



Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

**Extração Automática de Modelos CSP a
Partir de Casos de Uso**

Renata Bezerra e Silva de Araújo

Dissertação de Mestrado

Recife
18 de março de 2011

Universidade Federal de Pernambuco
Centro de Informática

Renata Bezerra e Silva de Araújo

Extração Automática de Modelos CSP a Partir de Casos de Uso

*Trabalho apresentado ao Programa de Pós-graduação em
Ciência da Computação do Centro de Informática da Uni-
versidade Federal de Pernambuco como requisito parcial
para obtenção do grau de Mestre em Ciência da Com-
putação.*

Orientador: *Prof. Dr. Juliano Manabu Iyoda*
Co-orientador: *Prof. Dr. Augusto César Alves Sampaio*

Recife
18 de março de 2011

To everyone I love

Acknowledgements

I would like to sincerely thank everyone that, somehow, has contributed to the existence of this work, either helping me directly or simply being part of my life and making me the person I am today.

First, I want to thank God for being always by my side during this journey, giving me strength in all moments I needed it.

I am also eternally grateful to my parents (“Lula” and “Cema”). Their support, advices, incentive, talks, jokes and unconditional love have encouraged me more and more to get here.

I do thank my family, that I love so much; special thanks for my grandmother, “vó Maria”, my brother Fábio, my cousins “Ló”, Zilá and Márcio and my brother’s wife, Juliana.

Thanks also to my boyfriend Pablo, who has brought a lot of joy to my life and had supported me with affection and love during difficult moments. After almost 6 years of friendship and more than one year of dating, we were able to construct a strong relationship, based on respect, kindness, fellowship and most importantly, love.

I really need to thank all my old friends that I love more each day. Special thanks to “Nando”, Renata, “Tati” and of course, “Lidinho”, who had provided me an entire year full of complicity, true friendship, support, fellowship and advices, during the time we have shared the same house.

I offer my thanks also for the all friends that I have made during my academic life. Special thanks to Aline, “Gras”, Thiago Burgo (“Thi”) and of course, Luciano (“Lulu”), who is always by my side when I need anything, and also helped me a lot in the beginning of the development of this work.

Finally, I want to thank my advisor, Juliano Iyoda, and my co-advisor, Augusto Sampaio, for their trust, support and great help. Also, I cannot be more grateful to Sidney Nogueira for his patience, dedication and support, in all moments I needed him to make this work achievable.

Apologies if I have forgotten someone here, I am not less thankful if it had happened. For all of you, my sincere thanks, I will be forever grateful. All of you have inspired me some way.

Live as if you were to die tomorrow. Learn as if you were to live forever.

—MAHATMA GANDHI

Resumo

No ciclo de vida de desenvolvimento de software, especificação de requisitos é uma atividade muito propensa a definições incorretas. Isto geralmente acontece porque esses documentos são normalmente escritos em linguagem natural, tornando muito alta a possibilidade de introduzir ambiguidades e interpretações errôneas. Por outro lado, a utilização de linguagem natural traz simplicidade e flexibilidade ao se especificar requisitos, considerando que esta é uma notação que pode ser compreendida tanto pelo cliente quanto pelo desenvolvedor.

Uma vez que projetos de software possuem documentos precisos, engenheiros de software que tenham bom conhecimento em linguagens formais podem criar manualmente uma especificação formal com o propósito de validar as propriedades do sistema. No entanto, esta criação manual pode não cobrir todos os requisitos ou podem conter inconsistências. Desta forma, a geração automática de modelos formais a partir de documento de requisitos parece ser uma boa solução para este problema. Para alcançar este objetivo, os documentos de requisitos devem ser simples, diretos, uniformes e sem ambiguidades. Para que isto aconteça, Linguagens Naturais Controladas (Controlled Natural Languages - CNL) são comumente utilizadas.

Este trabalho faz parte do projeto de Pesquisa e Desenvolvimento do CIn Brazil Test Center (CInBTCRD), que é uma cooperação entre a Motorola e o Centro de Informática da Universidade Federal de Pernambuco (CIn-UFPE). Em primeiro lugar, este trabalho propõe uma linguagem restrita (CNL) para definir casos de uso contendo uma noção de estado, os quais consideram dados de entrada, saída, guarda e atualização de variáveis, como um complemento para a descrição textual. Depois disso, uma tradução automática dessa linguagem para a álgebra de processos CSP foi proposta, a fim de permitir a análise formal de requisitos e geração de casos de teste. Finalmente, foi realizada a implementação e integração desta linguagem e sua tradução para CSP em uma ferramenta conhecida como TaRGeT, cujo propósito é a geração de casos de teste a partir de documentos de casos de uso que seguem um template padrão e são escritos utilizando uma CNL. A TaRGeT original não era capaz de lidar com definições de dados e as manipulações destes dados, e utiliza sistemas rotulados por transição (labelled transition systems) em vez de CSP, como formalismo.

Para ilustrar as técnicas propostas neste trabalho, um estudo de caso foi realizado no ambiente da Motorola, adaptando um exemplo de caso de uso real da indústria de modo a encaixá-lo no nosso template. O documento de caso de uso considera situações de envio e recebimento de SMS/MMS, contendo uma *feature* com 7 casos de uso, incluindo definições e manipulações de dados, relacionamentos entre casos de uso e 6 fluxos alternativos. O CSP gerado contém 570 linhas de código e a verificação de suas propriedades foi checada com sucesso utilizando-se a ferramenta FDR, um verificador de modelo para CSP.

Palavras-chave: Especificação de Casos de Uso, Linguagem Natural Controlada, Geração de Especificação Formal, CSP

Abstract

In the software development life cycle, requirements specification is an activity very prone to incorrect definitions. This commonly happens because these documents are usually written in natural language, making the possibility of introducing ambiguities and unclear interpretation very high. On the other hand, the use of natural language brings simplicity and flexibility when specifying requirements, once this is a notation which can be understood by both the customer and the system developer.

Once a project has precise documents, software engineers that have good knowledge in formal languages can manually create a formal specification in order to validate the system properties. The manual creation of formal models, nevertheless, may not cover all requirements or may contain inconsistencies. Thus, the automatic generation of formal models from requirements documents seems to be an effective solution for this problem. In order to automate the construction of formal models, the requirements documents should be simple, direct, unambiguous and uniform. In order to make this feasible, Controlled Natural Languages (CNL), with a precise syntax and semantics, can be used.

This research is part of the CIn Brazil Test Center Research and Development (CInBTCRD) team, which is a cooperation between Motorola Inc. and the Informatics Centre of the Federal University of Pernambuco (CIn-UFPE). In the first place, this work proposes a restricted language (a CNL) to define state based use cases, which considers input, output, guard and state update, as a complement to the textual description. After that, an automatic translation of this language to the CSP process algebra was proposed in order to allow formal analysis of requirements and test case generation. Finally, we implemented and integrated this language and its CSP translation in a framework of tools, known as TaRGeT, whose purpose is the generation of test cases from use case documents written in a CNL and following a specific template. The original TaRGeT was not able to deal with data definitions and manipulations, and uses labeled transition systems, rather than CSP, as formalism.

To illustrate the techniques proposed in this work, a case study was performed in Motorola's environment, adapting a real industrial use case example to fit in our template. The context of the use case document considers situations in the act of sending and receiving SMS/MMS; it contains a feature with 7 use cases, including data definitions and manipulations, relationship among use cases and 6 alternative flows. The generated CSP contains 570 lines of codes and the verification of its properties was successfully checked using the FDR tool, a model checker for CSP.

Keywords: Use case specification, Controlled Natural Language, Formal specification generation, CSP

Contents

1	Introduction	1
1.1	Objectives and Context	2
1.2	Dissertation Organization	4
2	Background	5
2.1	Formal Methods	5
2.1.1	CSP Notation	6
2.2	TaRGeT Overview	9
2.2.1	Use Case Specification Template	10
2.2.2	Feature and Use Case	10
2.2.3	Execution Flow	11
2.2.4	Relationship among use cases	12
2.3	Concluding Remarks	14
3	Use Case Extension	15
3.1	Extended Use Case Specification Overview	15
3.1.1	Data Definition	15
3.1.1.1	Name Type	16
3.1.1.2	New Type	18
3.1.1.3	Constants and Variables	19
3.1.2	Flow Step	20
3.1.2.1	Inputs	21
3.1.2.2	State Guard	22
3.1.2.3	Variable Assignments and Output Values	22
3.1.3	Use Case Relations	23
3.2	Complete Extended Use Case Template	23
3.3	Concluding Remarks	26
4	Implementation	27
4.1	XML Schema Extension	28
4.2	Architecture Extension	31
4.3	CSP Model Generation	34
4.3.1	Implementation of the CSP translation	39
4.4	Concluding Remarks	41

5	Case Study	42
5.1	Feature Types, Variables and Constants	42
5.2	Use Case UC1	44
5.3	Use Case UC2	44
5.4	Use Case UC3	48
5.5	Use Case UC4	49
5.6	Use Case UC5	50
5.7	Use Case UC6	51
5.8	Use Case UC7	52
5.9	Concluding remarks	54
6	Related Works	56
6.1	Processable specification generation from requirements	56
6.1.1	Supporting use case based requirements engineering	56
6.1.2	Attempto — From Specifications in Controlled Natural Language towards Executable Specifications	57
6.1.3	Automatic Transformation of Natural Language Requirements into Formal Specifications	61
6.1.4	Autonomous Requirements Specification Processing Using Natural Language Processing	63
6.1.5	Improved Processing of Textual Use Cases: Deriving Behaviour Specifications	65
6.2	Test cases generation from requirements	65
6.2.1	Automated Formal Specification Generation and Refinement from Requirement Documents	66
6.2.2	UML-Based Statistical Test Case Generation	67
6.2.3	An Approach for Supporting System-level Test Scenarios Generation from Textual Use Cases	68
6.2.4	Automatic Test Generation: A Use Case Driven Approach	68
6.3	Discussion	70
7	Conclusion and Future Work	72
7.1	Future Work	73
A	Complete BNF	76
B	Complete XML Schema for use case template	80
C	CSPm generated from <i>Important Messages</i> feature	90
D	CSPm generated from <i>Sending and Receiving SMS/MMS</i> feature	97

List of Figures

1.1	CInBTCRD initiatives overview	3
2.1	Use case specification example	11
2.2	Use case specification template	13
2.3	Use case specification template	14
3.1	Use case specification example with data	16
3.2	Data Definition template	17
3.3	BNF for a Name Type definition	17
3.4	Name Type definitions example	18
3.5	BNF for a New Type definition	18
3.6	New Type definitions example	18
3.7	BNF for Constant and Variable definitions	20
3.8	Constant and Variable definitions example	21
3.9	Extended flow step template	21
3.10	BNF for defining inputs	21
3.11	Input example	22
3.12	Guard example	22
3.13	BNF for defining assignments and outputs	23
3.14	Assignment and output example	23
3.15	Complete use case specification template	24
3.16	Important Messages Feature in the new template	25
4.1	TaRGeT Architecture	27
4.2	Relationship among use cases	28
4.3	Relationship among use cases in XML Schema	28
4.4	Data Definition	29
4.5	Data Definition in XML Schema	29
4.6	Flow step	30
4.7	Flow step in XML Schema	30
4.8	Class diagram for relationship among use cases	31
4.9	Class diagram for data definition	32
4.10	Class diagram for Name Type and New Type elements	32
4.11	Class diagram for expression	33
4.12	Class diagram for the extended flow step	33
4.13	Structure of the CSP model	34

4.14	CSP translation of Important Messages data definition	35
4.15	Generated CSP of UC02 flow	36
4.16	Generated CSP of UC03 memory, extension and flow	37
4.17	UC03 process	38
4.18	CSPGenerator templates	39
4.19	Data definition CSP code example	40
4.20	Flow step CSP code example	41
5.1	Data Definition of the feature 33629	43
5.2	Data Definition CSP Model of the feature 33629	43
5.3	Use case UC1 of the feature 33629	44
5.4	Generated CSP Model of UC1	44
5.5	Use case UC2 of the feature 33629	45
5.6	Generated CSP Model of UC2 - Part 1	46
5.7	Generated CSP Model of UC2 - Part 2	47
5.8	Use case UC3 of the feature 33629	48
5.9	Generated CSP Model of UC3	49
5.10	Use case UC4 of the feature 33629	50
5.11	Generated CSP Model of UC4	50
5.12	Use case UC5 of the feature 33629	51
5.13	Generated CSP Model of UC5	51
5.14	Use case UC6 of the feature 33629	52
5.15	Generated CSP Model of UC6	52
5.16	Use case UC7 of the feature 33629	53
5.17	Generated CSP Model of UC7	54
6.1	Example of use case diagram for a PM system	57
6.2	Use case describing a login procedure in a PM system	58
6.3	Extension use case	58
6.4	Grammar for conditions	59
6.5	Grammar for use case operations	59
6.6	Small excerpt of the SimpleMat specification	60
6.7	Insertion of a new substantive example	60
6.8	Specification example	61
6.9	Specific information required from the user	61
6.10	Step-by-step execution	62
6.11	ATM requirements specification	63
6.12	TLG specification of the ATM	64
6.13	Parse tree example	64
6.14	Proposed strategy overall process	66
6.15	Activities of the proposed approach within the software development process	67
6.16	Global methodology for requirement-based testing	69
6.17	Contracts of use cases open and close	70

LIST OF FIGURES

xii

7.1	Parameterization example	74
7.2	Use Case Main Flow Example	75

List of Tables

6.1 Comparison among related works	71
------------------------------------	----

CHAPTER 1

Introduction

Software development encompasses an extreme competitive market. Given that the system quality is an important fact to guarantee the company position in the market, great effort has been dedicated to ensure the product quality and customer satisfaction. The insertion of errors during the software development process decreases the final quality of the products and increases considerably the cost of fixing those errors.

The cost to make a modification in an implemented system, resulting from a requirement problem, is much greater than a modification during the design, requirements, analysis, or code phases [Som03b]. As said in [BBL76], the sooner a problem is found during the software development cycle the least expensive is to fix it. In this way, requirements elicitation is considered a fundamental task.

Once a project has precise documents, software engineers that have good knowledge in formal languages can create a formal specification in order to validate the system properties. Nevertheless, the manual creation of formal models may not cover all requirements or may lead to inconsistencies. Thus, the necessity of automatically generating formal models from requirements documents seems to be an important task to be accomplished. In order to automate the construction of formal models, the requirements documents should be simple, direct, unambiguous and uniform.

In order to make this feasible, an important ingredient in the process is the choice of language used to write requirements. Typically, a graphical or a textual language with a restricted syntax is adopted. The textual languages are normally called Controlled Natural Languages (CNL) [SLH03]. They contain a smaller and restricted grammar than the natural languages. Thus, they prevent the writer from introducing ambiguous and non-uniform sentences, retain readability and understandability by all stakeholders and also allow an automated derivation of formal specifications.

There are some academic researches about this automatic generation of formal models from requirements. Somé's work [Som06] proposes a restricted form of natural language for use cases such that automated derivation of specification is possible. Lee's [Lee01] work proposed to realize an automatic conversion of a requirements document written in a natural language into a formal specification language. The work of Cabral et. al. [CS08] proposes a Controlled Natural Language (CNL) used to write use case specifications according to a template. From these use cases, a complete strategy and tool enable the generation of process algebraic formal models in the CSP notation (our work belongs to the same project as this work, where our strategy is based on it). The details of these works and others more can be seen in Chapter 6.

Software testing is another very important activity in order to guarantee the quality of the products. Due to the enormous possibility of injecting human failures and its associated costs,

a really careful and well planned testing process is definitely necessary. The main role of software testing is to find defects in the product so that the development team can fix them on time, before the product reaches the customer. It can be done by verifying if all implemented requirements are according to their specification and by producing test cases which have high probability of revealing an error that was not found yet with reduced time and effort.

In this way, software testing is a hard task and can reach 50% of the total software development cost [Mye04]. In order to try to reduce the time dedicated to this activity a lot has been invested in test automation. The work of Elfriede Dustin *et. al.* [DRP99] shows that the automation of testing process activities leads to an average decrease of 75% of the total effort comparing to the effort of the same activities being performed manually.

There are a lot of approaches to automatic test generation based on a formal representation of the system [BBC⁺03]. Such approaches are based on a formal specification of the system and produce as output a set of formal test cases, where the traditional elements of a test case are written in some formal notation. One of the benefits of using formal test cases is the ability to guarantee properties about the tests created and being mechanically translated to some other notation more suitable for manual or automatic execution of tests on an implementation.

The work of Cabral *et. al.* [CS08], previously cited, also considers an automatic test generation having the CSP specification of the use case as input. Another work following this direction is the work of Nebut *et. al.* [NFTJ06], which proposes an approach for automating the generation of system test scenarios from use cases in the context of object-oriented embedded software. Another research in this context is Riebisch's work [RPG03]. It proposes an approach for generating system-level test cases based on UML use case models which can be further refined by state diagrams. The details of these works and others are described in Chapter 6.

1.1 Objectives and Context

This work is part of the CIn Brazil Test Center Research and Development project (CInBTCRD), which is a cooperation between Motorola Inc. and the informatics centre of the Federal University of Pernambuco (CIn-UFPE). The CInBTCRD focuses mainly on the definition of an integrated process for the generation, selection and evaluation of test cases for mobile applications. Figure 1.1 provides a vision of the main research project.

Starting from a use case template written using a restricted form of natural language (CNL), or from a UML specification, test cases can be automatically generated, calculating the test execution effort as well as analyzing code coverage. As Figure 1.1 illustrates, the test cases are generated from an intermediate notation, which in our case is the CSP [RHB97, Hoa85] process algebraic notation. This model is automatically generated based on the requirements.

The use cases are written in a CNL with a fixed grammar in order to allow the automatic and mechanized translation into CSP (the intermediate representation). As the context of this work is a research cooperation between CIn/UFPE and Motorola, related to mobile applications testing, the proposed CNL [TLB06] reflects this domain, but they can also be used to describe other domains of applications. The formal specification generated in CSP is used in the project as an internal model to the automatic generation of test cases.

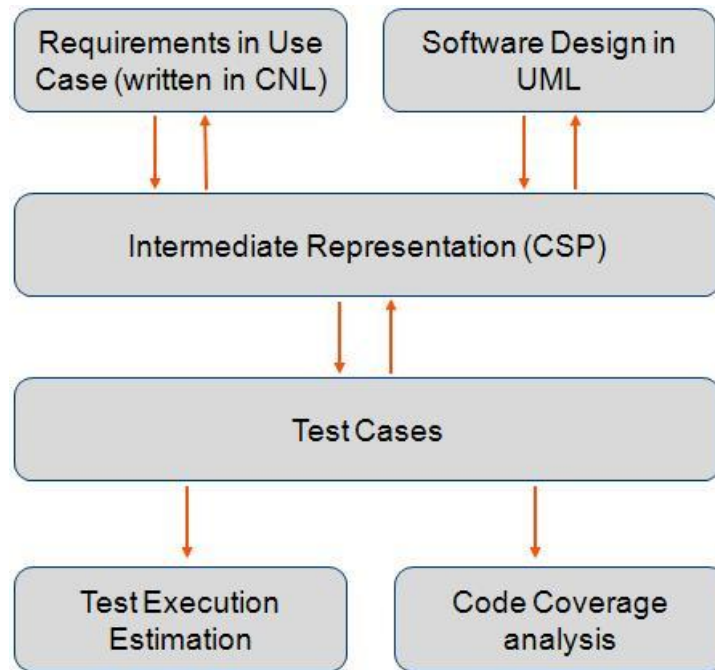


Figure 1.1 CInBTCRD initiatives overview

This work focuses on the following activities: definition of a notation for describing use cases using a CNL and automatic generation of a model in the CSP notation. Here, we propose a strategy that initially extends the current use case template used in CInBTCRD project (introduced in next chapter, Section 2.2.1) so as to consider data definitions and manipulation. Actually, the existing CNL used to describe use cases remains the same. We specifically propose a mix between the original notation with data fields and state manipulation (these new fields are introduced separated from the CNL sentences and so, they do not impacted in the CNL).

In this way, the first contribution is the definition of a language that allows a state based use case specification considering data (inputs, outputs, guards, variables, constants and types). The intention is that this language be as close to a natural language as possible, in order to allow requirements analysts to express the requirements models in a precise but user-friendly way.

The second contribution is the implementation of the automatic generation of the intermediate representation in CSP having the extended use case template (already considering data) as input. This project is integrated with a framework of tools, known as TaRGeT [CU10, NCT⁺07], which has been developed in the CInBTCRD context. This tool will be better explained in the next chapter.

1.2 Dissertation Organization

- Chapter 2 introduces an overview of the CSP notation and the TaRGeT tool, and provides the initial use case template.
- Chapter 3 presents the proposed use case template, where the notion of state is now considered. Every clause of the language to support data is explained in details.
- Chapter 4 shows how the mechanization of the CSP generation from a use case specification following the new template was implemented.
- Chapter 5 introduces a case study performed in order to investigate our strategy within a real application context.
- Chapter 6 discusses some works that are somehow related to ours.
- Chapter 7 finally summarizes our contributions and suggests topics for further research.

Background

According to the UML [OMG10] specification, a use case is “the specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system”. Every use case provides one or more scenarios that describe how the system interacts with end users or other systems. Use cases can be used to validate the system’s architecture and are a fundamental source to perform system tests.

Use case specifications capture system behaviour, possibly at different levels of abstraction. Therefore, depending on the developer’s need, use cases are created for different purposes. Nowadays, use cases are generally specified using a natural language. While on the one hand the use of natural language is quite simple and attractive, on the other hand, it is known that natural languages are prone to ambiguities and are also hard to be supported by automation tools. The latter issue is critical in the context of making the software development faster, in particular, for test case generation, which has a significant reduction in the time spent.

In order to make sure that use case specifications are uniform and have no ambiguities, Controlled Natural Languages (CNLs) can be used to describe them. They can be seen as subsets of natural languages with a small and more restricted grammar, making it possible to accomplish formal validations and transformations while still being easy to use and understand — features not possible using a formal language.

This chapter introduces some background to help understanding the strategy developed in this work. First of all, an introduction to formal methods focusing on the chosen notation (CSP) is presented (Section 2.1). Then, Section 2.2 introduces an overview of the TaRGeT tool [CU10, NCT⁺07] and the current template for use case specifications (written with a CNL) adopted in the Motorola context, where no data is considered.

2.1 Formal Methods

Formal methods means the mathematics and modeling applicable to the specification, design, and verification of software. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design.

Industrial-quality model checkers and advanced theorem provers make it possible to do sophisticated analyses of formal specifications in an automated or semi-automated mode, making these tools attractive for commercial use. Given a finite model of a system, it is possible to test automatically whether this model meets a given specification.

The choice of which notation to use is somewhat application dependent. The use of a no-

tation such as Z [Spi92] or VDM [Jon90] is indicated when state based aspects are specified. On the other hand, when there is the presence of distributed or concurrent characteristics in the system, it is better to use a process algebra, which is useful to specify and design concurrent and distributed system's behaviour of hardware and software. Examples of such notation are: CSP (Communicating Sequential Processes) [RHB97] and CCS (Communication and Concurrency) [Mil89].

CSP offers a rich set of operators that allows us to model use cases as communicating processes. We also model state information as separate memory processes whose data can be read and updated by the use case processes. Furthermore, the FDR tool [Gar97] is a model checker for CSP that allows the automation of a strategy to generate test cases as counter-examples of model verification.

