Fitting Standard Software to Customer Needs

Rajesh Veluswamy

LYNGBY 2003
M.Sc. Thesis Project
NR.00/99

IMM

Printed by IMM, DTU

Abstract

There is a given:

- an already existing software $\mathcal{A}$, that does not support some of customer requirements $\mathcal{R}$. Typically, $\mathcal{A}$ is extended with a sizeable software component $\mathcal{B}$, to meet this class of customer requirements $\mathcal{R}$. The combined system $\mathcal{A} \oplus \mathcal{B}$, is now capable of addressing the class of customer needs. The Main Development Centre (MDC) designs and develops $\mathcal{A} \oplus \mathcal{B}$.
- Also, each customer currently using $\mathcal{A}$, has his own specific requirements $\mathcal{R}'$, which are of type $\mathcal{R}$.

For $\mathcal{A} \oplus \mathcal{B}$ to be useful to the customer, the Requirements Engineer at the Customer Solution Centre (CSC), has to fit $\mathcal{A} \oplus \mathcal{B}$ with $\mathcal{R}'$ to generate a customer-specific $\mathcal{A}' \oplus \mathcal{B}'_{\mathcal{R}}$.
A CSC Requirements Engineer has a challenge to be able to elicit and specify right customer requirements $\mathcal{R}'$. The question is:

> How can the MDC assist a CSC Requirements Engineer to elicit and specify $\mathcal{R}'$ that would fit $\mathcal{A} \oplus \mathcal{B}$ to the specific needs of a customer?

This thesis addresses this question. It does so by proposing a *Solution Concept* that addresses this problem. The thesis carries out a case-study, with the aim of applying the Solution Concept to an actual industrial project (*Event Management*) within Microsoft Business Solutions (MBS).

KEYWORDS: Domain Analysis, Requirements Analysis, Formal Specification, Reuse, Domain-specific Language.

Preface

This document has been produced as part of a Masters Thesis carried out at Computer Science and Engineering (CSE) section of the Institute of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU). Work on the thesis has been supervised by Prof. Dines Bjørner.

I would like to thank Professor Dines Bjørner for his valuable inputs and guidance. I also would like to thank my colleagues at Microsoft Business Solutions, who have been extremely helpful in giving me valuable feedback.

Rajesh Veluswamy
Student No. : s010841
DTU Lyngby, 7th April 2003

Typographical Conventions

This section gives an overview of the typographical conventions used in this document.

Normal Text

Normal texts in this document are presented in the font , as used in this sentence.

Emphasized Text

Text which has been emphasised by the author is presented in the *font used here.*

Quoted Material

Quoted material is presented in one of two forms: Shorter quotations whose contents are directly integrated in the running text, are presented *"inline,such as this*".

> *Longer quotations whose contents are not necessarily integrated in the running text are formatted like this paragraph.*

Cross-references

As customary, cross-references are usually integrated in the running text, such as: See section 1.1 on page 25.

Terms and Definitions

Terms whose importance is deemed to be significant are *emphasised.*

Acronyms

The acronyms which are used in this document have been listed in Appendix A.1.

Special Notation

> `Special entities which are found throughout the document, like Formal specification, narratives, domain-specific language are presented in typewriter style as in this sentence.`

Document Structure

This document is divided into five parts. The content of each document parts is briefly described below.

Part I: Problem Analysis

This part presents the general problem area of Requirements Engineering and the major research challenges in this field. It highlights the main challenge addressed by this thesis, that of fitting standard software to specific needs of a customer. This is followed by the problem definitions formulation in Section 1.3 on page 21.

Part II: Solution Concept

This part presents the *Solution Concept* for the thesis problem statement. Further an approach, to systematically create a domain-specific language (DSL) is described. This part also, provides the theoretical background information about Domain-specific languages (DSL) and Domain and Requirements Analysis, that serve as input for design of DSL.

Part III: Case Study

This part presents the Case-study which was carried out with the thesis with the aim of applying the Solution Concept to an actual industrial project (*Event Management*) within Microsoft Business Solutions (MBS). As part of case-study, Domain analysis, Requirements analysis and Step-wise development of different solution approaches is done The result of analysis served as input to design an Event Management Language.

Part IV: Conclusion

This part provides a discussion of the work done for the thesis, identifies future work, and concludes by providing a brief summary of the results of the thesis.

Part V: Appendices

This part contains all the listings relevant to the document.

Contents

List of Tables

List of Figures

Part I

Problem Analysis

Chapter 1

Introduction

This chapter introduces Requirements Engineering in Section 1.1, as seen from the software engineering perspective. This is followed by a focus on the challenge of fitting a standard software to customer needs in Section 1.2 and a clear problem statement formulation in Section 1.3.

1.1 Requirements Engineering(RE)

1.1.1 Motivation

The last 25 years of research in the area of software engineering, has given intense focus towards RE. This is primarily been motivated by the observation of Bell and Thayer, that inadequate, inconsistent, incomplete, or ambiguous requirements are numerous and have a critical impact on the quality of the software [1]. They further concluded that:

> "The requirements for a system do not arise naturally; instead, they need to be engineered and have continuing review and revision."

A survey over 8000 projects undertaken by 350 US companies revealed that one third of the projects never completed and one half succeeded partially, with major cost overruns and delays [3]. When asked about the causes of such failure executive managers identified poor requirements as the source of problems. A similar survey in Europe over 3800 organizations in 17 countries concluded that most of the perceived software problems are in the area of requirements specification (>50%) and requirements management [4].

The important conclusion, among many researchers was to improve the quality of requirements, by adopting an engineering approach [6, 5, 1].

1.1.2 What is Requirements Engineering?

The role of RE in software engineering is clearly captured by Zave [2]:

> "Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families."

This definition, highlights that RE should capture the basic need/motivation for the development of a software system. Secondly, it refers to identification of functions and constraints on the software system which would meet the goals. Thirdly, it stresses the need of unambiguous specification of the software behaviour, so that software engineers can implement and test the requirements correctly. Finally, the definition refers to the changing nature of requirements over time, being driven by the need of the customer. Also, the need to reuse partial specification for fast development of families of software.

1.1.3 Requirements Engineering Process

RE involves the following intertwined activities [6, 5]:

- *Requirements Acquisition*
  This activity involves interacting with the customers, end users and analysts to find out about the application domain. It involves identifying the Stake-holders, interacting with them to find their main goals and needs, defining the boundaries in terms of problems to be solved. The usual techniques include questionnaires and surveys, interviews, use cases and scenarios, viewpoint-oriented elicitation [7] and ethnographic techniques like participant observation. The main artifacts of this activity are informal documents, that are needed to be further analysed for software requirements.

- *Modelling and analyzing requirements*
  The main goal of this activity is to create abstract descriptions that are amenable to interpretation [5]. This abstract description, forms the basis for analysis and reasoning. The usual techniques include Domain Modelling, Enterprise Modelling[15], Data Modeling like Entity-Relationship-Attribute (ERA), Object-oriented modelling technique, Requirements Modelling Language [23].

- *Specifying requirements*
  In this activity, the requirements and assumptions are formulated in a precise way. Considerable effort has been devoted formal methods, supported by automated tools, that enable engineers to capture and specify the software requirements unambiguously. Considerable effort has been devoted to design of Formal Methods such as Z specification languages like Z [11], VDM [13], RAISE [12], CSP [14].

- *Validation requirements*
  This concerns agreeing specified requirements with the Stake-holders. The usual techniques involves carrying out customer site-visits and presenting the requirements, inviting stake-holders for inspections and reviews. During requirements validation process, different types of checks are carried out to ensure validity, consistency, completeness in the requirements.

1.1.4 Research Challenges

Given such complexity of the requirements engineering process, some of the major challenges for RE in the years ahead include [5] :

1. *Development of new techniques for formally modelling and analysing properties of the environment:* This would support specification of the expectations that a software component has of its environment.

2. *Bridging the gap between requirements elicitation approaches based on contextual enquiry and more formal specification and analysis techniques*: Contextual approaches, such as those based on ethnographic techniques, provide a rich understanding of the organizational context for a new software system, but do not map well onto existing techniques for formally modelling the current and desired properties of problem domains.

3. *Reuse of requirements models:* Requirements within a specific domain are more likely to be similar than the software components implementing the them [6]. This encourages the development of reference models for specifying requirements, for application domains. This would reduce the effort of developing requirements models from scratch [6, 5]. Work on problem frames is a preliminary attempt to classify and characterize task patterns [8].

4. *Support for requirements practitioners:* Requirements engineers are the ones, who apply the techniques of elicitation, analysis, specification. There is a challenge to find ways to support the requirement engineers to be able to do these tasks more effectively.

5. *Richer models for capturing non-functional requirements*: These are requirements which are attributed to the quality of the software like reliability, scalability, usability, simplicity. [9]

6. *Better understanding of the impact of software architectural choices on the prioritisation and evolution of requirements*: While work in software architectures has concentrated on how to express software architectures and reason about their behaviourial properties, there is still an open question about how to analyse what impact a particular architectural choice has on the ability to satisfy current and future requirements, and variations in requirements across a product family [10].

The main goal of this thesis at a high level, is to contribute to the challenges of:

> *Supporting requirements practioners* by *Re-using requirements model*.
> (Bullets 3 and 4 above)

In order to be more realistic, we have collaborated with *Microsoft Business Solutions* (MBS) with the aim:

- To understand a typical Requirements Engineers problem, that is representative in the industry,
- To carry out a case-study.

One of the challenges MBS faces, is to successfully fit a *standard software* (from MBS) to meet the specific needs of the customer. This is further clarified in the following sections, by first giving requirements challenge specific to MBS in Section 1.2, and then formulating a problem definition in Section 1.3.

1.2 Challenge of Fitting Standard Software to Customer Needs

In a typical *product* development life-cycle at MBS, the requirements elicitation and specification takes place at two levels as show in Figure 1.1:

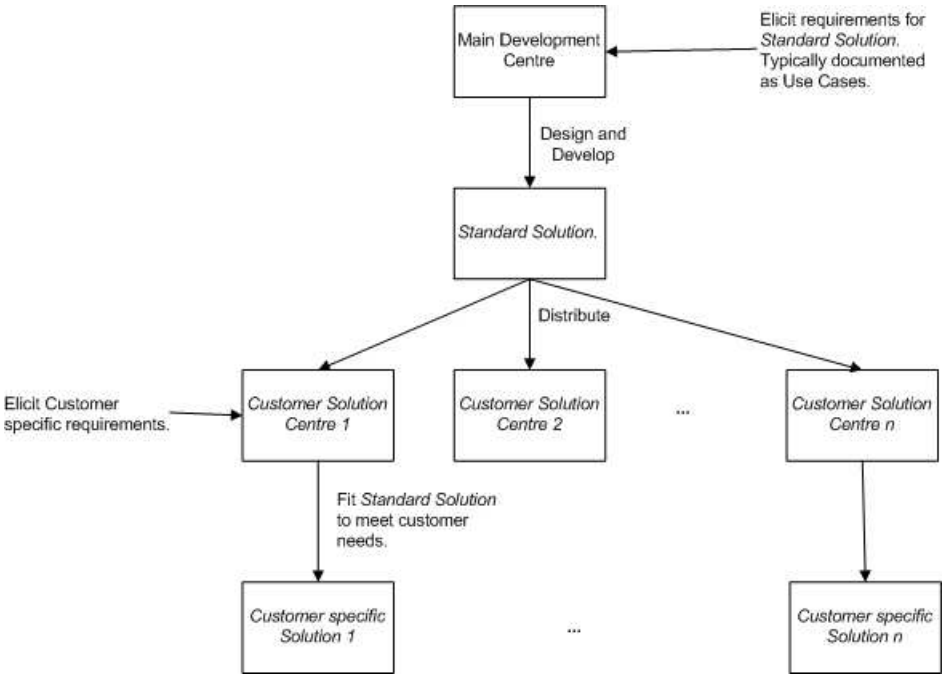


Figure 1.1: Requirements Challenge

- **Main Development Centre Level**
  The *Main Development Centre* (MDC) represents the software development house in MBS, where the development teams develop a *standard software*. The Standard software is characterised as generic in functionality, which could scale to a wider market as compared to a *customer-specific software*. The development team, typically carries out proper market research, makes a business case, spends good amount of time to understand the domain of the software and gather requirements from some representative Customers, Analysts and *Customer Solution Centres* (CSC). Typically, structured but informal methods like Use-cases are used to elicit and specify requirements. This is followed by development and distribution of the *standard solution* to a number of its CSCs.

- **Customer Solution Centre Level**
  The Customer Solution Centre (CSC) represents the software development houses, that is responsible for deploying a *customer-specific solution* that fits the specific needs of the customer. A CSC typically consists of consultants (Requirements engineers and Solution developers), who have a better understanding of the customers domain. They are often constrained with time (consultancy time) and limited documentation about the *standard solution,* that makes it a challenge for the *Requirements Engineer*:

  – To know type of requirements to "seek" from the customer, that could be met/satisfied by the *standard solution*.

– To capture these requirements and effectively communicate with both the customer and the CSC Solution developer.

1.3 Problem Definition

Given the above challenge, we shall now define the Thesis problem more precisely.

Problem Context

There is a given:

- an already existing software $\mathcal{A}$, that does not support some of customer requirements $\mathcal{R}$. Typically, $\mathcal{A}$ is extended with a sizeable software component $\mathcal{B}$, to meet this class of customer requirements $\mathcal{R}$. The combined system $\mathcal{A}\oplus\mathcal{B}$, is now capable of addressing the class of customer needs. The Main Development Centre (MDC) designs and develops $\mathcal{A}\oplus\mathcal{B}$.
- Also, each customer currently using $\mathcal{A}$, has his own specific requirements $\mathcal{R}'$, which are of type $\mathcal{R}$.

For $\mathcal{A}\oplus\mathcal{B}$ to be useful to the customer, the Requirements Engineer at the Customer Solution Centre (CSC), has to fit $\mathcal{A}\oplus\mathcal{B}$ with $\mathcal{R}'$ to generate a customer-specific $\mathcal{A}'\oplus\mathcal{B}'_{\mathcal{R}}$.
A CSC Requirements Engineer has a challenge to be able to elicit and specify right customer requirements $\mathcal{R}'$.

Problem Statement

The question is:

> How can the MDC assist a CSC Requirements Engineer to elicit and specify $\mathcal{R}'$ that would fit $\mathcal{A}\oplus\mathcal{B}$ to the specific needs of a customer?

Thesis Objectives

The main objectives of this thesis document are:

1. To propose a *solution concept*, that addresses the *problem statement* above. This is further elaborated in Part II on page 23.
2. To carry out a case-study, with the aim to apply the *solution concept* on a industrial project (*Event Management*) offered by Microsoft Business Solutions. This is further described in chapters of Part III on page 35.

Part II

Solution Concept

Chapter 2

Solution Concept

In this chapter, we first develop an understanding of the basic concepts in 7.3. This is followed by a Solution Formulation for the Problem Statement (Section 1.3) in 2.2.

2.1 Introduction

Figure 2.1 shows the general software development problem as presented by Jackson in [46]. Below we define the terms, base on which the solution concept is formulated.

Figure 2.1: The General Software Engineering Problem (Adapted from [46])

Domain

Domain refers to the application domain where there are some requirements to be met.
For example: Domain of Railways, Banking, Postal Service, Transportation, Sales and Purchase etc.

Requirement

We refer to requirement as some condition that needs to be met in the domain in order to meet some goals in the domain.
For example: Within Sales domain, it is a requirement that, after an order has been released, the customer must be notified with the delivery details.

Machine

A general purpose computer specialised with a Software Program. Machine interacts with a domain in order to ensure that, requirements in the domain are met. For the machine to control the domain (to solve domain problems), it has to have the knowledge of the domain *as it is*. That is, it should continuously maintains a representation (model) of the domain aspects. In other words, the machine has to keep an internal version of things that is going on in the domain or a model of the domain.

Challenge of Software Engineer

The challenge for a software engineer is to provide a prescription, that instructs a general-purpose computer, on how to interact with the domain to fulfill the requirements.

Software Program

A prescription, that instructs a general-purpose computer to a) capture the essential domain phenomenon b) enforce the desired conditions, and thereby fulfill the requirements in the domain.

Software Language

It is a language, that provides, syntax and semantics, to describe a prescription (*software program*) for the machine.

Specific Software Program -> Specific Machine

A prescription, that captures the essentials phenomenon of a customer- specific domain and enforces the desired conditions in order to meet the requirements specific to the customer domain.

Generic Software Program -> Many Specific Machines

A prescription, that captures the essentials phenomenon of a *class* of customer domain and is capable of enforcing the desired conditions in order to meet the requirements of a particular class of customer domains. This is normally called *Standard Software*. We are able to create a generic software program by identifying common basic phenomenon from the similar domains by abstraction.
For each customer, a specific machine could be generated by initialising the Standard Software with customers specific domain phenomenon and requirements.

2.2 Solution Formulation

In light of above concepts, we could infer that:

- A Standard Software $\mathcal{A} \oplus \mathcal{B}$, is a prescription to meet a specific class of requirements $\mathcal{R}$ of different customers.
- A Customer-specific software, can be instantiated from a Standard Software by augmenting it with customer-specific requirements $\mathcal{R}'$.
- Customer-specific requirements $\mathcal{R}'$ that could be met by the Standard Software $\mathcal{A} \oplus \mathcal{B}$, are of type $\mathcal{R}$.

For Example:
Microsoft has developed MS Office as a Standard Software $\mathcal{A} \oplus \mathcal{B}$, that addresses a class of requirements $\mathcal{R}$ (at a high-level):

- Document Management using Word, Excel
- Presentation using PowerPoint etc.

One customer might have a Documentation requirement $\mathcal{R}_1$ only, and his specific MS Office installation would need only MS Word. Another customer might have a requirement for both documentation and presentation $\mathcal{R}_2$, and his specific MS Office installation would need MS Word and Powerpoint. These customers specific requirements $\mathcal{R}_1$ and $\mathcal{R}_2$ are of type $\mathcal{R}$.

The above suggests that, we could assist a CSC Requirements Engineer to *elicit* customer-specific requirements $\mathcal{R}'$, by making him available with the Standard Software requirements $\mathcal{R}$, which are already found by the Main Development Centre. In other words, *Reuse* $\mathcal{R}$. With this approach, the Problem Statement (Section 1.3), could be refined as:

> How can we help a CSC Requirements Engineer to effectively reuse the Standard Software requirements found by the MDC ?

Typically the Standard Software requirements are specified informally in Natural Language or as Use-cases. These Use-cases could be handed over to a CSC Requirements Engineer for reuse, as a first step. They serve as an excellent example of usage of a Standard Software, which would help a CSC Requirements Engineer to seek for similar usage in the customers specific domain. Two main concerns related to using Use-cases as a reuse tool are:

- An informal nature of Natural language specification and Use-cases, makes it open for different interpretation (with reference to its initial intent) and hence ambiguous.
- Use-cases fail to provide the link of requirements to the Standard Software components and specifications, that are formal.

The above disadvantages could be approached by using a *Domain-specific language* (DSL), which is descriptive in nature, specific to the domain and have a formal syntax and semantics.
Based on the above analysis, we suggest that the MDC can assist a CSC Requirements Engineer:

1. By providing guidelines and examples of usage of the Standard Software as Scenarios (of Use-case) along with a clear specification of requirements in DSL. That would help the CSC Requirements Engineer to elicit similar requirements from the customer's specific domain. In Section 2.3, we describe our suggestion of an approach to create a domain-specific language.
2. By providing DSL itself with its clear syntax and semantics description, which could be used to specify customer-specific requirements.

2.3 Creating DSL

Figure 2.2, shows the process (along with the artifacts) we have considered to create a domain-specific language.

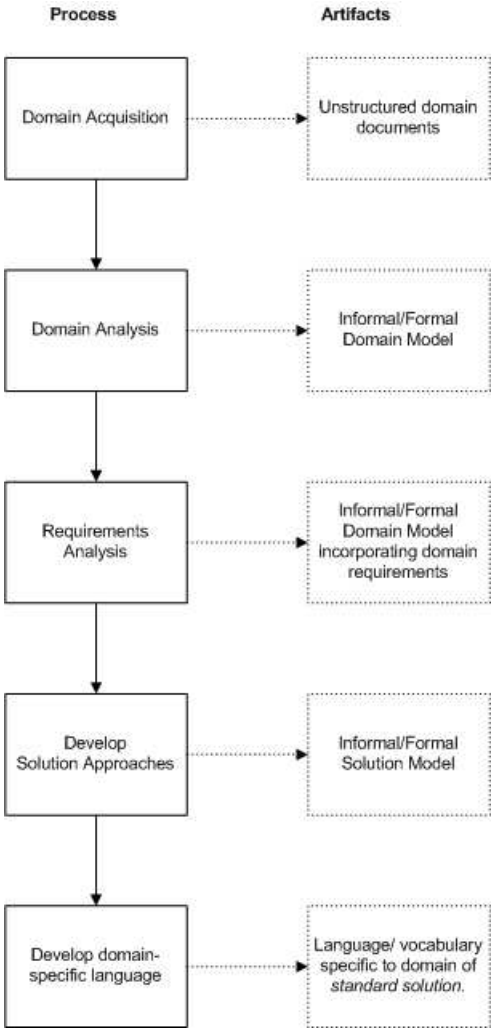


Figure 2.2: The Approach

Below is a description of the four major stages of the process, along with the description of the tools and techniques used and the artifacts:

- **Domain Acquisition**
  The main goal of this activity is to get to know a domain *as it is.* Elicitation techniques like Interviews, Site-visits, Use-cases/scenarios, View-point oriented analysis could be used. The information results in informal description of domain phenomenon. At this stage, we are not at all concerned with the requirements. The informal descriptions could be further structured using techniques like, Use Cases, Structured Natural Languages. The result of this activity is used for domain analysis.

- **Domain Analysis**
  In this activity, the informal domain information is analysed with the aim to identify basic domain aspects. *Abstraction* along with Formal Methods, is used as the base technique during *Domain Analysis*. This activity results in a informal and formal domain model.

- **Requirements Analysis**
  Once we have a better understanding of the domain, both informally and formally, we are in a position to seek requirements and create a requirements model by specifying them formally.

- **Develop Solution Approaches:**
  In this phase, we carry out step-wise development of different solution models, to get a better understanding of how the requirements could be met in the domain.

- **Develop Domain-specific Language**
  This activity involves developing a *language*, which could be used to express the link between, informal customer domain(and its requirements) and the formal Standard Software. The main advantage of a domain-specific language is that it is small, focused to solution domain, is descriptive and hence easily accessible to CSC Requirements Engineers. Further, it makes it possible to specify the customer specific requirements within the context of the Standard Solution, and more importantly, specify them unambiguously. An introduction to *Domain-specific language* is given in *Chapter* 3.

Chapter 3

Domain Specific Language

This chapter gives an introduction to domain-specific language, its characteristics, use and formalisation aspects.

3.1 Introduction

According to Scoot [21]:

> A Domain-specific language is a specification language that is restricted/focused to a particular application domain.

It is restricted or specific, in the sense that, it has a vocabulary and construct which caters to a specific application domain. Some well known domain-specific languages along with there domain is listed in Table 3.2 . For instance, LATEX is a language with a vocabulary and rules to typeset a document.

