSMs execution. Invalid PSM states are eliminated. Therefore, after an event was processed, there must be at least one PSM state for each satisfied PSM that is being verified. If there are no PSM states after an event was processed, a conclusion is that a checked PSM was violated during the BSMs execution and an error is produced.

The function *eventOcurred* in the algorithm presented in fig. 5.23 observes an execution of BSMs and filter events for referred operations in a checked PSM (line no. 5). The function knows about a list of all possible PSM states at a current moment of the execution (variable *psmStates*) that conform to a current

[18]In summer semester of 2012, students of *System Integration* course were instructed to use protocols in the context of interfaces. In the previous years, protocols were defined in the context of ports.

Figure 5.23: PSMs verification algorithm.

```
1  init() =
2     psmStates ← initializePSM()
3
4  eventOcurred(event) =
5     if ((not isReply(event) and getTarget(event) = myInstance)
          or (isReply(event) and getSource(event) = myInstance))
          and isReferred(event)
6        localStates ← psmStates
7        foreach psmState in localStates
8           if isReply(event)
9              postcall(psmState, event)
10          else
11             precall(psmState, event)
12       eliminateEliminatedAndDuplicatedPSMStates()
13       if psmStates = ∅
14          error
15
16 precall(psmState, event) =
17    transitions ← getTransitionsWithValidPreconditions(
          activeConf(psmState), event)
18    if transitions = ∅
19       setForElimination(psmState)
20    else
21       toTake ← choose(transitions)
22       foreach transition in transitions
23          if transition ≠ toTake
24             newState ← newPSMState(psmState)
25             psmStates ← psmStates ∪ (newState)
26             handleEvent(newState, event, transition)
27       handleEvent(psmState, event, toTake)
28
29 handleEvent(psmState, event, transition) =
30    if isSynchronous(event)
31       setLastSynchEventTransition(psmState, transition, event)
32    else
33       setLastSynchEventTransition(psmState, null, null)
34       if validatePostCondition(transition)
35          takeTransition(psmState, transition)
36          takeCompletionTransitions(psmState)
37       else
38          setForElimination(psmState)
39
40 postcall(psmState, event) =
41    transition ← getLastSynchEventTransition(psmState, event)
42    if null ≠ transition
43       if validatePostCondition(transition)
44          takeTransition(psmState, transition)
45          takeCompletionTransitions(psmState)
46       else
47          setForElimination(psmState)
```

execution trace and satisfy pre- and post-conditions. The PSM states that do not conform to the execution are eliminated (line no. 12); duplicates of PSM states are also eliminated (line no. 12). There must be at least one active PSM state after processing of an event otherwise a checked PSM is not valid (line no. 13). The list of references to PSM states must be copied (in line no. 6) before entering the loop (in line no. 7) to avoid possible modification of it during the event processing.

The *precall* function is called for call events (line no. 16). Transitions outgoing from an active state configuration of PSM that are specified for an operation referred in the event and have satisfied pre-conditions are calculated (line no. 17). If no such transitions are found, the PSM state is marked for elimination (line no. 19). If some transitions are found, one of them is chosen for the current PSM state (line no. 21) and for all other transitions, new PSM states are generated that are copy of this state (line no. 24) but take other possible transition (line no. 26). Then, the chosen transition in the current PSM state is taken (line no. 27).

In case of synchronous events, the transition to take and the completion transitions are not taken immediately in a PSM state, instead, the taken transition and the triggering event are stored in the PSM state (line no. 31) for later use (when a reply event arrives and *postcall* is called).

For asynchronous events, post-conditions are validated immediately (line no. 34), a transition to take is taken (line no. 35) and completion transitions are calculated (line no. 36). Note, during completion transition calculation, new PSM states can also be created (if there are more than one completion transitions outgoing from active configuration). If post-condition on the transition to take is not valid, a PSM state is set for elimination (line no. 38).

The *postcall* function is called for reply events (line no. 40). The function finds a corresponding transition for the last called synchronous call event in the current PSM state (line no. 42), a post-condition on the transition is validated (line no. 43). If the post-condition is satisfied, the transition and completion transitions are taken, otherwise the PSM state is set for elimination (line no. 47).

The algorithm splits checking of transitions triggered by synchronous call events in two phases (*precall* and *postcall*). It is because a post-conditions evaluation requires a contribution of pre- and post-states. A post-state is only available at the time a reply event arrives. In contrast, post-conditions for asynchronous events are evaluated immediately in *precall*.

5.9.2 Unreferred operations

Only operations referred in PSMs are used to fire protocol transitions. Unreferred operations "can be called for any state of the protocol state machine, and do not change the current state" [Obj09]. Therefore, in our approach it is allowed to call unreferred operations that are declared in interfaces but not referred in PSMs at any moment.

5.9.3 Pre- and post-conditions on protocol transitions

Protocol transitions can have pre- and post-conditions. A pre-condition "specifies the condition that should be verified before triggering the transition" [Obj09] while a post-condition is "the condition that should be obtained once the transition is triggered." [Obj09] In the approach presented in this thesis, pre- and post-conditions are specified in OCL language. [Obj06] Pre- and post-conditions on protocol transitions represent OCL pre- and post-conditions for operations.

On protocol transitions that have a referred operation, a pre-condition is evaluated before the operation was called; a post-condition immediately after a result of the operation is returned. In post-conditions on protocol transitions, it is possible to use `result` variable that refers to the operation result and variables with post-fix **@pre**, e.g. *x@pre* to refer to the value of x from before the operation call (from pre-state). It is important to note that for validation of pre-condition, we need to have a system state at a moment of the operation call (pre-state). To validate a post-condition, we need to have a pre-state, a post-state, and operation result.

The semantics for post-conditions on protocol transitions fired by asynchronous calls define that they are evaluated immediately after the moment of the call. Using `result` in post-conditions in case of events of asynchronous calls is considered an error because the asynchronous calls do not return results.

On protocol transitions that do not specify referred operation, pre-conditions are evaluated at the same moment as post-condition. It is possible to use `result` or post-fix **@pre** on these transitions as well, because they are fired at the moment when a post-state is already known.

If a post-condition failed on a protocol transition, the associated PMS state will be eliminated[19].

[19]This is one of the UML semantics variation points that we need to specify.

5.9.4 State invariants

State invariants are "conditions that are always true when this state is the current state." [Obj09] In the algorithm presented in fig. 5.23, an evaluation of state invariants is not visible. State invariants are not easy to check without observing all changes to a state of a system. State invariants are outside the scope of this project.

A possible solution to checking state invariants includes adding broadcasting of *change events* on every change of a system state during BESs execution, e.g. during an assignment (in the SAL), and listening to the events in PSM verification algorithm. If an invariant would be violated in a system state after a change event is processed, a current PSM state would be eliminated.

5.9.5 Example run of the verification of protocol state machines algorithm

In this section, we will see an example run of the PSM verification algorithm on the *check-in with tag* scenario. In this example we use the model of the toll system with components. The division to components decouples the system and introduces interfaces between different parts of the system allowing definition of PSMs in context of the interfaces.

Components and interfaces in the toll system are presented in fig. 5.24. The detailed class diagram of the toll system is presented in fig. 5.25. The BSMs of these classes are presented in appendices A and B.

Check-in with toll tag scenario and its realization after the scenario was successfully run are presented in fig. 5.26; the system states (object diagrams) before and after the scenario simulation are shown in fig. 5.27.

During the example run, we will be interested in two protocols: *BarrierToLane* (fig. 5.28) and *EnterpriseServerToLane* (fig. 5.29).

There are two separate instances of the algorithm running for each object implementing the interfaces with PSM (in this case: for *barrier* object with *BarrierToLane* protocol and for *server* object with *EnterpriseServerToLane* protocol).

At the beginning of execution of the scenario, in the initial phase of the algorithm in fig. 5.23, *init* function is called (line no. 1). Initial PSM states for the object instances implementing the interfaces with protocols are created (line no. 2).

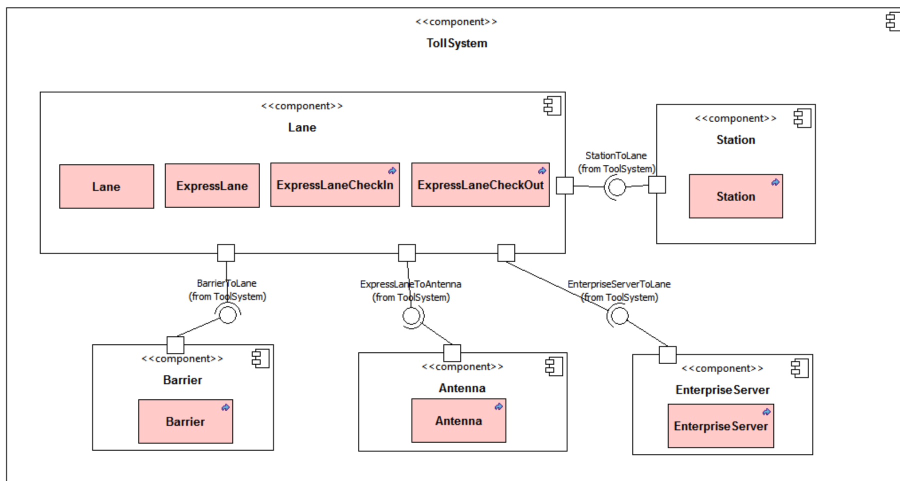

Figure 5.24: The components and interfaces of the toll system.

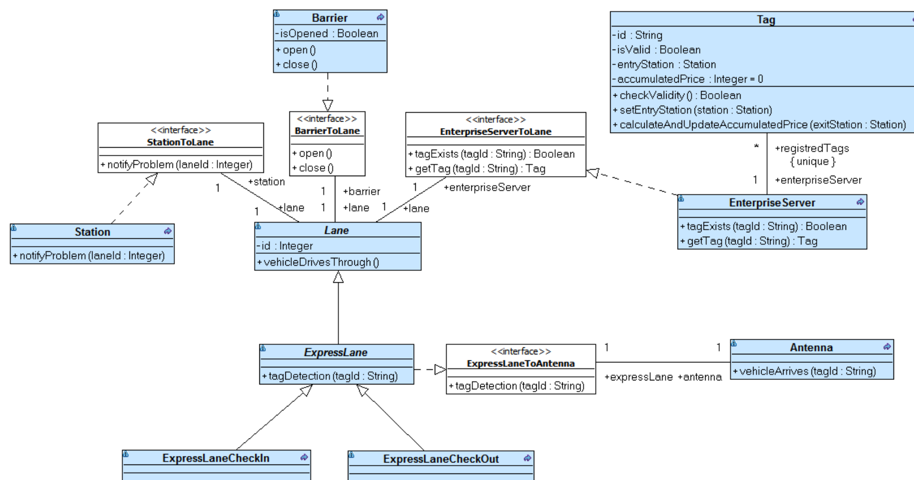

Figure 5.25: The class diagram of toll system with components. The interfaces visible here are defining borders of the components presented in fig. 5.24.

They are presented in fig. 5.31 and 5.30 marked with no. 1 markers in both figures.