Considering that our purpose is to introduce a notion of state (with memories) in a use case specification, CSP seems to be an interesting formal language for our goal; it provides us with a formalism that make possible the modeling of use cases communicating with local and global memories as well a refinement checker FDR [Gar97], a tool for automatic model checking and test case generation. That is why this process algebra seems to be an interesting choice in the context of our work. Now let us have a look in the CSP notation.

2.1.1 CSP Notation

CSP is a formal language for describing patterns of interaction in concurrent systems [Ros97]. CSP allows the description of systems in terms of component processes that operate independently, and interact with each other through message-passing communication. The language of CSP was designed for describing systems of interacting components and is supported by an underlying theory for reasoning about them [Sch99].

The conceptual framework taken by CSP is to consider components, or processes, as independent self-contained entities with particular interfaces through which they interact with their environment. This viewpoint is compositional, in the sense that if two processes are combined to form a larger system, that system is again a self-contained entity with a particular interface: a larger process.

The relationships between different processes, and the way each process communicates with its environment, are described using various process algebraic operators. Using this algebraic approach, quite complex process descriptions can be easily constructed from few primitive elements.

The interface of a process is described as a set of events. An event describes a particular kind of atomic indivisible action that can be performed or suffered by the process. In describing a process, the first issue to be decided must be the set of events that the process can perform. This set provides the framework for the description of the process. Each event has a unique name. One event may occur many times in the process behaviour. Along with events, datatypes are defined to structure the data transmitted between events. The set of datatypes and events, defined during the formal model specification of a specific domain application, is called an alphabet.

The behaviour of a CSP process is described in terms of events, which are instantaneous actions, like OPEN or CLOSE, that may transmit information. A primitive process can be

understood as a representation of a basic behaviour. There are two primitive processes: *Stop* and *Skip*. *Stop* is the process that does not communicate anything and is used to describe a broken system, as well as a deadlock situation. *Skip* is the process that always terminates successfully.

Some of the CSP operators and constructors will now be presented. Actually, some of the following constructions follow the ASCII (textual) form of CSP (CSPm), which can be readable and processable by a machine. The complete set of these constructors and operators can be found in the works of Roscoe [RHB97] or Hoare [Hoa85]. The structure of the explanation presented here is based on the master's thesis of Cabral [Cab06].

- **Datatype**

Whenever some information needs to be transmitted it is necessary to define its type and possible values through a datatype definition. Therefore, complex datatypes can be defined and used in the channel definitions (events with data transferring) to specify the communication channel. Now we introduce a simple example of a datatype declaration: *datatype SimpleColour = Red | Green | Blue*, where *SimpleColour* is the defined type and *Red*, *Green* and *Blue* are its possible values. Actually, this constructor is used in CSPm; in CSP there is no need of using the word *datatype*.

- **Nametype**

Nametype definitions associate a name with a type expression. An example of a name-type declaration is: *nametype SomeColours = Red, Green*, which associates the id *SomeColours* with some elements of the type *SimpleColour* (defined above). Like the previous constructor, the word *nametype* is used only in CSPm.

- **Channel**

An event is specified by the *channel* constructor. When an event is defined it is possible to determine if it will only represent an event or if it will communicate some data (a proper channel). An example of two channels declaration is: *channel in, out : SimpleColour*, where the channels *in* and *out* both carry a *SimpleColour*.

- **Process**

Processes are the basic unit to capture behaviour. Each process is defined by equations and, in general, a set of processes is used to specify a large system. Processes communicate with each other through synchronization in their events, and as explained, this communication may or may not carry data. The primitive processes *Skip* and *Stop* are terminal processes (no communications happen). Another kind of process constructors can be done using the operators explained below.

- **Prefix**

The prefix is the simpler operation involving a process. It defines a process engagement on an event and then the process behaviour is like the suffixed process. Let x be an event and P a process, then $x \rightarrow P$ represents the process that waits indefinitely by x , and then behaves like the process P .

- **Communication**

A process can pass to or receive information from other processes. In CSP, a communication is represented by a pair $c.v$ where c is the channel name and v is the message value sent by the channel. The process $P = c!v_1 \rightarrow c?v_2 \rightarrow Q$ initially sends v_1 through the channel c , receives the value v_2 through the channel v_2 and then behaves like Q . A channel can also send and receive information simultaneously, such as the process $P = c!v_1?v_2 \rightarrow Q$, where it simultaneously sends v_1 and receives v_2 through the channel c and then behaves like Q .

- **Sequential Composition**

Processes execute when they are invoked, and it is possible that they continue to execute indefinitely, retaining control over execution. It is also possible that control may pass to a second process because the first process reaches a particular point in its execution where it is ready to pass control. The mechanism for transferring control from a terminated process to another process is sequential composition. The sequential composition operator used is $(;)$. The process $P = A; B$ initially behaves like A , and then like B whenever A terminates successfully (*Skip*).

- **Internal and External Choices**

There are also the operators for external choice and internal choice. The operator \square is the external choice. In this case, the environment controls the choice between the options of behaviour. The process $a \rightarrow P \square b \rightarrow Q$ tries to communicate the initial events a and b . If the environment accepts to communicate a , the process starts to behave like P . On the other hand, if the environment accepts to communicate b , the process starts to behave like Q . The operator \sqcap represents the external choice in CSPm. The operator \sqcup is the internal choice. This operator is similar to the previous operator but denotes a process whose choice is made internally by the process with no control from the environment. The process $P \sqcup Q$ means that the choice between P and Q is non-deterministically defined. The process behaves like P or Q , arbitrarily. The operator $|\sim|$ represents the internal choice in CSPm.

- **Parallel Composition**

When two processes are put in concurrent execution, in general, we hope they could interact with each other. The interactions can be viewed as events that require the simultaneous participation of both processes. Let P and Q be processes with the same alphabet, $P \parallel Q$ represents a process in which P and Q must be synchronized in all events. So an event x only occurs when both processes are ready to accept it. The process $P \parallel [X] Q$ synchronizes P and Q in the events of the set X . P and Q can interact independently with the environment through the events outside the set X . The process $P \parallel\parallel Q$ allows P and Q to execute concurrently without synchronization between them. Each event offered to the interleave of two processes occurs only in one of them. If both are ready to accept that event, the choice between the processes is non-deterministic.

- **Interruption**

Consider the process $P \triangle Q$; it indicates that Q can interrupt the behaviour of P if an event offered by Q is communicated. In CSPm, the equivalent operator is $/\backslash$

- **Hiding**

Consider the CSP notation $P \backslash X$; it defines a process which behaves like P, communicating all its events except the events that belong to X, which become internal (invisible). The character “\” stands for the hiding operator, which is the same for CSPm notation.

- **Let ... within clause**

Additionally, consider the CSPm expression $P = \text{let } \dots \text{ within } Q$, which specifies the behavior of the process P as that defined by Q, in the context of local declarations to Q introduced by the *let ... within* clause.

CSP (actually, CSPm) is the notation we use to model use cases. We automatically transform use cases into CSPm processes which are, subsequently, used to generate test cases. In what follows we describe how use cases are specified in the TaRGeT tool (Section 2.2). Chapters 3 and 4 show how this transformation is extended to deal with data and with includes and extends relationships among use cases.

2.2 TaRGeT Overview

TaRGeT [CU10, NCT⁺07] stands for Test and Requirements Generation Tool. It is a tool for automatic test case generation from use case scenarios written in a CNL. TaRGeT automates a systematic approach to deal with requirements and test artifacts in an integrated way. The use cases are written following a XML schema, which was designed to contain the necessary information for generating test procedure, description and related requirements. Moreover, the tool can generate traceability matrices between test cases, use cases and requirements.

Three major aspects distinguish TaRGeT from other behavioural model-based testing tools: 1) the use of test purposes, provided by test engineers, to restrict the number of generated test cases and to focus on test cases that are more critical or relevant to a given task; 2) algorithms for eliminating similar test cases, reducing the test suite size without significant impact on effectiveness; and 3) use cases written in a controlled natural language as system input, which is more natural for engineers when contrasting to formal behaviour specification languages.

As said before, TaRGeT’s main functionality is the automatic generation of test cases from use cases scenarios written in a controlled natural language. Supporting this main functionality, TaRGeT features the following facilities:

- Automatic purpose-based generation of test suites, with adequate coverage criteria;
- Elimination of similar test cases according to parameters informed by the user;
- Selection of requirements or use cases to generate specific test suites;

- Use cases are flexible enough to handle different platforms or domains;
- Automatic generation of traceability matrices: Requirements \times Test Cases, Requirements \times Use Cases and Test Cases \times Use Cases;
- A consistency management that allows to compare test suites and keep the integrity of test cases id;
- A Controlled Natural Language (CNL) that defines some writing rules and a restricted vocabulary in order to avoid authors to introduce ambiguities into their test case steps;
- Two options for use case editing: the TaRGeT editor and Microsoft Word;
- Exporting test case suites in many formats, including the XML format for TestLink (a web-based test management and software testing execution tool);
- Friendly GUI to guide the generation processes.

In what follows we present the use case specification template currently used in TaRGeT. (We call *current* template, or *current* TaRGeT the version of TaRGeT currently being used, which does not contain any of the extensions proposed by this work.) The template presented below does not accept data manipulation yet. The extension for data manipulation is introduced in Chapter 3.

2.2.1 Use Case Specification Template

In this section we present an example according to the current template used by TaRGeT. Although includes and extends relationships among use cases have been recently introduced to the current template, they are not part of the current TaRGeT implementation yet. This development has also been done as part of this dissertation work. The subsequent sections explain the current template in more details.

2.2.2 Feature and Use Case

Use cases are initially grouped to form a feature. As can be seen in Figure 2.1, each feature contains an identification number and a name (in the example we can see the *Important Messages* feature, which contains three use cases - UC01, UC02 and UC03 - and is identified by the 11169 Id). This grouping is convenient for organization purposes. The use case itself also contains an identification number and a name, execution flows and may include lists of inclusions, extensions and extension points (for the relationship among use cases).

A use case can also be tagged as auxiliary, meaning that it is not activated. An auxiliary use case is a dependent functional unit that performs the specified behaviour as a consequence of its relation (inclusion or extension) with some activated use case. To specify an auxiliary use case in the template, it is necessary to enter the mark «auxiliary» in the end of the use case name (see the use case UC01 in Figure 2.1).

11169 – Important Messages**UC01 – Selecting Inbox Messages**

<<auxiliary>>

Main Flow

Step Id	User Action	System State	System Response
1M	Go to Inbox folder.		All Inbox Messages are displayed.
2M	Select inbox message(s).		Message(s) are highlighted.

UC03 – Delete Important Messages**Extends** (Important messages folder is not empty, UC02@Clean up)**Main Flow**

Step Id	Action	System State	System Response
1M	Select important message(s).		Message(s) are highlighted.
2M	Clean up selected message(s).		Clean is performed.

UC02 – Moving Messages from Inbox to Important Messages**Includes** UC01@START
Extension points Clean up: 1A**Main Flow**

Step Id	Action	System State	System Response
1M	Select "Move to Important Messages" option.	Message storage has enough space.	"Message moved to Important Messages folder" is displayed.

Alternative FlowsFrom Step: START
To Step: START

Step Id	Action	System State	System Response
1A	Select "Move to Important Messages" option.	Message storage has not enough space.	"Message storage has not enough space" is displayed, "Clean Up request" is displayed.

Figure 2.1 Use case specification example**2.2.3 Execution Flow**

Usually a use case can be used to specify usage scenarios, depending on user inputs and actions. Hence, each execution flow represents a possible sequence of steps that a user can take. Now we are going to have a look at the execution flow components.

- **Step** - A step is represented by the tuple (user action, system state, system response) and each of them is identified by an identifier (step id). The user action describes an operation accomplished by the user. The system state is an optional condition on the system state just before the user action is executed. Thus, it can be a condition on the current application configuration (setup) or memory status. The system response is a description of the operation result after the user action occurs based on the current system state. In Figure 2.1 we can see the step 1M of use case UC01, for example, which has no condition but has the user action "Go to inbox folder" and the system response "All Inbox Messages are displayed". In the step 1M of UC02 we can note the user action, system state and system response are specified.
- **Flow Types** - Execution flows are categorized as main, alternative or exception flows. The main execution flows represent the use cases happy path, which is a sequence of steps where everything works as expected. An alternative execution flow represents a choice situation; during the execution of a flow, such as the main flow, it may be possible to execute other actions, comprising choices. If an action from an alternative flow is executed, the system continues its execution behaviour according to the new path specification. Alternative flows can also begin from a step of another alternative flow; this enables reuse of specification.

The exception flows specify error scenarios caused by invalid input data or critical system states. Alternative and exception flows are strictly related to the user choices and to the system state conditions. The latter may cause the system to respond differently given the same user action. We can see in Figure 2.1 a main flow in every use case, and use case UC02, which contains an alternative flow.

- **Reference between Execution Flows** - There are situations when a user can choose between different paths. When this happens it is necessary to define one flow for each path. Every execution flow has one starting point, or initial state, and one final state. The `From step` field represents the initial state in which it contains the set of initial steps to be executed, while the `To step` field references the final state, also containing a set of steps, but in this case the last steps to be executed. After the `From step` items are executed, the first step from the specified execution flow is executed up to the last step, and after this last step, the control passes to the steps defined in the `To step` field.

Whenever the `From Step` field is defined as `START`, in the main flow, this means that the use case does not depend on any other, so it can be the starting point of the system usage. The other possibility is when the main flow `From step` field refers to other use case steps, meaning that this flow can be executed after a sequence of events had been performed in the correspondent use case. In the case of the `To step` field, when it is set to `END`, in any execution flow, this flow will terminate successfully after its last step is executed. Subsequently, the user can execute another use case that has the `From step` field set to `START`.

The `From step` and the `To step` fields are essential to define the application navigation and also enable the reuse of existing flows; a new scenario may start from a preexistent step from some flow. Every Main Flow has its `From step` set to `START` and its `To step` set to `END` as default, that is why they do not have to be specified for this flow type, as can be seen in Figure 2.1. The Alternative Flow in UC02 specifies that the initial state of this flow is the beginning of the use case, and when the execution of this flow ends it also comes back to the beginning.

2.2.4 Relationship among use cases

In order to allow relationships among use cases, the template includes a list of inclusions, a list of extensions and another list of extension points in a use case. The field `Includes` indicates where the behaviour of the included use cases will be added in the including use case. We can see an example in Figure 2.1, where UC01 is included after the `START` step of UC02. In this way, each inclusion is done by following the format: *UCID@Position*, where *UCID* represents the ID of the use case being included and *Position* is related to the position (`START`, `END` or a step ID) where this inclusion is done (see the `Includes` field of UC02 at Figure 2.1).

The field `Extension points` defines the extension points of a use case, where an extension point is a location internal to a use case, labeled by a name that allows extensions to add behaviour (optionally) to that point by referring to the label. In Figure 2.1 UC02 defines an extension point labeled `Clean up` after the step 1A. Extensions associated with such an extension point will assume control after the step 1A and resume before its continuation (`START`

step). We can see that the format of an extension is: *Label:Position*, where *Label* is the extension point's label (unique in the list), and *Position* is the step ID (or the words START and END) after which the behaviour of an extension use case associated with *Label* will be added.

The field `Extends` specifies the extension points to where an extension use case adds behaviour. Once again, Figure 2.1 shows that UC03 extends UC02 in the extension point `Clean up` provided the Important Messages folder is not empty. Each extension is a tuple of the form (*Condition*, *UCID@Label*), where *Condition* represents the (optional) condition for the extension, and *UCID@Label* the extension point *Label* in the extended use case *UCID* to where behaviour is added.

Note that use cases can only be related through inclusion and extension relations. The fields `From Step` and `To Step` are local references inside a use case. Figure 2.2 presents the complete use case template, showing all possible constructions of such a specification.

```

<Feature Id> - <Feature Name>
-----
<Use Case Id> - <Use Case Name>
-----
Includes <The use cases included by this use case>
Extension points <The extension points of this use case>
Extends <The use cases extended by this use case>

Main Flow
From Step: <from where this flow starts (step id) >
To Step: <to where this flow goes (step id) >

```

Step Id	User Action	System State	System Response
<Step id>	<Describe here a user action>	<System state related to the specified system response>	<Expected result after user action>

```

Alternative Flows
From Step: <from where this flow starts (step id) >
To Step: <to where this flow goes (step id) >

```

Step Id	Action	System State	System Response
<Step id>	<Describe here a user action>	<System state related to the specified system response>	<Expected result after user action>

```

Exception Flows
From Step: <from where this flow starts (step id) >
To Step: <to where this flow goes (step id) >

```

Step Id	Action	System State	System Response
<Step id>	<Describe here a user action>	<System state related to the specified system response>	<Expected result after user action>

Figure 2.2 Use case specification template

2.3 Concluding Remarks

We can see that a tool such as TaRGeT offers great facilities to test generation. However its use cases and test purposes do not take into consideration input and output data or the notion of state. If a representation like that could be included in the tool, CSP test purposes could describe test scenarios that match particular states of the specification. Based on Cabral's work, Nogueira *et al.* [NSM] proposes a strategy to generate CSP code from a use case document that considers data, and in this strategy, the translation of a complete CNL sentence is captured by a CSP event. The implementation of the CSP generation in our work is based on Nogueira's strategy, which is explained in details in Chapter 4.

In the CinBTCRD research project there was a first attempt of extending this control flow template with a notion of state, where it is possible to define types, constants and variables and to build constructions for state update, input, output and guards. Figure 2.3 shows a small example of this first attempt of specifying use case documents considering the state information. Note how the language used for this purpose is formal and also very difficult to understand and use. Formal specification is mixed together with Controlled Natural Language (see the text between square brackets).

<u>11169 - Important Messages</u>	<u>UC01 - Selecting Messages <<auxiliary>></u>														
Data Definition nametype Natural = {0..2} datatype Message = M.Natural var inbox: $\wp(\text{Message}) = \{M.0, M.1\}$ var selected: $\wp(\text{Message}) = \{\}$	Main Flow <table border="1"> <thead> <tr> <th>Step Id</th> <th>Action</th> <th>System State</th> <th>System Response</th> </tr> </thead> <tbody> <tr> <td>1M</td> <td>Go to inbox folder.</td> <td></td> <td>All inbox Messages are displayed.</td> </tr> <tr> <td>2M</td> <td>Select inbox message(s). [?x : $\wp(\text{inbox})$ - {\emptyset }]</td> <td></td> <td>Message(s) are highlighted. [selected := x]</td> </tr> </tbody> </table>			Step Id	Action	System State	System Response	1M	Go to inbox folder.		All inbox Messages are displayed.	2M	Select inbox message(s). [?x : $\wp(\text{inbox})$ - { \emptyset }]		Message(s) are highlighted. [selected := x]
Step Id	Action	System State	System Response												
1M	Go to inbox folder.		All inbox Messages are displayed.												
2M	Select inbox message(s). [?x : $\wp(\text{inbox})$ - { \emptyset }]		Message(s) are highlighted. [selected := x]												

Figure 2.3 Use case specification template

In Figure 2.3 we can see that it is possible to define some data, in the data definition field. The constructions *nametype* and *datatype* are from the CSPm language. To define variables, it was necessary to use the construction *var*, as can be seen in Figure 2.3. In this case we have two defined variables (*inbox* and *selected*) of the type $\wp(\text{Message})$, representing the power set of Message datatype. In the use case UC01, in step 2M, there is an input in the power set of *inbox* not considering the empty set. The other sentence in brackets represents an assignment of this input value to the variable *selected*.

That is why it was necessary to define a more friendly language aiming at making the specification easier and better to understand. In this way, this is the first contribution of this dissertation work. The next chapter (Chapter 3) presents in details this new language – with its BNF and examples – and the new data supporting use case template.

Use Case Extension

This chapter introduces how the use case specification template is extended to support the notion of state, as well as constructions for state update, input, output and guard. The intention here is to textually describe use cases — considering an explicit notion of state — in a way that it is described in a notation as close to a natural language as possible, unlike the one briefly presented in the end of the previous chapter, and at the same time, it is not so unrestricted as a common natural language.

In other words, the specification for data makes a bridge between a natural language description and a formal specification language, thus, it is more user-friendly and still fits its purpose. Once it is necessary to deal with variables, constants, types, and to manipulate them, the user of the proposed language needs to have some basic notion of a programming language.

3.1 Extended Use Case Specification Overview

This section introduces the extended use case specification, supporting data definition and data manipulation. The template for control was extended with new fields to consider data. Figure 3.1 presents a simple example just to provide an overview of the extended use case specification template. In this example a global constant a is defined with value 2; then, a variable b is declared in the scope of use case UC01 and has its initial value set to 0.

The use case UC01 has only one step just to provide the general idea of how it is possible to define inputs, guards, output and variable assignments. Note that, now, we mix natural language with our language. For example, the command “**Input x from $\{1,2,3,4\}$** ” means to input a value from the set $\{1,2,3,4\}$ - which is an element of the type `SetValue` (it will be introduced later) - and associates this value with the variable x (in the scope of the step). The expression $x + a \geq b$ tests if the sum of the input value and the value of the constant a exceeds the value of the variable b .

Output x outputs the inputed value and **$b := x$** assigns the inputed value to the variable b . This examples gives a flavour of our language. It is by no means a complete example. The subsequent sections explain all the new fields of this template in more details and introduces all constructs of the language.

3.1.1 Data Definition

Every feature has the *Data Definition* field, in which the data related to each feature can be specified. It is possible to define types (Name Types and New Types), constants and variables

1234 - Manipulating numbers**Data Definition**

Constant (Id)	Description	Value
a	Minimum value	2

UC01 - Selecting a number**Data Definition**

Variable (Id)	Description	Initial Value
b	Helps testing state	0

Main Flow

From Step: START

To Step: END

Step Id	User Action	System State	System Response
1M	Select a number. % Input x from {1,2,3,4} %	Selected number passes the test. % $x + b \geq a$ %	The number is displayed. % Output x, b := x %

Figure 3.1 Use case specification example with data

in the scope of the feature. Note that each use case also has this *Data Definition* field, thus representing the data specified in the scope of the use case. These kinds of data can be created by filling the tables shown in Figure 3.2. Its fields contain comments explaining how the template should be filled.

Each table is related to a specific data definition (Name Type, New Type, Constant and Variable), and each line of the table represents a tuple to create them. The first two elements of the tuples are the same for all of them, which are: an Id - an identifier - and a description of the introduced data. The third element depends on the type of data created (they are described in detail below). The idea of using tables to define data was based on the work of Blackburn *et al* [BBF97].

There are primitive types, such as *Bool* (boolean) and *Natural*. User defined types can also be created, by using the New Type construct. There is still the possibility of dealing with sets (we call SetValues), in which their elements can be of *Natural* type or the type of any New Type previously introduced. The following sections describe how each kind of data definition can be specified.

3.1.1.1 Name Type

A Name Type specification associates a name with a subset of elements of an existent type. In other words, it represents a pre-defined subset of elements of a specific type, and this subset is associated with a name. Thus, when this name is referenced, it means that it is just a synonym to the type expression it represents.