DSL	Application Domain
VHDL	Hardware Design
SQL	Database queries
LATEX	Typesetting
GAL	Video Device Drivers
BNF	Syntax
HTML	Hypertext web pages

Table 3.2: Example domain-specific languages (Adapted from [19])

They are also called *Little Languages* and *Application-specific languages.* In contrast to a general purpose language, a domain-specific language (DSL) is expressive uniquely over the specific features of programs in a given problem domain [22].

3.2 Characteristics of a DSL

Below is a list of characteristics common among existing DSLs:

- ***DSLs are usually small:*** They have a restricted vocabulary and rules. And are just large enough to express the basic abstractions and behaviour in the domain of concern.

- ***DSLs are usually declarative:*** In contrast to imperative languages that specify explicit sequence of steps to produce a result, DSLs describe the relationships.

- ***DSLs are less expressive power:*** Compared to a imperative language, DSL are less expressive due to its restricted focus. It has features clearly focused to express the specific domain requirements only.

- ***DSLs provide built-in abstractions:***

For example the **make** command in Unix. It is a utility to automatically determine which pieces of a large program need to be recompiled, and issues the commands to recompile them. The language of makefiles is *small* and mainly *declarative*. Its *expressive power* is limited to updating task dependencies. It hides implementation details like file last-modification time and provides domain abstractions such as file suffixes and implicit compilation rules. As a result, the user may concisely and precisely express update dependencies.
The declarative nature of DSL, makes is highly suitable as a specification language.

3.3 Design and Formalisation of DSL

The results of Domain analysis, Requirement analysis and Step-wise development of Solution Approaches, provide the necessary inputs for designing a Domain-specific language. A formal specification language ,according to Guttag et al [47]:

> Definition: A formal specification language is a triple, $< Syn, Sem, Sat >$, where Syn and Sem are sets and is a $Sat \subseteq Syn \times Sem$ relation between them. Syn is called the language's syntactic domain; Sem, its semantic domain; and Sat, its satisfies relation.
>
> Definition: Given a specification language, $< Syn, Sem, Sat >$, if $Sat(syn, sem)$ then syn is a specification of sem, and sem is a specificand of syn.
>
> Definition: Given a specification language, $< Syn, Sem, Sat >$, the specificand syn in Syn is the set of all specificands sem in Sem such that $Sat(syn, sem)$.

Informally, a formal specification language provides a notation(its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects specify each specification. A specification is a sentence written in terms of the elements of the syntactic domain. It denotes a specificand set, a subset of the semantic domain. A specificand is an object satisfying a specification. The satisfies relation provides the meaning, or interpretation , for the syntactic elements.
In the context of above definitions, Figure 3.1 on page 33 is intended to show how a domain-specific language fits-in to specify requirements by the CSC Requirements Engineer.

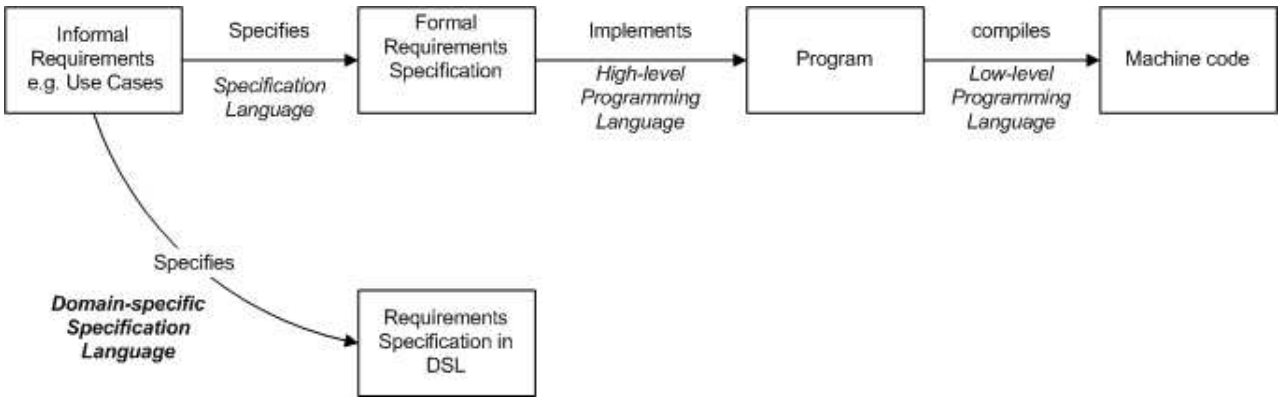


Figure 3.1: Syntactic, Semantic and Satisfy Relational context of DSL

From the figure we can observe the different syntactic Syn, semantic Sem domains and the satisfying relation Sat among them as listed in Table 3.3:

Language	**Satisfy Relation**	**Syntactic Domain**	**Semantic Domain**
RSL, Z etc.	Specifies	Informal Requirements	Formal Requirements Specification
High-level Programming Language	Implements	Formal Requirements Specification	Program
Low-level programming language	Compiles	Program	Machine Code
Domain-specific language	Specifies	Informal Requirements	Requirements Specification in domain-specific language

Table 3.3: Syntactic, Semantic and Satisfy Relational context

It is to be noticed that, the main users of a domain-specific language is the CSC Requirements Engineer and the CSC Solution Developers. This is in contrast to the specification languages like RSL, Z etc, to be used by the development team at the Main Development Centre (MDC).

Part III

Case Study

Chapter 4

Case Study Overview

This case study involves application of the approach mentioned in Chapter 2, to an industrial-scale project. We have used the *Event Management* (EM) project at Microsoft Business Solution(MBS), Denmark, for the same. A Standard Solution of EM is being developed at MBS on top of an existing *Simple Application System (SAS)*, that will be distributed to its Customer Solution Centres (CSCs). The CSCs in turn sell and fit the solution to the end customers needs.

4.1 Motivation

The choice of the EM project was motivated by the following:

- **From Thesis point of view**
  The *Event Management* project was in the initial stages of development process involving requirements gathering and specification. This gave us a great opportunity to utilize the domain knowledge being gathered and apply the domain modelling techniques.
- **From MBS point of view**
  Since the Standard EM Software was intended to be built on top of an existing Standard Software (SAS) (that is already in the market), the work on the Thesis could provide some inputs for the company to help CSC Requirements engineers to later deploy the *Event Management* Software.

4.2 Information Source

The information gathered during our investigation came from the following sources:

- On-Site Interviews
  We visited Microsoft Business Solutions, located in Copenhagen, regularly during the period of this thesis. Informal interviews were conducted with the team members of *Event Management* project. Interviews involved Product Manager, Program Manager, Developers, Tester, Usability Experts and Documentation engineers. Interview sessions were approximately two hours.
- Documentation
  We reviewed business and technical documents describing the business case, scope, use cases, prototype descriptions for the Event Management System. A list of Use Cases that were considered for this thesis are listed in Appendix B.

- Reviews by EMS Team
  To evaluate the usefulness of the results of our approach, particularly, the domain-specific language, we had the review of the same with the EMS team members.

Nevertheless, since the development of EMS was in progress, it posed a challenge to get enough and complete information. Further, because of confidentiality agreement, the *Event Management Project* documentation is not part of this document.

4.3 Analysis Overview

The goal of this section is to give an overview of the analysis and to give pointers to specific chapters that addresses the details.
Figure 4.1 on page 38 shows, how the domain of *standard solution* , is expected to evolve (from as *it-is* now to as it is intended *to-be*), as some of the requirements are met by the proposed *Event Management Solution.*

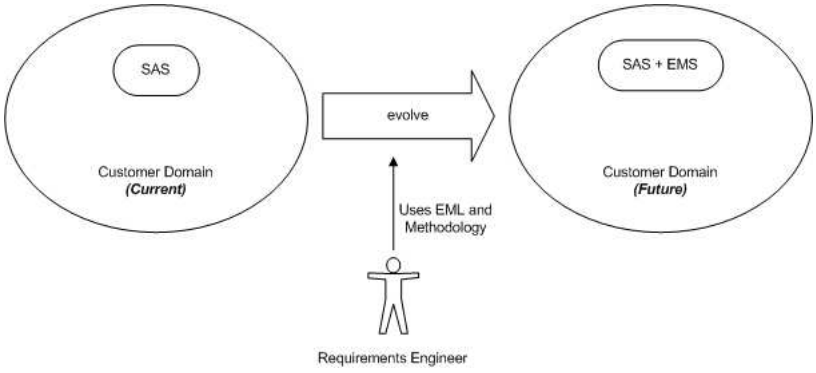


Figure 4.1: Evolution of Domain with EMS System

The following could be observed from the diagram:

- **Current Customer Domain**
  The *current* customer domain consists of the Simple Application System (SAS), its users and other external systems.

- **Evolution of Domain**
  The Requirements Engineer at a CSC has to identify the customer specific requirements for determining how to fit the proposed EMS to meet the needs of the customer. He is able to do that by capturing and specifying the requirements using an Event Management Language (EML) and a methodology to use it.

- **Future Customer Domain**
  The *future* customer domain would mean extending the *current* domain with Event Management System.

Given the above very high level overview of EMS solution with respect to the domain, we organise the chapters as below:

1. **Introduction to Event Management Project**
   A short introduction to the Event Management Project is given in Chapter 5 on page 41.

2. **Domain Analysis**
   In order to understand the requirements in the current domain, we need to first analyse different facets of the domain that constitute the need for Event Management System. See Chapter 6 on page 45 , for further details on the domain analysis of EMS.

3. **Requirements Analysis**
   Identify requirements that needs to be met by the proposed EMS. This is further detailed Chapter 7 on page 55.

4. **Solution Approaches**
   Stepwise analysis of four different solution approaches, in order to understand the EMS requirements better. Further details on different solution approaches for EMS, is given in Chapter 8 on page 65 .

5. **Design of Event Management Language**
   Results of Domain Analysis, Requirements analysis and Step-wise development of Solution Approaches, serve to define the syntax and semantics of a domain-specific language for EMS. This is elaborated in Chapter 9 on page 81.

6. **Guidelines for Requirements Engineer**
   This chapter provides overall guidelines and example usage scenarios and its specification in EML. This is further elaborated in Chapter 10 on page 99.

Chapter 5

Introduction to Event Management Solution

This section gives an overview of the *proposed* Event Management System in terms of its Need, Business problem statement, Target customer and a High level solution concept.

5.1 Need for EMS

The *Simple Application System (SAS)* as it is now, is not event-based. SAS tend to be information storages and interactive with the user to carry out a specific task. It assists the *application users* to maintain data, retrieve data and carry out some functions. The users make the choices to gather the information and carry out the tasks. Some of the common problems, expressed by the customers of SAS are:

- Slow and ineffective information flows between people and systems
- Problems in detecting events and exceptions leading to disruptions to production, deliveries and cash flow in general
- Difficulties in making decisions due to missing information or problems in reaching a person who can make the decision.
- Difficulties in providing the right and flexible information to customers or partners on time and in the right quality
- A lot of time spend on manual checks via reports

Here is an *example* that highlights the need.
As it is today, whenever a Salesperson enters a Sales Order in SAS, and if there is not enough credit for the customer, the user is warned about the credit problem. This may do well at an initial level. However, this situation requires many more people to act:

- Customer department responsible, may need to check with the bank whether the order is safe to confirm.
- The Manager needs to know, in certain situations, to ensure a dollar limit for that check.

- Bank may need information on the customer for approval.

SAS as it is now, deals with none of these flows. SAS shows a *credit limit* warning on the order and leaves it open to the choice of the Salesperson. The remainder is not supported. This is exactly where an *Event Management Solution* comes in. An *Event Management System,* would act on changes like completion of an activity or an occurrence of an exception, by notifying concerned user or application about the same. That is, an EMS is supposed to communicate extraordinary things between people and applications. And it is supposed to help information flow better than what is offered by SAS. The above problems translates to the following high-level *needs or goals* for the software system (EMS system):

- To be able to streamline existing business processes in order to run the business fast and lean.
- To increase the productivity of the employees by pushing relevant information to the employees:
  - By providing every employee the facility, to have access to relevant information in the easiest way.
- To provide timely and accurate information about exceptional situations, to be able to make the right decisions
- To improve the response time for the customers

5.2 Business Problem Statement

With more and more customer using transactions based systems (like SAS) to carry out the business tasks, it is getting harder to find out which tasks are complete and which are due. Further, the user has to analyse lot of transacted data, to figure out if an exception has occurred. In short, more and more communication is going on between the employees and the IT system rather than between employees as in the old days. This has resulted in the IT systems having most business information and a lack of visibility for an employee about the overall state of the company.
On a day to day basis, the users needs to be pro-actively notified of changes, exceptions, task completions, early warnings etc. in order to act fast. For the employees in the company to communicate more effectively between departments, would also require them communicate between applications.

5.3 Target Market and Customer

EMS is a new *standard solution* for the existing and new MBS customers.

- **New Customers:** Customers in this segment include companies that are purchasing a *Simple Application System* and in the offering want to include the *Event Management* functionality.
- **Upgrade Customers:** This segment includes existing customers on an older *Simple Application System,*who want to upgrade to include EMS.

5.4 Overall solution concept

Figure 5.1, shows the overall solution concept of the EMS with respect to the existing application areas of the *standard solution.*

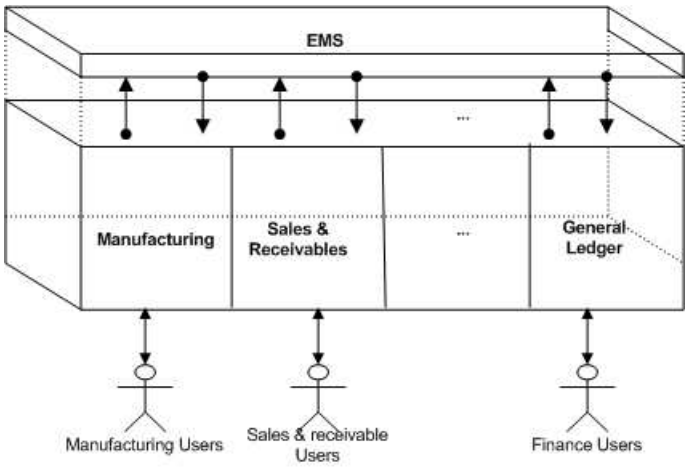


Figure 5.1: Simple Application System(SAS) extended with EMS.

We observe that, EMS has to serve as an application area that spans across all the specific application areas like Sales and Receivable, Manufacturing, General Ledger and others. It is intended to provide the flexible and pro active link between users of these application areas.
For Example:
Consider the scenario of a typical Shipment Workflow. Table 5.2 on page 43, shows the steps of the shipment workflow as it is done now without EMS.

Step	Description
1	Sales Order is released by a Salesperson.
2	The Shipment manager manually checks for requests for creating new Shipment Order..
3	Shipment manager creates a Shipment Order.
4	Shipment manager creates a Pick document and prints it.
5	Warehouse employee picks the items according to the Pick document.
6	Warehouse employee registers the Pick document.
7	Shipment manager posts the Shipment Order.
8	Sales person invoices the Sales Order.

Table 5.2: Shipment Workflow as it is, without EMS

Now consider Table 5.4 on page 44, which shows the same shipment workflow as envisaged with the EMS solution.

Step	Description
1	Sales Order is released by a Salesperson.
2	*The Shipment manager is notified by Email to create a new Shipment Order.*
3	Shipment manager creates a Shipment Order.
4	Shipment manager creates a Pick document and prints it.
5	*Warehouse employee is notified to do a pick.*
6	Warehouse employee picks the items according to the Pick document.
7	Warehouse employee registers the Pick document.
8	*The Shipment manager is notified to post the Shipment Order.*
9	Shipment manager posts the Shipment Order.
10	The Salesperson is notified to invoice the Sales Order.
11	Sales person creates an invoices related to the Sales Order.

Table 5.4: Shipment Workflow with EMS

Note that, the EMS proactively notifies the necessary information to users for a more effective workflow. For instance in this case, in step 2, the Shipment Manager is now actively notified by an Email about the release of a Sales Order, further, informing him to create a Shipment Order.

Chapter 6

Domain Analysis

In this chapter we shall analyse the application domain, which the *Event Management System* is envisioned to support. Domain of EMS is that part of customer domain, where the Events are happening or rather where Events are generated. Following methodical approach towards domain analysis is taken:

- First, an informal description of different aspects of the domain that participate in Event generation is given in Section 6.1.
- Then the *Simple Application System* (SAS) is identified as the main domain area of interest for analysis. A detailed static and dynamic analysis of is given in Sections 6.2, 6.3 and 6.4.
- The above leads to a complete model of *Simple Application System* in Section 6.5.

6.1 Domain of Events

The domain of Event Management System, is the application area where EMS is supposed to be deployed to meet the customers need. Figure 6.1 shows a representative domain for the EMS solution. It is typically a business house consisting of :

- Departments like Sales, Purchase, Accounts, Manufacturing etc.
- Employees working in these departments.
- Employees use Simple Application Software (SAS) software product, to carry out there daily activities like “Entering a new Sales Order”, “Printing Invoices”, “Registering receipt of goods from the Supplier” etc.
- They interact with customers, vendors, banks and other external partners and institutions.

Employees

Employees of the domain have different roles depending on the type of tasks they are doing. It can be observed that employees carry out there tasks in the following ways:

- Manually doing the tasks like making a phone call, giving a presentation etc

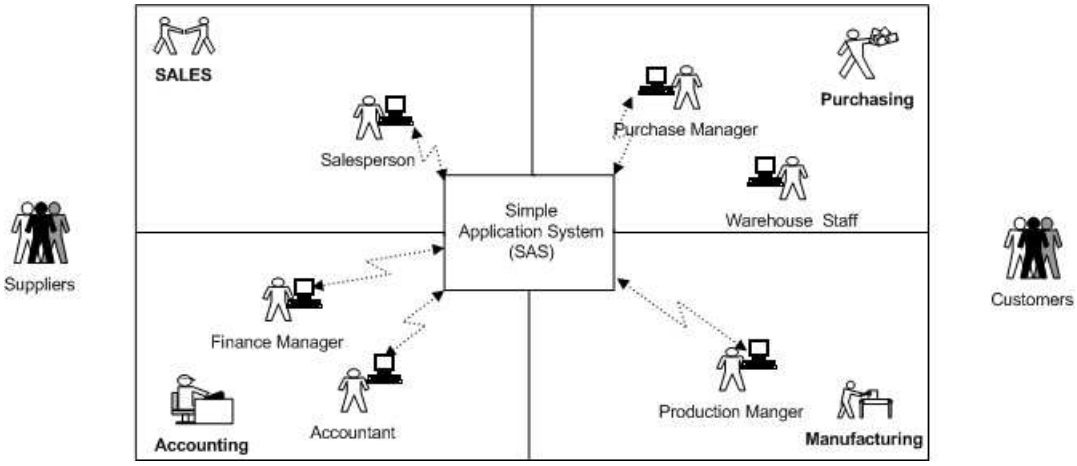


Figure 6.1: Domain of Event Management Solution

- Interacting with SAS and using the functionality offered by SAS.
- Use other systems like IT systems(Email), Mobile, Fax etc to carry out there tasks.

By analysing the Use Cases (Chapter B) and some of informal requirements of the proposed EMS solution, we found that most of them are around the activities involving Employees and their interaction with SAS software to do there daily tasks. This motivated us to explore the interaction and activities between the user and SAS.

6.2 What is a Simple Application System?

This refers to the *standard solution* from Microsoft Business Solution, that is already being used, by the customers in their specific domains. Figure 6.2 on page 46 shows an informal abstract model of Simple Application System (SAS).

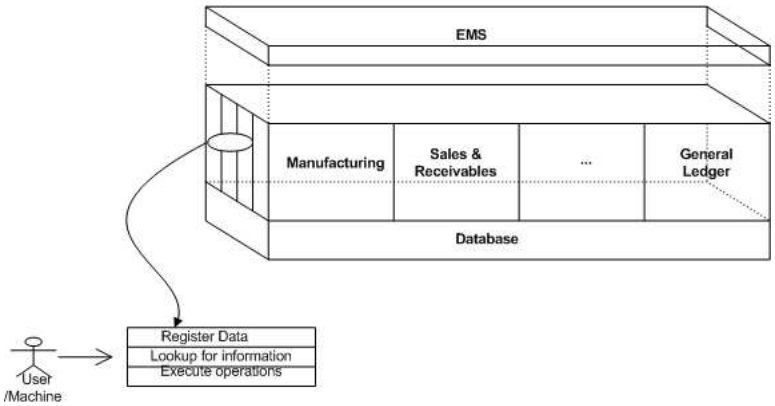


Figure 6.2: Informal model of Simple Application System

Following could be observed from the diagram:

- SAS supports the employees in the customer domain, to carry out tasks in different application areas like Manufacturing, Sales & Receivables, General Ledger etc. Each of the application area, provides specific business functionalities for its users to do his daily tasks.
  For example:
  A Salesperson interacts with Sales & Receivable application area to "Create Sales Order", "Create Sales Invoice", "Maintain Customer Addresses" etc.

- Employee of different departments use different application areas.
  For example:
  Salesperson, Sales Manager, Campaign manager normally use Sales & Receivables application area of SAS.

- Each application area in SAS supports the user to carry out following kinds of tasks:

  - **To Register Data**
    These are functionalities that provide a user, to maintain (Create, Update, Delete) basic information about the entities in the real world.
    For example: "Create Customer ","Create Sales Order", "Setup Salespeople", "Setup Shipment methods" etc.
  - **To do Lookups**
    These are functionalities that provide a user, to lookup for some information in order for doing his daily tasks.
    For example: "Check Sales Order Status for a Customer ","Get a Sales Statistics report", "Get a Customer Order Summary report", "Check if an order has been shipped" etc.
  - **To execute some Operations**
    These are functionalities that provide a user some sort of automated processing.
    For example
    * Perform some operations on the data registered - "Release a Sales Order", "Run MRP", "Cancel an order" etc
    * Perform some calculations/operations: "Implement price changes", "Calculate invoice discounts" etc.

6.3 Static Analysis of SAS

Below we abstract the static aspects from the above informal descriptions and state it more formally. The results below are described by a concise *narrative,* followed by a *formal specification* in RSL (Section **??**), and then an *example* instantiation.

- **Application System**
  The Simple Application System `A` consists of one or more, separately invocable application areas `AP`.
  Formal Specification

$$\texttt{A} = \texttt{AP-}\mathbf{set}$$

  Example:
  `{General Ledger, Manufacturing, Sales & Receivables,..., Customer Relationship Management}`

- **Application Area**

  Each application area AP offers one or more services S to its users. The services are of types: *Register, Lookup* and *Execute.*
  Formal Specification

  type APS = AP $\overrightarrow{m}$ S-**set**
  value obs_ servType: S → RegisterService | LookupService |ExecuteService

  Example:
  [Sales & Receivable ↦ {"Create Customer", "Post Sales Order", "Cancel a Sales Order",...} ....]

- **Application User**

  - Each application area AP has its own associated users U.
    Formal Specification

    type APU = AP $\overrightarrow{m}$ U-**set**

    Example:
    [Sales & Receivable ↦ {Salesperson, Sales Manager}, Purchase & Payable ↦{Purchaser} ....]

  - A user U(u) can interact/communicate with different application areas A(a) through a specific channel to the application area:
    Formal Specification

    channel {cua[u,a]:(R|V)|u:UIdx,a:AIdx}

    Annotations

    * For a in the index AIdx set of n indices, corresponding to n application areas.
    * For u in the index UIdx set of m indices, corresponding to m application users.
    * R and V designate, the requests and values respectively.