After the simulation is started, the two PSM verification algorithm instances observe the BSMs execution. The first two execution call events to arrive are:

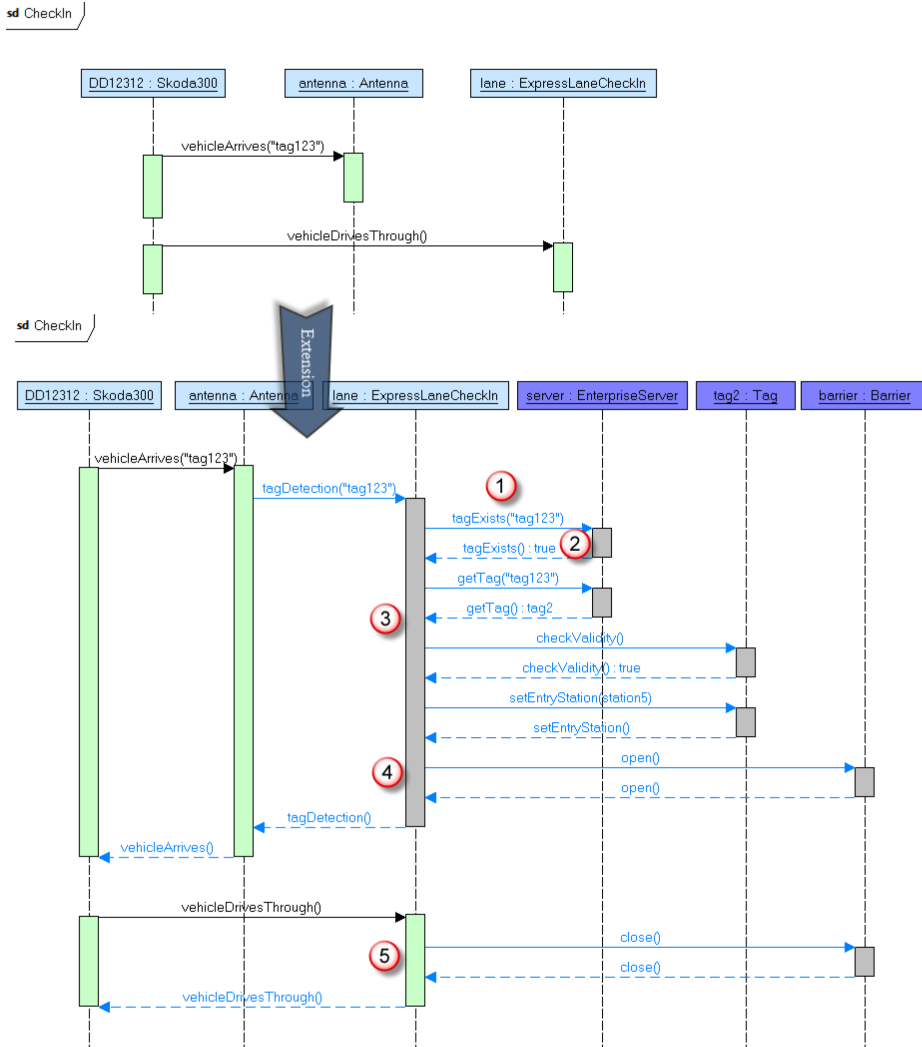

Figure 5.26: Scenario *check-in with toll tag* and its realization. Added elements in the realization of scenario have distinct colors. Numbers in circles are markers referred in a text.

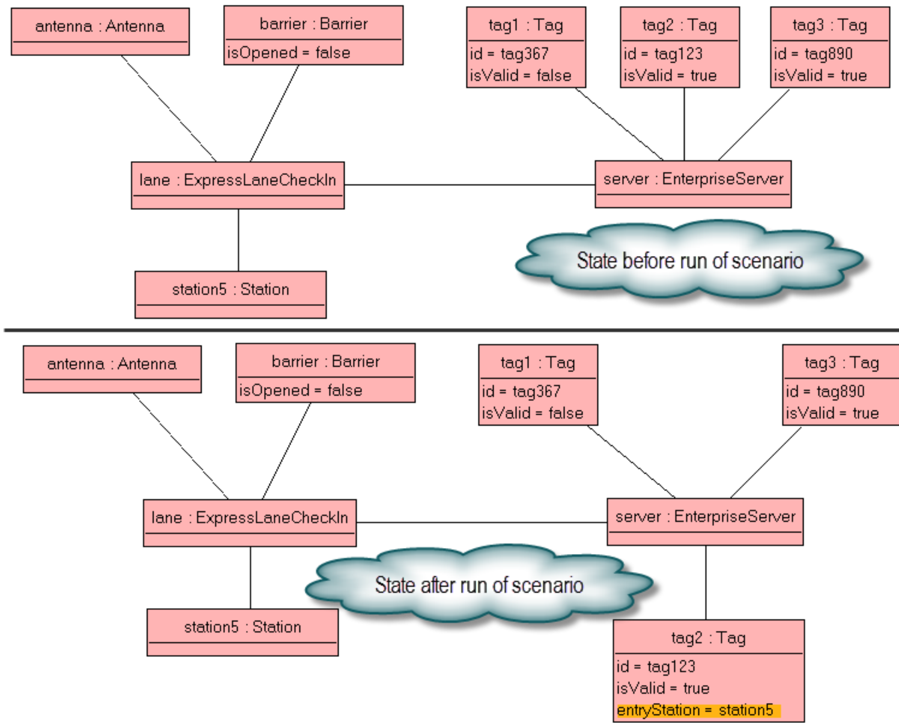

Figure 5.27: The states of the toll system captured before and after the scenario *check-in with toll tag* was simulated. An entry station in *tag2* was set to *station5* during the simulation.

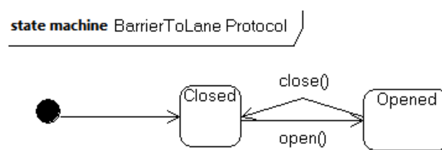

Figure 5.28: BarrierToLane PSM. Barrier must be opened before it can be closed again.

1. name: vehicleArrives, type: synchCall (from DD12312 to antenna)
2. name: tagDetection, type: synchCall (from antenna to lane)

They are both filtered out by the instances of the algorithm in line no. 5 because they are targeted to other object instances (not to *barrier* or to *server*).

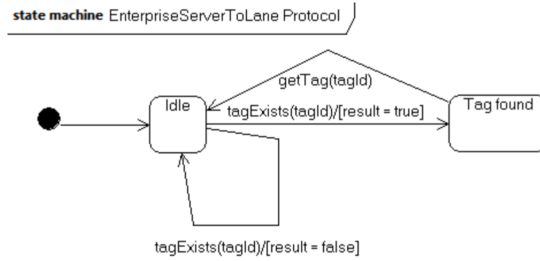

Figure 5.29: EnterpriseServerToLane PSM. The protocol forces to check for existence of a tag and receive a positive result (checked in post-condition) before *getTag()* can be called.

A next call event arriving is of more interest to us:

3. name: tagExists, type: synchCall (from lane to server)

This event is targeted to *server* (see marker 1 in fig. 5.26) and a corresponding operation is referred in *EnterpriseServerToLane* protocol (fig. 5.29); therefore it passes the filter for the algorithm's instance to which *server* object instance is assigned. Because it is a call event, *precall* is called for PSM state 1 in fig. 5.30 (only PSM state 1 exists at this point (marker 1)).

A result assigned to *transitions* in line no. 17 is two transitions available from *Idle* state having referred operation *tagExists*. One transition is chosen (line no. 21) and for other transitions, new PSM states are created (we have one more transition so PSM state 2 is created in fig. 5.30 (marker 2) that is a copy of PSM state 1 but firing other transition). Because the processed event is a synchronous call event, both PSM states do not change the active state configurations. They remember the transition to take (line no. 31) and will, possibly, take it when a reply event will arrive (and post-state will be known). A next event that arrives is the reply event:

4. name: ReplyOftagExists, type: reply (from server to lane)

Because it is the reply event for the referred operation and a source of it is *server* instance, it passes the filter in line no. 5 in *EnterpriseServerToLane* PSM verification algorithm instance. Then, *postcall* function is called (line no. 9) for both PSM state 1 and PSM state 2. In *postcall* function, the post-conditions are checked on both transitions referring to *tagExists* in both PSM states.

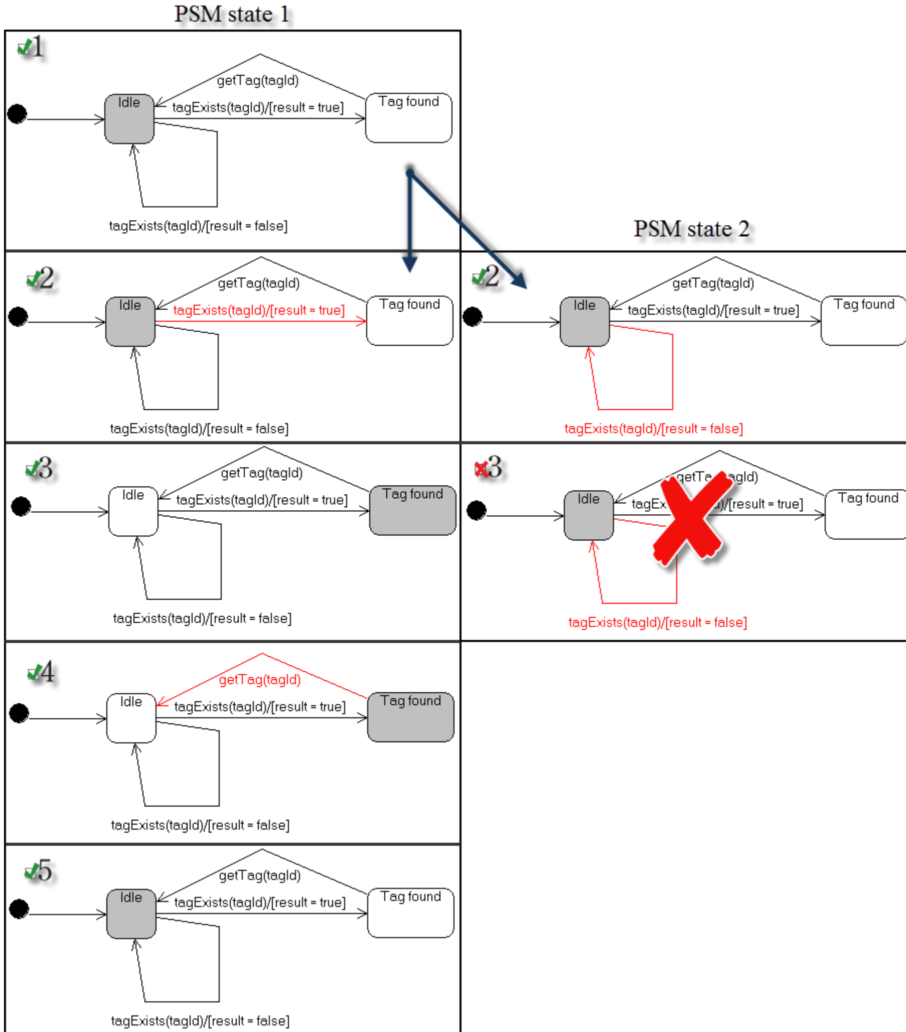

Figure 5.30: *EnterpriseServerToLane* protocol validation progress during the simulation of the scenario. In step 2, a second PSM state is created from the first one that takes another transition with different post-condition. The post-condition fails for PSM state 2 and the state is eliminated.

Because we know (see marker 2 in fig. 5.26) that *result* is equal to `true` during the simulation, condition in line no. 43 fails for the PSM state 2 and this state is eliminated (see fig. 5.30 (marker 3)). In contrast, the post-condition in PSM state 1 is satisfied and this PSM state changes the active configuration to *Tag found* state.

Even though the PSM state 2 was eliminated, the PSM state 1 is still present and the condition in line no. 13 is not satisfied[20].

The next two events that arrive are:

5. name: getTag, type: synchCall (from lane to server)
6. name: ReplyOfgetTag, type: reply (from server to lane)

First of these events causes remembering of the transition *getTag* for taking in *EnterpriseServerToLane* protocol for *server* instance (see fig. 5.30 (marker 4)). The second event, a reply, is going to change the active state configuration of PSM state 1 back to *Idle*.

From this moment (see marker 3 in fig. 5.26), in this scenario, there will be no more events affecting *EnterpriseServerToLane* protocol and there will be always at least one PSM state for this protocol meaning this protocol is consistent with the BSMs execution.

The next few events are not interesting from a point of view of any protocol and we will skip describing them in detail:

7. name: checkValidity, type: synchCall (from lane to tag2)
8. name: ReplyOfcheckValidity, type: reply (from tag2 to lane)
9. name: setEntryStation, type: synchCall (from lane to tag2)
10. name: ReplyOfsetEntryStation, type: reply (from tag2 to lane)

The next two events (see marker 4 in fig. 5.26) affect validation of *BarrierToLane* protocol for *barrier* object presented in fig. 5.31:

11. open, type: synchCall (from lane to barrier)

[20] If the condition in line no. 13 was satisfied, i.e. if we had no PSM states left, it would be the error of the PSM verification.