Figure 3.3 shows the BNF related to a Name Type definition, which can be represented by

Data Definition

Name Type (Id)	Description	Elements
<Name Type Id>	<A brief description of the Name Type>	<The Name Type elements >

New Type (Id)	Description	Elements
<New Type Id>	<A brief description of the New Type>	<The New Type elements >

Constant (Id)	Description	Value
<Constant Id>	<A brief description of the Constant>	<The Constant value >

Variable (Id)	Description	Initial Value
<Variable Id>	<A brief description of the Variable>	<The Variable initial value >

Figure 3.2 Data Definition template

the tuple (Name Type Id, Description, Elements). The fields in bold are values of the existent types.

```

NameType ::= ID Description NameTypeElements
ID ::= (([a-zA-Z_])+([a-zA-Z_0-9])*)?
Description ::= StringValue
NameTypeElements ::= SetLiteral | SetRange
SetLiteral ::= SetValue
SetRange ::= “[” NaturalValue “,” NaturalValue “]”

```

Figure 3.3 BNF for a Name Type definition

The field *Elements* can be specified in two ways, as can be seen in the BNF. One, which is called *SetRange*, can be used only for the Natural type and is used to specify the values in the range of the two Natural values introduced. This is illustrated in the first Name Type definition of the Figure 3.4, which presents some examples of Name Type definitions. In this first example, the Name Type *SomeNaturals* is composed of the values 0, 1 and 2.

The second way is by properly enumerating the desired elements, which can be of the Natural type or of some already defined New Type (just like the definition of a set value). The other two Name Type definitions in the Figure 3.4 illustrates this case: *SomeNaturals_2* introduces a Name Type with the values 0 and 4, and *SomeMessages* is composed by the values M.0 and M.2.

We assume here that these values have already been defined previously by using the New Type construct (see the next section). This is the only case in which there is no problem of using values that are specified after its use; in all other cases (definitions of New Types, Constants

Name Type (Id)	Description	Elements
SomeNaturals	Message identifier	[0,2]
SomeNaturals_2	Some naturals	{0,4}
SomeMessages	Some messages	{M.0, M.2}

Figure 3.4 Name Type definitions example

and Variables) this is not possible.

3.1.1.2 New Type

A New Type is used to associate a type name with atomic values. Alternatively, values can also be associated with tags. Figure 3.5 shows the BNF related to a New Type definition, which can be represented by the tuple (New Type Id, Description, Elements). Figure 3.6 presents some examples of the New Type definitions.

```

NewType ::= ID Description NewTypeElements
NewTypeElements ::= Enumeration | Indexing | BaseTypeList
Enumeration ::= ID | ID “,” Enumeration
Indexing ::= Tag SetRange
Tag ::= ID
BaseTypeList ::= BaseType | BaseType “|” BaseTypeList
BaseType ::= Tag “.” ID

```

Figure 3.5 BNF for a New Type definition

New Type (Id)	Description	Elements
Color	Some colors	green, blue, red
Phone	Some telephones	Phone[1,3]
Message	Phone message	M.SomeNaturals
ColorPhone	Some colors and phones	A.Color B.Phone

Figure 3.6 New Type definitions example

The New Type elements can be specified in three ways: *Enumeration*, *Indexing* and *BaseTypeList*, as can be seen in Figure 3.5. A New Type that is nothing more than just introducing each value is called *Enumeration*. An example is the New Type *Color*, in Figure 3.6; their

elements are the atomic values: green, blue and red. The *Indexing* and *BaseTypeList* associate values with tags. The *Indexing* associates a tag with the values defined in a range of Natural values. The New Type *Phone* exemplifies this case, where it comprises the elements: Phone1, Phone2 and Phone3. A *BaseTypeList* is just a list of *BaseType*, which is used to create elements by associating a tag with the elements of an existent type (making a disjoint union); this association is done by using a dot, and after the dot the ID required needs to be a Name Type or New Type ID.

The *Message* example illustrates a New Type composed by the elements M.0, M.1 and M.2 (recall that the Id *SomeNaturals* represents a previously defined Name Type containing the values 0, 1 and 2). See that these are the elements used in the definition of the Name Type *SomeMessages*. Another more complex example is the *ColorPhone* New Type, which uses a list with two *BaseType*, thus, it is composed of A.green, A.blue, A.red, B.Phone1, B.Phone2 and B.Phone3.

It is worth emphasizing that the values associated with a New Type Id can not be used as values of any other New Type Id of the whole use case document. After all, when the user define new types, the values of such type can only be associated to such type.

3.1.1.3 Constants and Variables

Constants and Variables are defined by assigning a name with a value expression. The difference between them is just that a Constant remains unchanged in all document while a Variable may have its value changed by an assignment. The Constant can be represented by the tuple (Constant Id, Description, Value) and the Variable is defined by the tuple (Variable Id, Description, Initial Value). Figure 3.7 shows the BNF related to Constant and Variable definitions.

A value (or initial value, in the case of a variable) is represented by an expression. As Figure 3.7 illustrates, an expression can be a single value, a unary expression, a binary expression or an Id (of another Constant or Variable). A single value may be of any of the types already explained (Natural, Boolean, Set or New Type). Now let us have a look at the operators used to build a unary expression; the first four operators described below are related to sets.

- “#” - the cardinality of a set; thus, it is an expression of type Natural.
- “powerSet of” - the power set of a set, which still returns a set.
- “is non-empty” - a boolean that verifies whether a set is non-empty.
- “is empty” - a boolean that verifies whether a set is empty.
- “not” - a boolean that negates another boolean expression.

Concerning binary expressions, the operators “+”, “-” and “*” are overloaded. When used with two expressions of type Natural, they just mean the standard operations of sum, subtraction and multiplication, respectively. If they are applied to sets, they mean union, set difference and intersection of sets, respectively.

The extended language also supports ordering operators “>”, “>=”, “<”, “<=”, which can be used for naturals and sets. Equality operators “=”, “!=” can be used for all types. The operators


```

Constant ::= ID Description Expression

Variable ::= ID Description Expression

Expression ::= Value | UnaryExp | BinaryExp | ID

Value ::= NaturalValue | BoolValue | SetValue | NewTypeValue

UnaryExp ::= "#" Expression
           | "powerSet of" Expression
           | Expression "is non-empty"
           | Expression "is empty"
           | "not" Expression

BinaryExp ::= Expression "+" Expression
           | Expression "-" Expression
           | Expression "*" Expression
           | Expression ">" Expression
           | Expression ">=" Expression
           | Expression "<" Expression
           | Expression "<=" Expression
           | Expression "=" Expression
           | Expression "!=" Expression
           | Expression "and" Expression
           | Expression "or" Expression
           | Expression "is in" Expression
           | Expression "is not in" Expression

```

Figure 3.7 BNF for Constant and Variable definitions

“and” and “or” are used for logical boolean expressions and the operators “is in” and “is not in” are used to verify if a value (first expression) is a member of a set expression.

Figure 3.8 introduces some simple examples to illustrate the definition of constants and variables. There are three constant definitions: the first associates the Id *MAX* with the natural value 2, then the Id *a* is assigned to the boolean value false, and finally, the Id *b* receives the New Type value M.0, which is of the type Message (recall the New Type example in Section 3.1.1.2).

In the variable definitions there are three Ids being associated with a set of Messages: *inbox*, *selected* and *important*. The variable *inbox* is initialized with the set {M.0, M.1}; the variable *selected* is initialized with an empty set; and the variable *important* contains a single element: M.2. The variables *c* and *d* are examples of a case of overloading, where the operator “+” represents the union of two sets (the values of the variables *inbox* and *selected*) (variable *c*), while *d* is assigned to the sum of two natural values (2 and 3).

3.1.2 Flow Step

The step specification of the execution flows was extended in order to support the manipulation of data. Now it is possible to define inputs, outputs, assignments and state guards in the

Constant (Id)	Description	Value
MAX	The maximum number of important messages	2
a	Some boolean	false
b	Some message	M.0

Variable (Id)	Description	Initial Value
inbox	Set of inbox messages	{M.0,M.1}
selected	Set of selected messages	{}
important	Set of important messages	{M.2}
c	Some set of messages	inbox + selected
d	Some natural	2 + 3

Figure 3.8 Constant and Variable definitions example

specification of each flow step. Data fields are enclosed between two “%” characters, and are not part of the CNL sentence. Hence, the proposed template extensions have no impact on the existing CNL [TLB06].

Figure 3.9 illustrates this idea and the next sections explain how the user can write each kind of data use.

Step Id	User Action	System State	System Response
<Step id>	<Describe here a user action> <%Include here a list of inputs %>	<System state related to the specified system response> <% Include here a guard %>	<Expected result after user action> <% Include here a list of outputs and assignments %>

Figure 3.9 Extended flow step template

3.1.2.1 Inputs

Inputs are supplied by actors and are associated with the *User Action* column (Figure 3.9). Figure 3.10 presents the BNF for inputs.

$$\text{InputList} ::= \text{Input} \mid \text{Input} \text{“,”} \text{InputList}$$

$$\text{Input} ::= \text{“Input” ID “from” Expression [“such that” Expression]}$$

Figure 3.10 BNF for defining inputs

A list of inputs is defined by separating each input definition with a comma. Looking at Figure 3.10, it is noteworthy to know that the input variable Id is in the scope of the step. The expression in which the input is realized (the first expression after the word “from”) has to be a set expression.

After this, it is optional (the brackets delimit an optional statement in a BNF) to define a restriction on the input variable. Thus, the expression after “such that” can be used for that purpose, and must be a boolean expression which makes a restriction in the input variable. Figure 3.11 shows an example of an input.

Step Id	User Action	System State	System Response
1M	Select inbox message(s). % Input x from powerSet of inbox such that x is non-empty %		

Figure 3.11 Input example

The input variable x represents the set of non-empty messages selected by the user from the *inbox* (recall that *inbox* is a previously defined variable representing the set of inbox messages). Formally, this input takes a value from the set $\wp(\text{inbox}) - \{\emptyset\}$, where $\wp(S)$ is the power set of S .

3.1.2.2 State Guard

State guard is a condition that enables the step to be performed; it is defined as a boolean expression placed in the *System State* column. Thus a guard is simply a boolean expression. This is an important contribution with respect to the previous template, where conditions are abstractedly captured by events, without any state information. Figure 3.12 illustrates the use of a state guard in the flow step.

Step Id	User Action	System State	System Response
1M		Message storage has enough space. % #(important + selected) <= MAX %	

Figure 3.12 Guard example

The guard of this example specifies the condition to execute the *User Action*: the cardinality of the union of the important messages and the selected messages must not exceed the maximum capacity of the important messages folder.

3.1.2.3 Variable Assignments and Output Values

The *System Response* column can be of two kinds: variable assignments and outputs. Variable assignments specify updates in the system state after the step finalization and output values are data sent back to the user. Figure 3.13 introduces the BNF associated with these commands.

A sequence of assignments and outputs are allowed in the *System Response* by separating them with a comma, as can be seen in the BNF. A variable assignment is specified by the definition of a new expression that will be now associated with the variable ID. In order to output an expression it is just necessary to specify an expression in any format of the expression BNF already presented.

```

OutputAssignList ::= Output
                  | Assign
                  | Output “,” OutputAssignList
                  | Assign “,” OutputAssignList

Output ::= “Output” Expression
Assign ::= ID “:=” Expression

```

Figure 3.13 BNF for defining assignments and outputs

Figure 3.14 illustrates an example involving outputs and assignments. The example shows the variable *important* (set of important messages) being updated to the expression “*important - selected*”, which means to remove the selected messages (variable *selected*) from the important messages folder. Variables not assigned to are assumed unchanged. The output expression *selected* completes the system response message by outputting the number of removed messages.

Step Id	User Action	System State	System Response
1M			Clean is performed. % <i>important</i> := <i>important - selected</i> , Output # <i>selected</i> %

Figure 3.14 Assignment and output example

3.1.3 Use Case Relations

Our use case specification still considers relationship among use cases. There is, however, a small difference with respect to the original template: the condition in the *Extends* relationship can now be expressed with a state guard. For example, the extension *Extends (Important Messages folder is not empty, UC02@Clean up)* can be replaced by *Extends (#important > 0, UC02@Clean up)*. The guard can also be expressed as *important is non-empty*, which is equivalent to *#important > 0*: it tests if the variable *important* has at least one element.

3.2 Complete Extended Use Case Template

Now that the language for data has been explained in detail, the complete template is presented in Figure 3.15, gathering every part of the new template that has been explained separately above. The purpose is to provide the general view of the new use case specification, where the new fields are now included.

Figure 3.16 introduces a complete example for the Feature *Important Messages*, presented in Chapter 2. This example illustrates how the data approach can be used in a real specification. The complete BNF can be found in Appendix A. Now let us have a look at the complete example, which puts together the pieces already explained above. As the *Data Definition* section of

<Feature Id> - <Feature Name>

Data Definition

Name Type (Id)	Description	Elements
<Name Type Id>	<A brief description of the Name Type>	<The Name Type elements >

New Type (Id)	Description	Elements
<New Type Id>	<A brief description of the New Type>	<The New Type elements >

Constant (Id)	Description	Value
<Constant Id>	<A brief description of the Constant>	<The Constant value >

Variable (Id)	Description	Initial Value
<Variable Id>	<A brief description of the Variable>	<The Variable initial value >

<Use Case Id> - <Use Case Name>

Data Definition

<Include here any data definition related to the use case>

Includes <The use cases included by this use case>

Extension points <The extension points of this use case>

Extends <The use cases extended by this use case>

Main Flow

From Step: <from where this flow starts (step id) >

To Step: <to where this flow goes (step id) >

Step Id	User Action	System State	System Response
<Step id>	<Describe here a user action> <Include here a set of inputs>	<System state related to the specified system response> <Include here a guard>	<Expected result after user action> <Include here a set of outputs and assignments>

Alternative Flows

From Step: <from where this flow starts (step id) >

To Step: <to where this flow goes (step id) >

Step Id	Action	System State	System Response
<Step id>	<Describe here a user action> <Include here a set of inputs>	<System state related to the specified system response> <Include here a guard>	<Expected result after user action> <Include here a set of outputs and assignments>

Exception Flows

From Step: <from where this flow starts (step id) >

To Step: <to where this flow goes (step id) >

Step Id	Action	System State	System Response
<Step id>	<Describe here a user action> <Include here a set of inputs>	<System state related to the specified system response> <Include here a guard>	<Expected result after user action> <Include here a set of outputs and assignments>

Figure 3.15 Complete use case specification template

11169 - Important Messages

Data Definition

Name Type (Id)	Description	Elements
SomeNaturals	Message Identifier	[0,2]

New Type (Id)	Description	Elements
Message	Phone Message	M.SomeNaturals

Constant (Id)	Description	Value
MAX	The maximum number of important messages	2

Variable (Id)	Description	Initial Value
inbox	Set of inbox messages	{M.0,M.1}
selected	Set of selected messages	{}
important	Set of important messages	{M.2}

UC03 - Delete Important Messages

Extends (#important > 0, UC02@Clean up)

Data Definitions

Variable	Description	Initial Value
selected	Set of selected messages	{}

Main Flow

Step Id	Action	System State	System Response
1M	Select important message(s). % Input x from powerSet of important such that x is non-empty %		Message(s) are highlighted. % selected:=x %
2M	Clean up selected message(s).		Clean is performed. % important := important - selected)%

UC01 - Selecting Inbox Messages

<<auxiliary>>

Main Flow

Step Id	User Action	System State	System Response
1M	Go to Inbox folder.		All Inbox Messages are displayed.
2M	Select inbox message(s). % Input x from powerSet of inbox such that x is non-empty %		Message(s) are highlighted. % selected:=x %

UC02 - Moving Messages from Inbox to Important Messages

Includes UC01@START

Extension points Clean up: 1A

Main Flow

Step Id	Action	System State	System Response
1M	Select "Move to Important Messages" option.	Message storage has enough space. % #(important + selected) <= MAX %	"Message moved to Important Messages folder" is displayed. % inbox:=inbox - selected, important:=important + selected %

Alternative Flows

From Step: START
To Step: START

Step Id	Action	System State	System Response
1A	Select "Move to Important Messages" option.	Message storage has not enough space. % #(important + selected) > MAX %	"Message storage has not enough space" is displayed, "Clean Up request" is displayed. % Output #(important + selected) - MAX%

Figure 3.16 Important Messages Feature in the new template

this example has already been introduced previously, we start from the description of the use cases.

The auxiliary use case UC01 specifies the selection of messages from the inbox messages folder. In the first step of this use case there are no data involved, thus, it was not necessary to use the extended language for data. The second step updates the set of selected messages with the messages that come from the input.

The use case UC02 includes UC01 after the *START* step, and defines an extension point named *Clean up*. The main flow of UC02 specifies the successful attempt of moving the *selected* messages to the *important* messages folder. In order to move the messages, the *selected* messages are removed from the *inbox* messages folder and added to the *important* messages folder.

The alternative flow of UC02 specifies the unsuccessful attempt to move those messages. The guard of step 1A is a complement to that of step 1M, thus, they are mutually exclusive. If the guard of step 1A holds, then a message is displayed indicating that a cleanup is required; the minimum number of messages to be deleted is indicated.

The use case UC03 extends UC02 with the *Clean up* extension if there is some messages in the *important* messages folder. It defines a local variable named *selected* which is initialized with an empty set. When a local variable has the same name as a global variable (overloading), references to the variable name will refer to the local one. In UC03 the messages selected from the important messages folder are assigned to the local variable (step 1M), and the same selected messages are removed from that folder.

3.3 Concluding Remarks

This chapter introduced the new template, where the new language to support data definition and manipulation is now included. The entire language constructs were explained in details, so that a software engineer responsible for building a use case specification is now able to make use of those new improvements inside the use case document. An example illustrating the usage of the new template was also presented.

The major advantage that this extension brings to the old use case specification is the explicit notion of state, as well as constructions for state update, input, output and testing conditions. For that purpose, the concept of memory in the scope of feature and use case was also included in order to embody the definitions of constants, variables and types. A real case study is presented in Chapter 5.

Implementation

The work of Cabral *et al.* [CS08] presents a strategy for generating formal specifications in CSP from a use case document. Based on this work, Nogueira *et al.* [NSM] extended this idea so that the *includes* and *extends* relationships among use cases and the data definition and manipulation are now supported. We extended the TaRGeT tool in order to mechanize the work proposed by Nogueira *et al.*

This chapter introduces the mechanization of the CSP generation from a use case specification following the template presented in the previous chapter. Actually, the generated model follows the syntax of CSPm [Gol04], which is the ASCII (textual) form of CSP, in which FDR scripts are expressed. It is the standard syntax accepted by CSP tools. Our implementation is conservative, which means that previous use case specifications that do not use our extensions still work in TaRGeT [NCT⁺07]. Similar to TaRGeT, our implementation is also in the Java language and based on the representation of a use case specification in XML.

Our first task was to extend the TaRGeT XML Schema to support the new features and to include the processing of these features. After that, the class responsible for generating the CSP specification (and its auxiliary classes) and the classes needed to represent the new features were created. Every new constructor of the language is supported by a class. For example, an input is represented by the class **Input**, a guard is represented by the class **Guard**, etc.

Figure 4.1 shows a high level view of TaRGeT's architecture before the inclusion of CSP generation (a) and after this inclusion (b). The following sections explain what was needed to be changed and to be created in TaRGeT in order to mechanize our CSP generator.

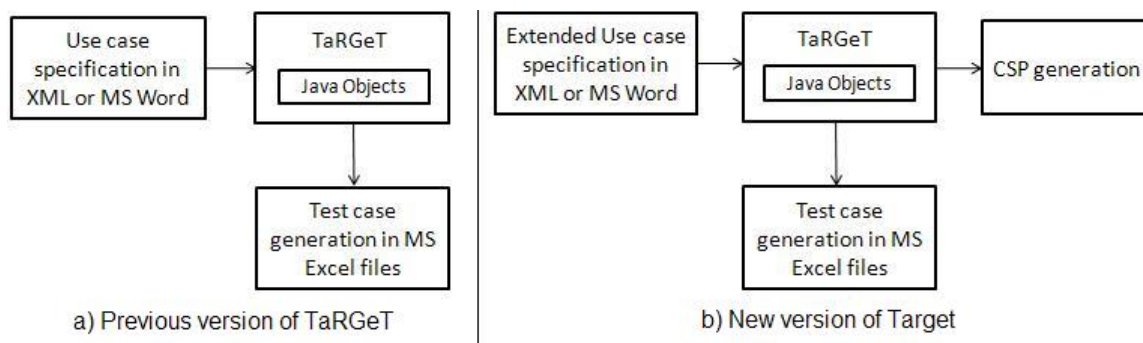


Figure 4.1 TaRGeT Architecture

4.1 XML Schema Extension

The XML Schema that represents the use case specification template was extended so as to support the data definition and manipulation. Our language was included as elements in the new Schema in a conservative way. Having in mind that the XML Schema is very large, we show in this subsection just the main parts added to support the new features. The complete Schema can be found in Appendix B.

Even though the extends and includes relationships had been already defined and included in the use case specification template, it was not implemented in the previous version of the TaRGeT tool. Thus, we also had to embody this feature in the XML Schema. Figure 4.2 illustrates the diagram and Figure 4.3 shows an excerpt of the Schema. This illustrates the three elements needed to be created to represent the inclusion and extension relationships among use cases: *include*, *extension* and *extensionPoint*.

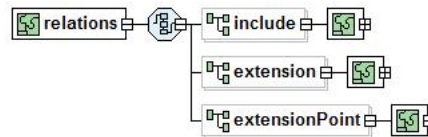


Figure 4.2 Relationship among use cases

```

<xs:complexType name="relations">
  <xs:sequence>
    <xs:element name="include" maxOccurs="unbounded" minOccurs = "0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="useCaseId" type="spec:useCaseId" />
          <xs:element name="position" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="extension" maxOccurs="unbounded" minOccurs = "0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="guard" type="spec:expression" />
          <xs:element name="useCaseId" type="spec:useCaseId" />
          <xs:element name="label" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="extensionPoint" maxOccurs="unbounded" minOccurs = "0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="label" type="xs:string" />
          <xs:element name="stepId" type="spec:stepId" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
  
```

Figure 4.3 Relationship among use cases in XML Schema

When the type of an element in the XML Schema has “spec:SomeType” this means that

“SomeType” is already defined as a primitive or complex type in the Schema. Thus, *useCaseId*, *expression* and *stepId* types in Figure 4.3 are complex types already defined. The *minOccurs* and *maxOccurs* requires that each kind of relationship contains zero or more elements, preserving the situations that does not consider relationships.

Figure 4.4 shows the diagram that represents the *dataDefinition* component, which represents the definition of Name Types, New Types, constants and variables; Figure 4.5 presents the corresponding XML Schema. Note that each kind of data definition follows the BNF presented in the previous chapter and the elements also have the *minOccurs* set to zero.

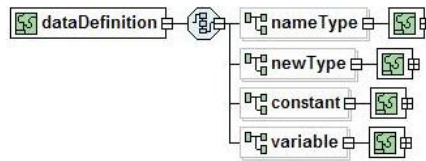


Figure 4.4 Data Definition

```

<xs:complexType name="dataDefinition">
  <xs:sequence>
    <xs:element name="nameType" maxOccurs="unbounded" minOccurs = "0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="id" type="spec:IdType" />
          <xs:element name="description" type="xs:string" />
          <xs:element name="nameTypeElements" type="spec:nameTypeElements" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="newType" maxOccurs="unbounded" minOccurs = "0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="id" type="spec:IdType" />
          <xs:element name="description" type="xs:string" />
          <xs:element name="newTypeElements" type="spec:newTypeElements" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="constant" maxOccurs="unbounded" minOccurs = "0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="id" type="xs:string" />
          <xs:element name="description" type="xs:string" />
          <xs:element name="constValue" type="spec:expression" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="variable" maxOccurs="unbounded" minOccurs = "0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="id" type="xs:string" />
          <xs:element name="description" type="xs:string" />
          <xs:element name="initialValue" type="spec:expression" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
  
```

Figure 4.5 Data Definition in XML Schema

Another important change was in the flow step in order to include the data manipulation (inputs, guards, outputs and assignments). In order to preserve the original Schema definition, the necessary elements were included but the old elements remains the same. Figure 4.6 illustrates the diagram for a flow step and Figure 4.7 shows the flow step Schema. Once again, the Schema represents exactly the BNF presented in the previous chapter.