    Example:
    A Salesperson uses different User-interfaces to create a Sales Order and to create a Campaign.

6.4 Dynamic Analysis of SAS

In this section, we we shall analyse the dynamic behaviour of the Simple Application System and its Users as a system.

6.4.1 System Behaviour

The system `Sys` can be modeled as a main process consisting of:

- a Simple Application system (SAS). SAS in turn consists of, `n` separately invocable application areas `A(a)` .
- Application user processes, `U(u)` running in parallel.

Formal Specification

$$\begin{aligned} \texttt{Sys}\textbf{: Unit} &\rightarrow \textbf{Unit} \\ \texttt{Sys()} &\equiv \{\texttt{U(u)|u:UIdx}\} \parallel \{\texttt{A(a)|a:AIdx}\} \end{aligned}$$

Figure 6.3 on page 49, schematically shows the complex of channels and processes. In the figure:

- `U(u)` and `A(a)` designates the user and application area processes respectively.
- The communication between the user and an application area is through the channel `cua[u,a]` . The user can both send and receive information through this channel.
- The communication between two application areas is through the channel `caa[a,a']` .

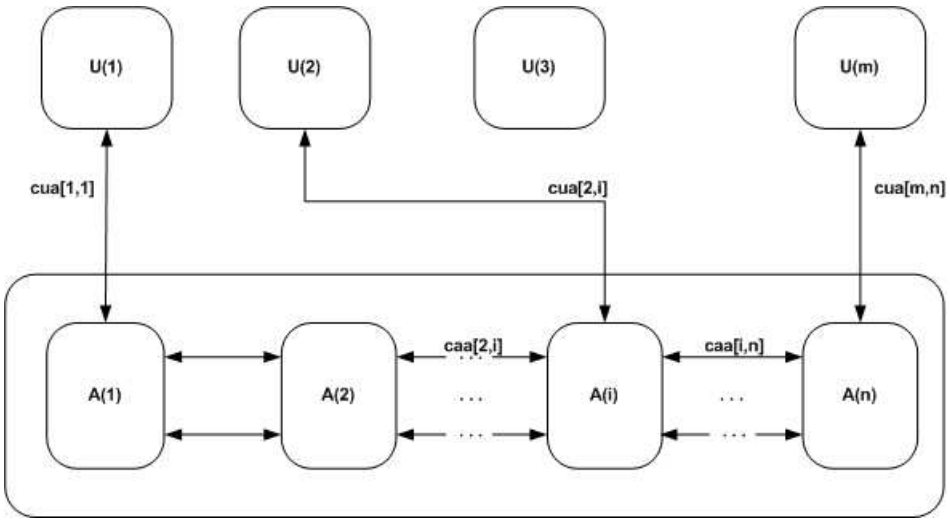


Figure 6.3: A Schematic Diagram: Users, Application areas and Channels

6.4.2 User Process behaviour

The user interacts with an application area, by making service-requests and receiving the results back.

Narrative

1. User `U(u)` makes a service request `r` to an application area `ap:AP` , through the communication channel `cua[u,a]` .
2. User waits for the response **v** to the request on the same channel `cua[u,a]`.
3. User while waiting for a response from a previous request(s), could send another request to an another application area.

Formal Specification:

```
U(u):u:UIdx  →  in, out {cua[u,a]|a:AIdx} Unit
       U(u)  ≡  let ap:AP = A1 ⊓ A2 ⊓ ... ⊓ An in
                 let r:R = gen_req(select_serv(ap),userinput:V) in
                  output (cua[u,obs_AIdx(ap)],r)
                  U(u) |||
                  {letv = input (cua[u,obs_AIdx(ap)) in U(u) end}
                 end
                end

                gen_req:S × V → R

                select_serv:AP → S
                select_serv(ap) ≡ ⊓{s | s:S • s ∈ aps(ap)}
```

Annotations

- `UIdx`and `AIdx` are an index set of Users and Application areas respectively.
- `ap:AP = A1 ⊓ A2 ⊓ ... ⊓ An` designates, the users choice to select different application areas depending on the task in hand.
- `output (cua[u,obs_AIdx(ap)],r)` designates, the user sending of a request `r` to the application over the channel `cua[u,a]`.
- **`U(u)` ||| {let `v` = `input` (`cua[u,obs_AIdx(ap)`) in `U(u)` end},** denotes that while the user can wait for a reponse from the application area, he can continue with new requests to other application areas. The choice that the user can choose to do nothing or other activities are not made explicit.
- The auxiliary functions denotes the following:

- `gen_req:S × V → R`
  Given a service and its input values, the function generates a service request.
- `select_serv:AP → S`
  `select_serv(ap) ≡ ⊓{s | s:S • s ∈ aps(ap)}`
  Given an application area, the function designates, a user making a non-deterministic external choice of a service.

- We introduce the following two macros for simplicity, that we shall use in this document without further mentioning:
  - `output` : **ChannelName** × **ARG** → Unit which means **ChannelName**!**ARG**.
    Typical usage of this macro is as follows:
    `output(channnel, val)`
  - `input` : **ChannelName** → Unit `RES` which means **ChannelName**!?. Typical usage of this macro is as follows:
    let `v=input(channel)` **in**...**end**

6.4.3 Application Area behaviour

Narrative

1. An Application area `A(a)` can receive service-requests `r` from both Users `U(u)` and other Application areas `A(a')`.
2. The Application area executes the service-request.
3. The Application area then sends the result back to service requester: User *or* another Application area. Or choose to do nothing.

Formal Specification:

```
A:a:AIdx  →  in,out {cua[u,a]|u:UIdx},
             {caa[a',a] | a':AIdx • a ≠ a'} Unit
   A(a)  ≡
         []  {let r:R = input(cua[u,a]) in
               let v = execute(r)in
                 output(cua[u,a],v) ⊓ skip
               end
             end
             ⊓
             let r:R = input(caa[a',a]) in
               let v = execute(r)in
                 output(caa[a',a],v) ⊓ skip
               end
```

end|u:UIdx,a′:AIdx};A(a)

execute:R → V

Annotations

- r:R= input(cua[u,a]) designates, receipt of a service-request from a user U(u)
- r:R = input(caa[a′,a]) designates, receipt of a service-request from an another application area A(a′) .
- Further, the choice of selection of the service requests from a User or another application area is internally non-deterministic.
- cua[u,a] designates, the communication channel between the user U(u) and the application area A(a) . And caa[a′,a] designates the communication channel between the application area A(a) and an another application area A(a′) .
- Auxiliary functions denote the following:
  - execute:R → V
    Given a service-request, the function carries out the service and returns the service result.

6.5 Domain Model of SAS

Below is a complete model of Simple Application System, taking into account both static and dynamic aspects analysed in the previous sections.

type R,V,U,S,AIdx,UIdx
AP==A1|A2| ...|An
APU=AP $\overrightarrow{m}$ U−**set**, APS=AP $\overrightarrow{m}$S−**set**

channel {cua[u,a]:(R|V)|u:UIdx,a:AIdx},
{caa[a,a′]:(R|V)|a,a′:AIdx • a ≠ a′}

value aps:APS,
obs_servType:S → Register|Lookup|Execute
obs_AIdx: AP→AIdx

Sys: **Unit** → **Unit**
Sys() ≡ {U(u)|u:UIdx} ‖ {A(a)|a:AIdx}

```
U(u):u:UIdx  →  in,out {cua[u,a]|a:AIdx} Unit
      U(u)   ≡  let ap:AP = A1 ⊓ A2 ⊓ ... ⊓ An in
                 let r:R = gen_req(select_serv(ap),userinput:V) in
                  output (cua[u,obs_AIdx(ap)],r)
                  U(u) |||
                  {letv = input (cua[u,obs_AIdx(ap)) in U(u) end}
                 end
                end

                gen_req:S × V → R

                select_serv:AP → S
                select_serv(ap) ≡ ⊓{s | s:S • s ∈ aps(ap)}

A(a):a:AIdx  →  in,out {cua[u,a]|u:UIdx},
                {caa[a′,a] | a′:AIdx • a ≠ a′} Unit
      A(a)   ≡
             []  {let r:R = input(cua[u,a]) in
                   let v = execute(r)in
                     output(cua[u,a],v) ⊓ skip
                   end
                  end
                  ⊓
                  let r:R = input(caa[a',a]) in
                   let v = execute(r)in
                     output(caa[a,a'],v) ⊓ skip
                   end
                  end|u:UIdx,a′:AIdx};A(a)

                execute:R → V
```

Chapter 7

Requirements Analysis

In this chapter we shall analyse the requirements for the which the *Event Management System* is envisioned to support. Following methodical approach towards requirement analysis is taken:

1. First, an informal description of the overall requirements for EMS is given in Section 7.1.

2. Then an informal and formal description of the following is given:

   (a) *Event* in Section 7.2

   (b) Notification in Section 7.3

   (c) Event-Notification Template in Section 7.4

   (d) Event Management System and State in Section 7.5

   (e) Event Notification Instance in Section 7.6

3. The above leads to a complete requirements model for EMS in Section 7.7.

7.1 Overall Event Management

Figure 7.1 on page 56, shows the current domain extended with EMS in more detail. It is to be noticed that a new EMS is an extension of existing SAS in a customer domain.

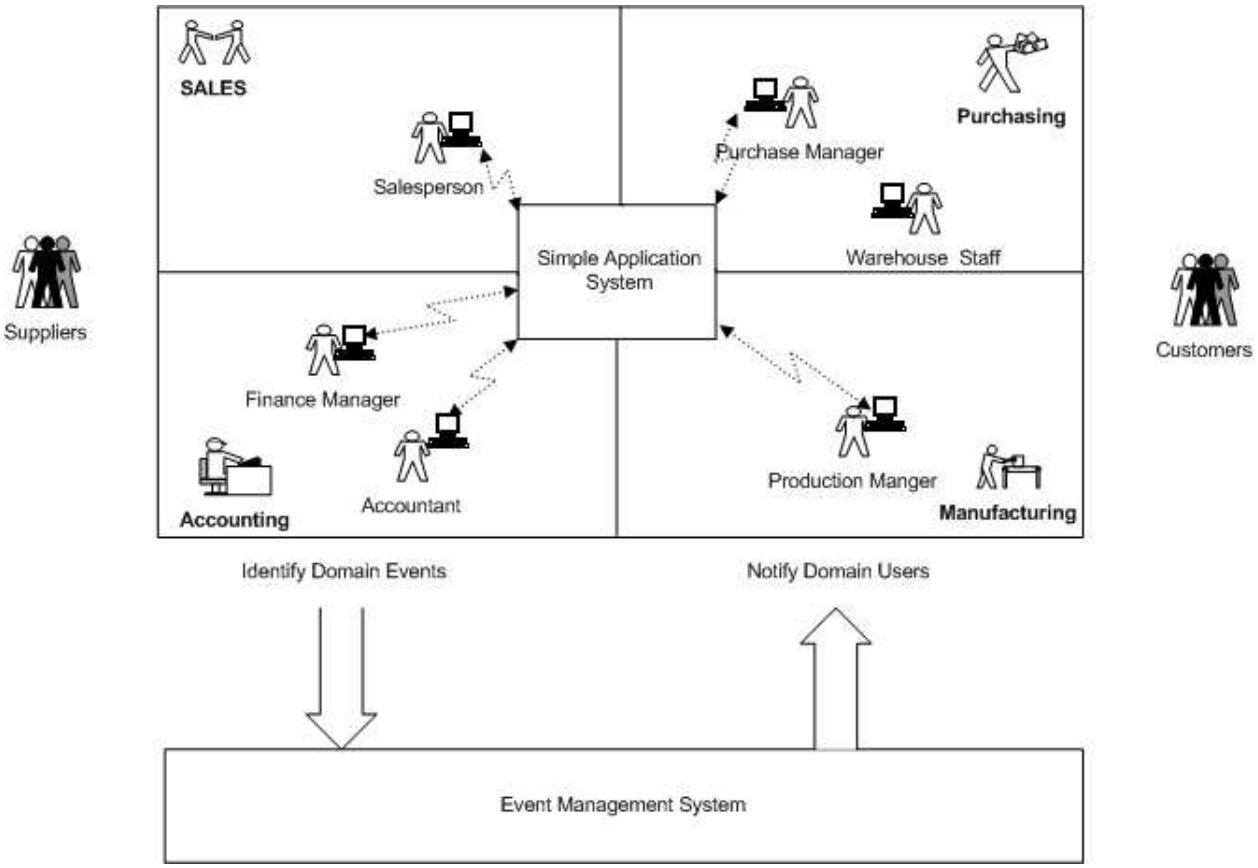


Figure 7.1: Domain extended with EMS

From the figure we can observe informally, that two high level requirements for EMS are:

1. Ability to identify an event
2. Ability to notify domain users

In the following sections, a detailed static and dynamic analysis of Event and Notification is given.

7.2 Event

We define the following event types with respect to SAS:

- ***User Event*** - These are events initiated by a SAS Application User by carrying out a *Task* in an application area of SAS, and meeting some condition.
  Example:
  "A Salesperson releases a Sales Order with prices and discount, in SAS, and the Profit-Margin is below 10%."

- ***SAS Event*** - These are events initiated by a SAS *Application itself* by evaluating some condition over its internal model of the domain. We would consider a scheduled event like time has reached 4pm, also as part of SAS event.

Example:
*"*The total costs related to travel has exceeded the budget level and a workflow is started to inform the budget owner*"*.

- ***External Event*** - These are events initiated by an IT system external to SAS.
  *Example: "*Receipt of a mail from a customer*"*

Static Analysis

1. **Event** `E` consists of:

   (a) an Initiator `I`: It can be SAS users, SAS itself or External.

   (b) an Event Task `ET`: It represents a task carried out by the Initiator.

   (c) an Event Rule `ER` over the data associated with the Event Task `T` .

   Formal Specification

   $$E = I \times ET \times ER$$

2. **Event Initiator** `I` are of types:

   (a) SAS User,

   (b) SAS itself or

   (c) an External System like Message Queue, Email, a RPC call.

   Formal Specification

   $$I = User | SAS | External$$

3. **Event Rule `ER`**, is a conditional expression over Task Data, that qualifies an incident as an Event.

   Formal Specification
   We can specify it as a function that returns a boolean.

   $$ER : V \rightarrow BOOL$$

Behaviour of an Event

Given a static event definition, we now define, how an event is generated. We consider an event `Event(e)` has occurred when:

- a Task `T` is invoked by the initiator `I` .
- and an associated event rule `ER` is satisfied.

*Formal Specificatio*n

```
Event:E  →  RS
Event(e)  ≡  let(i,et,er) = e in
               cases invoked(i,t) of
                FALSE → NOEVENT,
                res:RS →
                 if er(res) then res
                 else NOEVENT end
               end
             end

             invoked:I×T → FALSE|RS
```

Annotations

- `Event(e)` , designates the occurrence or non-occurrence of an Event.
- `RS` , designates the output of an Event, which can be a result `res` when an event occurs or `NOEVENT` when there is no event.
- The auxiliary functions denotes the following:
  - `invoked:I×T → FALSE|RS`
    Given an Initiator and Task, the function checks if the initiator `I` has carried out an associated Task `T`. And the function returns the result of the task.

7.3 Notification

Once an event has occurred, the recipient are notified to carry out some *Task* or just informed about the occurrence of the event to act appropriately. Accordingly we define two types of notifications:

- **Information Notification** - These includes notifying a recipient with simple information messages, alerts, warnings.
  For Example: "When the item inventory goes below reorder point, the Purchase Manager is alerted about the same."
- **Task Notification** - These includes notifying the recipient to carry out a task.
  For Example: "Once a new item is created, the Material Planner is notified to create a BOM structure for the item inside SAS."

Static Analysis

1. A Notification `N` consists of:

   (a) a Recipient **`R`,**

   (b) a notification Rule `NR`

   (c) a Channel `CH` of communication between EMS and the recipient,

   (d) and the message `M`.

   Formal Specification

$$N = R \times NR \times CH \times M$$

2. A recipient `R` can be:

   (a) Fixed Recipient,

   (b) Dynamic Recipient,

   (c) or a subscribed

   Formal Specification

$$R = Fixed | Dynamic | Subscribed$$

Behaviour of Notification

Once an event has occurred, a notification can take place. Given an event `e` and its results `res`, a notification process `Notification(e,res)` involves:

1. Identification of all the notifications `ns`, that needs to be sent as a result of the event occurrence.

2. For each notification `n`,

   (a) Notification Rule `nr` is identified.

   (b) The recipients are qualified for notification, by evaluating the Notification Rule against the event result, `nr(res)`

   (c) For the qualified recipients, the notification message `M` and the channel of communication `CH` is identified.

   (d) A notification of the message is send to the recipient on the channel.

Formal Specification:

```
Notification:E × RS  →   out {crcp[r]:M|r:RIdx} Unit
Notification(e,res)  ≡   let ns:N-set = ent(e) in
                     ‖   {let (r,nr,ch,m)=n in
                          if nr(res) then output(ch,m) end
                          end |n ∈ ns}
                         end
```

Annotations

- Notifications are sent to the recipients in parallel, that simultaneously, in particular sequence.
- {crcp[r]:M|r:RIdx}, designates the communication channel between the Notification Process and the recipient r. RIdx is an index set of all recipients. Note that, this channel used for specification, directly models the required notification channel of the domain.

7.4 Event-Notification Template

- We define an Event Notification Template ENT as a static definition, that relates an event E to one more intended notifications N−**set**.
  Formal Specification

$$\texttt{ENT} = \texttt{E} \ \overrightarrow{m} \ \texttt{N-set}$$

- Given an Event-Notification Template, it should be possible for new recipients to subscribe for Notifications.
  Formal Specification

$$\texttt{subscribe:ENT} \times \texttt{N} \rightarrow \texttt{ENT}$$

7.5 EMS System and State

Static Analysis

The overall Event Management System has a state Ψ, which consists of:

- Event-Notification Template State: A state ENTS, of an index-set ETIdx of defined or being defined Event-Notification Templates, ENT.
- Event-Notification Instances State: A state ENS, of an index-set EIdx of the Event-Notification Instances EN(e), that EMS currently executes. The state of an active Event-Notification template Φ is further undefined.

```
type
    Ψ, ENTS, ENS, Σ, ETIdx, EIdx
    Ψ = ENTS × ENS
    ENTS = ETIdx m→ ENT
    ENS = EIdx m→ Σ
    Σ = ETIdx × Φ
```

Dynamic Analysis

- An Event Management System `EMS` consists of one or more active Event-Notification Instances `EN(en)` running concurrently.

```
EMS:Unit → Unit
EMS() ≡ ‖ {EN(en)|en:ENIdx}
```

7.6 Event-Notification Instance

An Event-Notification Instance denotes one of the many active instances of an Event-Notification Template, which is being executed by the Event Management System.

Narrative

1. Each instance checks for the occurrence of an event.
2. If the response is a NOEVENT or FAULT, the instance skips and starts all over again.
3. If the response is an occurrence of an event with the results back, the instance carries out the notification.

Formal Specification:

```
                          / * Event Notification Process * /
EN(en):en:ENIdx → Unit
EN(en) ≡ let e:E = obs_event(en) in
           cases Event(e) of
             NOEVENT → skip,
             FAULT → skip,
             EVENT(res) → Notification(e,res)
           end
         end; EN(en)
         obs_event:ENIdx → E
```

Annotations:

- ENIdx is an index set of all active Event-Notification Instances EN(en). Each Event-Notification Instance is uniquely identified by a number en which belongs to the index set ENIdx.

7.7 Event Management Model

Below is a complete requirements model for the Event Management System, based on static and dynamic aspects analysed in the previous sections.

```
type      Ψ, ENTS, ENS, Σ, ETIdx, ENIdx, RIdx
          I, T, R, CH, M, V, RES, D, RIdx,
          E = I × ET × ER,
          ER = V → BOOL,
          I = User | SAS | External,
          N = R × NR × CH × M,
          R = Fixed | Dynamic | Subscribed
          NR = V → BOOL,
          ENT = E m~ N-set,
          ENTS = ETIdx m~ ENT
          ENS = ENIdx m~ Σ
          Σ = ETIdx × Φ

channel   {crcp[r]:M | r:RIdx}

value     ent:ENT
          subscribe:ENT × N → ENT

          / * Event Management Process * /
EMS:Unit  →  Unit
EMS()     ≡  ‖ {EN(en) | en:ENIdx}

          / * Event Notification Instance * /
EN(en):en:ENIdx  →  Unit
EN(en)           ≡  let e:E = obs_event(en) in
                    cases Event(e) of
                     NOEVENT → skip,
                     FAULT → skip,
                     EVENT(res) → Notification(e,res))
                    end
```

```
                                end; EN(en)

                                obs_event:ENIdx→E

                                /∗ Event ∗/
                  Event:E   →   RS
                  Event(e)  ≡   let(i,et,er) = e in
                                  cases invoked(i,t) of
                                   FALSE → NOEVENT,
                                   res:RS →
                                     if er(res) then EVENT(res)
                                     else NOEVENT end
                                  end
                                end

                                invoked:I×T → FALSE|RS

                                /∗ Notification ∗/
   Notification:E × RS    →   out {crcp[r]:M|r:RIdx} Unit
Notification(e,res)       ≡   let ns:N-set = ent(e) in
                          ‖   {let (r,nr,ch,m)=n in
                                 if nr(res) then output(ch,m) end
                               end |n ∈ ns}
                                end
```

Chapter 8

Solution Approaches

In this chapter we shall analyse four different solution approaches for an EMS solution. The chapter also attempts to clarify the differences among consecutive models and critically highlights the benefits and disadvantages. The chapter is organized as below:

- An informal description of Direct Event Management solution is given in Section 8.1.
- An informal description of Synchronous Event Management solution is given in Section 8.2.
- An informal description of Asynchronous Event Management solution is given in Section 8.3.
- An informal description of Asynchronous Event Management solution with the help of an external EM system, is given in Section 8.4.

In each of these sections, first the Solution concept is modeled as a diagram and informal description. This is followed by a formal specification of both static and dynamic aspects of the system.

8.1 Direct Event Management

8.1.1 Solution concept

In this solution, each Application area `A(a)` has its own *local* notification system to directly notify the recipients.
Figure 8.1 on page 66, shows a schematic diagram of a Simple Application System extended with Event Management functionality.