12. name: ReplyOfopen, type: reply (from barrier to lane)

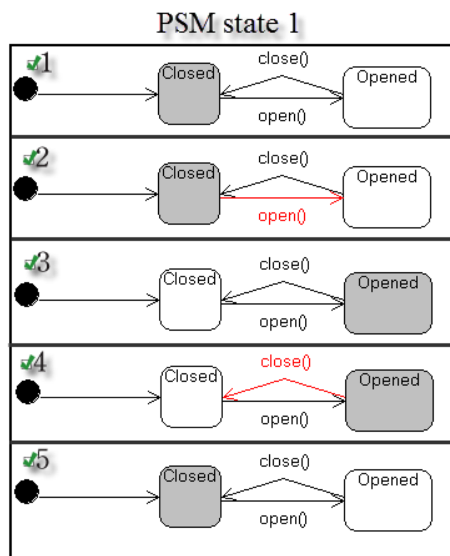

Figure 5.31: *BarrierToLane* protocol validation progress during the simulation of the scenario. During the validation only one PSM state is created (PSM state 1).

open event causes PSM state 1 in fig. 5.31 to mark a transition with a referred operation *open* (marker 2). Then, a reply event *ReplyOfopen* causes the transition to fire and PSM state 1 to change active state configuration to *Opened* (marker 3).

Then, the next events that we skip (they are filtered out by algorithm) are:

13. name: ReplyOftagDetection, type: reply (from lane to antenna)
14. name: ReplyOfvehicleArrives, type: reply (from antenna to DD12312)
15. name: vehicleDrivesThrough, type: synchCall (from DD12312 to lane)

More interesting events for protocols are (see marker 5 in fig. 5.26):

16. name: close, type: synchCall (from lane to barrier)

17. name: ReplyOfclose, type: reply (from barrier to lane)

These two events also affects the validation of *BarrierToLane* protocol causing firing of a transition with referred operation *close()* (see fig. 5.31 marker 4 and 5).

The last event in the execution of this scenario is also filtered out:

18. name: ReplyOfvehicleDrivesThrough, type: reply (from lane to DD12312)

Chapter 6

Tool

This chapter is about *EsculapaUML* tool that was developed for the purpose of the thesis. The first section contains general information about the tool and a methodology used; the next sections describe the functionalities, design and implementation notes of the tool. In the last section, testing methods are summarized.

6.1 General information

6.1.1 Name and license

For the purpose of this thesis the tool *EsculapaUML* was developed. The name EsculapaUML was chosen at the beginning of the project and comes from Greek god Asclepius. Tool's logo (see fig. 6.1) uses symbol of Rod of Asclepius. In the original context of medicine it is (according to Wikipedia):

> Snake as a symbol of rejuvenation, treatment of snakebite, and winding Guinea worms on a stick to remove them from the body.

Of course, in the UML consistency checking terminology it is:

- extend the life of complex UML models by making easy to introduce con-

sistent changes into them;

- if a consistency problem occurs during the design, show the reason of it in UML model;
- help with debugging of execution of UML model to facilitate detection of inconsistencies.

The tool is open-source and distributed on *Eclipse Public License 1.0*[1]. The tool (and the sources) can be downloaded from its web-site: `http://code.google.com/a/eclipselabs.org/p/esculapa-uml/`. The web-site contains also installation and running instructions.

Figure 6.1: Branding logo of EsculapaUML.

6.1.2 Agile methodology

This Master's project (including report and the tool) was executed in the agile methodology. We used this methodology because that at the beginning of the project it was unknown what will be the final outcome of the work so the decision was to plan just a little into a future.

The time-frame of the project was divided into 25 iterations (each of them one week long). The project itself was divided into small use-cases and even smaller tasks. For each of the task a projected priority and difficulty of realization was assigned. The white-board in the office was used during the project (see fig. 6.2). The yellow sticky notes represent the scenarios for the implementation of the tool and white ones the tasks for writing the report. Each note is assigned priority and placed for realization in iteration.

[1] `http://www.eclipse.org/legal/epl-v10.html`

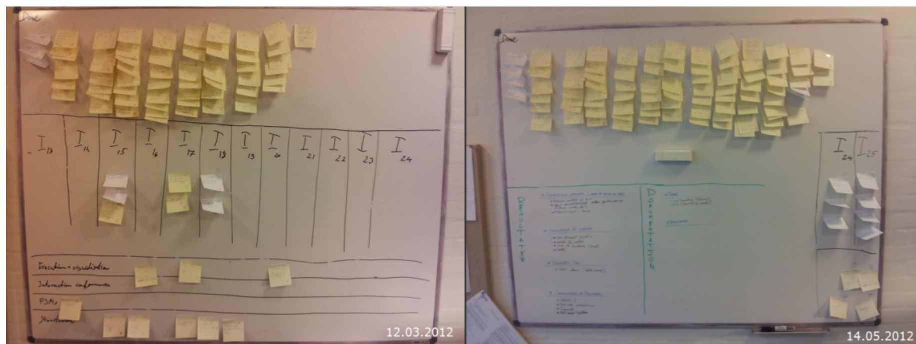

Figure 6.2: The white-board used to visualize iterations of the project at two different moments: in the middle of the project and close to the project finish. The scenarios (sticky notes) at the top of the white-board are placed in section *Done*. The iterations are presented in the table divided into columns *Ix*, where x is a number of iteration.

6.2 Functionalities

A purpose of the tool is to implement the approach described in chapter 5 and automatically find inconsistencies including the ones described in chapter 3 in the UML 2.2 models. In this section, functionalities of the tool are presented.

The overview on the use-case diagram of EsculapaUML tool is presented in fig. 6.3. Each of the use-cases visible in the diagram will be presented in this section.

6.2.1 Check scenario

The tool integrates with an existing UML graphical editor: Topcased. [FGC+06] Topcased enables user to see and design the UML models. It is possible to run the tool on the selected scenario from the editor from a pop-up menu in *Outline* view of the currently opened model (see fig. 6.4). After the simulation of the scenario is finished, the results (i.e. realization of the scenario) are presented in a sequence diagram in the editor without the necessity of reopening the model (see fig. 6.5).

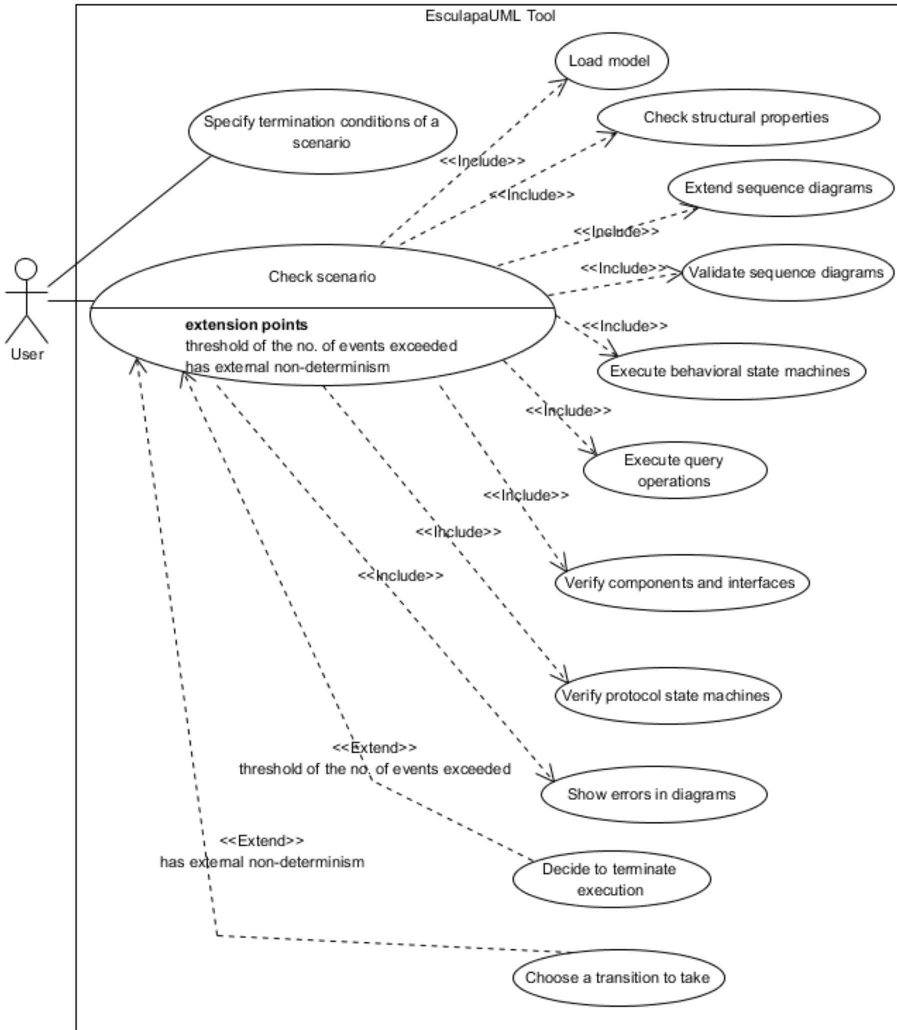

Figure 6.3: Use-case diagram with high-level use-cases implemented in EsculapaUML tool.

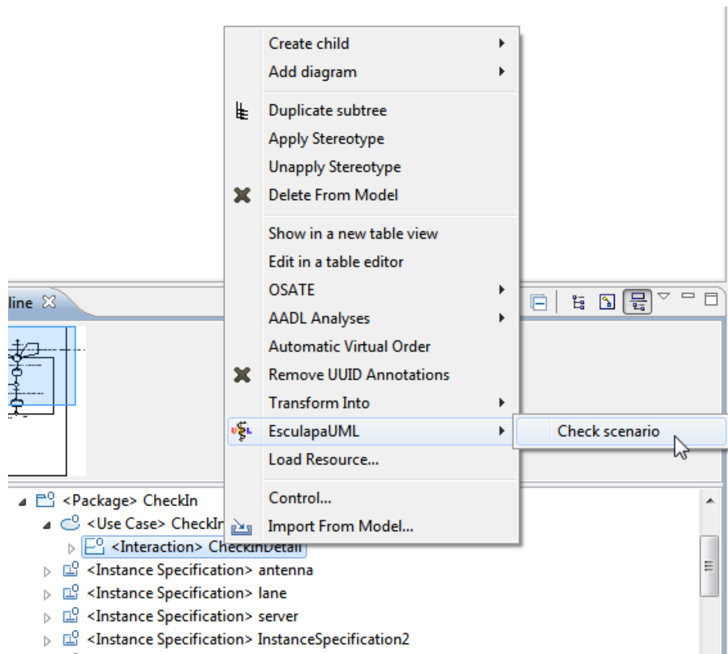

Figure 6.4: EsculapaUML pop-up menu in *Outline* view in Topcased.

6.2.2 Load model

The tool accepts UML models that are persisted in EMF model format (that is very similar to XML Metadata Interchange (XMI)). The editor loads a model and passes it to the tool.

6.2.3 Check structural properties

Before a scenario is checked during simulation and extended, the tool checks that the model structural properties of the elements used in the scenario are consistent. This includes checking e.g. for some of inconsistencies described in sections 3.1, 3.2, 3.3.