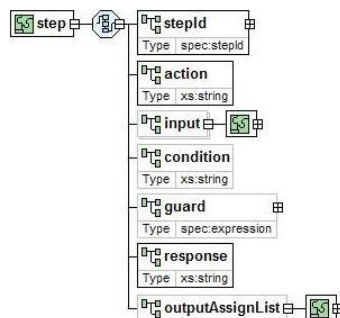


Figure 4.6 Flow step

```

<xs:complexType name="step">
  <xs:sequence>
    <xs:element name="stepId" type="spec:stepId" />
    <xs:element name="action" type="xs:string" />
    <xs:element name="input" minOccurs="0" maxOccurs="unbounded" >
      <xs:complexType>
        <xs:sequence>
          <xs:element name="var" type="spec:IdType" />
          <xs:element name="expression" type="spec:expression" />
          <xs:element name="restriction" type="spec:expression" minOccurs="0" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="condition" type="xs:string" minOccurs="0"/>
    <xs:element name="guard" type="spec:expression" minOccurs="0" />
    <xs:element name="setup" type="xs:string" maxOccurs="1" minOccurs="0"/>
    <xs:element name="response" type="xs:string" />
    <xs:element name="outputAssignList" minOccurs="0" >
      <xs:complexType>
        <xs:sequence>
          <xs:element name="output" type="spec:expression" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="assign" minOccurs="0" maxOccurs="unbounded" >
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Id" type="spec:IdType" />
                <xs:element name="expression" type="spec:expression" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
  
```

Figure 4.7 Flow step in XML Schema

It can be seen that the existing fields to build a step in the new template are: stepId, action, input, condition, guard, response and the list of outputs and assignments. Note that the new

features included have the minOccurs set to 0 (Figure 4.7), thus preserving the compatibility with the original Schema in which no data is included.

4.2 Architecture Extension

Looking forward to implementing the CSP model generation, TaRGeT had to be extended in order to support the new XML template. For that purpose, we needed to create new basic classes to support each new element created in the XML Schema. First of all, three entity classes were created to include the relationship among use cases: **UCExtension**, **UCInclude** and **UCExtensionPoint**, representing extension, inclusion and extension point, respectively. These classes were grouped into another entity class, called **UCRelations**, aiming at architecture organization.

Figure 4.8 shows the class diagram that represents the relationship among use cases. Recall that these relationships are just at use case level; therefore, it is not possible to create relationship among features. For that purpose, a **UCRelations** attribute was added in the **UseCase** entity class (already defined in TaRGeT). Note that the attribute - **auxiliary** - also had to be included in the **UseCase** class (and in the XML Schema) so as to identify whether a use case is auxiliary or not.

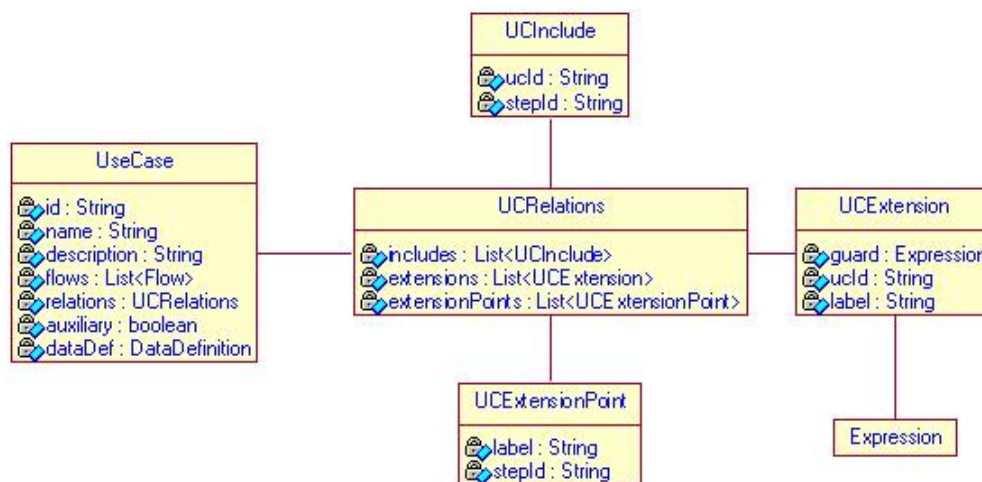


Figure 4.8 Class diagram for relationship among use cases

In order to implement the inclusion of data, other changes had to be performed in the architecture. We can divide this inclusion into two groups: data definition and data manipulation. To support the first group, a **DataDefinition** entity class was created, which embodies the definition of types (New Type and Name Type), constants and variables implemented by the classes **NameType**, **NewType**, **Constant** and **Variable**, respectively.

Having in mind that data can be defined both in the scope of a Feature and a use case, the **DataDefinition** class was included as an attribute of the corresponding classes **Feature** (already existent in the original TaRGeT) and **UseCase**. Figure 4.9 illustrates the class diagram.

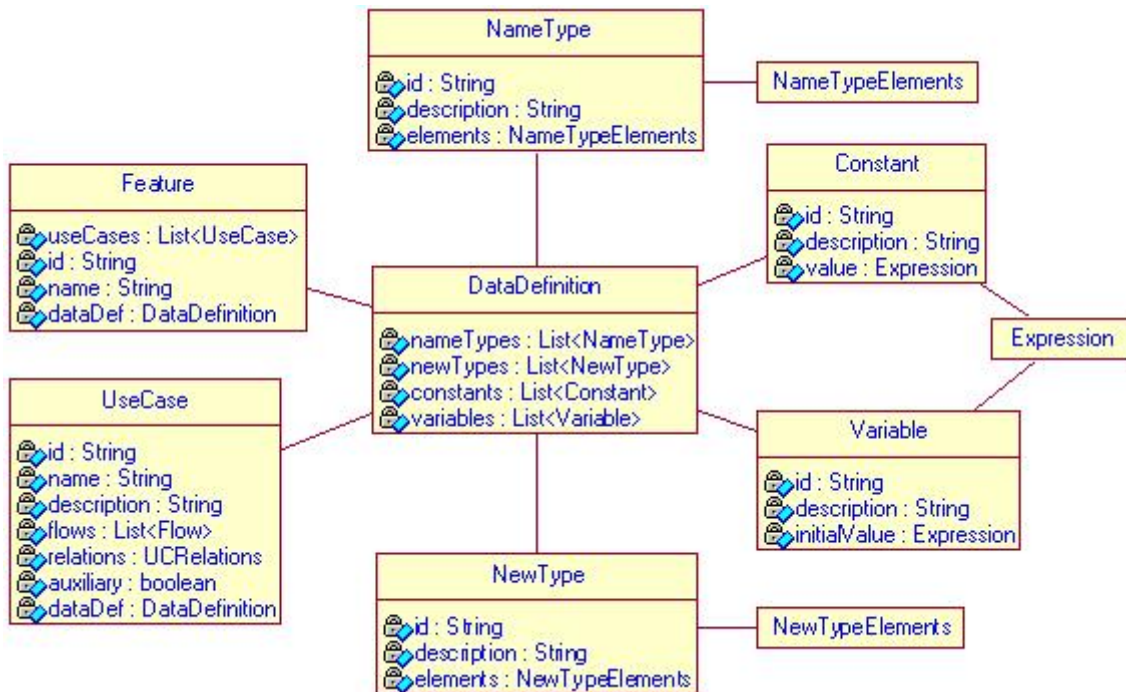


Figure 4.9 Class diagram for data definition

The **NameTypeElements** and **NewTypeElements** are two interfaces and were created to support the elements definition of Name Types and New Types. Figure 4.10 presents the class diagrams for these interfaces. Note that the entity classes **SetLiteral** and **SetRange** implements **NameTypeElements** and were included so that the two ways of defining the elements of Name Type could be implemented. Following the same idea, **BaseTypeList** (and **BaseType**), **Indexing** and **Enumeration** classes implements **NewTypeElements** and represents the three ways of defining the elements of a New Type (recall the data definitions introduced in Chapter 3).

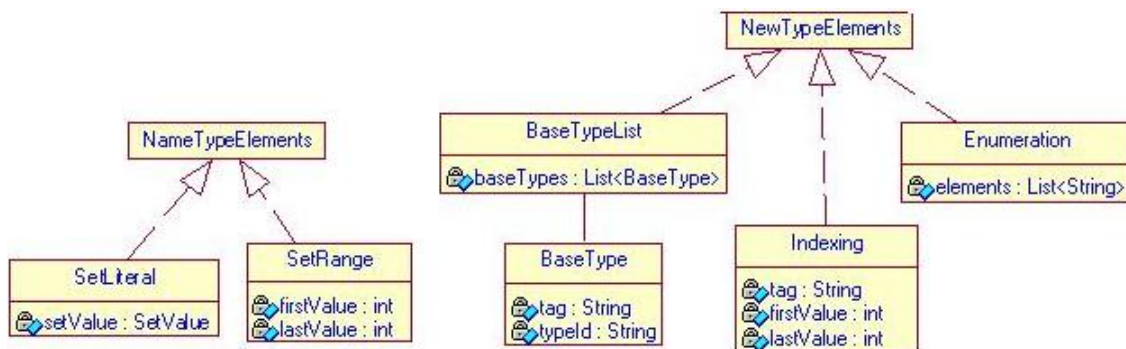


Figure 4.10 Class diagram for Name Type and New Type elements

To represent an expression — used for defining the values of constants, variables and the guard in an extension — the interface **Expression** was added. The classes **BinaryExpression**, **UnaryExpression**, **Id** and **Value** implement this interface and represent all kinds of possible

expressions. The **NaturalValue**, **BoolValue**, **SetValue** and **NewTypeValue** classes extends the parametrized abstract class **Value**, where the type of the *value* attribute depends on the subclass instantiated as can be seen in Figure 4.11.

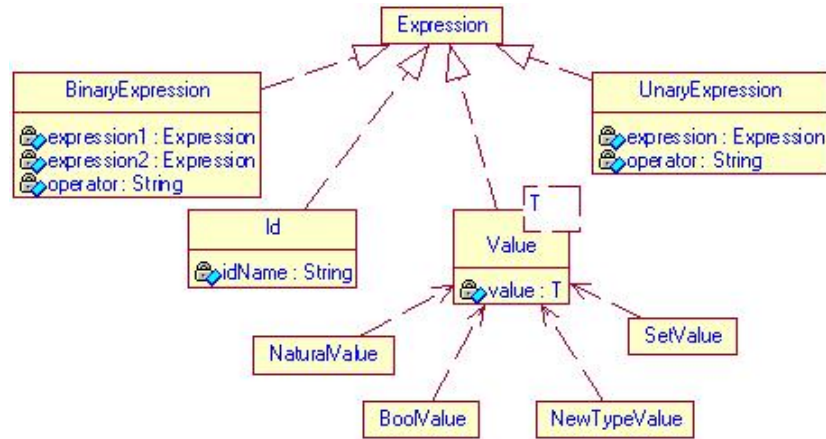


Figure 4.11 Class diagram for expression

To support data manipulation, the entity class **FlowStep** was extended with new attributes representing the inputs, guard, outputs and assignments. Figure 4.12 shows this architecture, where the entity classes **Input** and **Guard** represent the use of inputs and guard, respectively. For the purpose of architecture organization the class **OutputAssignList** encapsulates the classes **Output** and **Assign**, in which it is possible to output expressions and perform variable assignments, respectively.

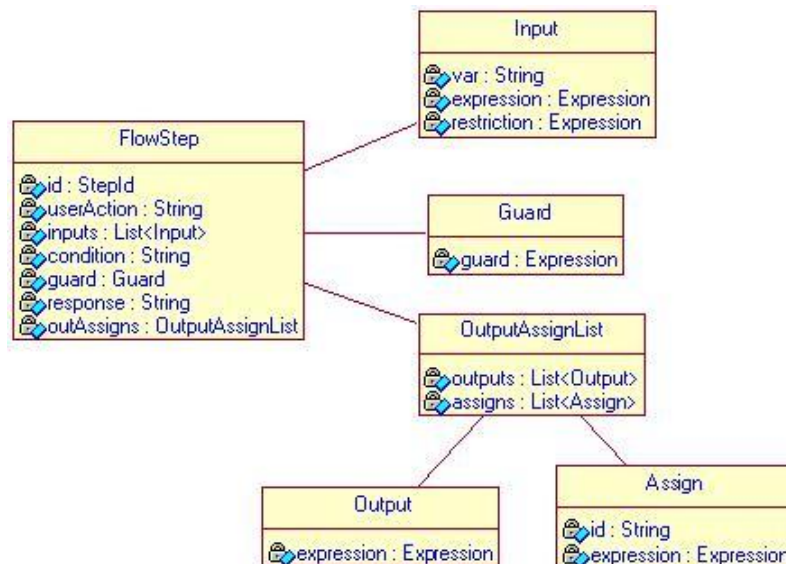


Figure 4.12 Class diagram for the extended flow step

4.3 CSP Model Generation

This section presents an overview of the CSP generation strategy adopted by this work. Figure 4.13 displays the structure of the CSP specification model with data. The process System models the behavior of the features F1, F2, ..., FN that reflect the document structure. Basically, for each document feature a CSP process is generated. Each feature gives rise to a process, and accordingly, each use case of a feature is modelled by a process (the FLOW processes in Figure 4.13).

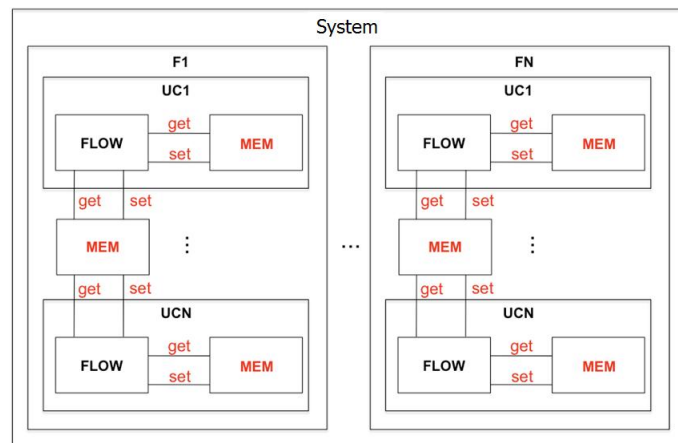


Figure 4.13 Structure of the CSP model

If the feature contains variables, additionally to the use case processes, there is another process (the MEM processes in Figure 4.13) that specifies the feature state; this process plays the role of an abstract memory. A memory process stores variable values and allows the use cases (also represented by processes) to read from and write to the memory. Local memories are added if a use case contains local variables.

Flow and memory processes communicate through special events that enable reading (get) and updating the memory (set). Such events are shared among flows and memory processes. A use case memory is only accessed by the use case flow, while a feature memory is shared among the feature use cases.

In what follows we incrementally present the translation of some elements of the example introduced in Chapter 3. We will see how the *Important Messages* feature is translated into its respective composition of CSP processes. Such a composition follows the structure depicted in Figure 4.13.

Figure 4.14 reintroduces the data definition of the *Important Messages* feature with its respective CSP model. This figure shows the translation of types, constants and variables declared in the template into CSP. As can be seen in Figure 4.14, a Name Type is transformed into a `nametype` in CSP and a New Type is mapped into a `datatype`. A constant is simply mapped to a constant in CSP.

The general rule for the names in the CSP model is that elements valid in the scope of a feature F are prefixed with $\#F__$, and those valid in the scope of a use case uc of the feature F

are prefixed with `fF_uc_`. Moreover, the types are preceded with the character `t` instead of `f`.

11169 - Important Messages

Data Definition

Name	Type (Id)	Description	Elements
SomeNaturals		Message Identifier	{0..2}

New Type (Id)	Description	Elements
Message	Phone Message	M.SomeNaturals

Constant (Id)	Description	Value
MAX	The maximum number of important messages	2

Variable (Id)	Description	Initial Value
inbox	Set of inbox messages	{M.0,M.1}
selected	Set of selected messages	∅
important	Set of important messages	{M.2}

```
--11169
datatype Var = f11169_inbox | f11169_selected
| f11169_important | f11169_UC03_selected
datatype Type = t11169_1.Set(t11169_Message)

-- 11169 data

-- types
nametype t11169_SomeNaturals = {0..2}
datatype t11169_Message = M.t11169_SomeNaturals

-- constants
f11169_MAX = 2

-- memory
b11169_MEM = {
(f11169_inbox,t11169_1.{(M.0),(M.1)}),
(f11169_selected,t11169_1.{}),
(f11169_important,t11169_1.{(M.2)})
}
```

Figure 4.14 CSP translation of Important Messages data definition

Each variable declared in the document is enumerated in the datatype `Var`. Figure 4.14 shows the datatype `Var` enumerating three variables in the scope of feature: `f11169_inbox`, `f11169_selected` and `f11169_important`. If there is any variable declared in the scope of a use case, it is also enumerated in this datatype. For example, supposing that a variable also named `selected` is declared in the scope of a use case `UC03`, the variable `f11169_UC03_selected` is also enumerated in the datatype `Var`.

Furthermore, each variable type is enumerated in the datatype `Type`, whose purpose is to represent the union of all the relevant types of the state variables; this is then used as the type of the variables placed in the memory. Each type is declared by a tag that identifies the type and the separator “.” followed by the set of values for the type. In this way, the datatype `Type` produces values in the form `typeTag.typeValue`, which are denominated tagged values.

For example, in Figure 4.14 we can see that `Type` enumerates the (single) type of the variables declared in the *Important Messages* feature. These variables are tagged with the identifier `t11169_1` (`typeTag`) and values in the superset of message sets (`typeValue`). Still in Figure 4.14, `b11169_MEM` is the initial binding for the three variables in the scope of the *Important Messages* feature. The binding of variables local to a use case is specified analogously, but in the memory created for that use case.

As already explained, get and set events are used by control flow processes to access memory processes. In the CSP translation these events are channels communicating values of type `Var.Type`, thus, they are able to communicate the pairs (variable, tagged value) whose values come from the already defined datatypes `Var` and `Type`, respectively. Based on these channels the process that models a memory can be defined. The strategy presented here considers an interleaving of processes, each one concerned with the store of a single variable, to represent

state information in CSP. In this way, we can say that the memory process is the interleaving among memory cells that carry variables and values from an initial binding. Supposing this memory process is already defined for the Important Messages feature, its memory model can be defined as `f11169_MEMORY = Memory(b11169_MEM)`.

To illustrate a CSP model example for a use case flow, Figure 4.15 reintroduces the use case UC02 of *Important Messages* feature and its corresponding generated CSP process. The process flow of the use case UC02 (`f11169_UC02_FLOW`) is parametrized by the use case extension (`f11169_UC02_Cleanup`). Its start step includes the use case UC01 that is sequentially composed with the continuations of the step (`f11169_UC02_1M [] f11169_UC02_1A`): an external choice between steps 1M and 1A because the *From Step* of the Alternative Flow is the *START* step.

UC02 - Moving Messages from Inbox to Important Messages

Includes UC01@START
Extension points Clean up: 1A

Main Flow

Step Id	Action	System State	System Response
1M	Select "Move to Important Messages" option.	Message storage has enough space. % #(important + selected) <= MAX %	"Message moved to Important Messages folder" is displayed. % inbox:= inbox - selected, important:= important + selected %

Alternative Flows

From Step: START
 To Step: START

Step Id	Action	System State	System Response
1A	Select "Move to Important Messages" option.	Message storage has not enough space. % #(important + selected) > MAX %	"Message storage has not enough space" is displayed. "Clean Up request" is displayed. % Output #(important + selected) - MAX%

```
f11169_UC02_FLOW(f11169_UC02_Cleanup) = let
f11169_UC02_START =
  f11169_UC01;
  f11169_UC02_1M [] f11169_UC02_1A
f11169_UC02_1M =
  get!f11169_important.t11169_1?important ->
  get!f11169_selected.t11169_1?selected ->
  get!f11169_inbox.t11169_1?inbox ->
  card(union(important,selected)) <= f11169_MAX &
  action11169_UC02_1M ->
  response11169_UC02_1M ->
  set!f11169_inbox!t11169_1.diff(inbox,selected) ->
  set!f11169_important!t11169_1.union(important,
  mem_update ->                               selected) ->
  SKIP
f11169_UC02_1A =
  get!f11169_important.t11169_1?important ->
  get!f11169_selected.t11169_1?selected ->
  card(union(important,selected)) > f11169_MAX &
  action11169_UC02_1A ->
  response11169_UC02_1A ->
  out1_11169_UC02_1A!(card(union(important,selected))
  - f11169_MAX) ->
  SKIP;
  (f11169_UC02_Cleanup [] SKIP);
  f11169_UC02_START
within f11169_UC02_START
f11169_UC02(f11169_UC02_Cleanup) =
f11169_UC02_FLOW(f11169_UC02_Cleanup)
```

Figure 4.15 Generated CSP of UC02 flow

Since a use case step can refer to variable values, current values must be in the context of the step process before being used. The process for the step 1M initially reads the referred variables (`important`, `selected` and `inbox`) from memory and places their values in the context, using the `get` channels (suppose these `get` and `set` channels are already defined and can communicate values of type `Var . Type`).

The subsequent behaviour is guarded by the state guard. If the guard expression holds (i.e. if `card(union(important, selected)) <= f11169_MAX` holds) the subsequent actions and response (also supposing that the events `action11169_UC02_1M` and `response11169_UC02_1M` are already defined) are performed and the system state is up-

dated accordingly: set events update variables `inbox` and `important`.

The auxiliary event `mem_update` indicates the finalization of a sequence of assignments. The process for the step 1A is created using the same idea of the step 1M. The difference is that there is no assignments in the System Response, but an output command. For instance, the output channel `out1_11169_UC02_1A` is used to communicate the value of the output expression.

After that, the extension `f11169_UC02_Cleanup` represents the behaviour of a use case that extends use case UC02 in the Clean up extension point. The step 1A continues as the start step (`f11169_UC02_START`) because of the *To Step* field. Because UC02 does not contain local variables (thus, no memory), the process `f11169_UC02` (`f11169_UC02_Cleanup`) is equivalent to its flow process (`f11169_UC02_FLOW(f11169_UC02_Cleanup)`).

We illustrate the use of extension, input and local memory in Figure 4.16, which presents the use case UC03 of the *Important Messages* feature and its generated CSP code. There is the need of a memory in this use case since a local variable `selected` is defined. After the initial binding (`b11169_UC03_MEM`) also supposes the definition of UC03 memory (`f11169_UC03_MEMORY`). The start process `f11169_UC03_START` is simply modeled by its continuation (`f11169_UC03_1M`). The process for the step 1M (`f11169_UC03_1M`) initially reads the set of important messages from memory (variable `important`).