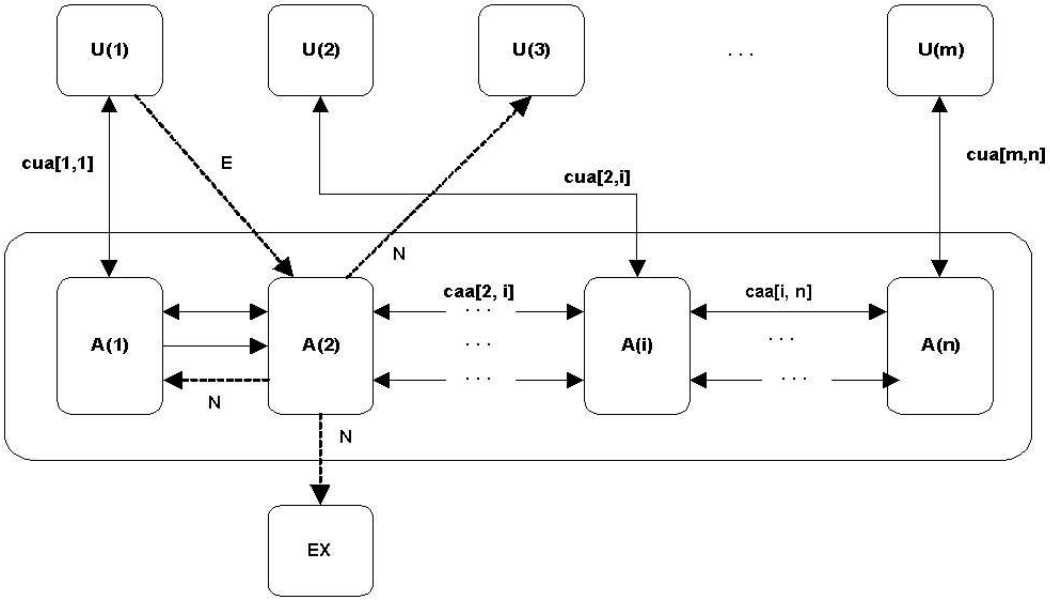


Figure 8.1: Simple Application with Direct Event Management

From the figure we observe that:
User `U(1)` does some activity in the Application area `A(2)` and which results in an event `E`. Then `A(2)` *directly* notifies the following recipients: user `U(3)` , another application area `A(1)` and an External System `EX`.

8.1.2 Extending SAS with Direct EMS

To extend the Simple Application System, with a direct EMS we need to extend the default behaviour of all those application areas `A(a)` , where we need a notification.

Narrative
The *italicised* line below marks the extension to the basic application area.

1. An Application area `A(a)` receives service-requests from Users `U(u)` and other Application areas`A(a')` .
2. The Application area executes the service-request as input.
3. The Application area then sends the result back to *service* requester, User *or* another Application area. Or choose to do nothing.

4. *The Application area notifies the recipients about the event, that is invocation of a service request by a user.*

   (a) `A(a)` identifies all the recipients who wants to be notified.

   (b) For each of the recipients , `A(a)` identifies the Notification Rules **NR**.

   (c) `A(a)` qualifies the recipients for notification based on their notification rule **NR.**

   (d) For the qualified recipients, **`A(a)`** identifies which notification needs to sent and on which channel. Then it sends the notification on the channel.

Formal Specification
Lines marked with numbers, shows the extension points.

```
A(a):a:AIdx  →  in, out {cua[u,a]|u:UIdx},
                {caa[a′,a] | a′:AIdx • a ≠ a′} Unit
      A(a)   ≡
             []  {
                 {let r:R = input(cua[u,a]) in
                  let v = execute(r) in
                    output(cua[u,a],v) ⊓ skip
                  end
                 end
                 ⊓
                 let r:R = input(caa[a′,a]) in
                  let v = execute(r) in
                    output(caa[a,a′],v) ⊓ skip
                  end
                 end}
                 EventNotification(e)                    (8.1)
                 |u:UIdx,a′:AIdx};A(a)

                 execute:R → V

EventNotification(e):e:E  →  Unit                        (8.2)
    EventNotification(e)  ≡  cases Event(e) of           (8.3)
                               NOEVENT → skip,           (8.4)
                              FAULT → skip,              (8.5)
                              EVENT(res) → Notification(e,res)(8.6)
                             end                         (8.7)

                             / ∗ Event ∗ /
```

```
Event:E  →  RS                                              (8.8)
Event(e)  ≡  let(i,t,er) = e in                             (8.9)
  cases invoked(i,t)of                                      (8.10)
   FALSE → NOEVENT,                                         (8.11)
   res:RS →                                                 (8.12)
     if er(res) then EVENT(res)                             (8.13)
     else NOEVENT end                                       (8.14)
  end                                                       (8.15)
end                                                         (8.16)

invoked:TRG → FALSE|RS                                      (8.17)

/ * Notification * /
Notification:E × RS  →  out {crcp[r]:M|r:RIdx} Unit         (8.18)
Notification(e,res)  ≡  let ns:N-set = ent(e)in             (8.19)
                    ||  {let (r,nr,ch,m)=nin                (8.20)
                     if nr(res) then output(ch,m) end       (8.21)
                    end |n ∈ ns}                            (8.22)
                    end                                     (8.23)
```

Annotations
`EventNotification(e)`, designates, the *local* notification system of the application area A(a).

8.1.3 Benefits and Limitations of Direct EMS

- **Benefit:** One of the advantages of this approach is that, it is easy and straight forward. This is suitable, when we have very specific and less number of Events to handle and do not foresee any major extension in future.
- **Limitation:** The biggest disadvantage of this approach is that, it disturbs the existing SAS system heavily. Because we have to modify each of those application areas in SAS, events of which we are interested in. It is very difficult to handle many Event-Notifications needs.

8.1.4 Model of Direct EMS

A complete model of the SAS extended with Direct Event Management is listed in Appendix E on page 125.

8.2 Synchronous Event Management

8.2.1 Solution concept

In this solution, each *Application area* notifies the occurrence of an event to a new Application area called Event Management Solution EMS. EMS in turn manages receiving events and notifying the relevant recipients. Figure 8.2 on page 69, shows a Simple Application System extended with Synchronous Event Management functionality.

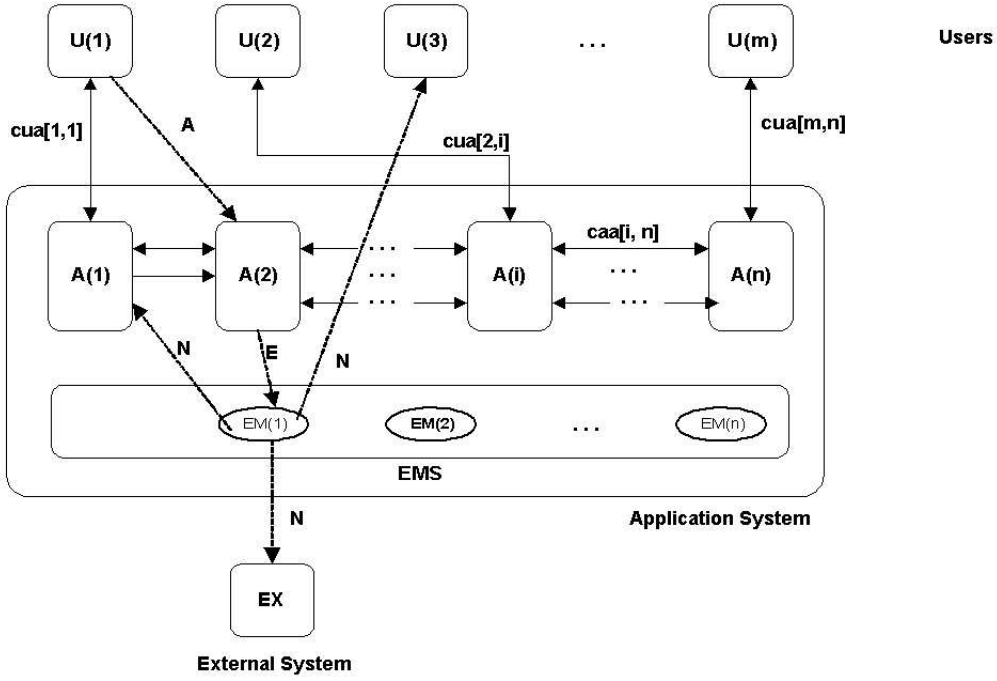


Figure 8.2: Simple Application with Synchronous Event Management

From the figure we observe that:
User `U(1)` does some activity `A` in the Application area `A(2)` . Then `A(2)` synchronously sends a Event message `E` to EMS. EMS evaluates the notification rules for each of the recipients and then *notifies* recipients:`N` **,** user `U(3)` , another application area `A(1)` and an External System `EX` .

8.2.2 Extending SAS with Synchronous EMS

To extend the Simple Application System, with a synchronous EMS we need to extend the default behaviour of all those application areas `A(a)` , where we need a notification. Further, we need to extend the SAS itself with a *new* application area called **EMS**.

- **Application Area Extension**
  Narrative

  The *italicised* line below marks the extension to the basic application area.

  1. An Application area `A(a)` receives service-requests from Users `U(u)` and other Application areas **`A(a')`**.

2. The Application area executes the service-request r .
3. The Application area then sends the result back to *service* requester, User *or* another Application area. Or choose to do nothing.
4. *The Application area notifies the Event Management application area about the occurrence of the event recipients about the event,*

Formal Specification

The numbered lines mark the extension points.

$$
\begin{array}{rcl}
\texttt{A(a):a:AIdx} & \rightarrow & \textbf{in},\textbf{out}\ \{\texttt{cua[u,a]}\,|\,\texttt{u:UIdx}\},\\
& & \{\texttt{caa[a'},\texttt{a]}\mid \texttt{a'}\texttt{:AIdx}\bullet \texttt{a}\neq\texttt{a'}\}\\
& & ,\textbf{out}\{\texttt{cae[a]}\}\textbf{Unit}\\
\texttt{A(a)} & \equiv & \\
& [] & \{\\
& & \{\textbf{let}\ \texttt{r:R} = \underline{\texttt{input}}\texttt{(cua[u,a])}\ \textbf{in}\\
& & \ \textbf{let}\ \texttt{v = execute(r)}\ \textbf{in}\\
& & \ \ \underline{\texttt{output}}\texttt{(cua[u,a],v)}\sqcap\ \textbf{skip}\\
& & \ \textbf{end}\\
& & \textbf{end}\\
& & \sqcap\\
& & \textbf{let}\ \texttt{r:R} = \underline{\texttt{input}}\texttt{(caa[a',a])}\ \textbf{in}\\
& & \ \textbf{let}\ \texttt{v = execute(r)}\ \textbf{in}\\
& & \ \ \underline{\texttt{output}}\texttt{(caa[a,a'],v)}\sqcap\ \textbf{skip}\\
& & \ \textbf{end}\\
& & \textbf{end}\}\\
& & \underline{\texttt{output}}\texttt{(cae[a],e)}\\
& & \texttt{|u:UIdx,a'}\texttt{:AIdx};\texttt{A(a)}\\
& & \\
& & \texttt{execute:R}\rightarrow\texttt{V}
\end{array}
\tag{8.24}
$$

Annotations
Here we give annotations for the extension parts only.

- output(cae[a],(obs_serv(r),res)), designates the notification by the application area A(a) to the Event Management application area about the occurrence of the event cae[a,a'] is the communication channel used.
- Comparing the application area behaviour in Synchronous EMS, with that of Direct EMS in Section 8.1.2, we notice that, this solution is more general, modular and extensible. It directly handles the disadvantages of Direct EMS.

- **New Synchronous EMS**
  EMS is the new application area, that will receive notification about occurrence of events from different application areas and then notify the subscribed users. Below a more precise description of the behaviour of EMS is given.

  Narrative

  1. Event Management Process `EMS` receives message from Application areas `A(a)` .
  2. `EMS` identifies all the recipients who have subscribed for the occurrence of an event.
  3. For each of the recipients, `A(a)` identifies the Notification Rules `NR` .
  4. `EMS` qualifies the recipients for notification based on their `NR` against the input value**.**
  5. For the qualified recipients, `A(a)` identifies what notification needs to be sent and on which channel. Then it sends the notification on the channel.

  Formal Specification

```
                          / * Synchronous Event Management Process * /
EMS():Unit   →   Unit
     EMS()   ≡
             []  {let e = input(cae[a]) in
                   cases Event(e) of
                     NOEVENT → skip,
                     FAULT → skip,
                     EVENT(res) → Notification(e,res))
                   end
                 end|a:AIdx}EMS()

                          / * Event * /
  Event:E    →   RS
  Event(e)   ≡   let (i,t,er) = e in
                   cases invoked(i,et) of
                     FALSE → NOEVENT,
                     res:RS →
                       if er(res) then EVENT(res)
                       else NOEVENT end
                   end
                 end

                 invoked:I×ET → FALSE|RS
```

```
                                  / * Notification * /
Notification:E × RS   →   out {crcp[r]:M|r:RIdx} Unit
Notification(e,res)   ≡   let ns:N-set = ent(e) in
                      ‖   {let (r,nr,ch,m) = n in
                            if nr(res) then output(ch,m) end
                           end |n ∈ ns}
                          end
```

Annotations

- Compared to Direct EMS in Section 8.1.2, we notice that the EMS application area behaves like the *local* notification system in Direct EMS.

8.2.3 Benefits and Limitations of Direct EMS

- **Benefit:** The main advantage of this approach is that, it is more modular and easily extensible. The EMS application area can receive event from the current and future application areas.
- **Disadvantage:** Despite of the above benefit, one still needs to modify each of the Application areas, to notify EMS about the occurrence of the event. If there are many events a customer is interested in, it means quite a big change in the existing standard solution SAS.

8.2.4 Model of Synchronous EMS

A complete model of the Application System extended with Synchronous Event Management is listed in Appendix F on page 129.

8.3 Asynchronous Event Management

8.3.1 Solution concept

In this solution, a new Application area called Event Management Solution EMS asynchronously polls and detects for the occurrence of the Event from the system state. EMS then processes the result and notifies the relevant recipients. Note that, the Application areas do not need to notify the occurrence of an event. Figure 8.3 on page 73, shows a Simple Application System extended with Asynchronous Event Management functionality.

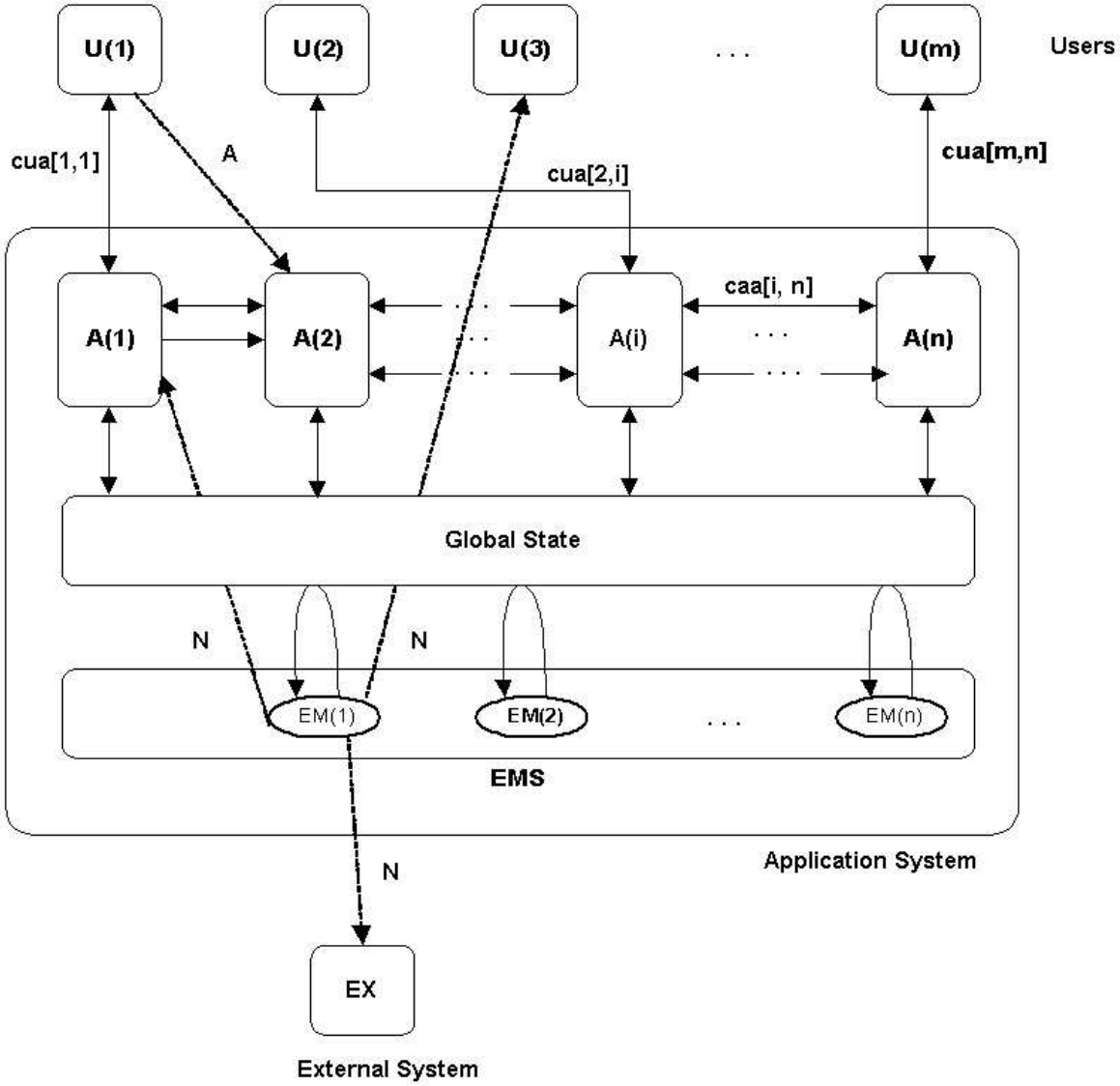


Figure 8.3: Simple Application with Asynchronous Event Management

From the figure we observe that:
User `U(1)` does some activity `A` in the Application area `A(2)` , that results in a change in the system state. Meanwhile, an instance of Event Monitor `EM(1)` polls the Global State for the occurrence of

the Event. EM(1) then *notifies* the recipients: user U(3) , another application area A(1) and an External System EX after evaluating the notification rules for each of the recipients.

8.3.2 Extending SAS with Asynchronous EMS

To extend the Simple Application System, with an asynchronous EMS we need to extend the SAS with a *new* application area called **EMS**.

New Asynchronous EMS
EMS is the new application area, that will poll different application areas for the occurrence of events and then notify the subscribed users. Below a more precise description of the behaviour of EMS is given.

Narrative

1. Event Management Process EMS consists of one or more Event Notification instances EN(en) running in parallel.
2. Each EN(en) checks if the Event Task et is invoked by Initiator I. The global state GS is observed to determine if the task is carried out.
3. If the response is a NOEVENT or FAULT, EN(en) skips and starts all over again.
4. If the response is an occurrence of an event with the results back, EN(en) carries out notification.
5. For each of the recipients, EN(en) identifies the notification rules NR .
6. EN(en) qualifies the recipients for notification based on their NR against the input value**.**
7. For the qualified recipients, EN(en) identifies the what notification needs to sent and on which channel. Then it sends the notification on the channel.
8. EN(en) starts all over again.

*Formal Specificatio*n

$$
\begin{array}{rcl}
 & & /* \text{ Asynchronous Event Management Process } */ \\
\texttt{EMS}:\mathbf{Unit} & \rightarrow & \mathbf{Unit} \\
\texttt{EMS()} & \equiv & \| \{\texttt{EN(en)} \mid \texttt{en:ENIdx}\} \\
 & & \\
 & & /* \text{ Event Notification Instance } */ \\
\texttt{EN(en)}:\texttt{en:ENIdx} & \rightarrow & \mathbf{Unit} \\
\texttt{EN(en)} & \equiv & \mathbf{let}\ \texttt{e:E = obs\_event(en)}\ \mathbf{in} \\
 & & \quad \mathbf{cases}\ \texttt{Event(e)}\ \mathbf{of} \\
 & & \quad\quad \texttt{NOEVENT} \rightarrow \mathbf{skip}, \\
 & & \quad\quad \texttt{FAULT} \rightarrow \mathbf{skip},
\end{array}
$$

```
                                  EVENT(v) → Notification(e,v))
                               end
                             end}EN(en)

                             obs_event:ENIdx→E

                             /∗ Event ∗/
                  Event:E  →  RS
                  Event(e) ≡  let (i,t,er) = e in
                               cases invoked(i,et) of
                                (false,v) → NOEVENT,
                                (true,v) →
                                 if er(v) then EVENT(v)
                                 else NOEVENT end
                               end
                             end

        invoked: I×ET  →  in cstg Unit ×BOOL × V
         invoked(i,et) ≡  let stg:Σ = input(cstg) in
                           let (b,v,i') = interpret(et)(stg) in
                            if b ∧ (i = i') then (true,v)
                            else (false,v) end
                           end
                         end

                         interpret:ET → Σ → BOOL×VAL×I

                   value
                         vstg:Σ
                GS:Unit  →  out cstg Unit
                   GS()  ≡  while true do
                             output(cstg,vstg) in
                            ....
                            end

                         /∗ Notification ∗/
Notification:E × V  →  out {crcp[r]:M|r:RIdx} Unit
Notification(e,v)   ≡  let ns:N-set = ent(e) in
```

```
  ||  {let (r,nr,ch,m)=n in
        if nr(res) then output(ch,m) end
       end |n ∈ ns}
      end
```

Annotations
Here we give annotations for the extension parts only.

- Comparing to Simple Application System, this solution, does not effect the behaviour of Application area at all.
- Compared to Direct EMS in 8.1.2, we notice that the EMS application area behaves like the *local* notification system in Direct EMS.
- Comparing with Synchronous EMS, we notice that, in this system, no direct message is sent to EMS at the point of event occurrence. Rather, EMS has the responsibility to *poll* for detecting if the Event has occurred.

8.3.3 Model of Asynchronous EMS

A complete model of the Application System extended with Asynchronous Event Management is listed in Appendix G on page 133.

8.4 External Asynchronous Event Management

8.4.1 Solution concept

This solution is similar to Asynchronous EMS except that, an external Event Management Solution EMS asynchronously polls the Simple Application via an interface **EMI,** to check for the occurrence of the Event from the system state. EMS then processes the result and notifies the relevant recipients. Figure 8.4 on page 77, shows a Simple Application System extended with Event Management functionality using an *external* Event Management Solution.