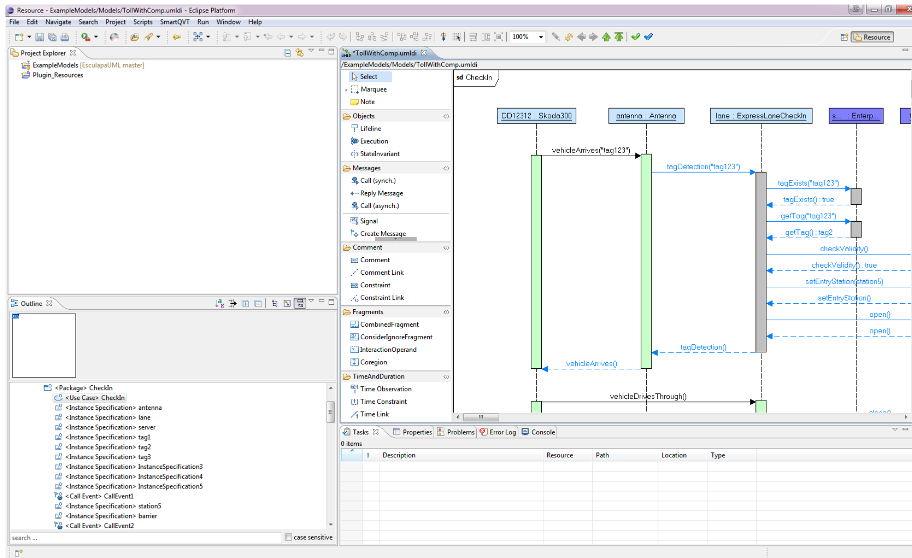

Figure 6.5: Topcased editor showing view on the sequence diagram just after the EsculapaUML extended the *Check-In with toll tag* successful scenario (see section 4.2).

6.2.4 Extend sequence diagrams

The tool is able to extend sequence diagrams representing the scenarios into the realizations of the scenarios (see section 5.6) by adding elements generated during BSMs execution to the original sequence diagram. This function is used mainly to check for existence of inconsistencies described in section 3.5.

In the current version of the tool, the supported elements of sequence diagrams that can be generated are: a lifeline, a synchronous message, an asynchronous message, a reply message, and the behavior execution specifications.

The elements that are not supported (they were outside the scope of the project) are: the interaction frames (conditions, assertions, etc.), a create message, and a delete message.

6.2.5 Validate sequence diagrams

Using the presented approach, the realization of the scenarios sequence diagrams (that include interactions between system's objects) can be validated by the tool against a behavior of a model. It is also possible to validate a partly extended sequence diagram that represents a scenario with only fragment of an interaction between system objects defined (with missing fragments). In this case, the missing fragments are completed by the extension mechanism.

6.2.6 Execute behavioral state machines

The extension and validation functionalities are based on the execution of BSMs (see section 5.7). The tool gives a possibility to an actor to execute arbitrary operation defined in a class in the UML model. The class must define its behavior in a BSM. The operation must have its trigger on a transition that can be fired at the moment of calling in order to execute.

The execution requires a system state to be defined in a model, i.e. the instance specifications of the classes and associations (links) must be defined in the model.

Instances can call each other during execution of BSMs by using effects on transition that are defined using the SAL (see section 5.8).

During the simulation, many run-time checks are in place, e.g. the one detecting inconsistency described in section 3.1.5.

The supported elements during BSMs execution are an initial pseudo-state, a state, a final state, an external transition, a composite state, a concurrent region, a guard in OCL, a choices pseudo-state, a junction pseudo-state.

Not supported elements (they were outside the scope of the project) are entry, exit, history, terminate pseudo-states, , internal transitions, state's entry, exit behaviors, and do-activities.

6.2.7 Execute query operations

The tool is able to execute query operations defined in a model (see section 5.7.6). The query operations are defined as OCL operation body expressions. A call to a query operation does not change a state of the system during execution.

6.2.8 Verify components and interfaces

The tool is able to check for all inconsistencies described in section 3.7. E.g., whatever components that have provided interfaces have indeed classes implementing these interfaces. Another check is for a conformance of a class that implements an interface with this interface.

6.2.9 Verify protocol state machines

The tool is capable of verifying the PSMs defined in context of interfaces (see section 5.9). The functionality detects inconsistencies described in section 3.6.

All UML elements except state invariants (see section 5.9.4) are supported in PSMs. The variables with `@pre` post-fix and `oclIsNew()` operator in post-conditions on the protocol transitions are not supported due to the limitations of the used OCL evaluator. The `result` variable in post-conditions is supported.

6.2.10 Show errors in diagrams

In case of any errors or warnings, their messages are presented to the user in Topcased editor and the errors indicators (small red icons in the diagrams) are shown on the parts of the model that are involved in these errors (see fig. 6.6). A user that clicks on an error message sees automatically the view on the erroneous model elements in the diagrams. A user that has the cursor hovering over an error indicator automatically sees the corresponding error messages.

6.2.11 Decide to terminate execution

There is a simple loop detection implemented based on the presented approach (see fig. 5.14). The simple loop detection will never detect complex loops (e.g. the ones that occur when two instances call each other). This is also known as the *halting problem*.

To avoid scenarios simulating forever, the tool has a build-in threshold of the number of executed events. If the threshold is exceeded it is likely that the scenario will never finish simulation and a user is notified. A user then can decide whatever to continue to simulate (and to adaptively increase the threshold) or to stop simulation.

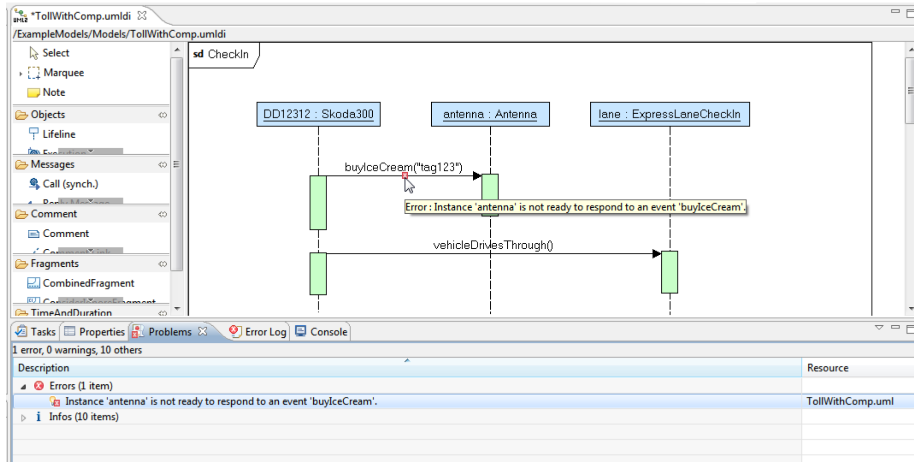

Figure 6.6: Topcased editor showing an error detected by EsculapaUML: the error message is presented in the *Problems* tab and also as an indicator in the diagram. The hint with the message is visible next to the cursor hovering over the error indicator.

6.2.12 Specify termination conditions of a scenario

The tool supports also other ways of terminating a scenario automatically:

- a user can add an annotation `max-global-events` to an interaction to limit the maximum number of events to execute in a scenario;
- a user can add an annotation `max-repetitive-subsequent-events` to an interaction to limit the maximum number of the same call events to execute subsequently in a scenario.

6.2.13 Choose a transition to take

A user can annotate specific BSM in a model as having external non-determinism[2] (see section 5.7.4). In this case, if there is more than one possible active transitions in an execution step (see fig. 6.7), the tool asks the user which transition to fire.

[2] A user adds an annotation with a key equal to `external-choice` and value `true`.

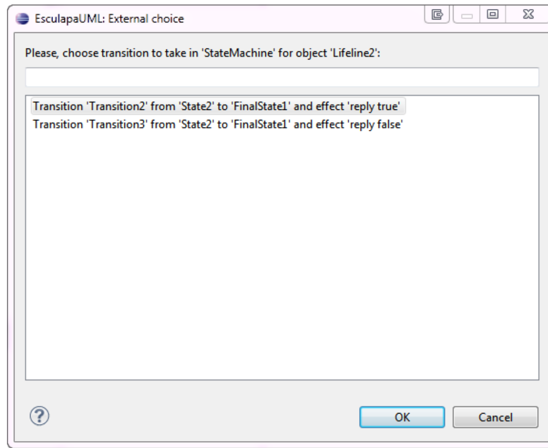

Figure 6.7: The window of the choice of transition to fire presented to a user.

6.3 Design

The tool consists of two separate Eclipse plug-ins (see fig. 6.8) that are described in this section.

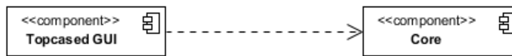

Figure 6.8: EsculapaUML plug-ins. The GUI plug-in depends on core plug-in. The core plug-in does not need the GUI plug-in to work.

6.3.1 Core plug-in

The core plug-in is invoked by a client (e.g. Topcased), accepts a loaded UML model and checks a scenario in the model by first statically checking the elements used in the scenario and then by the simulation of the scenario.

New elements of the scenario can be generated during the simulation (the extension of the sequence diagram described in section 5.6). The new generated elements are annotated in a model, i.e. the client can retrieve the changed model and see what elements were added during the simulation.

If any errors are detected during the simulation, they will be placed in the

diagnostics for the model[3]. The client can retrieve the diagnostics after the checking is finished.

The core plug-in is based on Eclipse Modeling Framework (EMF) [SBPM09] . EMF provides "basic framework for modeling" when other sub-projects "build on top of EMF core, providing such capabilities as model transformation, database integration, and graphical editor generation" [SBPM09][4]. One of the sub-projects that are used in the tool is UML2 project that "provides an EMF-based implementation of the UML 2.x metamodel" [SBPM09]. Other sub-project used is the OCL project that "defines APIs for parsing and evaluating OCL constraints and queries on Ecore or UML models"[5].

EMF was chosen as a framework because it implements the UML 2.2 metamodel and has decent libraries including the one to evaluate OCL queries on the UML models.

6.3.1.1 Packages

The core plug-in is divided into packages of which the most important are presented in fig. 6.9.

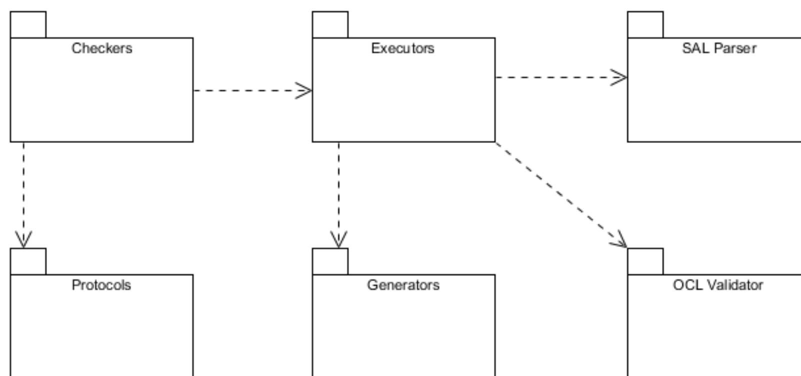

Figure 6.9: The most important packages in the core plug-in of EsculapaUML.

Checkers are responsible for checking the static structure of a model and, if

[3]The diagnostics are used in the EMF framework to capture the error information.

[4]The EMF framework known quotation is: "To model or to program, this is not the question" [SBPM09]. This sentence characterizes the essence of EMF that aims for modeling and programing to be considered the same thing using model-to-code generation facilities.

[5]`http://www.eclipse.org/modeling/mdt/?project=ocl`

there are no errors detected, initialize the simulation of a scenario. They are also responsible for errors management.

Protocols verify the PSMs during execution, they are initialized by *Checkers*.

Executors are responsible for the simulation of a scenario, including the execution of BSMs.

Generators are used by *Executors* to generate new interaction elements if necessary.

SAL Parser is used by *Executors* to parse SAL expressions.

OCL Validator is used by *Executors* to parse and evaluate OCL expressions.

6.3.1.2 Classes

The core plug-in has a complex structure and therefore in this section we will abstract out from the details and look only at the most important classes (see fig. 6.10) that help us understand how the design is realized.

The observer pattern in the core plug-in helps to decouple the different parts of the consistency checking algorithm. *Scenario Executor* observes the execution of the *Behavior Executors* (representing BSMs) via *Execution Coordinator* and reacts on call, reply and control flow events. *Scenario Executor* verifies and extends a sequence diagram representing a scenario.