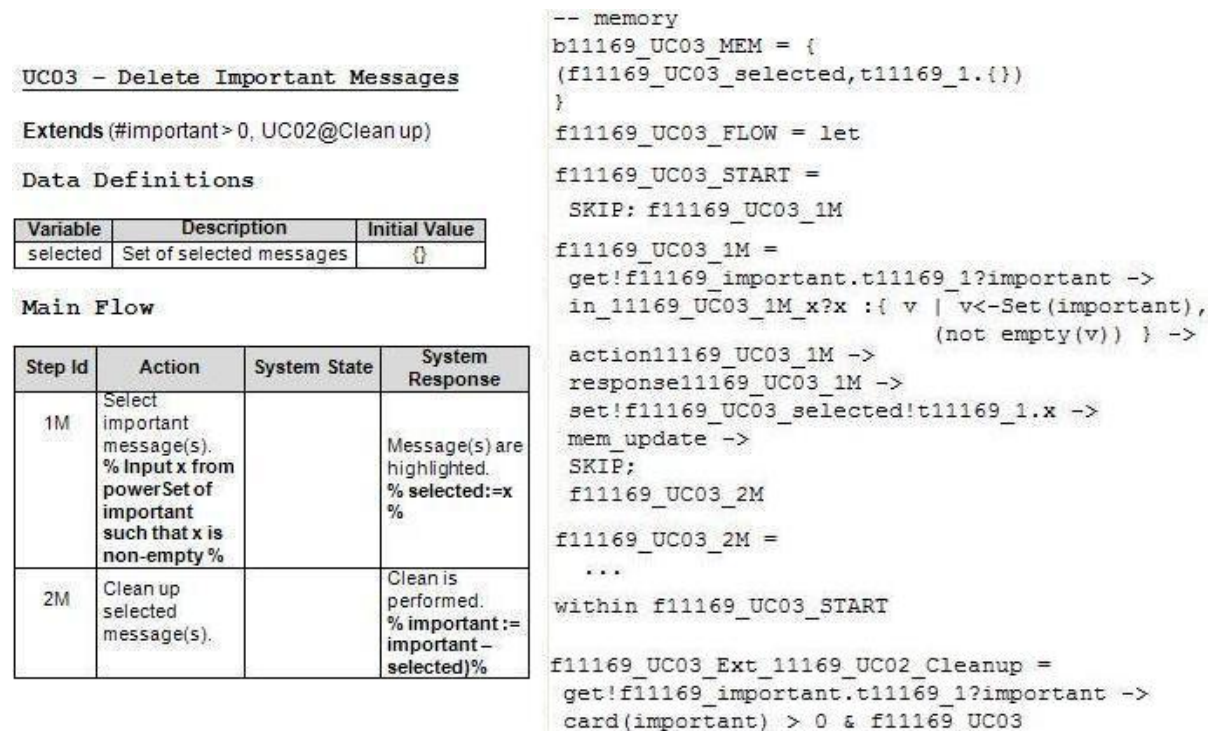


Figure 4.16 Generated CSP of UC03 memory, extension and flow

After that, it inputs a value in `x` (a non-empty subset of messages from the *important messages* folder). Note that the values for an input variable are specified in CSP by a previously defined channel (`in_11169_UC03_1M_x`). The process communicates the step action and

response, updates the set of selected messages (variable `selected`) and continues as step 2M. Considering that this step is simple and only has features already explained, it was omitted in Figure 4.16.

The extension is defined as the process `f11169_UC03_Ext_11169_UC02_Cleanup`, modeling the extension that the use case UC03 performs in the extension point `Clean up` of the use case UC02. It reads the variables referenced in its condition (variable `important`) and behaves as `f11169_UC03` if the extension condition (`card(important) > 0`) holds.

As we could see in the CSP model of UC02, a use case process has a subprocess for its control part (FLOW process), possibly parametrized by a list of extension points defined in the use case. When a use case contains local variables, the behaviour of the use case is modeled in CSP as the parallel composition of its control part and its memory, with synchronization on the events from memory, which represent the alphabet of the memory. This alphabet is internal to control and memory processes, so, they are hidden. Consequently, *get* and *set* events are synchronized between the flow and the memory processes and are invisible for the environment.

Because the memory process is a recursive, non terminating process, its direct composition with a flow process would also lead to nontermination; a parallel composition successfully terminates only when its argument processes do terminate. Consider the special event `success` that is not in the alphabet of the memory, neither in the use case alphabet. Such an event is used to define the process `success → Skip`, which communicates `success` and terminates, behaving like `Skip`. Figure 4.17 shows the CSP process that models UC03 behaviour. It terminates whenever the flow does.

```
f11169_UC03 =
(f11169_UC03_FLOW; success -> SKIP
 [|union(all1169_UC03_MEM,{success})|]
 f11169_UC03_MEMORY /\ (success -> SKIP)
) \ union(all1169_UC03_MEM,{success})
```

Figure 4.17 UC03 process

In this process, whenever the flow terminates with `success` (in the left-hand side of the parallel composition) it behaves as the process `success → Skip`. Because `success` is in the synchronization alphabet of the parallel composition, the event `success` can only be communicated if the same event is offered on the right-hand side of the parallel composition. At this point, the only possible behavior for `f11169_UC03_MEMORY \ success → Skip` is `success → Skip`. As a consequence, the resultant behavior of the composition becomes `success → Skip \ success`; since the only visible event is hidden (`success`), it is equivalent to `Skip`.

For brevity, some constructions of the generated CSP are omitted and not explained here. In order to have a complete and clear explanation about the whole translation of a use case specification following the idea of our template into a CSP specification see the work of Nogueira *et al.* [NSM]. The following subsection explains how the translation to CSP was implemented considering a use case document following the XML Schema presented as input.

4.3.1 Implementation of the CSP translation

This section introduces how the CSP generation was implemented from the classes introduced in Section 4.2. In order to provide the CSP translation from a use case document, the principal class **CSPgenerator** was implemented together with 17 auxiliary classes. The main idea behind **CSPgenerator**'s implementation is that it is composed of templates, each of them containing meta-variables that will be replaced by their respective concrete CSP terms. Figure 4.18 shows the templates created for that purpose: templates for data definitions, use cases, flows and flow steps.

```
String dataDefTemplate = "#variables# \n" +
    "#varTypes# \n\n" +
    "#tagFunction# \n" +
    "#nametypes#" +
    "#newtypes# \n" +
    "#constants# \n" +
    "#memory#";

String useCaseTemplate = "#dataDefinition# \n" +
    "#inputChannels#" +
    "#inputAlphabets#" +
    "#outputChannels#" +
    "#outputAlphabets#" +
    "#channels#\n\n " +
    "#alphabets#\n\n" +
    "#flowContent#" +
    "within #withinMark#\n\n" +
    "#endOfFlow#";

String flowTemplate = "#extension#" +
    "#FLOWID#" +
    "#stepContent#";

String stepTemplate = "#STEPID# = \n #INCLUSION_LIST#" +
    " #gets#" +
    " #inputs#" +
    " #guard#" +
    " #action#" +
    " #condition#" +
    " #response#" +
    " #outputs#" +
    " #sets#" +
    " SKIP; \n" +
    " #label#" +
    " #continuation#\n\n";
```

Figure 4.18 CSPGenerator templates

Now let us explain how this is done in more details. First of all, a method called *build* receives the list of features as parameter, and for each feature it calls the method *buildFeature*. This method then calls *buildDataDefinition*, that is the method responsible for constructing the CSP code according to the data definition template. Let us illustrate this process with the data definitions of the *Important Messages* feature (Figure 4.19).

```

--11169
datatype Var = f11169_inbox | f11169_selected
              | f11169_important
datatype Type = t11169_1.Set(t11169_Message)
tag(t11169_1.v) = t11169_1
-- 11169 data
-- types
nametype t11169_SomeNaturals = {0..2}
datatype t11169_Message = M.t11169_SomeNaturals
-- constants
f11169_MAX = 2
-- memory
b11169_MEM = {
  (f11169_inbox, t11169_1.{(M.0), (M.1)}),
  (f11169_selected, t11169_1.{}),
  (f11169_important, t11169_1.{(M.2)})
}

```

Figure 4.19 Data definition CSP code example

In order to build the first line of the code presented in Figure 4.19 (“datatype Var = . . .”), a method called *buildAllVariables* - created in an auxiliary class - is initially invoked in the method *build*. That method is responsible for creating a map that will contain the mappings between all variable ids with their respective types. After that, the method *buildDataDefinition*, which receives an object of the class *DataDefinition* as parameter, invokes the method *buildVar* of another auxiliary class so as to properly generate the string “datatype Var = f11169_inbox | f11169_selected | f11169_important” using the previously created map. In this particular case, it just iterates over each variable id to build the target string.

Following the same idea, we can build the complete structure of the CSP code to be generated. Once the CSP code is generated, we can see in Figure 4.19 how each meta-variable of the data definition template (dataDefTemplate in Figure 4.18) can be replaced by its respective CSP code. For example, **#variables#** is replaced by the CSP construct “datatype Var = . . .”. The meta-variable **#varTypes#** is replaced by “datatype Type = . . . ”, and so on. Recall that this code is generated by the method *buildDataDefinition*.

Back to the *buildFeature* method, after treating the case of data definition, a method named *buildUseCase* is needed to be called for each use case of the current feature. Following the same idea described before, the *buildUseCase* method is responsible for constructing the CSP code related to the use case template (remember this template in Figure 4.18). Thus, this method also has to call the *dataDefinition* method (supporting the case in which data local to a use case is defined), construct the rest of the CSP code related to use case, and finally, for each flow of the use case, call the method *buildFlow*.

The method *buildFlow* is responsible for creating the code related to the flow template and for calling the method *buildFlowStep*, for every flow step of the current flow. This method, like the others, replaces each meta-variable of the flow step template by their respective CSP code. Figure 4.20 reintroduces another example of the *Important Messages* feature, in this case

illustrating the creation of a CSP code for a flow step according to its template.

```
f11169_UC02_1A =
get!f11169_important.t11169_1?important ->
get!f11169_selected.t11169_1?selected ->
card(union(important,selected)) > f11169_MAX &
action11169_UC02_1A ->
response11169_UC02_1A ->
out1_11169_UC02_1A!(card(union(important,selected)) - f11169_MAX) ->
SKIP;
(f11169_UC02_Cleanup [] SKIP);
f11169_UC02_START
```

Figure 4.20 Flow step CSP code example

Looking at the flow step template (stepTemplate in Figure 4.18) and at Figure 4.20 we can see that **#STEPID#** is replaced by the respective flow step ID: f11169_UC02_1A. Considering that there is no inclusions in this step, **#INCLUSION_LIST#** is replaced with an empty string. Then, **#gets#** is replaced by the CSP code containing the needed variables memory reading, and so on.

In this way, doing that for every defined template, the complete CSP code can be generated. Just to finalize, there is also a method to build the feature behaviour (*buildFeatureBehaviour*), and another method responsible to build the system (*buildSystem*). To see the complete CSP code generated from the *Important Messages* feature specification see the Appendix C.

4.4 Concluding Remarks

As explained previously in this chapter, it was developed an implementation of automatic CSP generation from a use case specification following the template explained in Chapter 3. This implementation was done integrated with the framework of tools named TaRGeT, which also had to be extended to with new classes to support data definitions and their manipulation, as well as relationship among use cases, which was not considered inside TaRGeT yet.

Having a CSP model that represents the use case document in hands, it is then possible to automatically generate test cases. As proposed in the work of Nogueira *et al.* [NSM], test generation can be expressed as counter-examples of refinement checking, mechanized using the FDR tool. After test generation, state based test selection is also possible, which is an important task to reduce cost and time effort dedicated during software testing activities.

Case Study

This chapter presents a case study that was performed at the CInBTCRD project. In order to test the ideas and validate the approach proposed in this work, a use case document from Motorola was adapted to fit in the template used in this work, also including relationship among use cases and data fields. In order to use a shorter document, some requirements were not considered and a compacted version of the specification was used. Then, the CSP formal model of its specification was generated and the verification of its properties (non-determinism, deadlock and livelock freedom, for instance) was successfully checked using the FDR tool.

Considering that the case study was developed in the Motorola environment, the use case specification reflects the mobile devices domain. The use case we chose describes the sending and receiving of SMS (Short Message Service)/MMS (Multimedia Messaging Service) messages. Particularly, it deals with the act of adding multiple recipients to a single SMS/MMS message while being interrupted by a call, by the arrival of a new SMS or by the plug of the phone charger. The recipients to be added can be a phone number or an email address.

Another situation encompassed in this document is the receiving of an SMS with an email address in its body. This email address can be clicked on so that it is possible to send a text message to this address or store it to the Address Book. Besides that, the parsing of the From Address (in this case an email or a phone number) in SMS or MMS is also handled by the use case. The following sections present the use case document in more details.

5.1 Feature Types, Variables and Constants

This section presents the use case specification data definition in the scope of the feature. Figure 5.1 illustrates the declaration of the types, constants and variables that can be accessed by any use case of the feature 33629, whose name is *Sending and Receiving SMS/MMS*.

At first, let us take a look at the New Type definitions. There are two types of contacts data (numbers and addresses); thus, *number* and *address* represent the types which contain all (phone) numbers and (email) addresses, respectively. Because CSP has the integer type as built-in, it does not allow the use of integer values in the definition of other types; for that purpose we used the prefix “n_” before numbers. Also, it does not allow to use “@” and other special characters (only underscore) and dot can be only used to specify a basetype; thus, the emails are expressed with “_AT_” and “_DOT_”. The type *contact* is the union of numbers and addresses, which associate the tags Phone and Email to differentiate among them. The type *interaction* represent two types of interactions that can happen while composing a message: a dialog call can open (*dialog_call*) or an SMS notification can pop up (*SMS_notification*).

33629 - Sending and Receiving SMS/MMS

Data Definition

New Type (Id)	Description	Elements
number	Phone numbers	n_99887766, n_88446622
address	Email addresses	js_AT_gmail_DOT_com, ps_AT_ufpe_DOT_br
contact	All contacts data	Phone.number Email.address
interaction	Types of interaction	dialog_call, SMS_notification

Constant (Id)	Description	Value
max_selection	The maximum number of contacts selection	10
all_contacts	Set of all contact information	{Phone.n_99887766, Phone.n_88446622, Email.js_AT_gmail_DOT_com, Email.ps_AT_ufpe_DOT_br}
JonhSmith_info	Set of John Smith's information	{Phone.n_99887766, Email.js_AT_gmail_DOT_com}
UFPE_group	Set of UFPE contact information	{Phone.n_88446622, Email.ps_AT_ufpe_DOT_br}

Variable (Id)	Description	Initial Value
selected	Set of selected contacts	∅

Figure 5.1 Data Definition of the feature 33629

In the group of constants we define *max_selection* as the maximum number of contacts data that can be added to send a message. The constant *all_contacts* is the set containing all existing contact information. In the same way, *JohnSmith_info* composes the set of all information related to John Smith, and *UFPE_group* represents an existing group of UFPE students filtered from *all_contacts*. The only variable declared in the scope of this feature is *selected*, which represents the set of selected contacts.

```
--33629
datatype Var = f33629_selected | f33629_UC7_current_type | f33629_UC3_current_mode
datatype Type = t33629_1.Set(t33629_contact) | t33629_2.t33629_UC7_address_type
               | t33629_3.t33629_UC3_mode
-- 33629 data
-- types
datatype t33629_number = n_99887766 | n_88446622
datatype t33629_address = js_AT_gmail_DOT_com | ps_AT_ufpe_DOT_br
datatype t33629_contact = Phone.t33629_number | Email.t33629_address
datatype t33629_interaction = dialog_call | SMS_notification
-- constants
f33629_max_selection = 10
f33629_all_contacts = {(Phone.n_99887766), (Phone.n_88446622),
                      (Email.js_AT_gmail_DOT_com), (Email.ps_AT_ufpe_DOT_br)}
f33629_JohnSmith_info = {(Phone.n_99887766), (Email.js_AT_gmail_DOT_com)}
f33629_UFPE_group = {(Phone.n_88446622), (Email.ps_AT_ufpe_DOT_br)}
-- memory
b33629_MEM = {
  (f33629_selected, t33629_1.{} )
}
```

Figure 5.2 Data Definition CSP Model of the feature 33629

In Figure 5.2 we can see an excerpt of the generated CSP model from the set of data definitions presented. Note that in the *Var* datatype two more variables are defined: the variables *f33629_UC7_current_type* and *f33629_UC3_current_mode* are local to use cases UC7 and UC3 (described below), respectively. Remember that variables local to use cases are also defined in the top-level *Var* datatype declaration.

5.2 Use Case UC1

Figure 5.3 presents the use case UC1 of feature 33629, named *Editing SMS/MMS message*. As can be seen in the figure, this use case is responsible for creating an SMS or MMS message and it is an auxiliary use case, i.e, an use case that is not activated. It is a very simple use case, just containing CNL sentences, without any data fields.

UC1 - Editing SMS/MMS message <<auxiliary>>

Main Flow

Step Id	User Action	System State	System Response
1M	Choose the option of sending a SMS or MMS.		The message editing area is presented.
2M	Start editing the SMS or MMS message.		The entered message is displayed in the editing area.

Figure 5.3 Use case UC1 of the feature 33629

Figure 5.4 shows an excerpt of the generated CSP of the use case UC1. Once UC1 does not contain any data manipulation, we can see that each sentence of Figure 5.3 is translated in a CSP event. Once this use case contains no variable, its process (f33629_UC1) is equivalent to its flow process (f33629_UC1_FLOW). This part is omitted here, and in the others use case CSP excerpts, for purpose of brevity. To see the complete generated CSP, see the Appendix D.

```
f33629_UC1_FLOW = let
f33629_UC1_START =
  SKIP;
  f33629_UC1_1M
f33629_UC1_1M =
  action33629_UC1_1M ->
  response33629_UC1_1M ->
  SKIP;
  f33629_UC1_2M
f33629_UC1_2M =
  action33629_UC1_2M ->
  response33629_UC1_2M ->
  SKIP
within f33629_UC1_START
```

Figure 5.4 Generated CSP Model of UC1

5.3 Use Case UC2

Figure 5.5 presents the use case UC2, named *Adding multiple recipients*, which describes the possibility of adding multiple recipients to a single message composition. It can be seen that, firstly, this use case includes the behaviour of the use case UC1 (the inclusion is done in the START step), representing that, at first, it is needed to pick the option of sending a message before adding the recipients.

UC2 – Adding multiple recipients

Includes UC1@START

Extension points Interruption1:1M, Interruption2:1A, Interruption3:1B, Interruption4:1C, Interruption5:1D, Interruption6:1E

Main Flow

Step Id	User Action	System State	System Response
1M	Choose the recipients from All Contacts list. % Input x from powerset of all_contacts such that x is non-empty %	The number of selected contacts does not reach the maximum limit. % #x <= max_selection %	The chosen contacts are highlighted. % selected := x %
2M	Send the message.		The message is sent to the selected contacts.

Alternative Flows

From Step: START

To Step: START

Step Id	User Action	System State	System Response
1A	Choose the recipients from All Contacts list. % Input x from powerset of all_contacts such that x is non-empty %	The number of selected contacts reaches the maximum limit. % #x > max_selection %	A toast is shown to indicate that the maximum limit is reached; the user shall edit the selection to remove the exceeding number of contacts % Output #(x) – max_selection %

From Step: START

To Step: 2M

Step Id	User Action	System State	System Response
1B	Choose the recipients from a group filtered from All Contacts list. % Input x from powerset of UFPE_group such that x is non-empty %	The number of selected contacts does not reach the maximum limit. % #x <= max_selection %	The chosen contacts are highlighted. % selected := x %

From Step: START

To Step: START

Step Id	User Action	System State	System Response
1C	Choose the recipients from a group filtered from All Contacts list. % Input x from powerset of UFPE_group such that x is non-empty %	The number of selected contacts reaches the maximum limit. % #x > max_selection %	A toast is shown to indicate that the maximum limit is reached; the user shall edit the selection to remove the exceeding number of contacts % Output #(x) – max_selection %

From Step: START

To Step: 2M

Step Id	User Action	System State	System Response
1D	Select multiple phone numbers or emails from a same contact. % Input x from powerset of JohnSmith_info such that x is non-empty %	The number of selected contacts does not reach the maximum limit. % #x <= max_selection %	The chosen contacts are highlighted. % selected := x %

From Step: START

To Step: START

Step Id	User Action	System State	System Response
1E	Select multiple phone numbers or emails from a same contact. % Input x from powerset of JohnSmith_info such that x is non-empty %	The number of selected contacts reaches the maximum limit. % #x > max_selection %	A toast is shown to indicate that the maximum limit is reached; the user shall edit the selection to remove the exceeding number of contacts % Output #(x) – max_selection %

Figure 5.5 Use case UC2 of the feature 33629

UC2 also contains 6 extension points (interruptions in our case), labeled by the names *Interruption* and containing the steps where the extension use cases can be added. In this context, while adding the recipients it is possible to receive a call, receive a new SMS or be interrupted by the charger. These extension points are explained in the following sections.

The main flow of UC2 shows the addition of recipients chosen from a list where all contacts are available. The input variable x represents the contacts selected from the *all_contacts* set; non-selection is not allowed (see the restriction that x is non-empty). The system state guarantees that the maximum number of contacts is not reached (x must be less or equal to *max_selection*). Once this condition is satisfied, the input x is assigned to the set of selected contacts and the message can be sent.

The first alternative flow (containing step 1A) considers the situation when the number of contacts selected reaches the maximum, as can be seen in the system state. Because of that, the user is notified and advised to remove the exceeding number of contacts (the number of contacts to be removed is then output); thus, it comes back to the START step.

The two following alternative flows (containing the steps 1B and 1C) are very similar to the previous two flows; the only difference is that they are considering the case where the recipients are chosen from the subset of the complete list of contacts called *UFPE_group* (the input represents the set of non-empty contacts selected from that list). After that, again the two following flows (containing the steps 1D and 1E) are similar to 1B and 1C, but the selection comes from *JohnSmith_info*. Now, Figure 5.6 and Figure 5.7 present an excerpt of the generated CSP for UC2.

```
f33629_UC2_FLOW(f33629_UC2_Interruption1, f33629_UC2_Interruption2,
f33629_UC2_Interruption3, f33629_UC2_Interruption4, f33629_UC2_Interruption5,
f33629_UC2_Interruption6) = let
f33629_UC2_START =
  f33629_UC1;
  SKIP;
  f33629_UC2_1M [] f33629_UC2_1A [] f33629_UC2_1E [] f33629_UC2_1D []
  f33629_UC2_1C [] f33629_UC2_1B
f33629_UC2_1M =
  in_33629_UC2_1M_x?x : { v | v<-Set(f33629_all_contacts), (not empty(v)) } ->
  card(x) <= f33629_max_selection &
  action33629_UC2_1M ->
  response33629_UC2_1M ->
  set!f33629_selected!t33629_1.x -> mem_update ->
  SKIP;
  (f33629_UC2_Interruption1 [] SKIP);
  f33629_UC2_2M
f33629_UC2_2M =
  action33629_UC2_2M ->
  response33629_UC2_2M ->
  SKIP
f33629_UC2_1A =
  in_33629_UC2_1A_x?x : { v | v<-Set(f33629_all_contacts), (not empty(v)) } ->
  card(x) > f33629_max_selection &
  action33629_UC2_1A ->
  response33629_UC2_1A ->
  out1_33629_UC2_1A!(card(x) - f33629_max_selection) ->
  SKIP;
  (f33629_UC2_Interruption2 [] SKIP);
  f33629_UC2_START
```

Figure 5.6 Generated CSP Model of UC2 - Part 1

```

f33629_UC2_1B =
in_33629_UC2_1B_x?x : { v | v<-Set(f33629_UFPE_group), (not empty(v)) } ->
card(x) <= f33629_max_selection &
action33629_UC2_1B ->
response33629_UC2_1B ->
set!f33629_selected!t33629_1.x -> mem_update ->
SKIP;
(f33629_UC2_Interruption3 [] SKIP);
f33629_UC2_2M

f33629_UC2_1C =
in_33629_UC2_1C_x?x : { v | v<-Set(f33629_UFPE_group), (not empty(v)) } ->
card(x) > f33629_max_selection &
action33629_UC2_1C ->
response33629_UC2_1C ->
out1_33629_UC2_1C!(card(x) - f33629_max_selection) ->
SKIP;
(f33629_UC2_Interruption4 [] SKIP);
f33629_UC2_START

f33629_UC2_1D =
in_33629_UC2_1D_x?x : { v | v<-Set(f33629_JohnSmith_info), (not empty(v)) } ->
card(x) <= f33629_max_selection &
action33629_UC2_1D ->
response33629_UC2_1D ->
set!f33629_selected!t33629_1.x -> mem_update ->
SKIP;
(f33629_UC2_Interruption5 [] SKIP);
f33629_UC2_2M

f33629_UC2_1E =
in_33629_UC2_1E_x?x : { v | v<-Set(f33629_JohnSmith_info), (not empty(v)) } ->
card(x) > f33629_max_selection &
action33629_UC2_1E ->
response33629_UC2_1E ->
out1_33629_UC2_1E!(card(x) - f33629_max_selection) ->
SKIP;
(f33629_UC2_Interruption6 [] SKIP);
f33629_UC2_START
within f33629_UC2_START

```

Figure 5.7 Generated CSP Model of UC2 - Part 2

UC2 first behaves like UC1, followed by the external choice through `f33629_UC2_1M` (of main flow), `f33629_UC2_1A` to `f33629_UC2_1E` (of the alternative flows). Each step is then translated into processes with inputs, guards, and so on, depending on the data manipulation described in each one. For example, in `f33629_UC2_1M`, `x` is input through channel `in_33629_UC2_1M_x`. If the cardinality of `x` is less or equal to `f33629_max_selection` the events `action33629_UC2_1M` and `response33629_UC2_1M` are produced, and the assignment to `x` is done (`set!f33629_selected!t33629_1.x`). The other steps have similar translations. UC2 process (`f33629_UC2`) is also equivalent to its flow process.