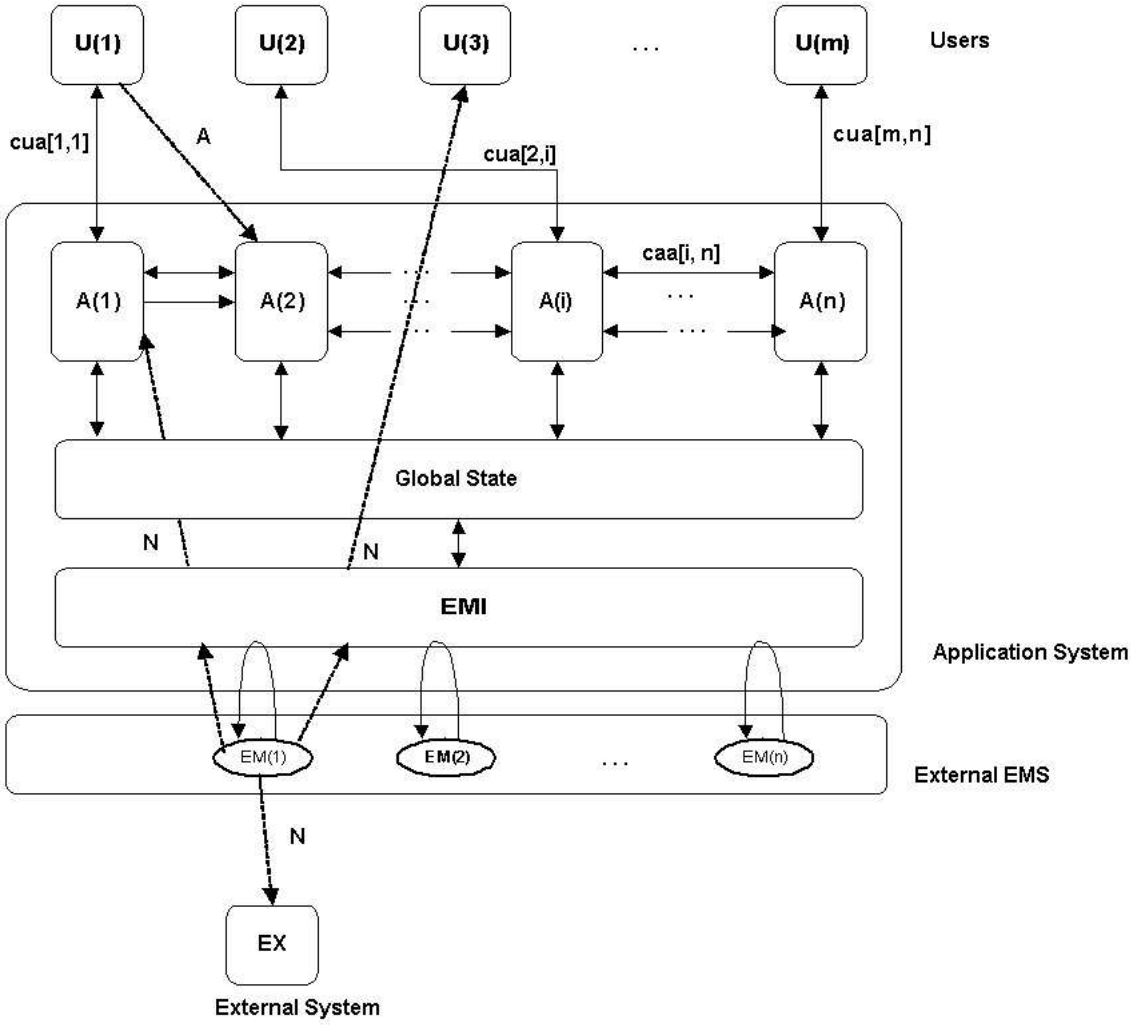


Figure 8.4: Simple Application with External Asynchronous Event Management

From the figure we observe that:
User `U(1)` does some activity `A` in the Application area `A(2)` , that results in a change in the system state. Meanwhile, an instance of Event Monitor `EM(1)` in the *external* Event Management Solution EMS polls the Global State via an Interface EMI**,** for the occurrence of the Event. `EM(1)` then *notifies* the recipients: user `U(3)` and another application area `A(1)` via `EMI` and an External System `EX` directly, after evaluating the business rules for each of the subscribers.

8.4.2 Extending SAS with external Asynchronous EMS

To extend the Simple Application System with an external asynchronous EMS, we need to have an external EMS and also extend SAS with an Event Management Interface **`EMI`** application area .

- **New External Asynchronous EMS**
  EMS is the new application area, that will pool the application area for the occurrence of events from different application areas and then notify the subscribed users. Below a more precise description of the behaviour of EMS is given.

  Narrative

  1. Event Management Process **`EMS`** consists of one or more Event Notification instances **`EN(en)`** running in parallel.
  2. **`EN(en)`** sends a request to Event Management Interface **`EMI`** in the Simple Application System to check for the occurrence of a specific Event.
  3. If the response is a NOEVENT or FAULT, `EN(en)` skips and starts all over again.
  4. If the response is an occurrence of an event with the results back, `EN(en)` carries out notification.
  5. For each of the recipients, `EN(en)` identifies the notification rules `NR` .
  6. `EN(en)` qualifies the recipients for notification based on their `NR` against the input value**.**
  7. For the qualified recipients, `EN(en)` identifies the what notification needs to sent and on which channel. Then it sends the notification on the channel.
  8. `EN(en)` starts all over again.

  *Formal Specificatio*n

```
                                  / * Event Management Process * /
        EMS:Unit   →   Unit
           EMS()   ≡   || {EN(en)|en:ENIdx}

                                  / * Event Notification Instance * /
EN(en):en:ENIdx   →   Unit
        EN(en)    ≡   let e:E = obs_event(en) in
                        cases output_ input(cemi[en],e) of
                          NOEVENT → skip,
                          FAULT → skip,
                          EVENT(v) → Notification(e,v))
                        end
                      end}EN(en)

                      obs_event:ENIdx→E
```

```
                                  / * Notification * /
Notification:E × RS  →   out {crcp[r]:M|r:RIdx} Unit
 Notification(e,v)   ≡   let ns:N-set = ent(e) in
                     ‖   {let (r,nr,ch,m)=n in
                           if nr(v) then output(ch,m) end
                          end |n ∈ ns}
                         end
```

Annotations
Here we give annotations for the extension parts only.

- `resp:RS=output_ input(cemi[e],obs_req(er,e))`, designates sending of a request to Event Management Interface `EMI` . This is followed by waiting for receipt of a response `resp` about occurrence of an event, from `EMI` .
- {`cemi[e]|e:EMIdx`}, designates the communication channel between EMS and EMI.

- **Event Management Interface**
  EMI is a new application area within SAS, that serves as the interface for the external EMS to communicate with other Application areas of SAS. Below a more precise description of the behaviour of **`EMI`** is given.

  Narrative

  1. Event Management Interface `EMI` inside the application waits for requests from Event Notification Instances `EN(en)`.
  2. `EMI` executes the request by executing appropriate service(business wrapper) to check for the occurrence of an Event. The global state `GS` is observed to determine if the task is carried out.
  3. `EMI` sends the response back or skips doing nothing.
  4. `EMI` starts all over again.

  *Formal Specificatio*n

```
                                  / * Event Management Interface * /
EMI:en:ENIdx  →   in, out {cemi[en]},
     EMI(en)  ≡   let (i,et,er):E = input(cemi[en]) in
                   cases invoked(i,et) of
                    (false,v) → NOEVENT,
                    (true,v) →
                     if er(v) then EVENT(v)
                     else NOEVENT end
                   end
                  end
```

```
invoked: I×ET → in cstg Unit ×BOOL × V
 invoked(i,et) ≡ let stg:Σ = input(cstg) in
                  let (b,v,i')=interpret(et)(stg)in
                   if b∧(i = i') then (true,v)
                   else (false,v) end
                  end
                 end

                 interpret:ET → Σ → BOOL×VAL×I

value
                 vstg:Σ
GS:Unit → out cstg Unit
 GS() ≡ while true do
         output(cstg,vstg) in
```

8.4.3 Model of External Asynchronous EMS

A complete model of the Application System extended with External Asynchronous Event Management is listed in Appendix H on page 137.

Chapter 9

Design of Event Management Language

In previous chapters, we have carried out Domain Analysis, Requirements Analysis and Different Solution Approaches for the proposed Event Management System (EMS). In this Chapter, we shall use the domain, requirements and the solution understanding, to *design* an Event Management Language (EML). EML is intended to be specific for the domain of EMS. It will embody the domain and requirements concepts in its constructs. It is our intention that EML would serve as a tool for the CSC *Requirements Engineer* to both elicit and specify customer specific requirements, that could be met by the proposed EMS solution.
EML language requirements and design goals are first described in Section 9.1. In Section 9.2, EMS Use Cases B are analysed to establish the syntax of EML. A model of EML syntax is then presented in Section 9.2.5.

9.1 Language Analysis

9.1.1 Language Requirements

Based on the Domain and Requirements Analysis of Event Management System, it should be possible, at the language level to:

- Express an Event
- Express one or more Notifications
- Express association of an Event to Notifications

9.1.2 Elements of Design

Below are some of the design consideration we have considered to design EML:

Users

The main user of this language is typically a *Requirements Engineers* at the various Customer Solution Centers (CSCs). They are expected to use EML to specify the customer specific requirements. Further, as a secondary user, Application Developers at various CSCs would *interpret* the specification written in EML, to configure the *standard solution* into a customer-specific solution.

Language Paradigm and Level

Since the main use of the language is to *specify* requirements by a *Requirements Engineer,* we decide EML to be *descriptive* and *high-level* in nature like "make" or "HTML". The language should be able to capture requirements that could be satisfied by the *standard* Event Management Solution (EMS). Further, if the *Requirements Engineer* is not able to express some requirements using EML, it could mean:

- That there is a need to build a custom component by a CSC Solution Developer, to meet those customer-specific requirements that could not be specified in EML. and hence could not be satisfied by satisfied by *standard* EMS.
- OR, it could serve as input for the EML language enhancement.

9.2 EML Syntax

In this section, an informal and formal description of EML syntax is given. We have developed the syntax iterating between the following two activities:

- Informally analysing and expressing the EMS Use Cases in a declarative manner. Section 9.2.1 presents an Example of a Use Case and its specification in EML.
- Analysing the declarative specification and formally specifying the abstract syntax in RSL. The is elaborated in Sections 9.2.2, 9.2.3 and 9.2.4.

9.2.1 Example EML Specification

Figure 10.1 shows, an example "Create new item"workflow scenario. (Based on the Use-case B.6 on page 117)

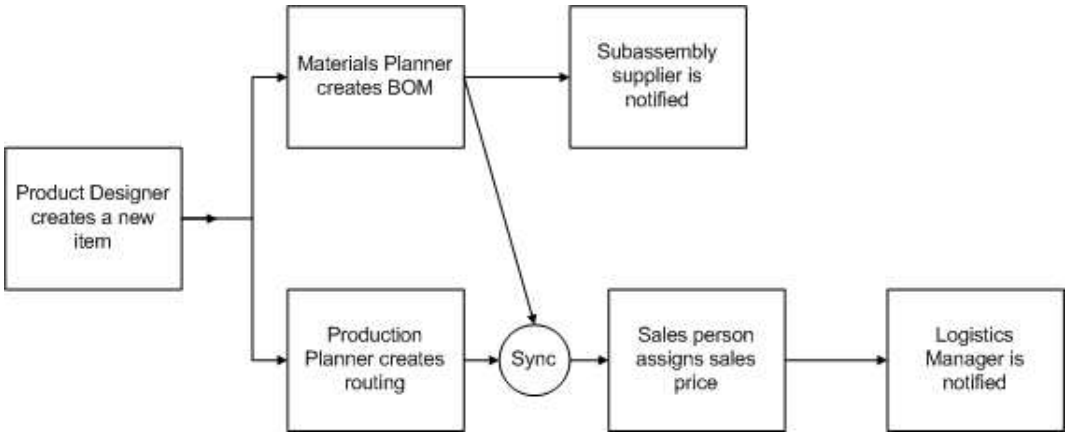


Figure 9.1: Workflow to create a new item in SAS

It involves the following steps:

1. The Product Designer creates a new item with its technical details only. The Item does not include information about BOM, cost, routing, and sales price.

2. The Materials Planner needs to be notified by Email to create BOM structure for the item. At the same time, the Production Planner would like to be informed by Email to create routing information.

3. After the Materials Planner creates BOM structure for the item and calculates unit costs for the new item, the supplier needs to be informed about the subcomponent details.

4. When the Production Planner has finished creating routing and the Materials Planner has created BOM structure for the item, Salesperson would like to be informed. So that the Saleperson can assign sales price for the item.

5. Once all the details are available for the item, the Logistics Manager would like to know which items to plan for production.

EML SPECIFICATION

```
 -- EventNotify
Event    "Production Designer SAS User Id"   "Create a new item."
         "(Item BOM Structure = Empty) AND (Routing = Empty) AND (Salesprice = 0.0)"
Notify   "Material Planner Email Id"
           Task("Create BOM structure for item.","Link to Item details")
Notify   "Production Planner Email Id"
           Task("Create routing for Item.","Link to Item details")
-- EventNotify
Event    "Material Planner SAS User Id"
          "Create BOM structure for item and calculate unit price."
Notify   "Supplier Email Id"    Info("New Item.","Link to Item and BOM details")
-- EventNotify
Event    "Production Planner SAS User Id"
          "Create routing and calculate capacity requirements for Item."
AND
Event    "Material Planner SAS User Id"
          "Create BOM structure for item and calculate unit price."
Notify   "Salesperson Email Id"
          Info("Item is ready for production and needs sales price.",
               "Link to Item details")
-- EventNotify
Event    "Salesperson SAS User Id"
          "Enter sales price for the item based on calculated cost."
Notify   "Logistics Manager Email Id"
          Info("New Item is created.","Link to Item and BOM details")
```

9.2.2 EML Specification

An EML Specification consists of a set of Event-Notification (EN) pairs.

$$\texttt{EML\_Spec} \;=\; (\texttt{E} \times \texttt{N})\text{-}\mathbf{set}$$

Annotations

- Within a Event-Notification `EN`, a notification `N` always follows an Event `E`.
  For Example:
  – *EventNotify*

```
Event   "Material Planner SAS User Id"
          "Create BOM structure for item and calculate unit price."
Notify  "Supplier Email Id"    Info("New Item.","Link to Item and BOM details")
```

- The sequence of Event-Notification EN within an EML-Specification is immaterial. The choice of **set** in the above specification, makes it explicit.

9.2.3 Event

- **Event**

  An Event E can be constructed by a SIMPLE-Event SE, an AND-Event AE or an OR-Event OE .

  - A SIMPLE-Event SE is constructed from an Initiator I, Event Task ET and an Event-rule ER. These are further elaoccurredborated below.
  - An AND-Event AE is constructed from two events E, a left event and a right event. Semantically, it means an event has occurred only when the two events have occurred.
  - An OR-Event OE is constructed from two events E, a left event and a right event. Semantically, it means an event has occurred when one of the two events has occurred.

  Formal Specification

```
E  ==  SE|AE|OE ...

SE  =  mkSE(i:I, et:ET, er:ER)
AE  =  mkAE(le:E, re:E)
OE  =  mkOE(le:E, re:E)
```

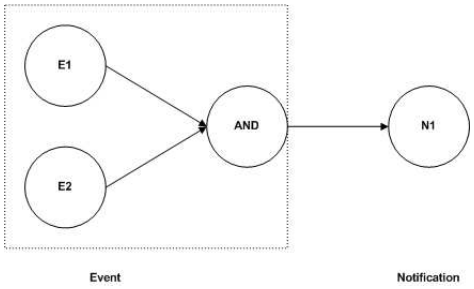
  Annotations
  mkSE(..),mkAE(..),mkOE(..) designate, the constructor functions that generate Simple Event, AND Event and OR Events respectively.

  For Example:

  1. Example of User Event, SAS Event and External Event

```
-- User Event
Event   "Salesperson SAS User Id"
          "Update Sales Order with price and discounts."
          "(Sales Order Status = Released) AND (Profit < 10%)"
-- SAS Event
Event   "SAS Application Id"   "MRP calculation run."
          "MRP status = Error "
-- External Event
Event   "Supplier Email Id"   "Replies by Email about Delivery."
          "Response = Delivery OK"
```

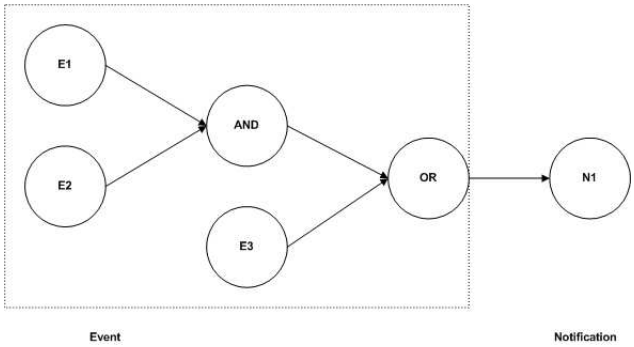
2. Figure 9.2, shows an EML-specification consisting of an AND Event `E1 AND E2` and the corresponding Notification `N`.




```
-- EventNotify
Event    "Production Planner SAS User Id"
          "Create routing and calculate capacity requirements for Item."
AND
Event    "Material Planner SAS User Id"
          "Create BOM structure for item and calculate unit price."
Notify   "Salesperson Email Id"
          Info("Item is ready for production and needs sales price.",
               "Link to Item details")
```

Figure 9.2: Example EML-Specification 3

3. Figure 9.3, shows an EML-specification consisting of an OR Event `(E1 AND E2) OR E3` and the corresponding Notification `N`. We assume AND is more associative than OR. This example illustrates, the power of recursive specification of AND and OR Events above.



$$\begin{array}{ll} \textbf{Event} & < parameter1 > \\ \texttt{AND} & \\ \textbf{Event} & < parameter2 > \\ \texttt{OR} & \\ \textbf{Event} & < parameter3 > \\ & \\ \textbf{Notify} & < parameter1 > \end{array}$$

Figure 9.3: Example EML-Specification 4

- **Initiator**

  An Initiator `I` of an Event `E` can be a SAS User `UI`, SAS Application area `SI` or an External user `EI`.These initiators can be constructed using unique ids: `UId`, `SId` and `EId` respectively.

```
I  ==  UI|SI|EI ...

UI   =   mkUI(UId)
SI   =   mkSI(SId)
EI   =   mkEI(EId)
```

  Examples:

  - A SAS user could be a Sales person with user id `RV` . The initiator can be constructed as `mkUI("RV")`.
  - An external user could be a Customer contact with email id `Cust@canon.com` . The initiator can be constructed as `mkEI("Cust@canon.com")`.

- **Event Task**

  An Event Task `ET` can be constructed by a SAS Task `ST` or an External Task `ExT`.

  Formal Specification

```
ET  ==  ST|ExT ...
ExT  =  mkExT(ExTId)
ST  ==  mkST(TId)
```

For Example:
In the below specification, "Update Sales Order with price and discounts." is a SAS Task.
– EventNotify

Event "Salesperson SAS User Id" "Update Sales Order with price and discounts."
"(Sales Order Status = Released) AND (Profit < 10%)"

- **Event Rule**

  An Event Rule ER can be constructed from:

  – a constant Const
  – an attribute A_n of a relation R_n in a database: R_nA_n
  – Equality constructor Eq
  – Logical equality constructor Eq
  – Logical AND constructor AND
  – Logical OR constructor OR
  – Logical LESS THAN constructor LT
  – Logical GREATER THAN constructor GT

  Formal Specification

```
   ER  ==  Const | RnAn| Eq | And | Or | Lt | Gt ...
Const   =  mkConst(v:VAL)
 RnAn   =  mkRnAn(rn:Rn, an:An)
   Eq   =  mkEq(ler:ER, rer:ER)
  And   =  mkAND(ler:ER, rer:ER)
   Or   =  mkOr(ler:ER, rer:ER)
   Lt   =  mkLt(ler:ER, v:VAL)
   Gt   =  mkGt(ler:ER, v:VAL)
```

  Examples:

  – In the below specification, "(Sales Order Status = Released) AND (Profit < 10%)" is an Event Rule.

Event "Salesperson SAS User Id" "Update Sales Order with price and discounts."
"(Sales Order Status = Released) AND (Profit < 10%)"

  – This Event rule: "(Sales Order Status = Released) AND (Profit < 10%)" translates to the abstract syntax

```
mkAND(mkEq(mkRnAn(Sales Order, Status), mkConst("Released"))
      ,mkLt(mkRnAn(Sales Order,Profit),mkConst("10")))
```

9.2.4 Notification

- **Notification**

  A Notification `N` can be constructed by an Information-Notification `IN` or a Task-Notification `TN`.

  – An Information-Notification `IN` is constructed from a Recipient `R`, a channel `CH` and information `Info` as message.
  – An Task-Notification `TN` is constructed from a Recipient `R`, a channel `CH` and Notification-task `NT` as message.

  Formal Specification

  ```
  N  ==  IN|TN...

  IN  =  mkIN(to:R, ch:CH, i:Info)
  TN  =  mkTN(to:R, ch:CH, nt:NT)
  ```

  Annotations

  – `mkIN(..),mkTN(..)` designate, the constructor functions that generate Information-Notification `IN` and Task-Notification `TN` respectively.

  Examples

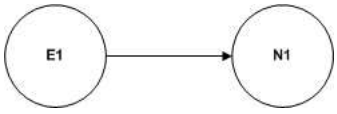
  1. Example of Task Notification

     Notify `"Material Planner Email Id"`
     `Task("Create BOM structure for item.","Link to Item details")`
     Notify `"Production Planner Email Id"`
     `Task("Create routing for Item.","Link to Item details")`

  2. Example of Information Notification

     Notify `"Salesperson Email Id"`
     `Info("Item is ready for production and needs sales price.",`
     `"Link to Item details")`
     Notify `"Logistics Manager Email Id"`
     `Info("New Item is created.","Link to Item and BOM details")`

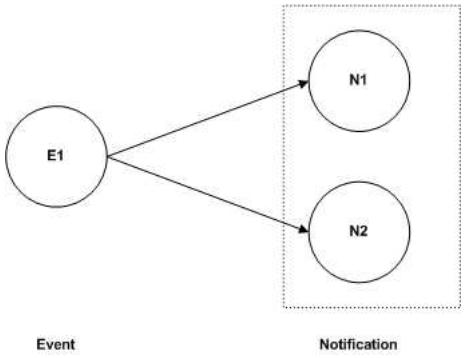
  3. Figure 9.4, shows an EML-specification with basic Notification `N1` as a result of an Event `E1`.



Event $< parameter1 >$
Notify $< parameter1 >$

Figure 9.4: Example EML-Specification 5

4. Figure 9.5, shows an EML-specification consisting of more than one Notifications `N1, N2` in response to an Event `E1`. It is assumed that all the Notifications `N1, N2...` takes place in parallel, independent of each other.




```
-- EventNotify
Event    "Production Designer SAS User Id"   "Create a new item."
          "(Item BOM Structure = Empty) AND (Routing = Empty) AND (Sales price = 0.0)"
Notify   "Material Planner Email Id"
           Task("Create BOM structure for item.","Link to Item details")
Notify   "Production Planner Email Id"
           Task("Create routing for Item.","Link to Item details")
```

Figure 9.5: Example EML-Specification 6

`R, CH, Info, NT` are further analysed and described below.

- **Recipient**

  A Recipient `R` of a Notification `N` can be a SAS User `UR`, SAS Application area `SR` or an External user `EXR`. These recipients can be constructed using unique ids: `UId`, `SId` and `EId` respectively.

```
R  ==  UR|SR|EXR ...

UR   =   mkUR(UId)
SR   =   mkSR(SId)
EXR  =   mkEXR(ExId)
```

  Examples:

  - A SAS user could be a Sales person with user id `RV` . The initiator can be constructed as `mkUR("RV")`.
  - An external user could be a Customer contact with email id `Cust@canon.com` . The initiator can be constructed as `mkEXR("Cust@canon.com")`.