Protocol Verifier instances also observe the execution and verify the PSMs. They react on call events (used to calculate pre-conditions) and reply events (used to calculate post-conditions).

Execution terminator also observes the execution and is able to cancel it whenever the conditions are violated (see sections 6.2.11 and 6.2.12).

Scheduler is used by *Behavioral Executor* to schedule and dispatch call events in a system.

6.3.2 Topcased GUI plug-in

The tool includes plug-in for the graphical user interface of Topcased 5.1.0 editor. The decision to use Topcased was made because the editor is implemented based on the Eclipse platform and the EMF framework. Topcased includes the UML

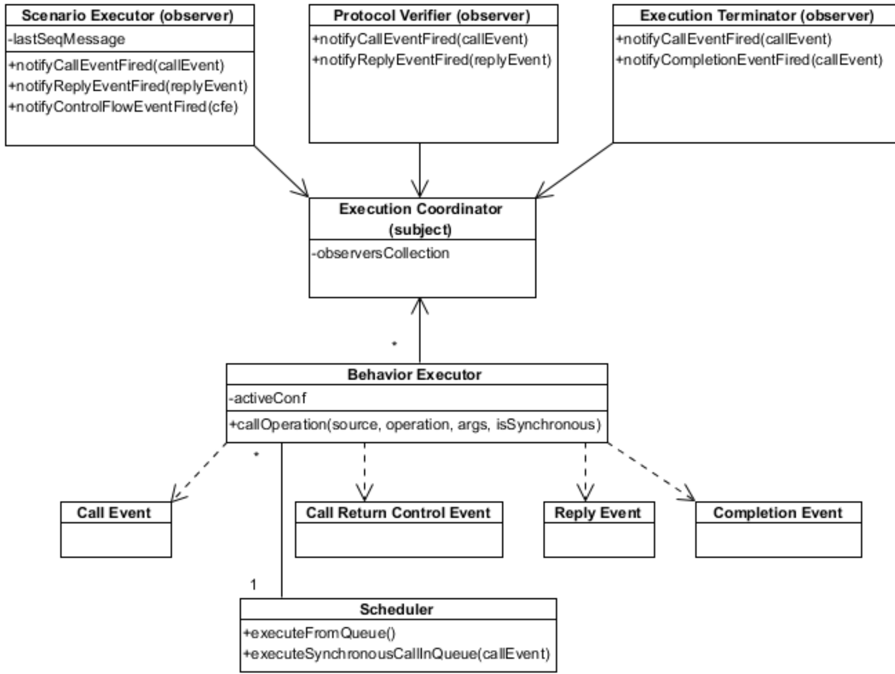

Figure 6.10: Some of the classes in the core plug-in that are important to understand the design.

2.2 editors (incl. sequence, state machines, and class and component diagrams) with the advanced view on the UML model: *Outline* view. Outline view enables modifying any possible property of a model's element.

Unfortunately, Topcased does not support the protocol state machines diagrams so the core plug-in can optionally accept the behavioral state machines (with the effects specified as post-conditions) as the protocol state machines.

The Topcased GUI plug-in adds necessary pop-up menu for EsculapaUML in Topcased. When a scenario is selected, first it fixes the model of the scenario (see bug description in section 6.4.3), then it initializes the core plug-in and passes the scenario and the currently opened model to the core plug-in.

After the checking of the scenario finishes, the GUI plug-in examines the changed model and finds the elements that were generated by the core plug-in in order to create a graphical representation for them in the sequence diagram and to create a layout for the new extended sequence diagram.

Next, the GUI plug-in retrieves the diagnostics for the model and checks if there are any errors detected. If there are some errors, the errors indicators are created in the diagrams and *Problems* view.

6.4 Implementation notes

6.4.1 EMF OCL bug 286931

The evaluation of query operations does not work in EMF OCL for UML. To avoid the submitted bug[6] the hack was developed in the core plug-in that changes the OCL environment before the simulation of a scenario starts so that the OCL evaluator can evaluate query operations correctly later. The hack development was critical for guards on transitions in BSMs to evaluate correctly.

6.4.2 EMF OCL bug with interfaces

The OCL evaluator has a problem when navigating "through" an interface. If one class accesses another class via an interface, the OCL evaluator will not correctly resolve the object's attributes and methods. E.g. evaluating the `self.barrier.isOpened()` guard on the transition, where `isOpened()` is the query operation and the `barrier` is instance specification of type `Barrier` connected through the link to `self`. The end of the link has type of the interface (where the same operation `isOpened()` is also declared). The guard will fail evaluating. This issue was reported on the OCL group[7] but the resolution of the issue didn't come on time to be included in this project. For the time being, the walk-around is to use `oclAsType`, e.g. `self.barrier.oclAsType(Barrier).isOpened()`. I used this type of guards in the model in appendix B to be able to execute the toll system with components.

6.4.3 Topcased bug #4014

During the tool development an important bug in Topcased was discovered and submitted to Topcased's bug-tracker[8]. The bug affects the consistency between what is presented in a sequence diagram graphically and what is stored

[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=286931
[7] http://www.eclipse.org/forums/index.php/t/328769/
[8] http://gforge.enseeiht.fr/tracker/?func=detail&atid=109&aid=4014&group_id=52

in a model with respect to the order of messages on lifelines. The description of the bug is: "Topcased wrongly saves and updates list 'coveredBy' for the Lifeline elements. The elements should be ordered according to their vertical position on Lifeline. Currently, when one moves a message in a way that order is changed, Topcased not always updates this order correctly. Also when one moves a message a little (so that the order is not changed), Topcased sometimes changes the order unexpectedly."

Because the resolution of the bug was critical for EsculapaUML (if a user wants to design the input sequence diagrams in the editor) and the Topcased team did not solve the bug, the hack was developed and implemented in GUI plug-in. The hack compares a model and a graphical representation of a sequence diagram and fixes the model if necessary so that the model with the correct order of messages will be always provided for the core plug-in.

6.5 Testing

During the tool development approx. 110 JUnit tests and the same number of test UML models were created that helped a lot during validation of the implementation. Some of the tests use the advanced UML models' diff provided by EMF Compare framework[9]. The framework gives the possibility to compare actual and expected results of the scenario extension.

[9] `http://www.eclipse.org/emf/compare/`

Chapter 7

Conclusions and Future Work

This chapter starts with the conclusions about the project and continues with a possible future work. At the end, the proposal of an evaluation is presented.

7.1 Conclusions

The aim of the project was to check the consistency between sequence diagrams and (behavioral and protocol) state machines diagrams in the UML models by simulating the use-case scenarios and extending them into realizations of the scenarios. During the project, the theoretical approach (including the consistency checking algorithm) was developed and then, based on that, the tool that realizes the approach was implemented. The project achieved its aim.

None of the approaches to consistency checking presented in section 1.3 generates realizations of scenarios from the scenarios. The approach in this thesis fulfills this research gap by giving a possibility to extend sequence diagrams representing scenarios into realization of the scenarios during the simulation.

The project also included the creation of the Simple Action Language (SAL) used for defining effects on transitions in UML behavioral state machines. The SAL enables designers to design UML state machines suitable for the execution and the consistency checking in a simple manner.

The approach uses the UML models directly during the simulation. The direct

use of UML models simplifies integration with editors and the tool development but it also reveals the high complexity of the UML meta-model.

During the project, we became aware of the limitations of the UML behavioral state machines' execution semantics that cannot deal with few types of the synchronous scenarios in the object-oriented systems without deadlocking. We see possible solutions for these limitations in section 5.7.8.

The UML semantics had to be defined precisely for some of the elements that were crucial for our consistency checking approach. For example, the FIFO dequeuing method was chosen for the event queue during the behavioral state machines execution. It also had to be chosen what happens when a post-condition failed in protocol state machines.

The implemented tool is able to detect the semantical inconsistencies between sequence and state machines diagrams during the design of the UML models. The tool also points-out a source of problems to a designer. The tool can simulate more complex systems, including the case-study of the toll system. The tool can extend sequence diagrams representing scenarios to realizations of the scenarios. The tool is integrated with UML editor (Topcased) and gives the possibility to the designer to use it continuously during model design process. Detected consistency errors are presented in UML diagrams in the editor in a concise way.

The approach is aware that the execution of a state machine might not terminate. For these scenarios that would never terminate the tool offers simple loop detection. The tool implements additional ways of detecting long-running scenarios based on events counter in which a user makes a decision to terminate the simulation process prematurely.

The overall summary is that, with the presented approach, it is possible to automatically detect the inconsistencies between sequence and state machines diagrams during design. The implemented tool is recommended for designers; it fulfills its requirements for showing the realizations of the use-case scenarios and it can be valuable to the designer to help him understand the possible problems in a model. Using the tool will eventually lead to the improvement of consistency of models.

7.2 Future work

The scope of the project was limited to the selected elements in the UML sequence and state machines diagrams. The approach can be further extended to support more UML elements, i.e. interactions frames, create and delete messages in the sequence diagrams.

The execution of behavioral state machines can support more constructs like an entry, an exit, a history and terminate pseudo-states, an internal transition, a state's entry and exit behaviors, do-activities.

As it was shown in section 5.9.4, the verification of protocol state machines can be extended to include the state invariants verification.

The SAL language can be extended by adding creator and destructor of instance specifications.

By further extending the OCL evaluator, we can also include checking of expressions with variables that refer to the pre-state in post-conditions (the variables with post-fix `@pre` and `oclIsNew` operator).

More diagrams types could be also added to the consistency checking algorithm, e.g. activity diagrams, use-case diagrams and communication diagrams. Supporting specific UML profiles will is also a possible extension.

Finally, the project could be extended by adding more functionality, e.g. a support of a step-by-step simulation could be added to the tool. It would be possible to construct a code generation facility that generates a source code in a chosen programming language from the simulated model.

7.3 Evaluation

To prove a hypothesis that the tool is helpful for students, we should conduct an empirical research with many students. Unfortunately, during this project we had no possibility to do it. The tool was not mature enough to be used by students at the time when the course *System Integration* was held in the summer semester of 2012 at DTU.

The possible experiment would be to construct a complex model (e.g. the complete toll system from fig. 4.2) with five hidden inconsistencies and to divide

the students that will check the model individually into two groups: one group will be given the tool to check the consistency and the other will check the consistency manually. If the students using the tool will all find and fix the inconsistencies significantly faster than the students checking manually then we have a good indication that the tool is helpful in finding the inconsistencies faster. If however, some students checking manually will fail finding some of the inconsistencies after a reasonable amount of time and all students using the tool will find all of the inconsistencies then we have a proof that the tool is helpful in finding the inconsistencies in complex models. The experiment should be conducted on the representative number of students to be valid.

From my personal experiences and looking back at my work logs for the case study model presented in appendix B designed before the tool was created and for a model of the same case study designed with the tool in the end of this project, I discovered that the tool shorten the time needed to create the consistent model by approx. 50%. That is, however, very weak evidence because I could also become a more experienced modeler.

Appendix A

Toll System without components

In this appendix, the model of the toll system that is able to realize the check-in and the check-out is described. This is a simplified version of the toll system that does not include and use the components, the interfaces and the protocol state machines.