5.4 Use Case UC3

Figure 5.8 shows the use case UC3, named *Calling interaction*. First this use case contains data defined in the UC3 scope: a New Type *mode*, which enumerates two types of phone modes (*selection_mode* and *call_mode*) and the variable *current_mode* (of *mode* type), which is initialized with the value *selection_mode*. This mode type represents the cell phone in 2 states (or modes). Selection mode defines the action of selecting contacts (in UC2) while call mode is when the user receives a call. We can also see that UC3 extends UC2 in all extension points contained in UC2. So, UC2 extended by UC3 represent the scenario of a call received while selecting the contacts before sending a message.

In the main flow, it can be seen that when the user receives a call in step 1M, a dialog is displayed. The output represents this situation: remember that *dialog_call* is a value of type *interaction*, which was defined in the scope of the feature (see Figure 5.1). A *dialog_call* interrupts the contacts' selection (that is why the current phone mode is changed to *call_mode*). In step 2M, when the user finishes the call, the phone returns to the selection state, thus, *current_mode* returns to *selection_mode*.

UC3 - Calling interaction

Data Definition

New Type (Id)	Description	Elements
mode	Phone mode	selection_mode, call_mode

Variable (Id)	Description	Initial Value
current_mode	The current phone mode	selection_mode

Extends (true, UC2@Interruption1), (true, UC2@Interruption2), (true, UC2@Interruption3), (true, UC2@Interruption4), (true, UC2@Interruption5), (true, UC2@Interruption6)

Main Flow

Step Id	User Action	System State	System Response
1M	The user receives a phone call.		A dialog is shown displaying the call, interrupting the selection of contacts. %current_mode := call_mode Output dialog_call %
2M	The user finishes the call.		The phone returns to the selection of contacts. %current_mode selection_mode %

Figure 5.8 Use case UC3 of the feature 33629

Figure 5.9 shows an excerpt of the generated CSP model of use case UC3. The type *mode* is translated to `t33629_UC3_mode` and the local variable *current_mode* becomes `f33629_UC3_current_mode`. Because UC3 contains a local variable, a local memory is then created (`b33629_UC3_MEM`). After that, every extension that UC3 performs in the extension points of UC2 is modeled by a process. The process for step 1M is modeled with events for user action and system response, followed an output and a local memory update. Step 2M is similar; it just does not contain the output. Once this use case contains vari-

able, its process (f33629_UC3) is modeled as the parallel composition of its control part (f33629_UC3_FLOW) and its memory (also omitted here).

```

-- types
datatype t33629_UC3_mode = selection_mode | call_mode
-- memory
b33629_UC3_MEM = {
  (f33629_UC3_current_mode,t33629_3.(selection_mode))
}
f33629_UC3_Ext_33629_UC2_Interruption1 = true & f33629_UC3
f33629_UC3_Ext_33629_UC2_Interruption2 = true & f33629_UC3
f33629_UC3_Ext_33629_UC2_Interruption3 = true & f33629_UC3
f33629_UC3_Ext_33629_UC2_Interruption4 = true & f33629_UC3
f33629_UC3_Ext_33629_UC2_Interruption5 = true & f33629_UC3
f33629_UC3_Ext_33629_UC2_Interruption6 = true & f33629_UC3
f33629_UC3_FLOW = let
  f33629_UC3_START =
    SKIP;
  f33629_UC3_1M
f33629_UC3_1M =
  action33629_UC3_1M ->
  response33629_UC3_1M ->
  out1_33629_UC3_1M!(dialog_call) ->
  set!f33629_UC3_current_mode!t33629_3.(call_mode) -> mem_update ->
  SKIP;
  f33629_UC3_2M
f33629_UC3_2M =
  action33629_UC3_2M ->
  response33629_UC3_2M ->
  set!f33629_UC3_current_mode!t33629_3.(selection_mode) -> mem_update ->
  SKIP
within f33629_UC3_START

```

Figure 5.9 Generated CSP Model of UC3

5.5 Use Case UC4

Figure 5.10 shows the use case UC4, named *New SMS interaction*, which represents the receiving of a new SMS while selecting the contacts to send a message. The list of extensions shows that this interaction extends UC2 in all extension points contained in UC2.

The main flow of UC4 shows that when the user receives a new SMS in step 1M, a notification of the new SMS is shown but does not interrupt the selection of contacts. That is why there is an output of *SMS_notification* and there is no changing of mode, once the task of selecting contacts is not interrupted (the notification can be simply a beep or the appearance of an icon). The change of mode on UC2 meant that the contact selection was interrupted for the call to be taken. After the call, the contact selection screen was enabled again. This interruption and change of screen does not happen in the use case UC4. So, we do not need a mode variable for

controlling the state of the phone.

UC4 - New SMS interaction

Extends (true, UC2@Interruption1), (true, UC2@Interruption2), (true, UC2@Interruption3), (true, UC2@Interruption4), (true, UC2@Interruption5), (true, UC2@Interruption6)

Main Flow

Step Id	User Action	System State	System Response
1M	The user receives a new SMS.		A notification of the new SMS is shown but does not interrupt the selection of contacts. % Output SMS_notification %

Figure 5.10 Use case UC4 of the feature 33629

Figure 5.11 shows an excerpt of the generated CSP model of use case UC4. This translation is very similar to that for UC3, except that there is no local data and memory and it contains only one step, with just one output. Once again, this use case does not contain variables, thus, its process (f33629_UC4) is modeled as its control (f33629_UC4_FLOW) part.

```
f33629_UC4_Ext_33629_UC2_Interruption1 = true & f33629_UC4
f33629_UC4_Ext_33629_UC2_Interruption2 = true & f33629_UC4
f33629_UC4_Ext_33629_UC2_Interruption3 = true & f33629_UC4
f33629_UC4_Ext_33629_UC2_Interruption4 = true & f33629_UC4
f33629_UC4_Ext_33629_UC2_Interruption5 = true & f33629_UC4
f33629_UC4_Ext_33629_UC2_Interruption6 = true & f33629_UC4
f33629_UC4_FLOW = let
f33629_UC4_START =
  SKIP;
  f33629_UC4_1M
f33629_UC4_1M =
  action33629_UC4_1M ->
  response33629_UC4_1M ->
  out1_33629_UC4_1M!(SMS_notification) ->
  SKIP
within f33629_UC4_START
```

Figure 5.11 Generated CSP Model of UC4

5.6 Use Case UC5

Figure 5.12 shows the use case UC5, named *Interaction with charger*. This use case describes the act of removing the cell phone from charger while selecting the contacts before sending a message. The list of extensions show this interaction, where UC5 extends UC2 in all extension points contained in UC2, just like UC3 and UC4.

The main flow of UC5 shows that when the user removes the cell phone from the charger in step 1M (the system state establishes the condition that there is a charger inserted in the phone), the action of removing the charger does not interrupt the selection of contacts nor displays a notification to the user. That is why there is no output and no changing of mode.

UC5 – Interaction with charger

Extends (true, UC2@Interruption1), (true, UC2@Interruption2), (true, UC2@Interruption3), (true, UC2@Interruption4), (true, UC2@Interruption5), (true, UC2@Interruption6)

Main Flow

Step Id	User Action	System State	System Response
1M	The user removes the cell phone from charger.	Phone has a charger inserted.	The action of removing the charger does not interrupt the selection of contacts.

Figure 5.12 Use case UC5 of the feature 33629

Figure 5.13 shows an excerpt of the generated CSP model of use case UC5. This translation is very similar to that for UC3 and UC4, where in this case, the steps are translated into events as there are no state change, no inputs, and no outputs in this use case. The process `f33629_UC5` is once again equivalent to its flow process.

```
f33629_UC5_Ext_33629_UC2_Interruption1 = true & f33629_UC5
f33629_UC5_Ext_33629_UC2_Interruption2 = true & f33629_UC5
f33629_UC5_Ext_33629_UC2_Interruption3 = true & f33629_UC5
f33629_UC5_Ext_33629_UC2_Interruption4 = true & f33629_UC5
f33629_UC5_Ext_33629_UC2_Interruption5 = true & f33629_UC5
f33629_UC5_Ext_33629_UC2_Interruption6 = true & f33629_UC5
f33629_UC5_FLOW = let
f33629_UC5_START =
  SKIP;
  f33629_UC5_1M
f33629_UC5_1M =
  action33629_UC5_1M ->
  response33629_UC5_1M ->
  SKIP
within f33629_UC5_START
```

Figure 5.13 Generated CSP Model of UC5

5.7 Use Case UC6

Figure 5.14 presents the use case UC6, which is named *Receiving an SMS with an email address attached to its body*. This use case describes the situation in which the user receives an SMS containing an email address in its body. When this happens, this address is highlighted,

allowing the user to click on it, and consequently, the options of sending a message to this email or storing it in the address book are available. As can be seen in Figur 5.14, this use case contains no data fields.

UC6 - Receiving an SMS with an email address attached to its body

Main Flow

Step Id	User Action	System State	System Response
1M	User receives a SMS which contains an email address on its body.		The email address is highlighted.
2M	User presses and holds the e-mail address.		At least the following options will be available: Send Text Message and Store to Address Book.

Figure 5.14 Use case UC6 of the feature 33629

Figure 5.15 then shows an excerpt of the generated CSP model of use case UC6. The translation is trivial: every action and system response is translated to an event. The CSP process is similar to that for UC1 (Figure 5.4). One more time, as the processes of the others use cases, the process of UC6 is equivalent to its flow process.

```
f33629_UC6_FLOW = let
f33629_UC6_START =
  SKIP;
  f33629_UC6_1M
f33629_UC6_1M =
  action33629_UC6_1M ->
  response33629_UC6_1M ->
  SKIP;
  f33629_UC6_2M
f33629_UC6_2M =
  action33629_UC6_2M ->
  response33629_UC6_2M ->
  SKIP
within f33629_UC6_START
```

Figure 5.15 Generated CSP Model of UC6

5.8 Use Case UC7

Figure 5.16 introduces the use case UC7, which is named *Parsing From Address in SMS or MMS*. It describes the situation where the From Address of a received SMS/MMS is highlighted so that the user can click on it and some options are available, just like in UC6. In UC7, the From Address can be either an email or a phone number (i.e, the sender can send the message

from his phone or from his email). We defined a New Type *address_type* enumerating the two options (*number* or *email*). A variable, named *current_type*, was also declared to store the current source type.

In the main flow of UC7 it can be seen that the source type of the From Address is input from a set containing the two available types. After that, this source type is stored in the *current_type* variable. Then, in the step 2M, the system state establishes the condition in which the From Address is a number: the guard *current_type = number* guarantees that. When the From Address is clicked on there are three possible options: make a call, send a message and store to address book. The alternative flow considers the opposite situation, in which the From Address is an email. In this, when the user clicks on the From Address, only two options are available: send a message and store to Address Book.

UC7 - Parsing From Address in SMS or MMS

Data Definition

New Type (Id)	Description	Elements
address_type	From address possible types	number, email

Variable (Id)	Description	Initial Value
current_type	The current type of the From address	number

Main Flow

Step Id	User Action	System State	System Response
1M	User receives a SMS or MMS. % Input sourceType from {number, email} %		The From Address of the SMS or MMS is highlighted. % current_type := sourceType %
2M	User presses and holds the number of the From address.	The From address is a number. % current_type = number %	At least the following options will be available: Voice Call, Send Text Message and Store to Address Book.

Alternative Flows

From Step: 1M
To Step: END

Step Id	User Action	System State	System Response
1A	User presses and holds the e-mail address.	The From address is an email. % current_type = email %	At least the following options will be available: Send Text Message and Store to Address Book.

Figure 5.16 Use case UC7 of the feature 33629

Figure 5.17 then shows an excerpt of the generated CSP model of use case UC7. The translation follows the same idea of the other use case. The type and variable defined are translated into `t33629_UC7_address_type` and `f33629_UC7_current_type`, respectively. A local memory (`b33629_UC7_MEM`) to store the variable (`f33629_UC7_current_type`) is created.

Step 1M is translated into the process which inputs the value `number` or `email` to the input variable `sourceType` and then produces the events for user action and system response; after that, it can go to the processes of the steps 2M or 1A. Step 2M and step 1A (of the alternative flow) are very similar; both of them need firstly to read the value of the local variable from memory (the `get` clause), then it is verified whether its value is equal `number` (step 2M)

or to email (step 1A). Finally, if the condition holds, the user action and system response are produced. Once UC7 contains variable, its process (f33629_UC7) is modeled as the parallel composition of its control part (f33629_UC7_FLOW) and its memory (also omitted here).

```

-- types
datatype t33629_UC7_address_type = number | email
-- memory
b33629_UC7_MEM = {
  (f33629_UC7_current_type, t33629_2.(number))
}
f33629_UC7_FLOW = let
f33629_UC7_START =
  SKIP;
  f33629_UC7_1M
f33629_UC7_1M =
  in_33629_UC7_1M_sourceType?sourceType : {v | v<-{(number), (email)}} ->
  action33629_UC7_1M ->
  response33629_UC7_1M ->
  SKIP;
  f33629_UC7_2M [] f33629_UC7_1A
f33629_UC7_2M =
  get!f33629_UC7_current_type.t33629_2?current_type ->
  current_type == (number) &
  action33629_UC7_2M ->
  response33629_UC7_2M ->
  SKIP
f33629_UC7_1A =
  get!f33629_UC7_current_type.t33629_2?current_type ->
  current_type == (email) &
  action33629_UC7_1A ->
  response33629_UC7_1A ->
  SKIP
within f33629_UC7_START

```

Figure 5.17 Generated CSP Model of UC7

5.9 Concluding remarks

An overview of the whole context of the use case specification used in the case study was described and some excerpts were presented, separately by each use case and their feature data definitions (some constructions of the CSP are omitted for brevity). The complete CSP model generated can be seen in Appendix D.

As said previously, this document was adapted from a real Motorola's specification to fit in our template, once the current Motorola's template for specifying use cases is different from ours. Also as already mentioned, we extended the original template with use case operations, state, input and output data. Besides that, we have produced a compacted version of the document so that performing the case study could be more easily achievable. The insertion of data and the use of relationship among use cases were also necessary in order to validate our approach in a real application context. We consider that the case study significantly illustrates the proposed templates and the translation strategy.

One thing we can notice is that most of the data defined in the scope of the feature are used only in the use case UC2. Thus, we can ask why those data are not defined in the scope of UC2. The answer for that question is: once we used a compacted version of the use case document in the case study, those data are only used in UC2, but, supposing that we can extend this use case specification to consider everything that was left behind, those data definitions might be used in other use cases.

We could also see that there are some restrictions in defining the elements of datatypes in CSP, where we cannot use numbers and strings containing special characters. Because of this, it was necessary to define the New Types in the use case document following the same restrictions. One thing that we can try to do in the future (considering that we have not thought about these restrictions earlier) is to treat these restrictions in the generated CSP model, in order to allow the user to enter with type elements like these (numbers and strings with special characters).

Some classical system properties (non-determinism, deadlock and livelock freedom, for instance) of the CSP formal model were successfully checked using the FDR tool. In the CInBTCRD project, this generated CSP model is the input to a test generation strategy, which is mechanized by the ATG tool [NSM08]. Our translation from use case to CSP ran on an AMD Turion 64, 1.4 Ghz, 1 MB of RAM, and took approximately 1 second. FDR (version 2.82, for Linux) ran on the same machine (in Ubuntu 8.10, emulated with a virtual machine) and the verification of the CSP model (for the three properties commented here) took 12 seconds.

The CSP model generator was also verified in other circumstances, not just in the CSP model generated for the case study and the Important Messages feature, in order to test other constructions of the proposed language and its generated CSP that is not approached in these cases.

Related Works

In this chapter some approaches that are somehow related to this work are described. The approaches are divided into two different domains: formal specification generation from requirements and test case generation from requirements. In general, the requirements documentation, in the approach considered here, makes use of a restricted form of natural language due to the problem of ambiguous, unclear and imprecise specifications.

6.1 Processable specification generation from requirements

This section introduces some works that aim at the generation, from a requirements document, of specifications which can be processed by a computer. These generated specifications also serve to verify properties of the requirements. The titles of the following subsections are relative to the name of each discussed work.

6.1.1 Supporting use case based requirements engineering

Somé's work [Som06] proposes an approach to support use case based requirements elicitation, clarification, composition and simulation. It introduces a restricted form of natural language for use cases such that automated derivation of specification is possible while readability and understandability of use cases by all stakeholders is retained.

This approach is supported by a tool called UCED (*Use Case Editor* [Som07]) that takes a set of related use cases written in a restricted form of natural language and generates an executable specification integrating the partial behaviours of the use cases. The work uses UML (Unified Modeling Language [OMG10]) class diagrams for the specifications and semantics of use cases, and also for the syntactic analysis and specification generation.

Figure 6.1 shows a use case diagram modelled in UML. The system under consideration is a Patient Monitoring System (PM System), which is used to monitor patients' vital signs in a hospital.

The relationship among use cases (inclusion and extension) are also supported, as can be seen in the Figure 6.1. This work distinguishes two kinds of use case descriptions: normal use cases and extension use cases. The former is presented in Figure 6.2 while the latter is showed in the Figure 6.3.

There are two basic components of use cases: conditions and operations. The restricted form of natural language proposed in this work is related with the description of these elements. Figure 6.4 outlines the grammar for conditions, which are predicative sentences describing

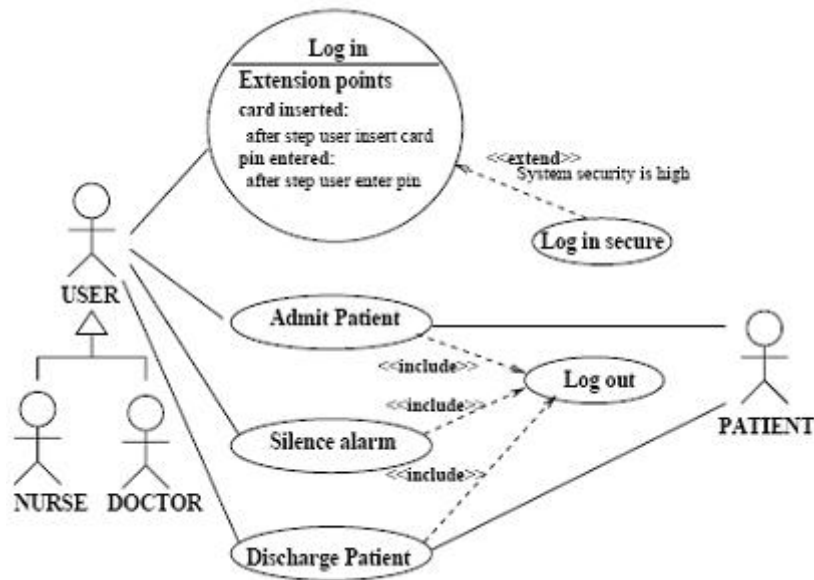


Figure 6.1 Example of use case diagram for a PM system

situations prevailing within the system and its environments. Figure 6.5 shows the syntax for operations.

The UCED tool uses an algorithm for the generation of a hierarchical type of finite state transition machines from use cases. The algorithm can be found in [Som03a, Som04]. A state machine generated from a use case includes all the use case scenarios. It is possible to find a sequence of transitions in the state machine corresponding to the sequence of operations of each scenario. State machines have the property of being executable and used as prototypes. Consequently, simulation can be efficiently applied within this approach, and because of that, UCED includes a simulator tool that allows use cases simulation using generated state machines as prototypes.

It can be seen that the language of this presented work seems to be a more structured language, with a well defined syntax for conditions and operations, while in our work it just happens in the use of data, where we have a specific BNF to defined the data fields. Also, the generated model of this work is more operational (using state machines), and ours considers a process algebra (CSP). There is one direction of this work whose final purpose includes test case generation (see Section 6.2.3).

6.1.2 Attempto — From Specifications in Controlled Natural Language towards Executable Specifications

The work of Schwitter et. al [SF96] proposes a language specification - *Attempto Controlled English (ACE)* - which is a subset of natural language that can be accurately and efficiently processed by a computer, but is expressive enough to allow natural usage. It is a textual view for writing functional requirements specifications so as to solve the problem of ambiguous, imprecise and unclear specifications.

Title: Log in

Primary Actor: User

Participants:

Goal: A User wants to identify herself in order to be able to use the PM system to perform a task such as admitting a patient or changing silencing an alarm.