- **Channel**

  A Channel `CH` of a Notification `N` can be Email, Phone, Fax,SMS.

```
CH  ==  Email|Phone|Fax|Fax|SMS ...
```

- **Notification Info**

  A Notification Information `Info` consists of static information.

  ```
  Info = mkInfo(d:Description)
  ```

- **Notification Task**

  A Notification Task `NT` can be constructed by:

  - Manual Task `MT`
  - SAS Task `ST`
  - External Task `ExT`
  - Automatic SAS Task `AST`

  Formal Specification

  ```
  NT  == MT|ExT|AST|ST ...
  MT  =  mkMT(d:Description)
  ExT =  mkExT(ExTId)
  AST =  mkAST(ExTId)
  ST  == mkST(TId)
  ```

9.2.5 EML Syntax Specification

Below is a complete specification for the EML syntax.

```
EML_Spec  =  (E × N)-set

      E  ==  SE|AE|OE ...

     SE  =  mkSE(i:I, tsk:ET, er:ER)
     AE  =  mkAE(le:E, re:E)
     OE  =  mkOE(le:E, re:E)

      I  ==  UI|SI|EI ...
     UI  =  mkUI(UId)
     SI  =  mkSI(SId)
     EI  =  mkEI(EId)

     ER  ==  Const | RnAn| Eq | And | Or | Lt | Gt ...
  Const  =  mkConst(v:VAL)
   RnAn  =  mkRnAn(rn:Rn, an:An)
```

```
  Eq  =  mkEq(ler:ER, rer:ER)
 And  =  mkAND(ler:ER, rer:ER)
  Or  =  mkOr(ler:ER, rer:ER)
  Lt  =  mkLt(ler:ER, v:VAL)
  Gt  =  mkGt(ler:ER, v:VAL)

  ET  ==  ST|ExT ...
 ExT  =  mkExT(ExTId)
  ST  ==  mkST(TId)

   N  ==  IN|TN...

  IN  =  mkIN(to:R, ch:CH, i:Info)
  TN  =  mkTN(to:R, ch:CH, nt:NT)

   R  ==  UR|SR|EXR ...

  UR  =  mkUR(UId)
  SR  =  mkSR(SId)
 EXR  =  mkEXR(ExId)

  CH  ==  Email|Phone|Fax|Fax|SMS ...

Info  =  mkInfo(d:Description)


  NT  ==  MT|ExT|AST|ST ...
  MT  =  mkMT(d:Description)
 ExT  =  mkExT(ExTId)
 AST  =  mkAST(ExTId)
  ST  ==  mkST(TId)
```

9.3 EML Semantics

We will sketch the essentials of the semantics by focusing on SIMPLE-Event `se:SE` and Notification `N` in our treatment of EML-Spec. The semantic description also assumes, the *solution approach* is an Event Management Solution based on an External EMS system as explained in Section 8.4.

9.3.1 Semantic of EML Specification

Given a EML-Specification `EML_Spec`, an environment `ENV` and a state of SAS Σ, a EML-Specification could be interpreted by the meaning function$\mathcal{M}_{\texttt{EML_Spec}}$ as described below:

- An Event `e` is detected ..
- If the Event has occurred, then the associated Notification `n` is carried out. This is followed by repetition of the interpretation of the *remaining* Event-Notification pairs.
- If the Event has not occurred, the interpretation step is repeated until the event has occurred.

Formal Specification

$$
\begin{array}{rcl}
\mathcal{M}_{\texttt{EML\_Spec}} & : & \texttt{EML\_Spec} \rightarrow \texttt{ENV} \rightarrow \Sigma \rightarrow \Sigma \\
\mathcal{M}_{\texttt{EML\_Spec}}\,\texttt{(ems)}\,(\rho)\,(\sigma) & \equiv & \textbf{if } \texttt{ems} = \texttt{\{\}} \textbf{ then skip} \\
 & & \textbf{else} \\
 & & \quad \textbf{let } \texttt{(e,n):(E}\times\texttt{N)} \bullet \texttt{(e,n)} \in \texttt{ems} \textbf{ in} \\
 & & \quad\quad \textbf{let } \texttt{(b,v)} = \mathcal{M}_{\texttt{Event}}\,\texttt{(e)}\,(\rho)\,(\sigma) \textbf{ in} \\
 & & \quad\quad\quad \textbf{if } \texttt{b} \textbf{ then} \\
 & & \quad\quad\quad\quad \mathcal{M}_{\texttt{Notification}}\,\texttt{(n,v)}\,;\, \mathcal{M}_{\texttt{EML\_Spec}}\,\texttt{(ems}\backslash\texttt{\{(e,n)\})}\,(\rho)\,(\sigma) \\
 & & \quad\quad\quad \textbf{else} \\
 & & \quad\quad\quad\quad \mathcal{M}_{\texttt{EML\_Spec}}\,\texttt{(ems)}\,(\rho)\,(\sigma) \\
 & & \quad\quad\quad \textbf{end} \\
 & & \quad\quad \textbf{end} \\
 & & \quad \textbf{end} \\
 & & \textbf{end}
\end{array}
$$

Annotations

- If the EML-Specification `ems` is empty, the whole interpretation process is skipped.
- The meaning functions $\mathcal{M}_{\texttt{Event}}\,\texttt{(e)}\,(\rho)\,(\sigma)$ and $\mathcal{M}_{\texttt{Notification}}\,\texttt{(n,v)}$ designates, the semantics of the Event `e` and Notification `n` respectively. These are further described below.

9.3.2 Semantic of Event

Given an Event `E`, an environment `ENV` and a state of SAS Σ, an Event could be interpreted by the meaning function $\mathcal{M}_{\texttt{Event}}$, which states that an Event has occurred when:

- An Event Task `et` has been carried out and

- The Event Initiator is same as the Event-Task actor.
- Event Rule `er` on the Task output `v` is satisfied.

Formal Specification

$$
\begin{array}{rcl}
\mathcal{M}_{\text{Event}} & : & \texttt{E} \rightarrow \texttt{ENV} \rightarrow \Sigma \rightarrow \texttt{BOOL} \times \texttt{VAL} \\
\mathcal{M}_{\text{Event}}\,\texttt{(e)}\,(\rho)\,(\sigma) & \equiv & \textbf{let}\ \texttt{mkSE(i:I, et:ET, er:ER) = e}\ \textbf{in} \\
 & & \quad \textbf{let}\ \texttt{(b,v,i')} = \mathcal{M}_{\text{EventTask}}\,\texttt{(et)}\,(\rho)\,(\sigma)\ \textbf{then} \\
 & & \quad\quad \textbf{if}\ \texttt{b} \wedge \texttt{(i = i')} \wedge \mathcal{M}_{\text{EventRule}}\,\texttt{(er)(v)}\ \textbf{then}\ \texttt{(true,v)} \\
 & & \quad\quad \textbf{else}\ \texttt{(false,v)}\ \textbf{end} \\
 & & \quad \textbf{end} \\
 & & \textbf{end} \\
 & & \\
\mathcal{M}_{\text{EventTask}} & : & \texttt{ET} \rightarrow \texttt{ENV} \rightarrow \Sigma \rightarrow \texttt{BOOL} \times \texttt{VAL} \times \texttt{I} \\
 & & \\
\mathcal{M}_{\text{EventRule}} & : & \texttt{ER} \rightarrow \texttt{VAL} \rightarrow \texttt{BOOL} \\
\mathcal{M}_{\text{EventRule}}\,\texttt{(er)(v)} & \equiv & \textbf{cases}\ \texttt{er}\ \textbf{of} \\
 & & \quad \texttt{mkConst(v)} \rightarrow \texttt{v} \\
 & & \quad \texttt{mkR}_n\texttt{A}_n\texttt{(r}_n\texttt{:R}_n\texttt{, a}_n\texttt{:A}_n\texttt{)} \rightarrow \texttt{obsR}_n\texttt{A}_n\texttt{(r}_n\texttt{,a}_n\texttt{,v)} \\
 & & \quad \texttt{mkEq(ler:ER,rer:ER)} \rightarrow \\
 & & \quad\quad\quad (\mathcal{M}_{\text{EventRule}}\,\texttt{(ler)(v)} = \mathcal{M}_{\text{EventRule}}\,\texttt{(rer)(v)}) \\
 & & \quad \ldots \\
 & & \textbf{end} \\
 & & \\
\texttt{obsR}_n\texttt{A}_n & : & \texttt{R}_n \times \texttt{A}_n \times \texttt{V} \rightarrow \texttt{V}
\end{array}
$$

Annotations

- The meaning function $\mathcal{M}_{\text{EventTask}} : \texttt{ET} \rightarrow \texttt{ENV} \rightarrow \Sigma \rightarrow \texttt{BOOL} \times \texttt{VAL} \times \texttt{I}$ designates, the occurrence of an Event-Task `et` . The output of the function are:
  - If the Task has occurred (`true/fasle`),
  - The Task output `VAL` and
  - The Task Initiator `I`
- The meaning function $\mathcal{M}_{\text{EventRule}} : \texttt{ER} \rightarrow \texttt{VAL} \rightarrow \texttt{BOOL}$ designates, the evaluation of the Event-Rule `ER` against the Task-Output `V`. The output of the function is `true/false`.

9.3.3 Semantic of Notification

Given a Notification `N` and an Event-Task output `V`, a Notification could be interpreted by the meaning function $\mathcal{M}_{\text{Notification}}$. A Notification is considered to have occurred when:

- The Notification rule NR is satisfied, and
- A message Info or Task is sent to the recipient R.

Formal Specification

$$
\begin{array}{rcl}
\mathcal{M}_{\text{Notification}} & : & \texttt{N} \times \texttt{V} \rightarrow \textbf{Unit} \\
\mathcal{M}_{\text{Notification}}\,\texttt{(n,v)} & \equiv & \textbf{cases}\ \texttt{n}\ \textbf{of} \\
 & & \quad \texttt{mkIN(to:R,nr:NR,i:Info)} \rightarrow \\
 & & \quad\quad \mathcal{M}_{\text{NotificationtRule}}\,\texttt{(nr,v)} \\
 & & \quad\quad \wedge\ \underline{\texttt{output}}\,(\mathcal{M}_{\text{Recipient}}\,\texttt{(to)},\ \mathcal{M}_{\text{Info}}\,\texttt{(i,v)}) \\
 & & \quad \texttt{mkTN(to:R,nr:NR,nt:NT)} \rightarrow \\
 & & \quad \mathcal{M}_{\text{NotificationtRule}}\,\texttt{(nr,v)} \\
 & & \quad \wedge\ \underline{\texttt{output}}\,(\mathcal{M}_{\text{Recipient}}\,\texttt{(to)},\ \mathcal{M}_{\text{NotificationTask}}\,\texttt{(nt,v)}) \\
 & & \textbf{end}
\end{array}
$$

$$
\mathcal{M}_{\text{NotificationRule}} \ : \ \texttt{NR}\times\texttt{V} \rightarrow \texttt{BOOL}
$$

$$
\begin{array}{rcl}
\mathcal{M}_{\text{Recipient}} & : & \texttt{R} \rightarrow \texttt{CH} \\
\mathcal{M}_{\text{Recipient}}\,\texttt{(r)} & \equiv & \textbf{cases}\ \texttt{r}\ \textbf{of} \\
 & & \quad \texttt{mkUR(UId)} \rightarrow \texttt{obs\_channel(UId)} \\
 & & \quad \texttt{mkSR(SId)} \rightarrow \texttt{obs\_channel(SId)} \\
 & & \quad \texttt{mkExR(ExId)} \rightarrow \texttt{obs\_channel(ExId)} \\
 & & \quad \ldots \\
 & & \textbf{end}
\end{array}
$$

$$
\texttt{obs\_channel} \ : \ \texttt{(UId|SID|ExId)} \rightarrow \texttt{CH}
$$

$$
\mathcal{M}_{\text{Info}} \ : \ \texttt{Info}\times\texttt{V} \rightarrow \texttt{Info}
$$

$$
\begin{array}{rcl}
\mathcal{M}_{\text{NotificationTask}} & : & \texttt{NT}\times\texttt{V} \rightarrow \texttt{TaskId}\times\texttt{V} \\
\mathcal{M}_{\text{NotificationTask}}\,\texttt{(nt,v)} & \equiv & \textbf{cases}\ \texttt{nt}\ \textbf{of} \\
 & & \quad \texttt{mkMT(MTId)} \rightarrow \texttt{(MTId,v)} \\
 & & \quad \texttt{mkST(STId)} \rightarrow \texttt{(STId,v)} \\
 & & \quad \ldots \\
 & & \textbf{end}
\end{array}
$$

Annotations

- The meaning function $\mathcal{M}_{\text{NotificationRule}} : \texttt{NR}\times\texttt{V} \rightarrow \texttt{BOOL}$ designates, the condition that must be satisfied for sending a message(Info/Task) to a recipient.

- The meaning function $\mathcal{M}_{\text{Recipient}}$: R → CH designates, the channel CH of the recipient R, to which the message is to be sent.
- The meaning function $\mathcal{M}_{\text{Info}}$: Info×V → Info designates, generation of Information Info that needs to be sent to the recipient.
- The meaning function $\mathcal{M}_{\text{NotificationTask}}$: NT×V → TaskId×V designates, generation of Task TaskId and its input V, that needs to be notified to the recipient.

9.3.4 EML Semantics Specification

$$
\begin{array}{rcl}
\mathcal{M}_{\text{EML\_Spec}} & : & \text{EML\_Spec} \to \text{ENV} \to \Sigma \to \Sigma \\
\mathcal{M}_{\text{EML\_Spec}}(\text{ems})(\rho)(\sigma) & \equiv & \textbf{if } \text{ems} = \{\} \textbf{ then skip} \\
 & & \textbf{else} \\
 & & \quad \textbf{let } (\text{e},\text{n}) : (\text{E}\times\text{N}) \bullet (\text{e},\text{n}) \in \text{ems} \textbf{ in} \\
 & & \quad\quad \textbf{let } (\text{b},\text{v}) = \mathcal{M}_{\text{Event}}(\text{e})(\rho)(\sigma) \textbf{ in} \\
 & & \quad\quad\quad \textbf{if } \text{b} \textbf{ then} \\
 & & \quad\quad\quad\quad \mathcal{M}_{\text{Notification}}(\text{n},\text{v}) ; \mathcal{M}_{\text{EML\_Spec}}(\text{ems} \setminus \{(\text{e},\text{n})\})(\rho)(\sigma) \\
 & & \quad\quad\quad \textbf{else} \\
 & & \quad\quad\quad\quad \mathcal{M}_{\text{EML\_Spec}}(\text{ems})(\rho)(\sigma) \\
 & & \quad\quad\quad \textbf{end} \\
 & & \quad\quad \textbf{end} \\
 & & \quad \textbf{end} \\
 & & \textbf{end}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{M}_{\text{Event}} & : & \text{E} \to \text{ENV} \to \Sigma \to \text{BOOL}\times\text{VAL} \\
\mathcal{M}_{\text{Event}}(\text{e})(\rho)(\sigma) & \equiv & \textbf{let } \text{mkSE}(\text{i:I}, \text{et:ET}, \text{er:ER}) = \text{e} \textbf{ in} \\
 & & \quad \textbf{let } (\text{b},\text{v},\text{i}') = \mathcal{M}_{\text{EventTask}}(\text{et})(\rho)(\sigma) \textbf{ then} \\
 & & \quad\quad \textbf{if } \text{b} \wedge (\text{i} = \text{i}') \wedge \mathcal{M}_{\text{EventRule}}(\text{er})(\text{v}) \textbf{ then } (\text{true},\text{v}) \\
 & & \quad\quad \textbf{else } (\text{false},\text{v}) \textbf{ end} \\
 & & \quad \textbf{end} \\
 & & \textbf{end}
\end{array}
$$

$$
\mathcal{M}_{\text{EventTask}} : \text{ET} \to \text{ENV} \to \Sigma \to \text{BOOL}\times\text{VAL}\times\text{I}
$$

$$
\begin{array}{rcl}
\mathcal{M}_{\text{EventRule}} & : & \text{ER} \to \text{VAL} \to \text{BOOL} \\
\mathcal{M}_{\text{EventRule}}(\text{er})(\text{v}) & \equiv & \textbf{cases } \text{er} \textbf{ of} \\
 & & \quad \text{mkConst}(\text{v}) \to \text{v} \\
 & & \quad \text{mkR}_\text{n}\text{A}_\text{n}(\text{r}_\text{n}:\text{R}_\text{n}, \text{a}_\text{n}:\text{A}_\text{n}) \to \text{obsR}_\text{n}\text{A}_\text{n}(\text{r}_\text{n},\text{a}_\text{n},\text{v}) \\
 & & \quad \text{mkEq}(\text{ler:ER},\text{rer:ER}) \to \\
 & & \quad\quad (\mathcal{M}_{\text{EventRule}}(\text{ler})(\text{v}) = \mathcal{M}_{\text{EventRule}}(\text{rer})(\text{v}))
\end{array}
$$

```
    ...
end

obsR_nA_n : R_n × A_n × V → V

M_Notification : N × V → Unit
M_Notification(n,v) ≡ cases n of
    mkIN(to:R,nr:NR,i:Info) →
        M_NotificationtRule(nr,v) ∧ output(M_Recipient(to), M_Info(i,v))
    mkTN(to:R,nr:NR,nt:NT) →
        M_NotificationtRule(nr,v) ∧ output(M_Recipient(to), M_NotificationTask(nt,
end

M_NotificationRule : NR×V → BOOL

M_Recipient : R → CH
M_Recipient(r) ≡ cases r of
    mkUR(UId) → obs_channel(UId)
    mkSR(SId) → obs_channel(SId)
    mkExR(ExId) → obs_channel(ExId)
    ...
end

obs_channel : (UId|SID|ExId) → CH

M_Info : Info×V → Info

M_NotificationTask : NT×V → TaskId×V
M_NotificationTask(nt,v) ≡ cases nt of
    mkMT(MTId) → (MTId,v)
    mkST(STId) → (STId,v)
    ...
end
```

9.4 Discussion

We wish to comment on how the solution model of External Asynchronous EMS (Section 8.4 on page 77) relates to the semantic model of EML. To do so let us consider the following EML semantic:

$$
\begin{array}{rcl}
\mathcal{M}_{\text{EML\_Spec}} & : & \texttt{EML\_Spec} \rightarrow \texttt{ENV} \rightarrow \Sigma \rightarrow \Sigma \\
\mathcal{M}_{\text{EML\_Spec}}\,\texttt{(ems)}\,(\rho)\,(\sigma) & \equiv & \textbf{if } \texttt{ems} = \texttt{\{\}} \textbf{ then skip} \\
 & & \textbf{else} \\
 & & \quad \textbf{let } \texttt{(e,n)} : \texttt{(E}\times\texttt{N)} \bullet \texttt{(e,n)} \in \texttt{ems} \textbf{ in} \\
 & & \quad\quad \textbf{let } \texttt{(b,v)} = \mathcal{M}_{\text{Event}}\,\texttt{(e)}\,(\rho)\,(\sigma) \textbf{ in} \\
 & & \quad\quad\quad \textbf{if } \texttt{b} \textbf{ then} \\
 & & \quad\quad\quad\quad \mathcal{M}_{\text{Notification}}\,\texttt{(n,v)}\,;\, \mathcal{M}_{\text{EML\_Spec}}\,\texttt{(ems}\backslash\texttt{\{(e,n)\})}\,(\rho)\,(\sigma) \\
 & & \quad\quad\quad \textbf{else} \\
 & & \quad\quad\quad\quad \mathcal{M}_{\text{EML\_Spec}}\,\texttt{(ems)}\,(\rho)\,(\sigma) \\
 & & \quad\quad\quad \textbf{end} \\
 & & \quad\quad \textbf{end} \\
 & & \quad \textbf{end} \\
 & & \textbf{end}
\end{array}
$$

- When in the above model we write σ , it refers to the global-state $\texttt{stg}:\Sigma$ of Simple Application System (SAS) in the External Asynchronous EMS solution model. The state of SAS is polled to detect occurrence of an Event.
- The meaning function $\mathcal{M}_{\text{Event}}\,\texttt{(e)}\,(\rho)\,(\sigma)$ corresponds to `invoked: I×ET →` **in** `cstg` **Unit** `×BOOL × V` function within Event Management Interface `EMI`, in the External Asynchronous EMS solution model.
- $\mathcal{M}_{\text{Notification}}\,\texttt{(n,v)}$ corresponds to `Notification:E×RS →` **out** $\{\texttt{crcp[r]:M|r:RIdx}\}$ **Unit** in the External Asynchronous EMS solution model.

Chapter 10

Guidelines for Requirements Engineers

The aim of this chapter is to provide guidelines in terms of example usage scenarios with there specification in EML for a CSC Requirements Engineer. The intention is that, it would enable a CSC Requirements engineer, to be first elicit and then specify customer-specific requirements in EML. It is assumed that, a CSC Requirements Engineer is knowledgeable about the customer's specific domain. The chapter is organised around the following different Event Management Functionality offered by the Standard Event Management software:

- **Exception Management**
  To alert employees when some exceptional situation happens. See section 10.1 for example scenarios and there specification in EML.

- **Event Based Workflow**
  To send alerts about task completion, which in turn, enables to have a workflow of Tasks. See section 10.2 for example scenarios and there specification in EML.

- **Pro-active Information**
  Proactively send information like "status of Sales Order" , to customers, suppliers and people inside the company. See section 10.3 for example scenarios and there specification in EML.

10.1 Exception Management

Scenario 1

Consider the below exception scenario:
"A Master Resource Planning(MRP) engine is started in SAS on Friday evening at 5pm and is let to run during the weekend. If an error occurs during the MRP run, the Production Planner would like to be notified about the exception by an SMS on his mobile. " (Based on Use-case B.3 on page 116)

EML SPECIFICATION

```
 -- EventNotify
Event   "SAS Application Id"   "MRP calculation run."    "MRP status = Error "
Notify  "Production Planner Mobile No."
          Info("Error during MRP run.", "Link to error details,")
```

Scenario 2

Consider the below exception scenario:
"The Sales Manager has a goal to ensure that profit margin on any sales doesn't fall below 10%. For this, Sales Manager would like to be alerted by Email, whenever there is a Sales with a profit less than 10%." (Based on Use-case B.1 on page 115)

EML SPECIFICATION

```
 -- EventNotify
Event   "Salesperson SAS User Id"   "Update Sales Order with price and discounts."
         "(Sales Order Status = Released) AND (Profit < 10%)"
Notify  "Sales Manager Email Id"
          Info("Sales order below profit margin.","Link to Sales Order details,")
```

Scenario 3

Consider the below exception scenario:
"The Production Manager is carrying out his daily production activity. The initiated operations are causing delay/back orders. Both the Production Manager and the Salesperson would like to know about the possible delay in production, so that they can plan for the sales or inform the customer as appropriate. Production Manager and Sales person would like to be notified on there phone and mobile respectively. " (Based on Use-case B.2 on page 115)

EML SPECIFICATION

```
 -- EventNotify
Event   "SAS Application Id"   "Register production time."
         "Delayed operations or back-orders"
Notify  "Production Manager Email Id"
         Info("Delay in production activity.","Link to activity details.")
Notify  "Sales Manager Mobile No."
         Info("Sales Order could be delayed due to delay in production activity.",
              "Link to sales order details.")
```

Scenario 4

Consider the below exception scenario:
"The Purchaser would like to know if the inventory of a critical item X goes below Item Reorder point. So that he can plan for the items replenishment." (Based on Use-case B.4 on page 116)

EML SPECIFICATION

```
-- EventNotify
Event   "Production Manager SAS User Id"   "Consume critical item for production."
         "Inventory of item < Reorder Point"
Notify  "Purchaser Email Id"
          Info("Low inventory for item X.","Link to Item inventory details,")
```

Scenario 5.