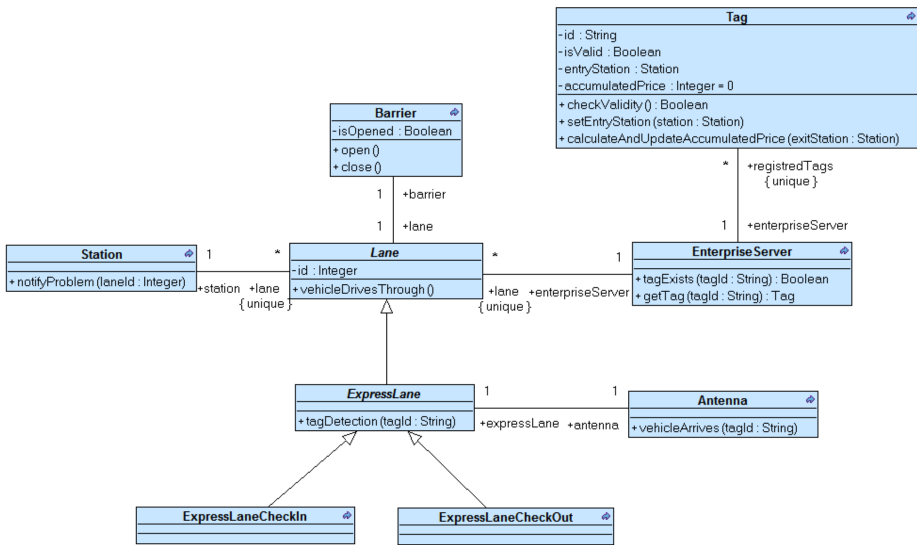

Figure A.1: Toll system's class diagram. Toll system is composed of *Stations* that have many *Lanes*. In our case, we provide lanes kinds that can be used for express check-in and check-out. Each *Lane* has a connection to *EnterpriseServer*. *EnterpriseServer* is responsible for keeping information about registered *Tags* in the system.

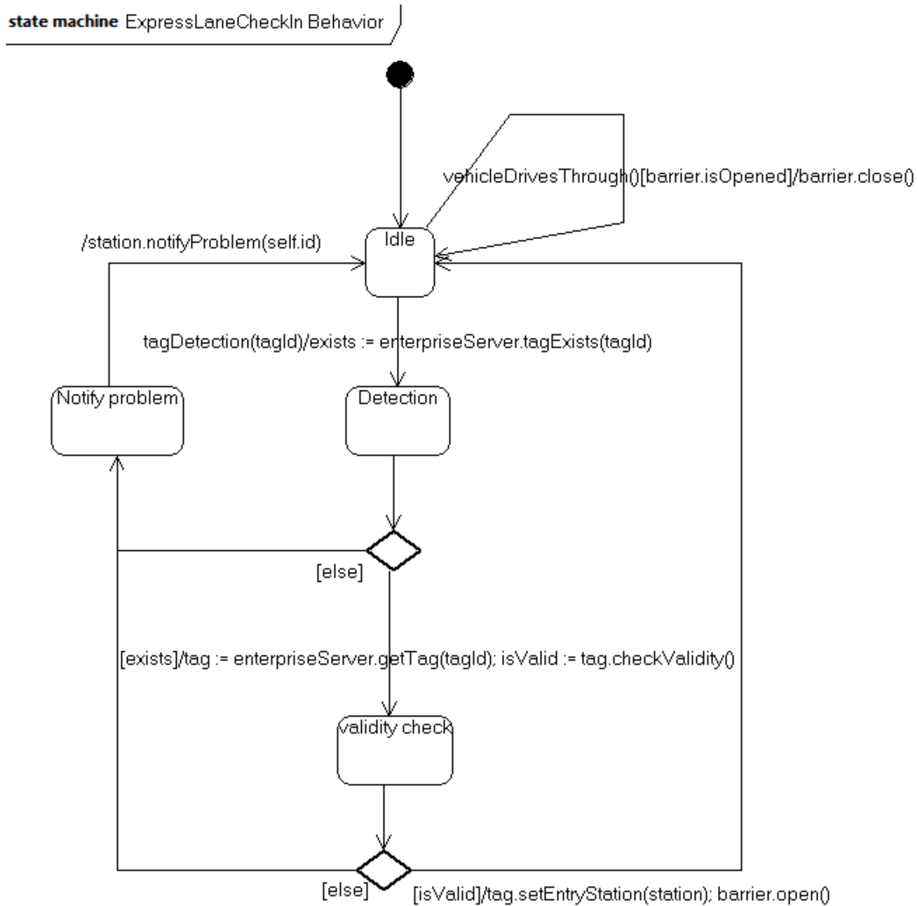

Figure A.2: Behavior of ExpressLaneCheckIn class. After successful validation of the tags, barrier is opened. Note, in case of problems with the tags, the station is notified.

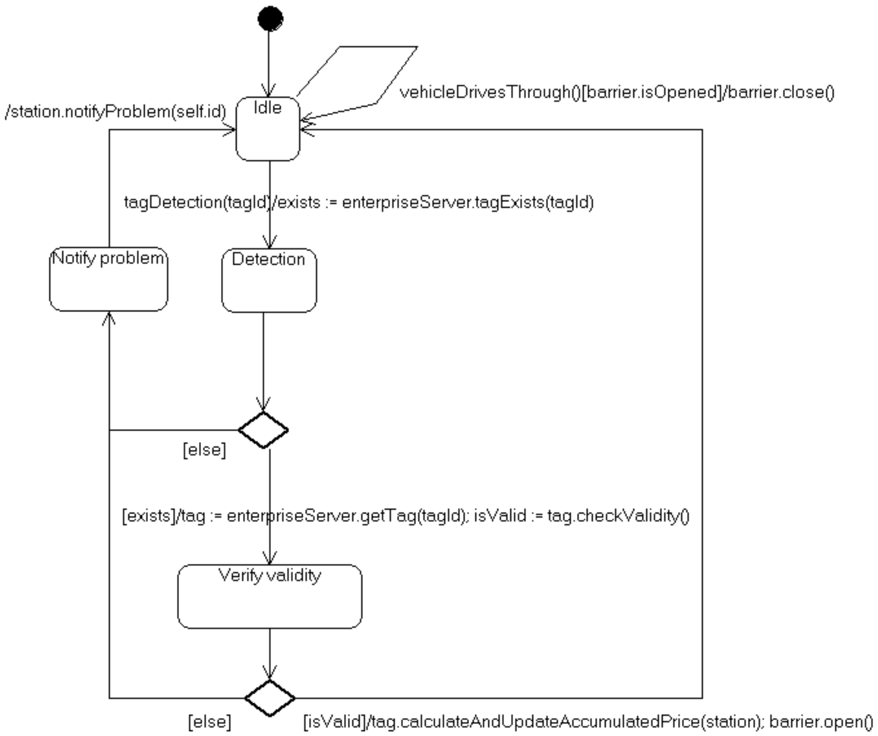

Figure A.3: Behavior of ExpressLaneCheckOut class. After successful validation of the tag, the tag is updated with the price of the trip and barrier is opened. Note, in case of problems with the tags, the station is notified.

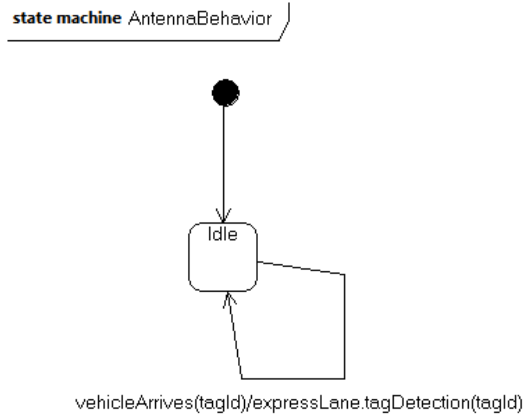

Figure A.4: Behavior of Antenna class. After *vehicleArrives* operation is called, express lane is notified about new detection of tag.

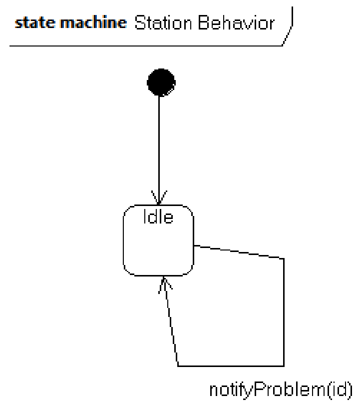

Figure A.5: Behavior of Station class. *Station* can handle problems with the tags.

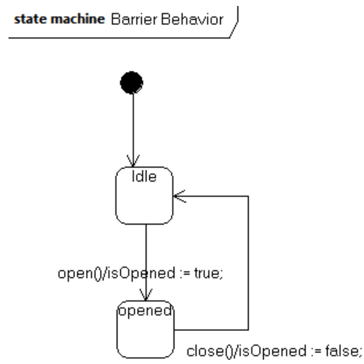

Figure A.6: Behavior of Barrier class. *Barrier* can be opened and then closed.

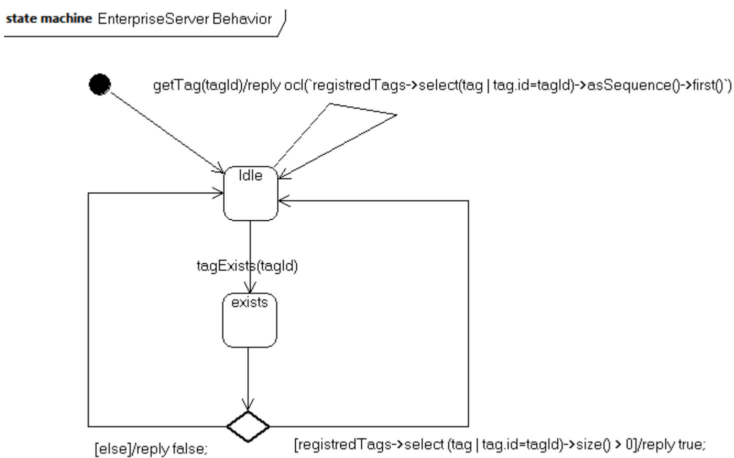

Figure A.7: Behavior of EnterpriseServer class. *EnterpriseServer* checks for existence of the tag and returns the tag from existing tags collection.

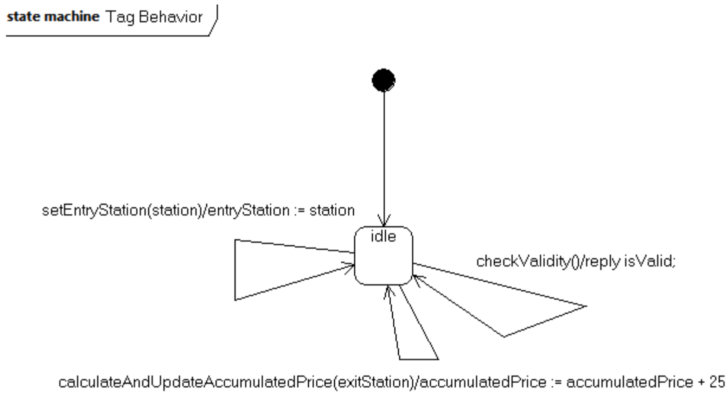

Figure A.8: Behavior of Tag class. *Tag* checks for self-validity and also knows how to calculate and update its price based on entry and exit stations. The algorithm here is a simplification that always adds 25 to the accumulated price.

Appendix B

Toll System with components

In this appendix, the model of the toll system that is able to realize the check-in and the check-out is described. This is a full version of the toll system that includes and uses the components, the interfaces and the protocol state machines. The behaviors of most of the classes are the same as in appendix A. Here, only the behaviors of the classes that differ are shown.

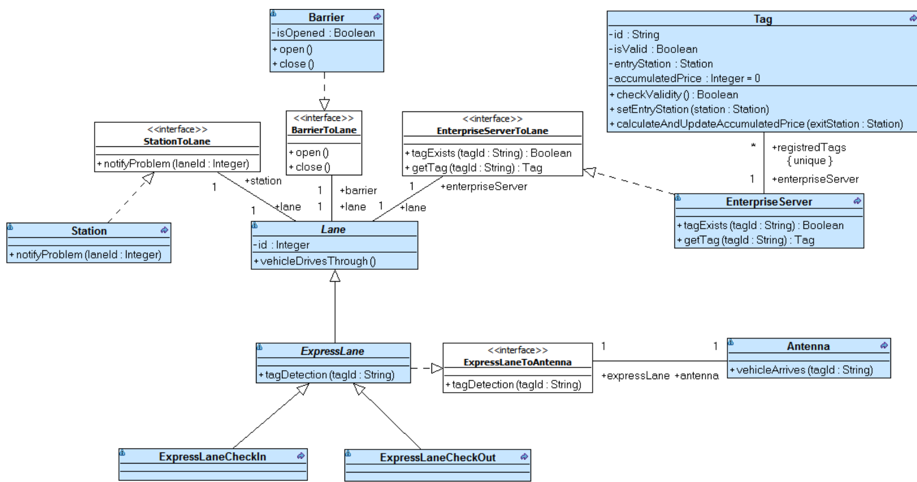

Figure B.1: Toll system's class diagram. Toll system is composed of *Stations* that have many *Lanes*. In our case, we provide lanes kinds that can be used for express check-in and check-out. Each *Lane* has a connection to *EnterpriseServer*. *EnterpriseServer* is responsible for keeping information about registered *Tags* in the system. The interfaces visible here are defining borders of components presented in fig. B.2.