Precondition: PMSystem is ON

Postcondition: User is logged in

Steps:

- 1: User inserts a Card in the card slot
 - Extension Point ==> card inserted
- 2: PMSystem asks for PIN
- 3: User types her PIN
 - Extension Point ==> pin entered
- 4: PMSystem validates the User identification
- 5: IF the User identification is valid THEN PMSystem displays a welcome message to User
- 6: AFTER 45 sec PMSystem ejects the User Card

Alternatives:

- 1a: User Card status is irregular
 - 1a1: PMSystem starts System status alarm
 - 1a2: AFTER 20 sec PMSystem ejects Card
- 2a: AFTER 60 seconds
 - 2a1: PMSystem starts System status alarm
 - 2a2: AFTER 20 sec PMSystem ejects Card
- 4a: User identification is invalid AND User number of attempts is less than 4
 - 4a1 GO TO Step 2
- 4b: User identification is invalid AND User number of attempts is equal to 4
 - 4b1: PMSystem starts System status alarm
 - 4b2: AFTER 20 sec PMSystem ejects Card

Figure 6.2 Use case describing a login procedure in a PM system

Title: Log in secure

Parts: At extension point card inserted

- 1: System logs transaction

At extension point pin entered

- 1: System logs transaction

Figure 6.3 Extension use case

```

<condition>  -> <acondition> "and" <condition>
  | <acondition> "or" <condition>
  | "("<condition>)"
  | <negation> <condition>
<acondition> -> [<determinant>] <entity> [<verb>] <value>
<determinant> -> "a" | "an" | "the"
<negation>    -> "not" | "no"
<entity>     -> <concept> | <attribute>
<concept>   -> (<word>)+ {member of the model concepts}
<attribute> -> (<word>)+ {attribute of concept}
<verb>      -> {derived from to be in present tense}
<value>     -> (<word>)+ {value of the entity}
  | <comparison> {entity is non-discrete ?}
<comparison> -> <comparator> <word>
<comparator> -> ">" | "<" | "=" | "<=" | ">=" | "<>"
  | "greater than" | "less than" | "equal to" | "different to"
  | "greater or equal to" | "less or equal to"

```

Figure 6.4 Grammar for conditions

```

<operation_spec> -> <concept_operation> | <branching_statement>
  | <useCase_inclusion>
<concept_operation> -> [<before_delay>] [<after_delay>]
  [<condition_spec>] <operation_reference>
<condition_spec> -> "IF" <condition> "THEN"
<operation_reference> -> (<word>)+ {derived from an operation of
  the current entity}
<after_delay> -> "AFTER" <duration_spec>
<before_delay> -> "BEFORE" <duration_spec>
<duration_value> -> <duration_value> <duration_unit>
<branching_statement> ->
  "GO" "TO" "Step" <word> {corresponding to a step label}
<useCase_inclusion> -> [<condition_spec>]
  "INCLUDE" <use-case-name>

```

Figure 6.5 Grammar for use case operations

The subset of controlled natural language proposed gives a well-defined syntax and semantics that can serve as a suitable view of a logic language. In ACE declarative sentences can be combined by constructors to compose more elaborate sentences, while certain forms of anaphora and ellipsis leave the language concise and natural. Furthermore, interrogative sentences are placed at the user's disposal for verifying the translated specification text.

Finite verbs can only be used actively, in the simple present tense, and in their third person singular and plural forms. In this way, users are supported to express statements that are always true. Figure 6.6 shows an example of a small excerpt of an ACE specification for a simple automated teller machine, called SimpleMat.

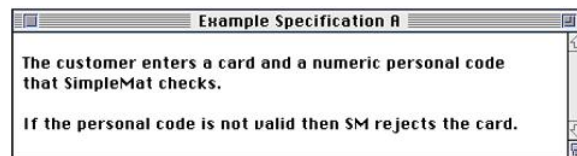


Figure 6.6 Small excerpt of the SimpleMat specification

Attempto accepts specifications in ACE and translates them into discourse representation structures (a structured form of first-order predicate logic), and then into Prolog [SS94]. Parsing errors and ambiguities to be resolved by the user are reported back by the dialog component. The specification text is parsed by a top-down recursive-descent parser that comes free with Prolog. Top-down parsing is very fast for short sentences but for longer composite sentences the exponential costs of backtracking can slow down the parsing.

There are two tools together with Attempto: a *Lexical Editor* and a *Spelling Checker*. The first one allows users to interactively modify and extend the full-form lexicon while the system parses the specification text. Figure 6.7 presents a screen shot of how a non-expert would add the common noun “customer” to the lexicon. The user has to enter the singular and the plural forms, to select the gender and to decide between the classes of count nouns and mass nouns.

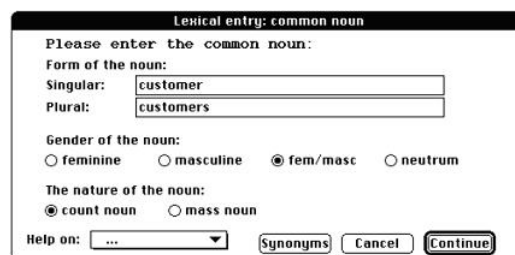


Figure 6.7 Insertion of a new substantive example

The second tool - *Spelling Checker* - allows users to determine whether all words of a specification text are in the lexicon. This spelling checker is invoked automatically if (part of) a specification text cannot be parsed.

The knowledge base can be used for simulation or prototyping by executing it. As it stands, however, the specification does not provide all the necessary information and needs to be enhanced in three situations:

- A chronological order of events has to be established, e.g. it has to make sure that during the simulation the event of entering a card has to precede the event of checking the personal code.
- Many relations representing events, e.g. I/O operations, are not only truth-functional, but also cause side-effects. These side-effects can be defined by interface predicates in the form of Prolog clauses.
- Finally, the execution needs some situation specific information, or scaffolding (this point can be better understood below, where we introduce how the execution is performed).

Now, suppose that the user enters the specification text showed in Figure 6.8. Its execution needs situation specific information and queries the user, e.g. to get an instance of a specific customer. In this way, the execution of the above specification text leads to the dialog showed in Figure 6.9. Side-effects of events are simulated by simply printing out a trace, informing the user - step-by-step - about the relevant events that have been triggered, as exemplified in Figure 6.10.

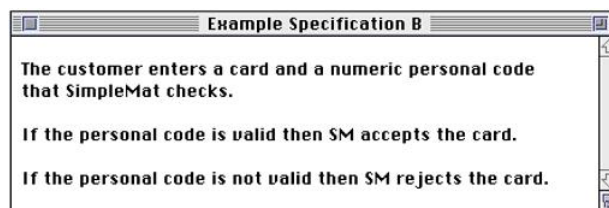


Figure 6.8 Specification example

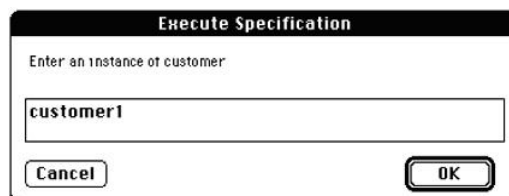
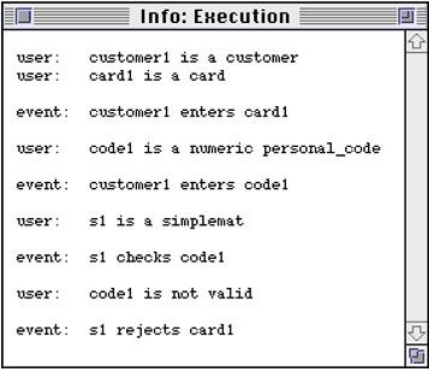


Figure 6.9 Specific information required from the user

We can see that the language used to specify a use case in this work allows the user to add new words, requires some specific information to the user; it has some interaction. In our approach there is no interaction with the user like that. Another difference that we can emphasize is that Attempto includes NL processing and does not propose a way of generating test cases.

6.1.3 Automatic Transformation of Natural Language Requirements into Formal Specifications

The main objective of Lee's work [Lee01] is to provide an automatic conversion of a requirements document written in a natural language (NL) into a formal specification language, once,



```

Info: Execution
user:  customer1 is a customer
user:  card1 is a card
event: customer1 enters card1
user:  code1 is a numeric personal_code
event: customer1 enters code1
user:  s1 is a simplemat
event: s1 checks code1
user:  code1 is not valid
event: s1 rejects card1

```

Figure 6.10 Step-by-step execution

even though there are many formal specification languages, it is not easy for a domain expert to learn and use them in practice.

This generation process has three phases. The first phase is the automated translation from a requirements documentation written in NL into a knowledge base. The second phase is to convert this knowledge base into a formal specification, known as *Two Level Grammar* (TLG [vW65]). TLG is used to construct a bridge between a NL requirements document and a formal specification language. In the last phase, the translation from TLG into a formal specification language, called *Vienna Development Method* (VDM [BJ78]) is made.

The unstructured document with requirements is automatically converted into a structured knowledge base by decomposing and abstracting the information. Given a requirements document, first each sentence is parsed into a list of words (decomposition). Syntactic and semantic information is retrieved from these parsed sentences to build up the knowledge base (abstraction).

In this way, the knowledge base is constructed using data structures. A conceptual data structure is used to store the vocabularies whereas a contextual data structure is used to store the sentences and the contexts.

Figure 6.11 shows a requirements specification example of an Automatic Teller Machine (ATM). Using the conceptual and contextual representations (knowledge base) generated in the first phase, the TLG specification shown in Figure 6.12 is automatically obtained. The logic programming model of TLG was extended in this work to include object oriented programming functionalities. Because of the NL-like syntax of TLG the translation from the knowledge base is relatively straightforward.

The generated TLG specification is used as an input to generate a VDM specification extension, known as VDM++ [DvK92], which supports object orientation as well.

It can be noted that this work is different from ours (and a lot of other works) because it does not use a restricted form of natural language, but uses the language in its natural form. Besides that, the generated model in this work includes object oriented programming features (as we said, VDM++ includes object oriented features) and also does not have its effort directed to test case generation.

```

The bank keeps the list of accounts.
Each account has three integer data fields;
  ID, PIN, and balance.
The ATM machine has 3 service types;
  withdraw, deposit, and balance check.
For each service first it verifies ID and PIN
  from the bank.

Withdraw service withdraws an amount from
the account of ID with PIN in the bank in
the following sequence:
First it gets the balance of the account
of ID from the bank,
if the amount is less than or equal to
the balance then
  it decreases the balance by Amount,
  updates the balance of the account of ID
  in the bank,
  and then outputs the new balance.

Deposit service deposits an amount to the
account of ID with PIN in the
bank in the following sequence:
First it gets the balance of the account
of ID from the bank,
it increases the balance by Amount,
updates the balance of the account of ID
in the bank,
and then outputs the new balance.

Balance check service check the balance of
the account of ID with PIN in Bank
in the following order:
It gets the balance of the account of ID
from the bank,
and then outputs the balance.

```

Figure 6.11 ATM requirements specification

6.1.4 Autonomous Requirements Specification Processing Using Natural Language Processing

The work of MacDonell et. al. [MMC05] describes the architecture of an autonomous requirements specification processing system that utilizes a limited version of a natural language processing (NLP) system and an interactive user interface system.

As a way to decrease the problems and ambiguities introduced in a requirements documentation, this work is therefore focused on the verification of requirements specification analysis with a view to producing a design model - a use case diagram, an entity-relationship model or similar. The system has the goal of automatically extracting objects of interest from a requirements document that is being processed by a systems analyst.

The system is composed of three modules: the first - a tokeniser - reads sentences from a document, the second module parses each sentence and extracts all unique noun terms (an NLP

```

class Bank.
  Account:: {ID Integer PIN Integer Balance Integer}*.
end class Bank.

class ATM.
  ID, PIN, Balance :: Integer.
  Amount :: Integer.

  withdraw Amount from account of ID with PIN in Bank
    giving Balance :
    verify ID and PIN from Bank,
    get Balance of the account of ID from Bank,
    if Amount <= to Balance then
      Balance := Balance - Amount endstmt
    update Balance of the account of ID in Bank
  endif.

  deposit Amount to account of ID with PIN in Bank
    giving Balance :
    verify ID and PIN from Bank,
    get Balance of the account of ID from Bank,
    Balance := Balance + Amount,
    update Balance of the account of ID in Bank.

  check balance of account of ID with PIN in Bank
    giving Balance :
    verify ID and PIN from Bank,
    get Balance of the account of ID from Bank.

end class ATM.

```

Figure 6.12 TLG specification of the ATM

tool), and the third module - a term management system - performs the filtering of unimportant terms, the classification of the remaining terms into one of three categories (function, entity, or attribute), and the insertion of objects of interest into a project knowledge base.

In the second module, each sentence is parsed by a syntactic parser based on a chart parsing technique [Ear70] with a context-free grammar (CFG) that is augmented with constraints. The current prototype system uses a dictionary with about 32000 entries and 79 rules. A parse tree example for the sentence “*A system requires entry of patient’s information*” can be visualized in Figure 6.13. The term management system allows the user to select terms to create classes

```

(S (NP (DET “A”) (NOUN “system”))
  (VP (VERB “requires”)
    (NP (NP (NOUN “entry”))
      (PP (OF “of”) (NP (POSSADJ “patient’s”)
        (NOUN “information”)))))).

```

Figure 6.13 Parse tree example

of objects of interest and to also manage the term's addition to and deletion from the defined class.

Unlike our work, this work includes natural language processing. Their final purpose is also different, aiming at producing design models such as entity-relationship model, while ours generates a process algebra (CSP) and includes test case generation.

6.1.5 Improved Processing of Textual Use Cases: Deriving Behaviour Specifications

The work of Drazan *et. al.* [DM07] proposes a new method for processing textual requirements (described using a subset of natural language) based on the scheme earlier described in [Men04], which focus on the derivation of behaviour specification, such as UML state machines. The new method allows processing the commonly used complex sentence structures, obtaining more descriptive behaviour specifications, which may be used to verify and validate requirements and to derive the initial design of the system.

While the original method [DM07] is applicable to sentences following the guidelines of [Coc00, Gra00], this work aims at broadening its range of use. Industry use cases often do not adhere to these guidelines, making use of complex sentences. Abstracting from the common patterns and proposing rules applicable to a broad range of industrial use cases is the main focus of this work.

For complex sentences, there is an increased risk that the statistical parser would return an incorrect parse tree. An additional goal is to propose a metric to evaluate parse tree quality and select the best parse tree if more than one is available.

It was developed a prototype tool [Dra06] implementing the improved method proposed. The method was evaluated on a substantial collection of use cases (307 use case steps), consisting of examples from methodology sources [Lar01, Cor03] respected by the industry, and also on the collection of sample use cases used in the earlier work.

The new method has proven to be more reliable also on the original test data. Part of the improvement is due to the ability of the new method to select a correct parse tree for sentences where the original method failed. The detailed results obtained in this case study are available in the appendices of [Dra06].

We can see that this approach differs from ours in some points: it includes natural language processing, the generated model is more operational (state machines), aiming at deriving the initial design of the system. Most importantly, it has evaluated the proposed approach in an experimental/industrial validation, which in our case can be considered a weakness.

6.2 Test cases generation from requirements

This section introduces some works that aims at reducing the cost of testing through the generation of test cases from a requirements document. The titles of the following subsections are relative to the name of each discussed work.

6.2.1 Automated Formal Specification Generation and Refinement from Requirement Documents

The work of Cabral et. al. [CS08] proposes a Controlled Natural Language (CNL), a subset of English, used to write use case specifications according to a template. From these use cases a complete strategy and tools enable the generation of process algebraic formal models in the CSP notation. Cabral's work is related to the same project that this work belongs to, in the research cooperation between CIn-UFPE and Motorola, called CInBTCRD.

Because the context of this work includes Motorola environment, the proposed CNL reflects this domain. The generated formal model is used in this project as an internal model to automatically generate test cases, both in Java (for automated ones) and in CNL (for manually executed). Thus, this work provides both a formal specification generation in CSP and test cases generation from the requirements document, addressing to the two sections discussed in this chapter.

The use cases are considered in views that represent different abstraction levels of the application specification: there are user and component views. A refinement relation between these views is also explored, which is the major contribution of their work; the use of CSP is particularly relevant in this context: its semantic models and refinement notions allow precisely capturing formal relations between user and component views.

Some tools were developed in order to support the adopted approach. There is a plug-in to Microsoft Word 2003 [SSLM04] to allow checking adherence of the use case specifications to the CNL grammar. Another tool has been developed to automate the translation of use cases written in CNL into CSP; FDR [Ros95], a CSP refinement checker, is used to check refinement between user and component views.

Figure 6.14 shows the strategy overview. After System Requirements are described in an abstract way, defining what the system is intended to perform, user view use cases are created based on requirements analysis. This first set of use cases designs the ways actors interact with the system. Later, component view use cases are created based on the user view use cases and the adopted System Architectural Information.

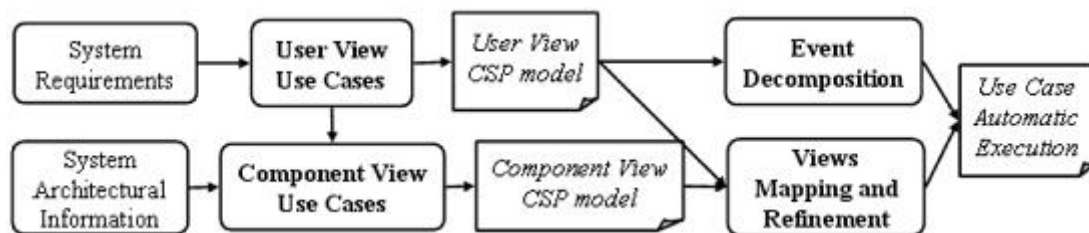


Figure 6.14 Proposed strategy overall process

The user view use cases are translated into a user view use model and the component view use cases are translated into a component view use model. Then, based on these models, the relation between user and component use cases is established. After that, the automatic execution is possible.

As just said previously, this work belongs to the same project as ours. The idea of our work

is based on this presented strategy, and includes relationship among use cases and a notion of state to the use case template and its generated formal model. The template used in our work only considers the user view. The generated CSP model is also different, where the translation of a entire sentence is captured by a CSP event.

6.2.2 UML-Based Statistical Test Case Generation

The work of Riebisch et. al. [RPG03] proposes an approach for generating system-level test cases based on UML use case models and refined by state diagrams. These models are transformed into usage models [WPT95] to describe both system behaviour and usage. The usage model serves as input for automated statistical testing [Sel99]. The approach is characterized by the following features:

- It is usage-oriented and specification-based, thus performing black box testing.
- It is intended for system-level testing, since a given specification usually describes the system's overall functionality rather than that of units or components in isolation.
- It aims at statistical (reliability) testing rather than fault detection.

The method is intended for integration into an iterative software development process model. The activity diagram (Figure 6.15) shows the sequence of activities in the context of the software development process.

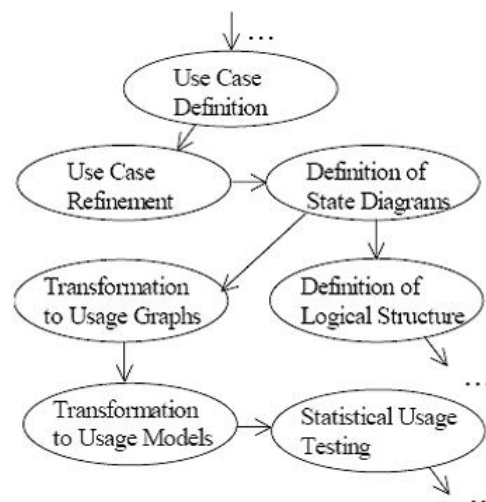


Figure 6.15 Activities of the proposed approach within the software development process

The resulting test cases are suited to be carried out in conventional ways, i.e., either manually or using test tools. The method is supported by an XML-based tool for model transformation. The approach described in this work is implemented in the tool UsageTester [Goe02] to provide a proof of concept and to support the method application.

We can see that the final purpose of this work is close to ours, considering the generation of test cases. In the case of this work it is also necessary to refine the use case models with state diagrams, which in our work there is not this need. Besides that, the intermediate model generated is usage-oriented.

6.2.3 An Approach for Supporting System-level Test Scenarios Generation from Textual Use Cases

Somé's work [SC08] describes an approach proposed toward test cases generation from requirements captured as use cases. This work uses the restricted form of natural language described in [Som06], which was already presented in this chapter, to overcome the problem of use cases specification ambiguities.

A second challenge, which is to ensure an adequate coverage of the sequences of actions defined by each use case, is resolved with the generation of a control-flow state machine from use cases. After that, traditional code coverage techniques are used to derive test sequences.

Finally, a third problem is that important sequential constraints between use cases are usually left out of the use cases and only assumed implicitly. They overcome this problem by inferring use case sequential relations based on a comparison of use case pre-conditions and post-conditions. This allows the combination of use cases behaviour in a global control-flow based state model.

As said before, this work is based on [Som06], but, while the state machine generation approach discussed now is based on use cases control-flow, the basis of state machine synthesis on the previous work is the domain operation pre-conditions and post-conditions. Synthesis based on domain operations produces state machines that depend on how operation pre/post-conditions are specified.

As mentioned in [Som06], resulting state machines may exhibit extra behaviours from those defined in the use cases. Synthesis based on use cases control-flow on the other hand, results in state machines that are an exact reflection of the use cases. They are therefore, more appropriate for test derivation. In addition, no specification of operations is required.

Test scenarios are generated using depth-first traversal of the generated control flow-based state machines according to criteria inspired from traditional white-box code coverage. In their approach, concrete (executable) test cases need to be manually derived from test scenarios because of an "abstraction gap" between test scenarios and concrete test cases.

These works which generates more operational models, such as state machines, the test case generation is based in explicit algorithms that runs through the model. In the case of a process algebra, such as CSP, our strategy to generate test cases is based in model checking; test cases are generated as counter-examples of models verification, without needing to define explicit generation algorithms.

6.2.4 Automatic Test Generation: A Use Case Driven Approach

It is well known that formal methods can be used both for validating requirements as well as for reducing the cost of testing through automatic test case generation. With this purpose, the work of Nebut et. al [NFTJ06] proposes to start from established practices and gently lead them

toward formally exploitable models. The focus is in an approach for automating the generation of system test scenarios from use cases in the context of object-oriented embedded software.

With the aim to eliminate the problem of specifying ambiguous use cases in natural languages, the work proposes to build UML use cases enhanced with contracts (based on use cases pre and postconditions) as they are defined in [NFTJ06, Coc97]. Lifting up Meyer's Design By Contract [Mey92] idea to the requirement level, it is proposed to make these contracts executable by writing them in the form of requirement-level logical expressions. Based on those more formalized (but still high-level) requirements, a simulation model of the use cases is defined.

Figure 6.16 summarizes the two-phase method to automatically generate functional test scenarios from requirement artefacts. The first phase of the method (steps (a) to (c) in Figure 6.16), aims at generating test objectives from a use case view of the system described using the contract idea.

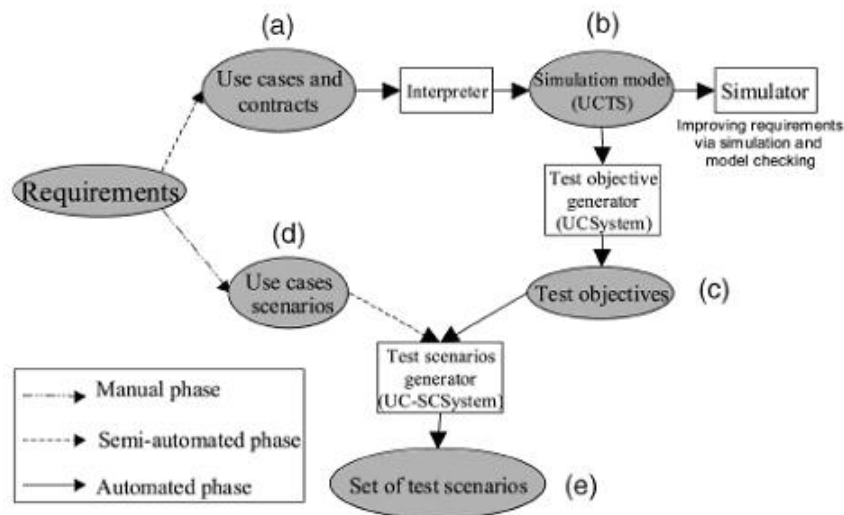


Figure 6.16 Global methodology for requirement-based testing

These contracts are used to infer the correct partial ordering of functionalities that the system should offer, and by using them, the actors involved in a use case can be considered as parameters of this use case. The use case contracts are first-order logical expressions combining predicates with logical operators. The precondition expression is the guard of the use case execution. The postcondition specifies the new values of the predicates after the execution of the use case. Figure 6.17 exemplifies a use case specification with contracts for a virtual meeting.