Consider the below exception scenario:
"Whenever inventory of an item is about to expire within 10 days, the Production Manager and Salesperson needs to notified, for taking suitable actions." (Based on Use-case B.8 on page 119)

EML SPECIFICATION

```
 -- EventNotify
Event    "SAS Application Id"   "Check for Item expire date."
          "(Expire date - Current Date) = 10 days)"
Notify   "Production Manager Email Id"
          Info("Items are about expire in 10 days.","Link to Items")
Notify   "Salesperson Email Id"
          Info("Items are about expire in 10 days.","Link to Items")
```

10.2 Event Based Workflow

Scenario 1

Figure 10.1 shows, an example "Create new item"workflow scenario. (Based on the Use-case B.6 on page 117)

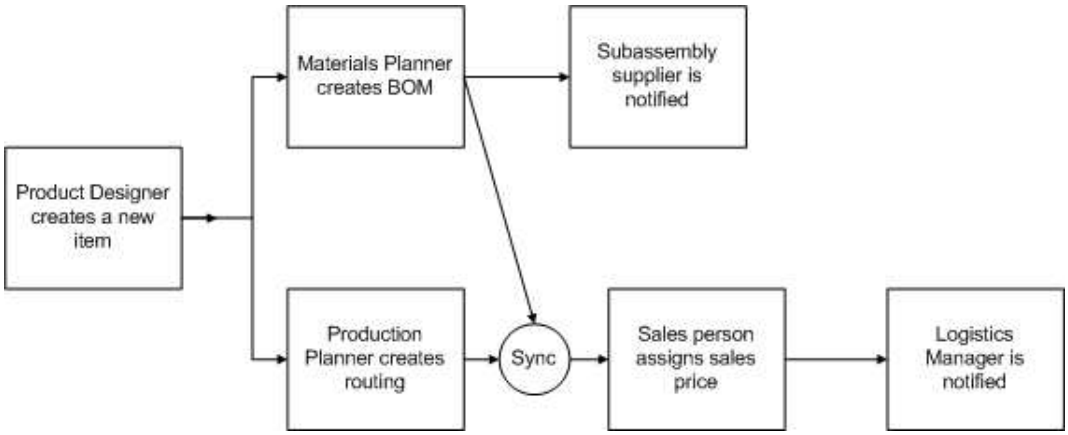


Figure 10.1: Workflow to create a new item in SAS

It involves the following steps:

1. The Product Designer creates a new item with its technical details only. The Item does not include information about BOM, cost, routing, and sales price.

2. The Materials Planner needs to be notified by Email to create BOM structure for the item. At the same time, the Production Planner would like to be informed by Email to create routing information.

3. After the Materials Planner creates BOM structure for the item and calculates unit costs for the new item, the supplier needs to be informed about the subcomponent details.

4. When the Production Planner has finished creating routing and the Materials Planner has created BOM structure for the item, Salesperson would like to be informed. So that the Salesperson can assign sales price for the item.

5. Once all the details are available for the item, the Logistics Manager would like to know which items to plan for production.

```
 -- EventNotify
Event    "Production Designer SAS User Id"   "Create a new item."
         "(Item BOM Structure = Empty) AND (Routing = Empty)
          AND (Sales price = 0.0)"
Notify   "Material Planner Email Id"
           Task("Create BOM structure for item.","Link to Item details")
Notify   "Production Planner Email Id"
           Task("Create routing for Item.","Link to Item details")
-- EventNotify
Event    "Material Planner SAS User Id"
          "Create BOM structure for item and calculate unit price."
Notify   "Supplier Email Id"
          Info("New Item.","Link to Item and BOM details")
-- EventNotify
Event    "Production Planner SAS User Id"
          "Create routing and calculate capacity requirements for Item."
AND
Event    "Material Planner SAS User Id"
          "Create BOM structure for item and calculate unit price."
Notify   "Salesperson Email Id"
          Info("Item is ready for production and needs sales price.",
               "Link to Item details")
-- EventNotify
Event    "Salesperson SAS User Id"
          "Enter sales price for the item based on calculated cost."
Notify   "Logistics Manager Email Id"
          Info("New Item is created.","Link to Item and BOM details")
```

Scenario 2

Figure 10.2 shows, an example "Sales/Shipping" workflow scenario (Based on Use-case B.5 on page 117)

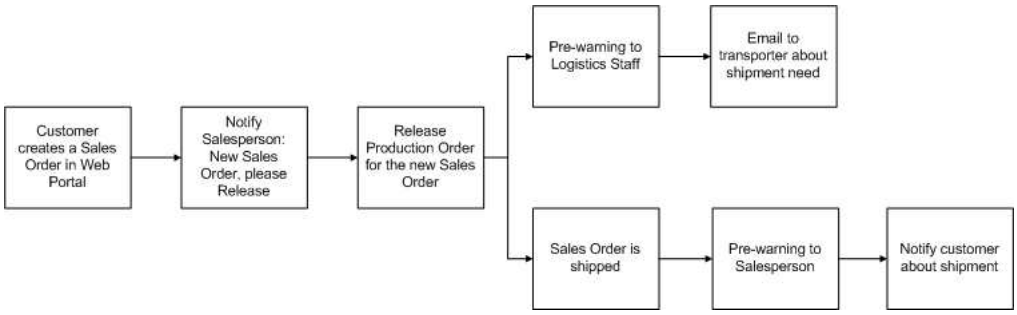


Figure 10.2: Workflow - Sales and Shipment Notifications

It involves the following steps:

1. A customer places an order for item X, via a Portal on the web.
2. A Salesperson is notified, about creation of a new Customer Order with order details.
3. Salesperson creates and releases a new Production Order for the Customer Order.

4. Logistics Staff is given a pre-warning , to Accept/Decline sending an email to the transportation supplier.
5. The Logistics Staff accepts to send email to the transportation supplier.
6. An Email is sent to the transportation supplier, with item and shipping details.
7. After shipping, the Warehouse Worker changes Sales Order status to shipped.
8. A pre-warning is sent to Salesperson to Accept/Decline, to send an email to the customer informing about the shipping-date.
9. When the Salesperson accepts sending of email to Customer, an email is sent to the customer.

EML SPECIFICATION

– *EventNotify*

```
Event   "Customer Web portal Id"   "Place a new order."
Notify  "Salesperson SAS User Id"
         Info("New order created.","Link to Order details")
-- EventNotify
Event   "Salesperson SAS User Id"   "Create and release a Production Order."
Notify  "Logistics Staff SAS User Id"
         Task("Accept/Decline sending of email to transportation supplier.",
              "Link to Order details")
-- EventNotify
Event   "Logistics Staff SAS User Id"
         "Accepts to send an email to transportation supplier."
Notify  "Transportation Supplier Email Id"
         Info("New item to be shipped.","Link to Item and Shipping details")
-- EventNotify
Event   "Warehouse Worker SAS User Id"   "Update Sales Order status."
         "Order Status = Shipped"
Notify  "Salesperson SAS User Id"
         Task("Accept/Decline sending of email to customer.","Link to order details")
-- EventNotify
Event   "Salesperson SAS User Id"   "Accepts to send an email to customer."
Notify  "Customer Email Id"
          Info("The order shipment details.","Link to order details")
```

Scenario 3

Figure 10.3 shows, an example "Supplier Reminder" workflow scenario (Based on Use-case B.7 on page 118)

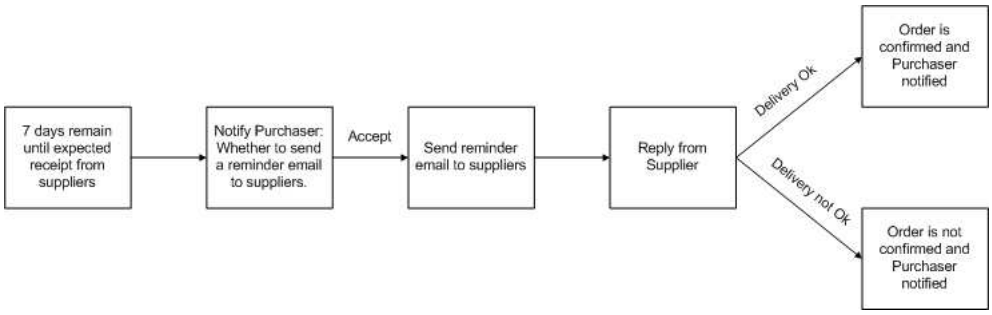


Figure 10.3: Workflow - Supplier Reminder

It involves the following steps:

1. Purchaser goes about daily work of procurement and supplier management. 7 days remain until expected receipt from some suppliers.
2. Purchaser would like to be notified to accept or decline, whether to send a reminder email to the suppliers.
3. When the purchaser accepts sending of reminders, a reminder email is sent to the suppliers with order details.
4. When the supplier replies to the reminder email and responds about Delivery as “OK”, the Order is updated as “Confirmed” and the Purchaser is notified by SMS about the scheduled delivery.
5. When the supplier replies to the reminder email and responds about Delivery as “Not OK”, the Order is updated as “Not Confirmed” and the Purchaser is notified by SMS about problem in delivery.

EML SPECIFICATION

```
 -- EventNotify
Event   "SAS Application Id"   "Check expected receipt date."
         "(Expected Receipt Date - Current Date) = 7days"
Notify  "Purchaser SAS User Id"
         Task("Accept/Decline to send reminder email to suppliers.",
              "Link to Order details")
-- EventNotify
Event   "Purchaser SAS User Id"   "Accept/Decline to send reminder email to suppliers."
         "Response = Accept"
Notify  "Supplier Email Id"
         Task("Respond whether delivery is OK/Not OK.","Link to Order details")
-- EventNotify
Event   "Supplier Email Id"   "Replies by Email about Delivery."
         "Response = Delivery OK"
Notify  "SAS Application Id"
         Task("Update Order to confirmed.","Link to Order details")
Notify  "Purchaser Mobile No."    Info("Scheduled Delivery.","Link to Order details")
-- EventNotify
Event   "Supplier Email Id"   "Replies by Email about Delivery."
         "Response = Delivery not OK"
Notify  "SAS Application Id"
         Task("Update Order to not confirmed.","Link to Order details")
Notify  "Purchaser Mobile No."
         Info("Delivery Problem.","Link to Order details")
```

10.3 Proactive Info

Scenario 1.

Consider the below exception scenario:
”The customer requests the Salesperson to cancel the Sales Order. This also means, to cancel the Production Order associated with the Sales Order. The Production Planner would like to be notified by Email in good time the orders which needs to be canceled, to avoid unnecessary production .” (Based on Use-case B.10 on page 119)

EML SPECIFICATION

```
 -- EventNotify
Event    "Salesperson SAS User Id"   "Cancel a Sales Order."
          "Order Status = Canceled"
Notify   "Production Manager Email Id"
          Info("Sales Order is canceled","Link to Sales Order, Production Orders")
```

Scenario 2.

Consider the below exception scenario:
"Once the output from a production order has been received at the warehouse, the Warehouse Personnel is to be notified on his Mobile to pick the items. Also, the Sales Manager is informed in SAS Task List, about completion of production order." (Based on Use-case B.9 on page 119)

EML SPECIFICATION

```
 -- EventNotify
Event    "SAS App Id"   "Register output from production order."
Notify   "Warehouse Person Mobile No."
          Info("Item to be picked","Link to Production Order Details")
Notify   "Sales Manager SAS User Id"
          Info("Production Order is complete and ready to be picked",
               "Link to Production Order Details")
```

Part IV

Conclusion

Chapter 11

Conclusion

11.1 Summary of Contributions

In summary, the main contributions of this thesis are as follows:

1. Contribution to the following areas of Requirements Engineering research challenges Section 1.1.4:

   (a) *Reuse of requirements models*
   We have demonstrated that the requirements found during development of Standard Software by MDC could be reused by the CSC *Requirements Engineer.*

   (b) *Support for requirements practitioners*
   We have shown the importance of domain-specific language (DSL) as a tool for the CSC Requirements Engineer, to both elicit and specifying customer specific requirements.

   (c) *Bridging the gap between requirements elicitation approaches based on contextual enquiry and more formal specification and analysis techniques*
   We have shown, how DSL could serve as a link between the informal elicitation techniques (Use Cases in our case) and formal specification in RSL.

   (d) *Better understanding of the impact of software architectural choices on the prioritisation and evolution of requirements*:
   Modelling he different solution approaches, explicitly brought out benefits and limitations among different approaches.

2. Contribution to the Event Management Project at Microsoft Business Solutions:

   (a) Proposal of an approach for creating a domain-specific language at the MDC.

   (b) Requirements model for *Event Management.*

   (c) *Event Management Language* as a tool for CSC *Requirements Engineers.*

11.2 Limitations

One of the major shortcoming of this thesis, has been that we could not evaluate of the Event Management Language with real CSC Requirements Engineers. Because of this we are not able to conclude

about the usefulness of EML in the field. On the other hand, we were able to get feedback from MBS Event Management Project Team. The main feedback were:

- They were able to use the example specification to present Event Management functionality to other teams in the company. They found it more precise than Use-cases or other type of descriptions.
- They were able to make the Use-cases more precise after looking at the basic constructs and examples of EML specification.
- dInterestingly, the test engineer of project team, found it very useful to define functional Test Cases.

11.3 Conclusions

The following inferences could be drawn based on the work done in this thesis are:

1. Partial application of Formal specification and modelling techniques, can be very effective to business software applications.
2. Domain-specific language serve as a link between the informal elicitation techniques (Use Cases in our case) and formal specification in RSL.
3. Step-wise development of different solution approaches help to understand the requirements better and possibly unravel benefits and limitations.

11.4 Future Work

1. **Deriving DSL elements from Formal Models**
   It could be of value to find out, once we have a formal domain and requirements model, how can we derive a domain-specific language from it. Work in this area would help software engineers, who are not that proficient in language design.
2. **Evaluation of EML**
   It would be worth investigating further, how EML and specification examples in EML are perceived by the CSC Requirements Engineers. That would serve as great input for further work in this direction.
3. **Application Generator for EMS**
   Based on the models of EML, an application generator could be developed, which would generate customer-specific EMS based on requirements specified in EML.

Part V

Appendices

Appendix A

Glossary of Key Terms

This appendix contains an alphabetized list of the acronyms and key terms used in this thesis.

A.1 Acronyms

CSC Customer Solution Center

DSL Domain Specific Language

EMS Event Management System

MDC Main Development Center

RE Requirements Engineering

RSL/CSP Raise Specification Language / Communicating Sequential Processes

SAS Simple Application System

Appendix B

EMS Use Cases

This appendix lists the Use Cases from the EMS project. The Use Cases are modified to accommodate the proprietary requirements of EMS Project at Microsoft Business Solutions.

B.1 Profit margin below limit

Context The Sales Manager wants to be notified, if the profits on item of group B goes below 10%.

Actors Sales Manager, Salesperson.

Goal Make sure that sales price levels ensure a certain profit margin.

Sequence

Actor action	System response
1. Sales Manager/Salesperson enters sales order with prices and discounts.	
2. One day, a released/posted sales order for an item in group B has a profit lower than 10%.	**3.** A notification is sent to the Sales Manager by Email, with links to the Sales Order and information about the Salesperson who is handling the Sales Order.
4. Sales Manager acts accordingly.	

B.2 Order Delayed - Production/Sales Activity Control

Context The delivery date of a planned sales order is pushed due to a slack in the production line.

Actors Production Manager, Salesperson

Goal To notify actors that production order can not meet promised ending date.

Sequence

Actor action	System response
1. Production Manager performs daily production activity	
2. Initiated operations are causing delay/back orders.	**3.** System notifies the Production Manager and the Sales person about a possible delay in delivery, by Email and SMS respectively. The message includes details about the delaying activity.
4. Production Planner and Salesperson act accordingly.	

B.3 MRP run error

Context As part of normal procedure, the MRP is set up to calculate a new materials plan during the weekend. It is crucial that the MRP result is ready on Monday morning when production must be restarted. If run errors are not detected until Monday, it will be too late to rerun.

Actors Production Planner.

Goal To alert actor of critical problem outside business hours.

Sequence

Actor action	System response
1. The MPS/MRP are set to start Friday evening and run during the weekend.	**2.** The system run the MPS/MRP planning according to planned.
3. An error occurs in the MPS/MRP and the MRP run is stopped.	**3.** The system registers the MRP errors. A notification is sent to the Production Planner on his mobile phone.

B.4 Inventory below reorder point

Context The Purchase Manager would like to keep track of critical item inventory for his order planning.

Actors Purchase Manager

Goal Notify the purchaser that item inventory is below reorder point - if critical.

Sequence

Actor action	System response
1. The inventory of a critical item X falls below the reorder point, due to sale of the item.	**2.** System registers the low inventory. A notification is sent to the purchaser by Email. It contains the item inventory information like Current Stock, Reorder point and a link to the Item.

B.5 Collaboration - Sales and shipping notifications

Context A customer places an Order which results in a new Sales Order for the Item, which needs to manufactured and shipped. When the production ending date is known, the transportation supplier is notified. When the item is shipped, the customer is notified.

Actors Salesperson, Logistics Staff, Warehouse Worker, Transportation supplier, Customer

Goal Improve customer service by providing shipping information. To make the process more effective by proactively providing information to the next in the process.

Sequence

Actor action	System response
1. A customer places an order for item X, via a Portal on the web.	**2.** The system registers an open Sales Order. And informs the Salesperson by a notification to his Application Inbox. The Sales Order details are included.
3. The Salesperson receives a notification about the new sales order and creates a Production Order for the Sales Order.	
4. The Production Planner releases the Production Order for production.	**5.** The system registers the Production Order as released. And sends a pre-warning to the Logistics Staff to his Application Inbox, in order to Accept/Decline, sending an email to the transportation supplier.
6. The Logistic Staff receives the notification and accepts that the email be sent to the transportation supplier.	**7.** The system sends an email to the transportation supplier. The email contains, information about items, weight, number of pallets, pick-up date, delivery address etc
8. After shipping, the Warehouse Worker changes Sales Order status to shipped.	**9.** The system registers the Sales Order as shipped. And sends a pre-warning to Salesperson, about the shipping-date-email for the customer.
10. The Salesperson receives the notification and accept that the email is sent to the Customer informing about the shipping details.	**11.** The system sends the email to the customer stating the delivery date and order details

B.6 Create new end item

Context The Product Designer has configured a new end item and proceeds to enter his data on a new item card in Attain. This causes related tasks for numerous interdependent actors.

Actors Product Designer (responsible for the configuration of new items), Materials Planner (in charge of BOM structures, unit costs and procurement), Production Planner (must define routing and BOM to prepare new item for production), Sales person (must assign a sales price to new item and prepare it for sale) , Logistics Manager (in charge of all aspects of the internal supply chain)

Goal Optimize the workflow around the creation of a new item in a production company.

Sequence

Actor action	System response
1. Product Designer creates a new item with its technical details only. The Item does not include information about BOM, cost, routing, and sales price.	**2.** System registers new incomplete item. It informs Materials Planner by Email to create BOM structure for the item. At the same time, system informs the Production Planner by Email to create routing information. Links to the new item is given.
3. Materials Planner creates BOM structure for the item and calculates unit costs for the new item.	**4.** The system registers the BOM structure and the item unit cost. And notifies the suppliers of new item details, by Email.
5. At the same time, Production Planner creates routing and calculates capacity requirements.	**6.** System synchronises the two tasks and notifies the sales person that the item is ready for production and needs a sales price.
7. Salesperson enters sales price for the item based on the calculated costs.	**8.** System registers that new item has complete set of data and notifies logistics manager that a new end item is created.

B.7 Remind supplier of delivery

Context The Purchaser may issue reminders to suppliers, either as part of a standard procedure to ensure that all inbound deliveries are on schedule, or as a special precaution against failure by specific unstable suppliers.

Actors Purchaser, Supplier

Goal To avoid disruption in supply chain due to delivery failure. Also, to know about potential delivery problems as early as possible.

Sequence

Actor action	System response
1. Purchaser goes about daily work of procurement and supplier management. 7 days remain until expected receipt from supplier XX	**2.** System notifies the Purchaser in his Application Inbox, to Accept/Decline, to send a reminder to the Suppliers by email.
2. Purchaser accepts, sending of reminders.	**3.** System sends reminder email to supplier XX. The email contains order details and options: “Delivery Ok” / “Delivery not ok”
4a. The supplier receives the Email and replies as “Delivery Ok”.	**5.** The system updates the order as confirmed and notifies the Purchaser about scheduled delivery.
4b. The supplier receives the Email and replies as “Delivery Not Ok”.	**6.** The system updates the order as not confirmed and notifies the Purchaser about the problem in delivery.

B.8 Inventory expire date passed

Context The Production Manager and Salesperson wish to know about inventory expire dates.

Actors Production Manager / Salesperson.

Goal Inform actors in time about expire for suitable action to be taken.

Sequence

Actor action	System response
1. The items in the inventory are about to expire in 10 days.	**2.** The system notifies the Production Manager and Salesperson, with a list of items which are about to expire and there details.
3. The Production Manager and Salesperson take suitable actions.	

B.9 Output to inventory

Context The output from a production order has been received at the warehouse and the information has to be passed on.

Actors Sales Manager/Warehouse Personnel

Goal To notify sales personnel that production output has been received in inventory

Sequence

Actor action	System response
1. Output from a production order is registered in inventory.	**2.** The system registers for the inbound of produced items. And notifies the Warehouse Personnel to pick the items and alerts the Sales Manager.
3. The recipients act accordingly.	

B.10 Order/demand canceled

Context An order is canceled and the components are not needed.

Actors Production Planner

Goal To notify the the Production Planner that the order has been canceled and the demand has changed.