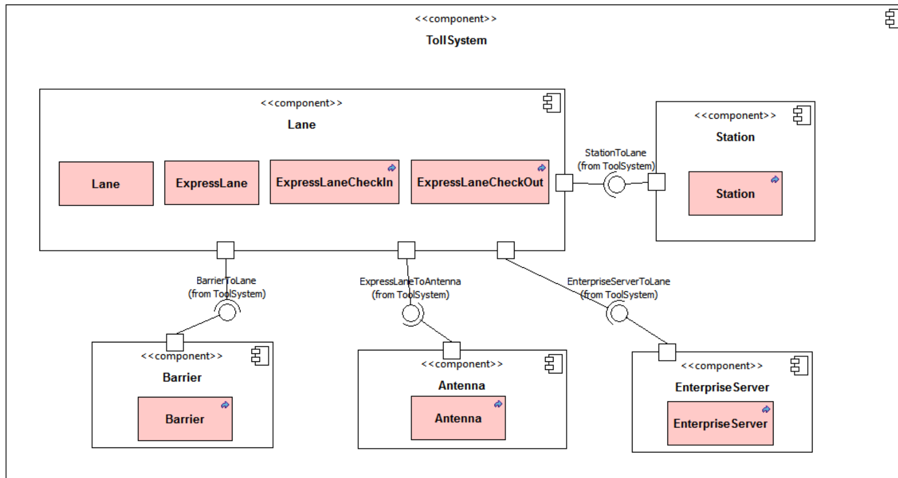

Figure B.2: Toll system's component diagram. Components are shown with classes inside them. Toll system is composed of *Lane*, *Station*, *Barrier*, *Antenna* and *EnterpriseServer* components. Provided and required interfaces are visible in lollipop notation. Note, *Tag* class is not visible in any component, reason is, it is outside components.

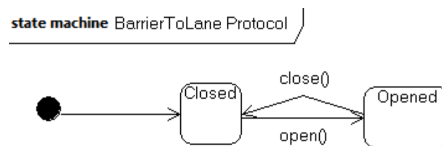

Figure B.3: BarrierToLane protocol state machine. Barrier must be opened before it can be closed again.

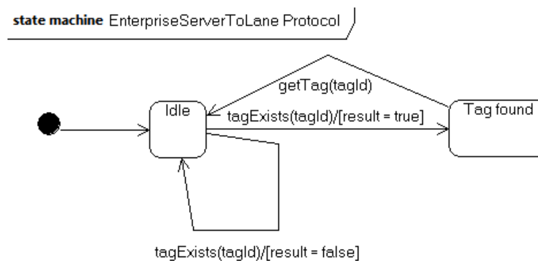

Figure B.4: EnterpriseServerToLane protocol state machine. Protocol forces to check for existence of a tag with `true` result (checked in post-condition) before *getTag()* can be called.

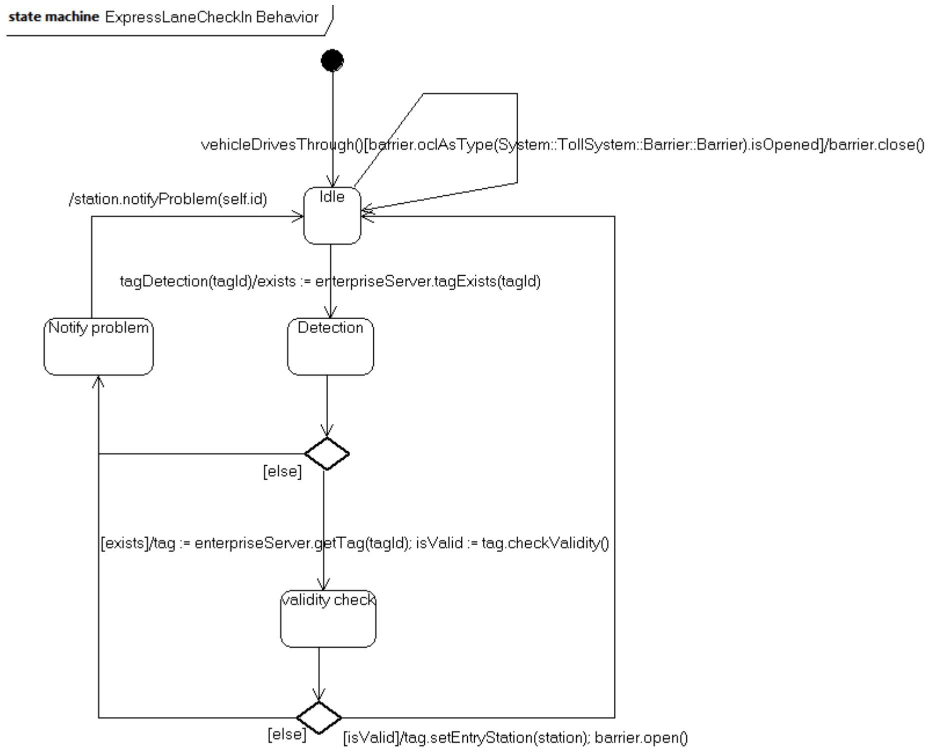

Figure B.5: Behavior of ExpressLaneCheckIn class. After successful validation of the tags, barrier is opened. Note, in case of problems with the tags, the station is notified. In comparison to fig. A.2, one guard uses `oclAsType` to check for value in *Barrier* (it is equivalent to query operation behavior).

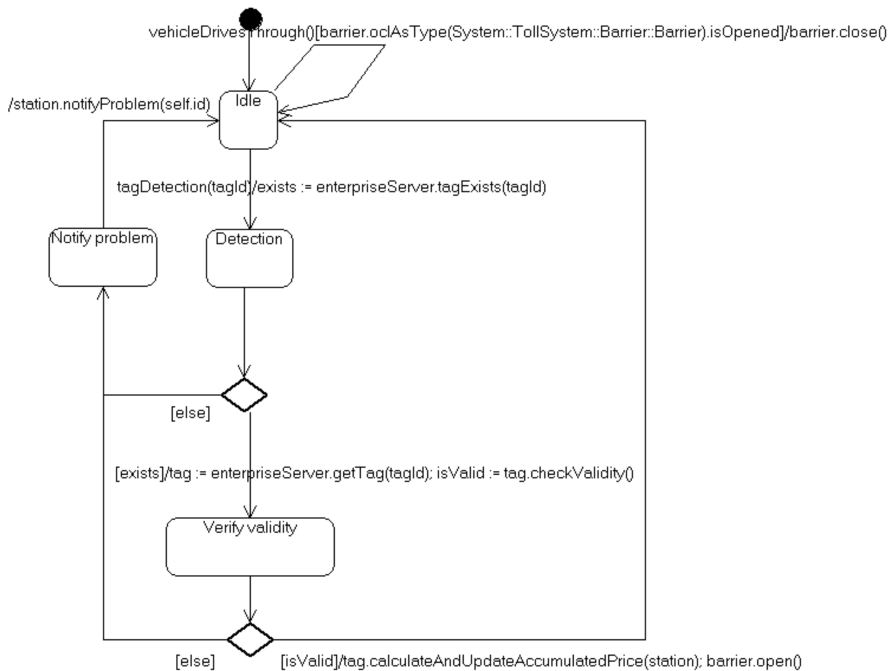

Figure B.6: Behavior of ExpressLaneCheckOut class. After successful validation of the tag, the tag is updated with the price of the trip and barrier is opened. Note, in case of problems with the tags, the station is notified. In comparison to fig. A.3, one guard uses `oclAsType` to check for value in *Barrier* (it is equivalent to query operation behavior).

Appendix C

Scenarios and realizations of scenarios in Toll System

In this appendix, the scenarios and the realizations of the scenarios with view on the object diagrams of the toll system are shown. The scenarios and the realizations are the same for both versions of the toll system presented in appendices A and B.

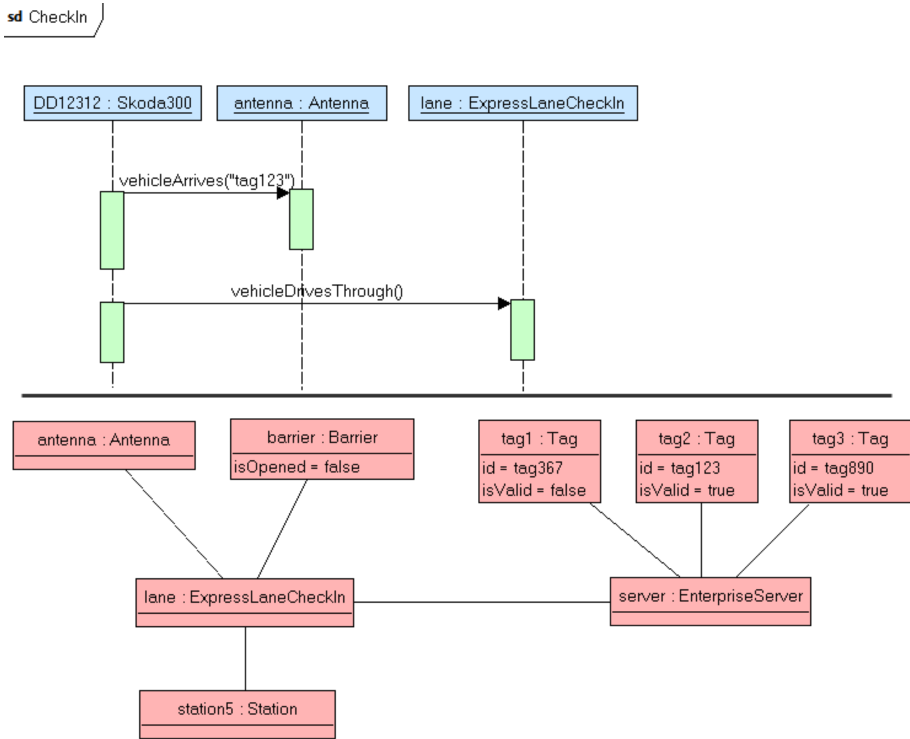

Figure C.1: *Check-in with toll tag* scenario and the corresponding object diagram **before** the realization.