Sequence

Actor action	System response
1. The Sales person cancels an Sales Order and the associated production order is no longer needed.	**2.** The system registers the Sales Order as canceled and notifies the Production Planner by Email, about the Canceled Orders with Order details.
3. The Planner runs a net change MPS/MRP	

Appendix C

Domain Model of Simple Application System (SAS)

```
type        R,V,U,S,AIdx,UIdx
            AP==A1|A2| ...|An
            APU=AP m→ U−set, APS=AP m→S−set

channel     {cua[u,a]:(R|V)|u:UIdx,a:AIdx},
            {caa[a,a′]:(R|V)|a,a′:AIdx • a ≠ a′}

value       aps:APS,
            obs_servType:S → Register|Lookup|Execute
            obs_AIdx:  AP→AIdx

Sys: Unit   →  Unit
  Sys()     ≡  {U(u)|u:UIdx} ‖ {A(a)|a:AIdx}

U(u):u:UIdx →  in,out {cua[u,a]|a:AIdx} Unit
  U(u)      ≡  let ap:AP = A1 ⊓ A2 ⊓ ... ⊓ An in
                let r:R = gen_req(select_serv(ap),userinput:V) in
                output (cua[u,obs_AIdx(ap)],r)
                U(u) |||
                {letv = input (cua[u,obs_AIdx(ap)) in U(u) end}
                end
               end

            gen_req:S × V → R
```

```
                    select_serv:AP → S
                    select_serv(ap) ≡ ⊓{s | s:S • s ∈ aps(ap)}

A(a):a:AIdx →       in,out {cua[u,a]|u:UIdx},
                    {caa[a′,a] | a′:AIdx • a ≠ a′} Unit
      A(a) ≡
           []       {let r:R = input(cua[u,a]) in
                     let v = execute(r)in
                       output(cua[u,a],v) ⊓ skip
                     end
                    end
                    ⊓
                    let r:R = input(caa[a',a]) in
                     let v = execute(r)in
                       output(caa[a,a'],v) ⊓ skip
                     end
                    end|u:UIdx,a′:AIdx};A(a)

                    execute:R → V
```

Appendix D

EMS Requirements Model

```
type      Ψ, ENTS, ENS, Σ, ETIdx, ENIdx, RIdx
          I, T, R, CH, M, V, RES, D, RIdx,
          E = I × ET × ER,
          ER = V → BOOL,
          I = User | SAS | External,
          N = R × NR × CH × M,
          R = Fixed | Dynamic | Subscribed
          NR = V → BOOL,
          ENT = E m⃗ N-set,
          ENTS = ETIdx m⃗ ENT
          ENS = ENIdx m⃗ Σ
          Σ = ETIdx × Φ

channel   {crcp[r]:M | r:RIdx}

value     ent:ENT
          subscribe:ENT × N → ENT

          / * Event Management Process * /
EMS:Unit  →  Unit
EMS()     ≡  ‖ {EN(en) | en:ENIdx}

          / * Event Notification Instance * /
EN(en):en:ENIdx  →  Unit
EN(en)           ≡  let e:E = obs_event(en) in
                      cases Event(e) of
                        NOEVENT → skip,
```

```
                                  FAULT → skip,
                                  EVENT(res) → Notification(e,res))
                                 end
                                end; EN(en)

                                obs_event:ENIdx→E

                                /* Event */
                    Event:E  →  RS
                    Event(e) ≡  let (i,t,er) = e in
                                 cases invoked(i,t) of
                                  FALSE → NOEVENT,
                                  res:RS →
                                    if er(res) then EVENT(res)
                                    else NOEVENT end
                                 end
                                end

                                invoked:TRG → FALSE|RS

                                /* Notification */
  Notification:E × RS  →  out {crcp[r]:M|r:RIdx} Unit
Notification(e,res)    ≡  let ns:N-set = ent(e) in
                       ‖   {let (r,nr,ch,m) = n in
                             if nr(res) then output(ch,m) end
                            end |n ∈ ns}
                           end
```

Appendix E

SAS with Direct EMS

type

R,V,U,S,AIdx,UIdx
AP==A1|A2| ...|An
APU=AP $\overrightarrow{m}$ U−**set**, APS=AP $\overrightarrow{m}$S−**set**

Ψ, ENTS, ENS, Σ, ETIdx, ENIdx,RIdx
I,T,R,CH,M,V,RES,D,RIdx,
E=I×ET×ER,
ER=V→BOOL,
I = User|SAS|External,
N = R×NR×CH×M,
R = Fixed|Dynamic|Subscribed
NR = V → BOOL,
ENT = E$\overrightarrow{m}$N−**set**,
ENTS = ETIdx $\overrightarrow{m}$ ENT
ENS = ENIdx $\overrightarrow{m}$ Σ
Σ = ETIdx ×Φ

channel

{cua[u,a]:(R|V)|u:UIdx,a:AIdx},
{caa[a,a′]:(R|V)|a,a′:AIdx • a ≠ a′}
{crcp[r]:M|r:RIdx}

value

aps:APS,
obs_servType:S → Register|Lookup|Execute
obs_AIdx: AP→AIdx

```
Sys: Unit → Unit
Sys() ≡ {U(u)|u:UIdx} ‖ {A(a)|a:AIdx}

U(u):u:UIdx → in,out {cua[u,a]|a:AIdx} Unit
U(u) ≡ let ap:AP = A1 ⊓ A2 ⊓ ... ⊓ An in
 let r:R = gen_req(select_serv(ap),userinput:V)
 output (cua[u,obs_AIdx(ap)],r)
 U(u) |||
 {letv = input (cua[u,obs_AIdx(ap)) in U(u) end}
 end
end

gen_req:S × V → R

select_serv:AP → S
select_serv(ap) ≡ ⊓{s | s:S • s ∈ aps(ap)}

A(a):a:AIdx → in,out {cua[u,a]|u:UIdx},
{caa[a′,a] | a′:AIdx • a ≠ a′} Unit
A(a) ≡
[] {
{let r:R = input(cua[u,a]) in
 let v = execute(r)in
  output(cua[u,a],v) ⊓ skip
 end
end
⊓
let r:R = input(caa[a′,a]) in
 let v = execute(r)in
  output(caa[a,a′],v) ⊓ skip
 end
end}
EventNotification(e)
|u:UIdx,a′:AIdx};A(a)

execute:R → V
```

```
                                  /* Event Notification Function */
EventNotification(e):e:E  →       Unit
    EventNotification(e)  ≡       cases Event(e) of
                                    NOEVENT → skip,
                                   FAULT → skip,
                                   EVENT(res) → Notification(e,res))
                                  end

                                  /* Event */
                  Event:E  →      RS
                  Event(e)  ≡     let (i,t,er) = e in
                                   cases invoked(i,t) of
                                    FALSE → NOEVENT,
                                    res:RS →
                                      if er(res) then EVENT(res)
                                      else NOEVENT end
                                   end
                                  end

                                  invoked:TRG → FALSE|RS

                                  /* Notification */
        Notification:E × RS  →    out {crcp[r]:M|r:RIdx} Unit
        Notification(e,res)  ≡    let ns:N-set = ent(e) in
                             ||   {let (r,nr,ch,m)=n in
                                   if nr(res) then output(ch,m) end
                                  end |n ∈ ns}
                                  end
```

Appendix F

SAS with Synchronous EMS

type

R,V,U,S,AIdx,UIdx
AP==A1|A2| ...|An
APU=AP $\overrightarrow{m}$ U−**set**, APS=AP $\overrightarrow{m}$S−**set**

Ψ, ENTS, ENS, Σ, ETIdx, ENIdx,RIdx
I,T,R,CH,M,V,RES,D,RIdx,
E=I×ET×ER,
ER=V→BOOL,
I = User|SAS|External,
N = R×NR×CH×M,
R = Fixed|Dynamic|Subscribed
NR = V → BOOL,
ENT = E$\overrightarrow{m}$N−**set**,
ENTS = ETIdx $\overrightarrow{m}$ ENT
ENS = ENIdx $\overrightarrow{m}$ Σ
Σ = ETIdx ×Φ

channel

{cua[u,a]:(R|V)|u:UIdx,a:AIdx},
{caa[a,a′]:(R|V)|a,a′:AIdx • a ≠ a′}
{crcp[r]:M|r:RIdx}

value

aps:APS,
obs_servType:S → Register|Lookup|Execute
obs_AIdx: AP→AIdx

```
Sys: Unit  →  Unit
Sys()  ≡  {U(u)|u:UIdx} ‖ {A(a)|a:AIdx} ‖ EMS()

U(u):u:UIdx  →  in,out {cua[u,a]|a:AIdx} Unit
U(u)  ≡  let ap:AP = A1 ⊓ A2 ⊓ ... ⊓ An in
          let r:R = gen_req(select_serv(ap),userinput:V) in
           output (cua[u,obs_AIdx(ap)],r)
           U(u) |||
           {letv = input (cua[u,obs_AIdx(ap)) in U(u) end}
          end
         end

         gen_req:S × V → R

         select_serv:AP → S
         select_serv(ap) ≡ ⊓{s | s:S • s ∈ aps(ap)}

A(a):a:AIdx  →  in,out {cua[u,a]|u:UIdx},
                {caa[a',a] | a':AIdx • a ≠ a'}
                ,out{cae[a]}Unit
A(a)  ≡
      []  {
          {let r:R = input(cua[u,a]) in
           let v = execute(r)in
             output(cua[u,a],v) ⊓ skip
           end
          end
          ⊓
          let r:R = input(caa[a',a]) in
           let v = execute(r)in
             output(caa[a,a'],v) ⊓ skip
           end
          end}
          output(cae[a],e)
          |u:UIdx,a':AIdx};A(a)

          execute:R → V
```

```
                                  / * Event Management Process * /
EMS():Unit                  →     Unit
EMS()                       ≡
                            []    {let e = input(cae[a]) in
                                    cases Event(e) of
                                      NOEVENT → skip,
                                      FAULT → skip,
                                      EVENT(res) → Notification(e,res))
                                    end
                                  end|a:AIdx}EMS()

                                  / * Event * /
Event:E                     →     RS
Event(e)                    ≡     let (i,t,er) = e in
                                    cases invoked(i,t) of
                                      FALSE → NOEVENT,
                                      res:RS →
                                        if er(res) then EVENT(res)
                                        else NOEVENT end
                                    end
                                  end

                                  invoked:TRG → FALSE|RS

                                  / * Notification * /
Notification:E × RS         →     out {crcp[r]:M|r:RIdx} Unit
Notification(e,res)         ≡     let ns:N-set = ent(e) in
                            ||    {let (r,nr,ch,m) = n in
                                    if nr(res) then output(ch,m) end
                                  end |n ∈ ns}
                                  end
```

Appendix G

SAS with Asynchronous EMS

type

```
R,V,U,S,AIdx,UIdx
AP==A1|A2| ...|An
APU=AP m-> U-set, APS=AP m->S-set

Ψ, ENTS, ENS, Σ, ETIdx, ENIdx,RIdx
I,T,R,CH,M,V,RES,D,RIdx,
E=I×ET×ER,
ER=V→BOOL,
I = User|SAS|External,
N = R×NR×CH×M,
R = Fixed|Dynamic|Subscribed
NR = V → BOOL,
ENT = E m->N-set,
ENTS = ETIdx m-> ENT
ENS = ENIdx m-> Σ
Σ = ETIdx ×Φ
```

channel

```
{cua[u,a]:(R|V)|u:UIdx,a:AIdx},
{caa[a,a′]:(R|V)|a,a′:AIdx • a ≠ a′}
{crcp[r]:M|r:RIdx}
```

value

```
aps:APS,
obs_servType:S → Register|Lookup|Execute
obs_AIdx:  AP→AIdx
```

```
Sys: Unit → Unit
Sys() ≡ {U(u)|u:UIdx} ‖ {A(a)|a:AIdx} ‖ EMS()

U(u):u:UIdx → in,out {cua[u,a]|a:AIdx} Unit
U(u) ≡ let ap:AP = A1 ⊓ A2 ⊓ ... ⊓ An in
        let r:R = gen_req(select_serv(ap),userinput:V) in
         output (cua[u,obs_AIdx(ap)],r)
         U(u) |||
         {letv = input (cua[u,obs_AIdx(ap)) in U(u) end}
        end
       end

       gen_req:S × V → R

       select_serv:AP → S
       select_serv(ap) ≡ ⊓{s | s:S • s ∈ aps(ap)}

A(a):a:AIdx → in,out {cua[u,a]|u:UIdx},
              {caa[a′,a] | a′:AIdx • a ≠ a′}
              ,out{cae[a]}Unit
A(a) ≡
     [] {
       {let r:R = input(cua[u,a]) in
         let v = execute(r)in
           output(cua[u,a],v) ⊓ skip
         end
       end
       ⊓
       let r:R = input(caa[a′,a]) in
         let v = execute(r)in
           output(caa[a,a′],v) ⊓ skip
         end
       end}|u:UIdx,a′:AIdx};A(a)

       execute:R → V
```

```
                          /∗ Event Management Process ∗/
EMS:Unit          →       Unit
EMS()             ≡       ‖ {EN(en)|en:ENIdx}

                          /∗ Event Notification Instance ∗/
EN(en):en:ENIdx   →       Unit
EN(en)            ≡       let e:E = obs_event(en) in
                            cases Event(e) of
                              NOEVENT → skip,
                              FAULT → skip,
                              EVENT(v) → Notification(e,v))
                            end
                          end}EN(en)

                          obs_event:ENIdx→E

                          /∗ Event ∗/
Event:E           →       RS
Event(e)          ≡       let(i,t,er) = e in
                            cases invoked(i,et) of
                             (false,v) → NOEVENT,
                             (true,v) →
                              if er(v) then EVENT(v)
                              else NOEVENT end
                            end
                          end

invoked:  I×ET    →       in cstg Unit ×BOOL × V
invoked(i,et)     ≡       let stg:Σ = input(cstg) in
                            let (b,v,i') = interpret(et)(stg) in
                              if b ∧ (i = i') then (true,v)
                              else (false,v) end
                            end
                          end

                          interpret:ET → Σ → BOOL×VAL×I

value
```

```
                        vstg:Σ
GS:Unit      →  out cstg Unit
   GS()      ≡  while true do
                  output(cstg,vstg) in
                  ....
                  end

                        /∗ Notification ∗/
Notification:E × V  →  out {crcp[r]:M|r:RIdx} Unit
Notification(e,v)   ≡  let ns:N-set = ent(e)in
                    ‖  {let (r,nr,ch,m)=n in
                        if nr(res) then output(ch,m) end
                       end |n ∈ ns}
                       end
```

Appendix H

SAS with External Asynchronous EMS

type

R,V,U,S,AIdx,UIdx
AP==A1|A2| ...|An
APU=AP $\overrightarrow{m}$ U−**set**, APS=AP $\overrightarrow{m}$S−**set**

Ψ, ENTS, ENS, Σ, ETIdx, ENIdx,RIdx
I,T,R,CH,M,V,RES,D,RIdx,
E=I×ET×ER,
ER=V→BOOL,
I = User|SAS|External,
N = R×NR×CH×M,
R = Fixed|Dynamic|Subscribed
NR = V → BOOL,
ENT = E$\overrightarrow{m}$N−**set**,
ENTS = ETIdx $\overrightarrow{m}$ ENT
ENS = ENIdx $\overrightarrow{m}$ Σ
Σ = ETIdx ×Φ

channel

{cua[u,a]:(R|V)|u:UIdx,a:AIdx},
{caa[a,a′]:(R|V)|a,a′:AIdx • a ≠ a′}
{crcp[r]:M|r:RIdx}

value

aps:APS,
obs_servType:S → Register|Lookup|Execute
obs_AIdx: AP→AIdx

```
Sys: Unit → Unit
Sys() ≡ {U(u)|u:UIdx} ‖ {A(a)|a:AIdx} ‖ EMI() ‖ EMS()

U(u):u:UIdx → in,out {cua[u,a]|a:AIdx} Unit
U(u) ≡ let ap:AP = A1 ⊓ A2 ⊓ ... ⊓ An in
        let r:R = gen_req(select_serv(ap),userinput:V) in
         output (cua[u,obs_AIdx(ap)],r)
         U(u) |||
         {letv = input (cua[u,obs_AIdx(ap)) in U(u) end}
        end
       end

       gen_req:S × V → R

       select_serv:AP → S
       select_serv(ap) ≡ ⊓{s | s:S • s ∈ aps(ap))}

A(a):a:AIdx → in,out {cua[u,a]|u:UIdx},
              {caa[a′,a] | a′:AIdx • a ≠ a′}
              ,out{cae[a]}Unit
A(a) ≡
     [] {
       {let r:R = input(cua[u,a]) in
         let v = execute(r)in
           output(cua[u,a],v) ⊓ skip
         end
       end
       ⊓
       let r:R = input(caa[a′,a]) in
         let v = execute(r)in
           output(caa[a,a′],v) ⊓ skip
         end
       end}|u:UIdx,a′:AIdx};A(a)

       execute:R → V


       / ∗ Event Management Interface ∗ /
```

```
EMI:en:ENIdx → in,out {cemi[en]},
EMI(en) ≡ let (i,et,er):E = input(cemi[en]) in
  cases invoked(i,et) of
   (false,v) → NOEVENT,
   (true,v) →
    if er(v) then EVENT(v)
    else NOEVENT end
  end
end

invoked: I×ET → in cstg Unit ×BOOL × V
invoked(i,et) ≡ let stg:Σ = input(cstg) in
  let (b,v,i') = interpret(et)(stg) in
   if b ∧ (i = i') then (true,v)
   else (false,v) end
  end
end

interpret:ET → Σ → BOOL×VAL×I

value

vstg:Σ
GS:Unit → out cstg Unit
GS() ≡ while true do
  output(cstg,vstg) in

/ * Event Management Process * /
EMS:Unit → Unit
EMS() ≡ ‖ {EN(en)|en:ENIdx}

/ * Event Notification Instance * /
EN(en):en:ENIdx → Unit
EN(en) ≡ let e:E = obs_event(en) in
  cases output_ input(cemi[en],e) of
   NOEVENT → skip,
   FAULT → skip,
   EVENT(v) → Notification(e,v))
  end
```

```
                        end}EN(en)

                        obs_event:ENIdx→E

                        /∗ Notification ∗/
Notification:E × RS  →  out {crcp[r]:M|r:RIdx} Unit
 Notification(e,v)   ≡  let ns:N-set = ent(e) in
                     ‖  {let (r,nr,ch,m)=n in
                          if nr(v) then output(ch,m) end
                         end |n ∈ ns}
                        end
```

Bibliography

[1] T.E. Bell and T.A. Thayer : *Software Requirements: Are They Really a Problem?*. Proc. ICSE-2: 2nd International Conference on Software Engineering. San Francisco, 1976, 61-68

[2] Zave, P. : *Classification of Research Efforts in Requirements Engineering*. ACM Computing Surveys, 29(4):315-321.

[3] The Standish Group, "*Software Chaos*", http://www.standishgroup.com.

[4] European Software Institute, "*European User Survey Analysis*", Report USV_EUR 2.1, ESPITI Project, January 1996.

[5] Bashar Nuseibeh & Steve Easterbrook: *Requirements Engineering: A Roadmap*. ACM 2000.

[6] Axel van Lamsweerde: *Requirements Engineering in the Year 00: A Research Perspective*.

[7] Ian Sommerville, *Software Engineering, 6th Edition,* Pearson Education Limited, 2001.

[8] M. Jackson: *Software Requirements & Specifications - A Lexicon of Practice, Principles and Prejudices.* ACM Press, Addison-Wesley, 1995.

[9] Mainbaum, T. S .E. : *Mathematical Foundations of Software Engineering: A Roadmap*. In this volume.

[10] Garlan, D. : *Software Architecture: A Roadmap.* In this volume.

[11] Spivey, J. M. : *Introducing Z: a Specification Language and its Formal Semantics.* Cambridge University Press, Cambridge, 1998.

[12] Nielsen, M., Havelund, K., Wagner, K., and George, C.: *The RAISE language, methods and tools.* Formal Aspects of Computing 1, 85-114, 1989.

[13] Jones, C. B. : *Systematic Software Development Using VDM*. Prentice-Hall International, New York, 1986.

[14] Hoare, C. A. R. : *Communicatinf Sequential Processes.* Prentice-Hall Internaltional, 1985.

[15] Loucopoulos, P., Kavakli, E. *Enterprise Modelling and the Telelogical Approach to Requirements Engineering.* International Journal of Intelligent and Cooperative Information Systems, 4(1):45-79, 1995.

[16] David M. Weiss: *Defining families: commonality analysis,* Lucent Technologies Bell Laboratories, 1997.

[17] J. L. Bentley: *Programming Pearls: Little Languages,* Communications of the ACM,29(8), 711-721, August 1986.

[18] D. Bruce: *What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation*. In Kamin [43], pages 17-35.

[19] J. Heering: *Application Software, Domain-Specific Languages and Language Design Assistants*, Software Engineering(SEN), SEN-R0010 May 31, 2000.

[20] A. van Deursen, P. Klint, J.M.W. Visser: *Domain-Specific Languages,* Software Engineering(SEN), SEN-R0032 November 30, 2000.

[21] Scott Thilbault: *Domain-Specific Languages: Conception, Implementation and Application,* Phd Thesis, Univeristy de Rennes, October 1998.

[22] Scott Thilbault, Renaud Marlet, Charles Consel: *A Domain Specific Language for Video Device Drivers: from Design to Implementation,* Univeristy de Rennes, 1997.

[23] S.J. Greenspan, A. Borgida and J. Mylopoulos: *A Requirements Modeling Language and its Logic*, Information Systems, 11(1), 1986,pages 9-23.

[24] James Neighbors.: *Software Construction Using Components. PhD thesis, University of California, Irvine, 1980.*

[25] R. McCain.: *Reusable Software Component Construction: A Product-Oriented Paradigm.* In Proceedings of the 5th AIAA/ACM/NASA/IEEE Computers in Aerospace Conference, Long Beach, CA, pp: 125-135, October 21-23, 1985.

[26] Ruben Prieto-Diaz: *Domain Analysis: An Introduction.* Software Engineering Notes, 15(2), April 1990.

[27] A. Dardenne, Axel van Lamsweerde, and S. Fikas. Goal*:Directed Requirements Acquisition. Science of Computer Programming*, 20:3-50, 1993.

[28] R. Darimont and Axel van Lamsweerde. *Formal Renement Patterns for Goal:Driven Requirements Elaboration.* In Proc. FSE'4, Fourth ACM SIGSOFT Symp. on the Foundations of Software Enginering, pages 179-190. ACM, October 1996.

[29] Axel van Lamsweerde and L. Willemet. *Inferring Declarative Requirements Specication from Operational Scenarios*. IEEE Transaction on Software Engineering, pages 1089-1114, 1998.

[30] *Special Issue on Scenario Management.* IEEE Trans. on Software Engineering, December 1998.

[31] Pamela Zave and Michael A. Jackson. *Techniques for partial specication and specication of switching systems*. In S. Prehn and W.J. Toetenel, editors, VDM'91: Formal Software Development Methods, volume 551 of LNCS, pages 511-525. Springer-Verlag, 1991.

[32] Pamela Zave and Michael A. Jackson. *Requirements for telecommunications services: an attack on complexity.* In Proceedings of the Third IEEE International Symposium on Requirements Engineering (Cat. No.97TB100086), pages 106{117. IEEE Comput. Soc. Press, 1997.

[33] Pamela Zave. *Classication of Research Eorts in Requirements Engineering.* ACM Computing Surveys, 29(4):315{321, 1997.

[34] B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships between Multiple Views in Requirements Specications. IEEE Transactions on Software Engineering, 20(10):760-773, October 1994.

[35] John Mylopoulos, L. Chung, and E. Yu. From Object-Oriented to Goal-Oriented Requirements Analysis. CACM: Communications of the ACM, 42(1):31-37, January 1999.

[36] A. Hunter and B. Nuseibeh. Managing Inconsistent Specications: Reasoning, Analysis and Action. ACM Transactions on Software Engineering and Methodology, 7(4):335-367, October 1998.

[37] Axel van Lamsweerde, R. Darimont, and E. Letier. Managing Con in Goal-Driven Requirements Engineering. IEEE Transaction on Software Engineering, 1998. Special Issue on Inconsistency Management in Software Development.

[38] Axel van Lamsweerde and E. Letier. Integrating Obstacles in Goal-Driven Requirements Engineering. In Proc. ICSE-98: 20th International Conference on Software Enginereering, Kyoto, Japan, April 1998. IEEE Computer Society Press.

[39] Axel van Lamsweerde and L. Willemet. Handling Obstacles in Goal{Driven Requirements Engineering. IEEE Transaction on Software Engineering, 2000. Special Issue on Exception Handling.

[40] Jackson, M. , Zave, P.: *Domain Descriptions.* 1st International Symposium on Requirements Engineering (RE'93), San Diego, USA, 4-6 January 1993, pp. 56-64.

[41] Dines Bjorner: *The SE Book,The ABZ of The Theory and Practice of Software Engineering*

[42] Janet Suleski, Catherine Quirk: *Supply Chain Event Management: The antidote for Next Year's Supply Chain Pain*, AMR Research Inc. 2001.

[43] Formal Methods Specification and Verification Guidebook For Software and Computer Systems, Volume II: A Practitioner's Companion, May, 1997, NASA

[44] The RAISE SPECIFICATION language, The RAISE Language Group

[45] Concurrent and Real-time Systems, The CSP Approach, Steve Schneider

[46] M.A. Jackson, *Problem Analysis Using Small Problem Frames,* South African Computer Journal 22, Special Issue on WOFACS'98, pp47-60, 1999.

[47] J.V. Guttag, J.J. Horning, and J.M. Wing, *"Some Remarks on Putting Formal Specifications to Productive Use.*", Science of Computer Programming, North-Holland, Vol. 2, No. 1, Oct. 1982, pp. 53-68.