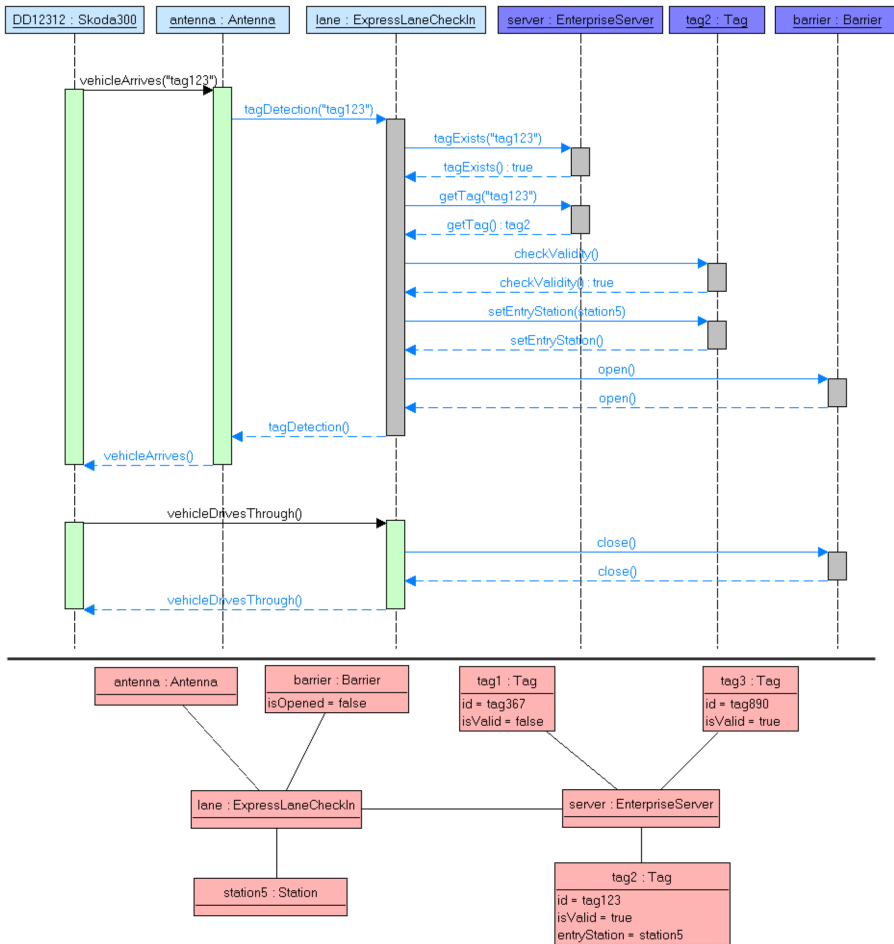

Figure C.2: *Check-in with toll tag* realization of scenario and the corresponding object diagram **after** the realization.

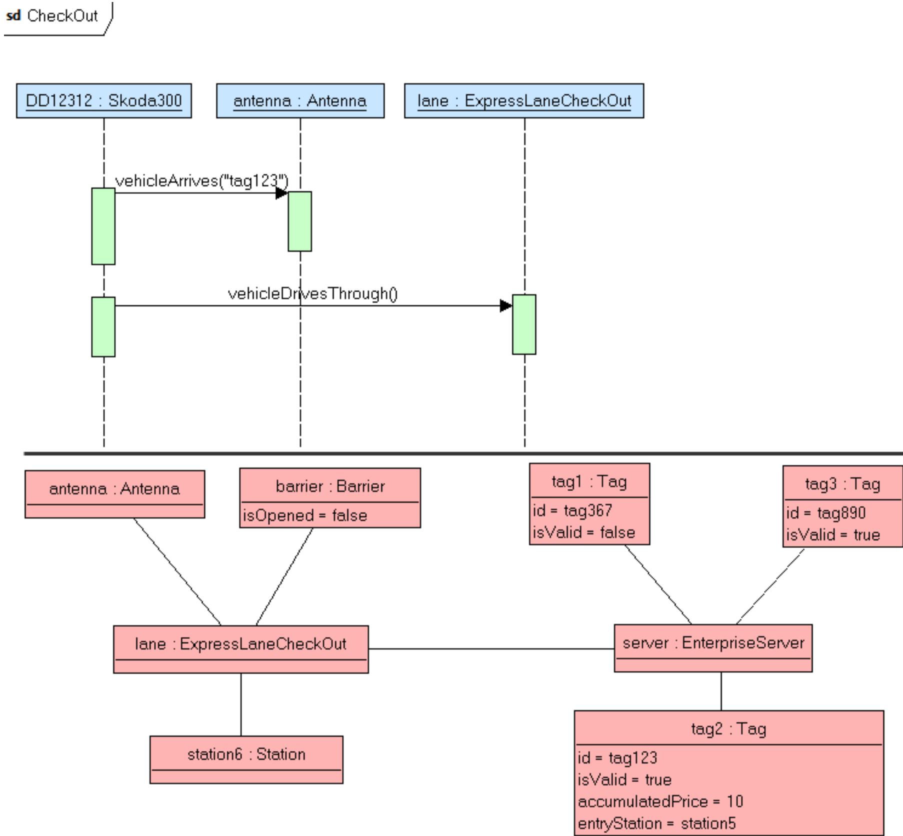

Figure C.3: *Check-out with toll tag* scenario and the corresponding object diagram **before** the realization.

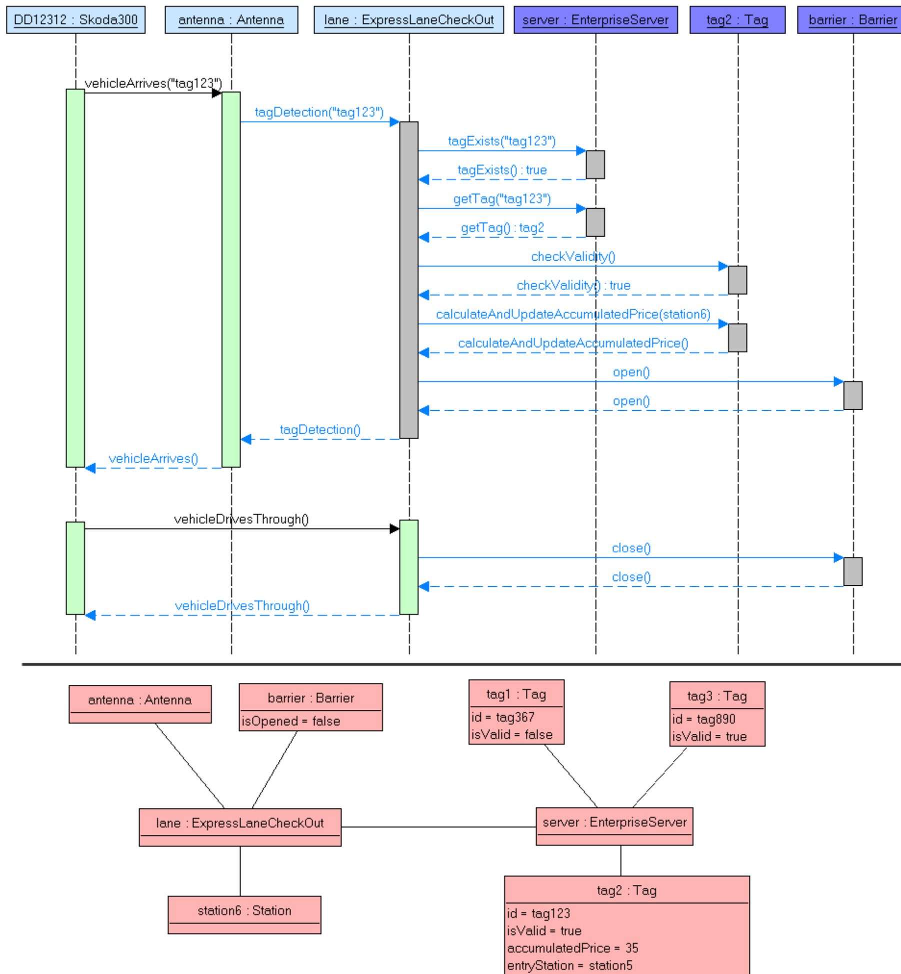

Figure C.4: *Check-out with toll tag* realization of scenario and the corresponding object diagram **after** the realization.

Bibliography

[BK12] Hubert Baumeister and Patrick Könemann. Project description: Toll system, version 1: March 27. 2012.

[DDd03] Alexandre David, Johann Deneux, and Julien d'Orso. A formal semantics for UML statecharts. Technical Report 2003-010, Uppsala University, 2003.

[DH04] Karsten Diethers and Michaela Huhn. Vooduu: Verification of object-oriented designs using uppaal. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 139–143. Springer Berlin / Heidelberg, 2004.

[Dub06] Jori Dubrovin. Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, March 2006.

[DV10] Ruta Dubauskaite and Olegas Vasilecas. The approach of ensuring consistency of uml model based on rules. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, CompSysTech '10, pages 71–76, New York, NY, USA, 2010. ACM.

[Egy00] Alexander Franz Egyed. Heterogeneous view integration and its automation. Technical report, PhD thesis, USC, 2000.

[Egy01] Alexander Egyed. Scalable consistency checking between diagrams-the viewintegra approach. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 387–, Washington, DC, USA, 2001. IEEE Computer Society.

[EKHG01] Gregor Engels, Jochem M. Küster, Reiko Heckel, and Luuk Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. *SIGSOFT Softw. Eng. Notes*, 26:186–195, September 2001.

[Eva03] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FGC+06] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The topcased project: a toolkit in open source for critical aeronautic systems design. In *Embedded Real Time Software (ERTS)*, Toulouse, February 2006.

[Ger05] Eran Gery. Rhapsody: A pragmatic a roach to model-driven development, 2005.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, 30:31–42, 1997.

[HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[Hol04] J. Holt. *UML for systems engineering: watching the wheels*. IEE professional applications of computing series. Institution of Electrical Engineers, 2004.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, June 1992.

[Kna04] Alexander Knapp. Semantics of uml state machines. Technical report, Institut für Informatik, Ludwig-Maximilians-Universität München, 2004.

[KTM08] Fabrice Kordon and Yann Thierry-Mieg. Experiences in model driven verification of behavior with uml. In *Monterey Workshop*, pages 181–200, 2008.

[LMT09] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of uml model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009. Quality of UML Models.

[LTY03] B. Litvak, S. Tyszberowicz, and A. Yehudai. Behavioral consistency validation of uml diagrams. In *Software Engineering and Formal Methods, 2003.Proceedings. First International Conference on*, pages 118 – 125, sept. 2003.

[Obj06] Object Management Group. Object Constraint Language, OMG Available Specification, Version 2.0. Technical report, May 2006.

[Obj09] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, V2.2. Technical report, February 2009.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[Shi06] Yoshiyuki Shinkawa. Inter-model consistency in uml based on cpn formalism. In *Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 411–418, Washington, DC, USA, 2006. IEEE Computer Society.

[SKB05] S. Sengupta, A. Kanjilal, and S. Bhattacharya. Automated translation of behavioral models using ocl and xml. In *TENCON 2005 2005 IEEE Region 10*, pages 1 –6, nov. 2005.

[TMH08] Yann Thierry-Mieg and Lom-Messan Hillah. Uml behavioral consistency checking using instantiable petri nets. *ISSE*, 4(3):293–300, 2008.

[TS03] Jennifer Tenzer and Perdita Stevens. Modelling recursive calls with uml state diagrams. In *PROC. 6 TH INT. CONF. FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING (FASE 03). VOLUME 2621 OF LECT. NOTES COMP. SCI*, pages 135–149. Springer, 2003.

[Tsi00] Aliki Tsiolakis. Consistency analysis of uml class and sequence diagrams based on attributed typed graphs and their transformation. In *ETAPS 2000 workshop on graph transformation systems*, pages 77–86, 2000.

[Tsi01] Aliki Tsiolakis. Semantic analysis and consistency checking of uml sequence diagrams. Technical Report 2001-06, Technische Universität Berlin, Department of Computer Science, April 2001. Diplomarbeit.

[UNKC08] M. Usman, A. Nadeem, Tai-Hoon Kim, and Eun-Suk Cho. A survey of consistency checking techniques for uml models. In *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, pages 57–62, December 2008.

[VdH94] W. Van der Hoek. Unravelling nondeterminism: on having the ability to choose. In *Proceedings of the sixth international conference on Artificial intelligence : methodology, systems, applications: methodology, systems, applications*, AIMSA '94, pages 163–172, River Edge, NJ, USA, 1994. World Scientific Publishing Co., Inc.

[VVP00] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. In Hartmut Ehrig and Gabriele Taentzer, editors, *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, page 14–21, Berlin, Germany, March 25–27 2000.

[WFZZ05] Hongyuan Wang, Tie Feng, Jiachen Zhang, and Ke Zhang. Consistency check between behaviour models. In *Communications and Information Technology, 2005. ISCIT 2005. IEEE International Symposium on*, volume 1, pages 486 – 489, oct. 2005.