There are some limitations of the contractualized use case model:

- *Difficulty of building a contractualized use case model.* The declarative definition of such contract expressions forces the requirement analyst to be precise and rigorous in the semantics given to each use case and, thus, may not be so easy to build. To decrease this complexity, there is an editor tool to manage the predicates and guide the design of contracts.

```

UC open(u:participant;m:meeting)
pre created(m) and moderator(u,m) and not closed(m)
and not opened(m) and connected(u)
post opened(m)

UC close(u:participant; m:meeting)
pre opened(m) and moderator(u,m)
post not opened(m) and closed(m) and
forall(v:participant) {not entered(v,m) and
not asked(v,m) and not speaker(v,m) }

```

Figure 6.17 Contracts of use cases open and close

- *Numeric types.* The requirements that can be expressed with contracts on the use cases are high level ones, e.g., they are not suitable to handle complex data types (including arithmetic calculus for example).
- *Restriction on postcondition.* In this model, there is a restricted usage of the postconditions: the postconditions must be deterministic.

Back to the Figure 6.16, still in the first phase, a prototype tool (UCSystem) builds a simulation model from the use cases and their contracts (step (b) of Figure 6.16) and generates correct sequences of use cases (step (c) of Figure 6.16), called *test objectives*.

The second phase (steps (c) to (e) of Figure 6.16) aims at generating *test scenarios* from the test objectives. To go from the test objectives to the test scenarios, additional information is needed, specifying the exchanges of messages involved between the environment and the system. Such information can be attached to a given use case in the form of several artefacts: sequence diagrams, state machines, or activity diagrams. For simplicity, this work deal with sequence diagrams, which are called use case scenarios.

The principle of the transformation from test objectives to test scenarios is inspired by Briand *et. al.* [BL02] and consists of replacing each use case of the test objective by one of its use case scenarios, using the prototype tool UC-SCSystem.

We can see that this work proposes some different strategies in order to generate test cases. At first, the use cases specification are enhanced with the idea of contracts, making them more object-oriented, and consequently, less prone to ambiguities and possible to perform simulation. But, as previously said, it has some limitations due to this contractualized use cases and also there is the need to attach additional information to use cases in order to generate test cases.

6.3 Discussion

This section introduces a comparison between all cited works and ours. As we could notice, all of these works are applicable to requirements specification as the input for the strategy. To facilitate the understanding, Table 6.1 presents some features and shows which works contain each feature. This table shows that this work neither supports NL processing nor industrial

	Uses CNL	Formal specification generation	Test generation	Automatic extraction	Data manipulation	NL processing	Industrial validation
[Som06]	x	x		x			
[SF96]	x	x		x		x	
[Lee01]		x		x			
[MMC05]				x		x	
[DM07]	x			x		x	x
[CS08]	x	x	x	x			
[RPG03]			x				
[SC08]	x		x	x			
[NFTJ06]	x		x	x			
Our work	x	x	x	x	x		

Table 6.1 Comparison among related works

validation; the test case generation is provided by the work of Nogueira *et al.* [NSM], which belongs to the same project as ours. We can also try to include the NL processing in the future, analyzing the possibility of integrating with the work reported in [Lei06].

Each of these works has a different way of writing the requirement specification. As we could see, some needed additional information, such as state diagrams, other introduced the idea of contracts, other used a non-restricted language, and so on. Every different use case description like those depends pretty much on the final purpose of each strategy.

It is worth emphasizing that almost every work that produces a formal specification could easily lead to test case generation. Then, the works cited here that do not generate test cases could do that if it was they wanted to.

Conclusion and Future Work

It is well known that quality is one of the most important goals of any company that produces software. Some improvements during previous steps of the software life cycle can be accomplished in order to hit this target. The first artifact to give a special attention is the requirements specification, since the sooner a problem is found during the software development cycle the cheaper is to fix it [BBL76].

Guaranteeing that a software is as free of bugs as possible is another very important point to increase its quality. In this way, an essential activity for that purpose is software testing, whose main role is to find defects in the product, so that the development team can fix them before the product reaches the customer. One way to achieve that is to produce test cases that present high probability of revealing a fault that was not identified yet, with a minimum amount of time and effort. In addition to that, the test case itself should be reliable. Automatic generation of tests from requirements (preferably formal requirements) gives us more confidence on the soundness of the test suite.

Even though most of the software engineers frequently hesitate in using formal models, the benefits that they bring to software quality are enormous. For example, a formal specification can be used to verify the overall system properties, and besides that, it promotes a better and more precise description of the system behaviour, therefore contributing to avoid the introduction of errors.

As software developers, our work is to create mechanisms that make it possible to automate tasks to our clients. It is more than natural that we bring automation to our environment too, in order to develop software better and faster. This work puts together every point made above so as to improve product quality and facilitates software engineers job; in this way, this dissertation proposes an automatic generation of CSP formal models from use case specifications. In our domain of application, such a CSP model is used to represent feature use cases, which are able to describe inputs, outputs, guards and variables assignment, in addition to the specification of a control flow.

In order to translate use cases into CSP, we proposed a Controlled Natural Language (CNL) to guarantee the absence of ambiguity and inconsistent sentences. The first contribution of our work was the specification of a language for use case specification that is as friendly as possible and that allows us to define and manipulate data, inputs and outputs. The CNL description of the use cases [TLB06], combined with the data fields and state manipulation we proposed, is automatically translated to CSP process algebraic models. The generated specification is the formal representation of the system, which holds the same system behaviour as specified by the use case document. Since a CSP specification is inherently stateless, we model state information as a separate process, which is accessed by the application via get and set channels.

With the CSP model in hand, the system properties (such as deadlock, livelock and non-determinism) can be easily verified through the use of the FDR model checker. Besides that, the generated CSP may be used as input for the generation of feature test cases, as reported in the work of Nogueira *et al.* [NSM], which proposes a strategy to automatically generate test cases from CSP. Although Nogueira *et al.* have proposed a strategy for automation, no concrete implementation of it had been implemented previously. The CSP processes accepted by the test generation strategy proposed by Nogueira *et al.* are fully compatible to the CSP we generate from use cases. After that, state based test selection is also possible, which is an important task to reduce cost and time effort dedicated during software testing activities, besides producing sound (reliable) test cases.

This work is part of a research effort between Motorola and the Informatics Center/UFPE for improving Motorola's software testing process. Thus, we conducted a case study in the Motorola environment in order to validate the proposed strategy. The Motorola's use case specification had to go through some adjustments to make it fit in the template, as well as some definitions of data and its manipulation were created in conformance to Motorola's behavioral specification and to our template. Both the language supporting data and its generated CSP model worked well in a real case study of industry.

Besides the definition of a language to capture data definitions and their manipulations, and the implementation of an automatic generation of CSP models from use case documents, this dissertation also has contributed to provide some improvements in the strategy proposed by Nogueira's work. During the development of the software for CSP generation we faced some situations in the strategy that had not being considered originally. For example, the translation of an input sentence into its correspondent CSP event was not generic enough to support different situations of restriction in the input variable. Thus, by implementing a translation proposed originally on paper, we could pay more attention to all those situations and take them into account in the original strategy for generating the CSP from use cases.

7.1 Future Work

In what follows we outline some improvements to complement the work presented here.

- **Type verification of data fields**

Our CSP generator reads every data field and translate to CSP sentences considering that all types of constants and variables (and their manipulation) are correctly specified. Then, one future work is to verify all types of the specified data, checking if everything was defined and used in the way it should be and if there are any inconsistencies. For example, suppose that a variable of type Natural was defined, and then this variable is assigned to a Set. This should result in a type error.

- **Inclusion of parameterization in the use case template and in the implementation of the CSP generation**

Another future work is to propose a way of defining parameters in the use case specification. Also a way of using these parameters in the use case steps, in a friendly way, needs

to be proposed. For example, suppose that a parameter `FNAME` is defined in the scope of a feature; and Figure 7.1 shows a use case using this parameter. In this situation, it is possible to make the selection of messages generic, and so, depending on the parameter value, the use case represents the messages selection in different folders. After that, the implementation also has to be extended to support this new feature, thus, the generated CSP shall consider the parameterization, corresponding to the use case specification.

UC1 - Selecting Messages <<auxiliary>>

Main Flow

Step Id	Action	System State	System Response
1M	Go to <code>\$FNAME\$</code> folder.		All <code>\$FNAME\$</code> Messages are displayed.
2M	Select <code>\$FNAME\$</code> message(s).		Message(s) are highlighted.

Figure 7.1 Parameterization example

- **Integrate this work with the work of Nogueira *et al.* [NSM]**

The work of Nogueira *et al.* [NSM] describes a strategy for the automatic generation of test cases from use case templates that capture control flow, state, input and output. In other words, it accepts a use case specification in the CSP process algebra as input to generate test cases. This approach allows test scenario selection based on particular traces or states of the model. The test generation is expressed as counter-examples of refinement checking, mechanized using the FDR tool. Taking everything into consideration, the CSP generated by our work can be used as an input to Nogueira's work to automatically generate test cases and to capture test selection by CSP test purposes (CSP processes that describe the properties of interest to be captured by the generated tests). The final outcome of this integrated system (our generated CSP model used as input for test case generation) is a set of test cases translated back to CNL (for manual execution).

- **Apply an abstraction approach to automatically transform infinite CSP models into finite ones**

Because the FDR tool enumerates data, which easily leads to state space explosion, we plan to apply the abstraction approach proposed by Mota *et al.* [MBS02, DFM09] to automatically transform infinite CSP models into finite ones with behavior preservation. Such an abstraction corresponds to find equivalent classes of data that are suitable to select a representative and finite set of use case behaviors and avoid state space explosion.

- **Translation of CNL sentences into sentences containing data**

One future work that could also be developed is to find a way to implement a translation of the CNL sentences, written in every use case steps that would contain data sentences, to the respective data fields. For example, Figure 7.2 shows a use case flow without our

data fields. The idea of this proposed work would be to translate the sentence “Select inbox message(s).” to the data field sentence “Input x from powerSet of inbox such that x is non-empty” and the sentence “Message(s) are highlighted.” to the sentence “selected := x”. That translation would be done for the whole use case document.

UC01 - Selecting Inbox Messages <<auxiliary>>

Main Flow

Step Id	User Action	System State	System Response
1M	Go to Inbox folder.		All Inbox Messages are displayed.
2M	Select inbox message(s).		Message(s) are highlighted.

Figure 7.2 Use Case Main Flow Example

- **Parsing the use case specification in MS Word template to XML**

Once the implementation of the CSP generation takes as input a use case document in a XML format, an important work to do is to parse the use case document written in MS Word format to its representation in XML, following the Schema used in this work. The type verification of all data definitions and their use (the first future work proposed here) can be done first; and after every data constructors are accepted, the parsing would be executed.

- **Treating CSP restrictions in the CSP model generator**

As we could see in Section 5.9, there are some restrictions in defining the elements of datatypes in CSP, where we can not use numbers and strings containing special characters. Because of this, it is necessary to define the New Types in the use case document following the same restrictions. One thing that we can try to do in the future is to treat these restrictions in the generated CSP model, in order to allow the user to enter with type elements like these.

APPENDIX A

Complete BNF

This appendix presents the complete BNF for the use case specification, including the data definition and manipulation. The fields in bold are considered primitive types (or a user defined type in the case of a `NewTypeValue`).

Document ::= FeatureList

FeatureList ::= Feature | Feature FeatureList

Feature ::= FID FeatureDef

FID ::= ([0-9])+

FeatureDef ::= FName [DataDefinition] UseCaseList

FName ::= **StringValue**

DataDefinition ::= NameTypeList | NewTypeList | ConstantList | VariableList

NameTypeList ::= Nametype | Nametype NameTypeList

NameType ::= ID Description NameTypeElements

ID ::= (([a-zA-Z_])+([a-zA-Z_0-9])*)?

Description ::= **StringValue**

NameTypeElements ::= SetLiteral | SetRange

SetLiteral ::= **SetValue**

SetRange ::= “[” **NaturalValue** “,” **NaturalValue** “]”

NewTypeList ::= NewType | NewType NewTypeList

NewType ::= ID Description NewTypeElements

NewTypeElements ::= BaseTypeList | Indexing | Enumeration

BaseTypeList ::= BaseType | BaseType “|” BaseTypeList

BaseType ::= Tag “.” TypeID

Tag ::= ID

TypeID ::= ID

Indexing ::= Tag SetRange

Enumeration ::= NewTypeId | NewTypeId “;” Enumeration

NewTypeId ::= ([a-zA-Z_0-9])+([a-zA-Z_.0-9])*

ConstantList ::= Constant | Constant ConstantList

Constant ::= ID Description Expression

VariableList ::= Variable | Variable VariableList

Variable ::= ID Description Expression

UseCaseList ::= UseCase | UseCase UseCaseList

UseCase ::= UCID UCName [RelationList] [DataDefinition] FlowList

UCID ::= ([0-9a-zA-Z_])+

UCName ::= **StringValue**

RelationList ::= IncludeList | ExtendList | ExtensionPointList

IncludeList ::= Include | Include “;” IncludeList

Include ::= UCID “@” Position

Position ::= StartEnd | StepID

StartEnd ::= “START” | “END”

StepID ::= (([0-9])+([A-Z])+)?

ExtendList ::= Extend | Extend “,” ExtendList

Extend ::= “(” Guard “,” UCID “@” Label “)”

Label ::= **StringValue**

Guard ::= Expression

Expression ::= Value | UnaryExp | BinaryExp | ID

Value ::= **NaturalValue** | **BoolValue** | **SetValue** | **NewTypeValue**

UnaryExp ::= “#” Expression
 | “powerSet of” Expression
 | Expression “is non-empty”
 | Expression “is empty”
 | “not” Expression

BinaryExp ::= Expression “+” Expression
 | Expression “-” Expression
 | Expression “*” Expression
 | Expression “>” Expression
 | Expression “>=” Expression
 | Expression “<” Expression
 | Expression “<=” Expression
 | Expression “=” Expression
 | Expression “!=” Expression
 | Expression “and” Expression
 | Expression “or” Expression
 | Expression “is in” Expression
 | Expression “is not in” Expression

ExtensionPointList ::= ExtensionPoint | ExtensionPoint “,” ExtensionPointList

ExtensionPoint ::= Label “:” Position

FlowList ::= Flow | Flow FlowList

Flow ::= FlowType FromStep ToStep StepList

FlowType ::= “Main Flow” | “Alternative Flow”

FromStep ::= “From Step: ” Position

ToStep ::= “To Step: ” Position

StepList ::= Step | Step StepList

Step ::= StepID Action [InputList] Condition [Guard] Response [OutputAssignList]

Action ::= **StringValue**

InputList ::= Input | Input “,” InputList

Input ::= “Input” ID “from” Expression [“such that” Expression]

Condition ::= **StringValue**

Response ::= **StringValue**

OutputAssignList ::= Output
 | Assign
 | Output “,” OutputAssignList
 | Assign “,” OutputAssignList

Output ::= “Output“ Expression

Assign ::= ID “:=” Expression

APPENDIX B

Complete XML Schema for use case template

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="user-view.target.v20071129"
  xmlns:spec="user-view.target.v20071129">

  <!-- defines the featureId type -->
  <xs:simpleType name="featureId">
    <xs:restriction base="xs:string">
      <xs:pattern value="([0-9])+" />
    </xs:restriction>
  </xs:simpleType>

  <!-- definition of useCaseId type -->
  <xs:simpleType name="useCaseId">
    <xs:restriction base="xs:string">
      <xs:pattern value="([0-9a-zA-Z_])+" />
    </xs:restriction>
  </xs:simpleType>

  <!-- definition of step Id type -->
  <xs:simpleType name="stepId">
    <xs:restriction base="xs:string">
      <xs:pattern value="(([0-9])+([A-Z])+)?" />
    </xs:restriction>
  </xs:simpleType>

  <!-- defines the Id type -->
  <xs:simpleType name="IdType">
    <xs:restriction base="xs:string">
      <xs:pattern value="(([a-zA-Z_])+([a-zA-Z_0-9])*)?" />
    </xs:restriction>
  </xs:simpleType>

  <!-- defines the new type Id value -->
```

```

<xs:simpleType name="newTypeValue">
  <xs:restriction base="xs:string">
    <xs:pattern value="([a-zA-Z_.0-9]*)?" />
  </xs:restriction>
</xs:simpleType>

<!-- defines the Set Value -->
<xs:complexType name="setValue">
  <xs:sequence>
    <xs:element name="element" type="spec:expression"
      maxOccurs="unbounded" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<!-- defines the possible types and a value -->
<xs:complexType name="typedValue">
  <xs:choice>
    <xs:element name="Natural" type="xs:unsignedInt"/>
    <xs:element name="Bool" type="xs:boolean"/>
    <xs:element name="SetValue" type="spec:setValue"/>
    <xs:element name="NewType" type="spec:newTypeValue"/>
  </xs:choice>
</xs:complexType>

<!-- defines the flowType type -->
<xs:simpleType name="flowType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Main_Flow" />
    <xs:enumeration value="Alternative_Flow" />
  </xs:restriction>
</xs:simpleType>

  <!-- defines the unaryOperator type -->
<xs:simpleType name="unaryOperator">
  <xs:restriction base="xs:string">
    <xs:enumeration value="#" />
    <xs:enumeration value="powerset_of" />
    <xs:enumeration value="is_non-empty" />
    <xs:enumeration value="is_empty" />
    <xs:enumeration value="not" />
  </xs:restriction>
</xs:simpleType>

```

```

<!-- defines the binaryOperator type -->
<xs:simpleType name="binaryOperator">
  <xs:restriction base="xs:string">
    <xs:enumeration value="+" />
    <xs:enumeration value="-" />
    <xs:enumeration value="*" />
    <xs:enumeration value="&gt;" />
    <xs:enumeration value="&gt;=" />
    <xs:enumeration value="&lt;" />
    <xs:enumeration value="&lt;=" />
    <xs:enumeration value="=" />
    <xs:enumeration value="!=" />
    <xs:enumeration value="and" />
    <xs:enumeration value="or" />
    <xs:enumeration value="is_in" />
    <xs:enumeration value="is_not_in" />
  </xs:restriction>
</xs:simpleType>

<!-- defines the expression type -->
<xs:complexType name="expression">
  <xs:choice>
    <xs:element name="value" type="spec:typedValue" />
    <xs:element name="unaryExpression" >
      <xs:complexType>
        <xs:sequence>
          <xs:element name="unaryOperator"
            type="spec:unaryOperator" />
          <xs:element name="expression"
            type="spec:expression" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="binaryExpression" >
      <xs:complexType>
        <xs:sequence>
          <xs:element name="expression1"
            type="spec:expression" />
          <xs:element name="binaryOperator"
            type="spec:binaryOperator" />
          <xs:element name="expression2"
            type="spec:expression" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:complexType>

```

```

        </xs:complexType>
    </xs:element>
    <xs:element name="Id" type="spec:IdType" />
</xs:choice>
</xs:complexType>

<!-- defines the nameTypeElements type -->
<xs:complexType name="nameTypeElements">
    <xs:choice>
        <xs:element name="setLiteral"
type="spec:setValue" />
        <xs:element name="setRange" >
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="firstValue"
type="xs:unsignedInt" />
                    <xs:element name="lastValue"
type="xs:unsignedInt" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:choice>
</xs:complexType>

<!-- defines the newTypeElements type -->
<xs:complexType name="newTypeElements">
    <xs:choice>
        <xs:element name="baseType" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="tag" type="spec:IdType" />
                    <xs:element name="typeId" type="spec:IdType" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="indexing" >
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="tag" type="spec:IdType" />
                    <xs:element name="firstValue"
type="xs:unsignedInt" />
                    <xs:element name="lastValue"
type="xs:unsignedInt" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:choice>
</xs:complexType>

```



```

        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="enumeration" >
    <xs:complexType>
        <xs:sequence>
            <xs:element name="element"
                type="spec:newTypeValue" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>

<!-- beginning of definition of type dataDefinition -->
<xs:complexType name="dataDefinition">
    <xs:sequence>
        <xs:element name="nameType" maxOccurs="unbounded"
            minOccurs = "0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="id" type="spec:IdType" />
                    <xs:element name="description" type="xs:string" />
                    <xs:element name="nameTypeElements"
                        type="spec:nameTypeElements" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="newType" maxOccurs="unbounded"
            minOccurs = "0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="id" type="spec:IdType" />
                    <xs:element name="description" type="xs:string" />
                    <xs:element name="newTypeElements"
                        type="spec:newTypeElements" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="constant" maxOccurs="unbounded"
            minOccurs = "0">
            <xs:complexType>
                <xs:sequence>

```

```

        <xs:element name="id" type="xs:string" />
        <xs:element name="description"
            type="xs:string" />
        <xs:element name="constValue"
            type="spec:expression" />
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="variable" maxOccurs="unbounded"
    minOccurs = "0">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="id" type="xs:string" />
            <xs:element name="description" type="xs:string" />
            <xs:element name="initialValue"
                type="spec:expression" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>

<!-- beginning of definition of type relations -->
<xs:complexType name="relations">
    <xs:sequence>
        <xs:element name="include" maxOccurs="unbounded"
            minOccurs = "0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="useCaseId"
                        type="spec:useCaseId" />
                    <xs:element name="position" type="xs:string" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="extension" maxOccurs="unbounded"
            minOccurs = "0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="guard" type="spec:expression" />
                    <xs:element name="useCaseId"
                        type="spec:useCaseId" />
                    <xs:element name="label" type="xs:string" />
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

```

```

        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="extensionPoint" maxOccurs="unbounded"
minOccurs = "0">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="label" type="xs:string" />
            <xs:element name="stepId" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>

<!-- beginning of definition of type Flow -->
<xs:complexType name="flow">
    <xs:sequence>
        <xs:element name="flowType" type="spec:flowType" />
        <xs:element name="description" type="xs:string" />
        <xs:element name="fromSteps" type="xs:string" />
        <xs:element name="toSteps" type="xs:string" />
        <!-- beginning of definition of element step -->
        <xs:element maxOccurs="unbounded" name="step">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="stepId" type="spec:stepId" />
                    <xs:element name="action" type="xs:string" />
                    <xs:element name="input" minOccurs="0"
maxOccurs="unbounded" >
                        <xs:complexType>
                            <xs:sequence>
                                <xs:element name="var" type="spec:IdType"/>
                                <xs:element name="expression"
type="spec:expression" />
                                <xs:element name="restriction"
type="spec:expression" minOccurs="0" />
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="condition" type="xs:string"
minOccurs="0" />
                    <xs:element name="guard" type="spec:expression"

```

```

minOccurs="0" />
<xs:element name="setup" type="xs:string"
maxOccurs="1" minOccurs="0"/>
<xs:element name="response" type="xs:string" />
<xs:element name="outputAssignList" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="output"
type="spec:expression"
minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="assign" minOccurs="0"
maxOccurs="unbounded" >
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Id"
type="spec:IdType" />
            <xs:element name="expression"
type="spec:expression" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
<!-- end of definition of type flow -->

```

```

<!-- beginning of definition of type useCaseType -->
<xs:complexType name="useCaseType">
  <xs:sequence>
    <xs:element name="id" type="spec:useCaseId" />
    <xs:element name="name" type="xs:string" />
    <xs:element name="auxiliary" type="xs:boolean"
default="false" minOccurs="0"/>
    <xs:element name="description" type="xs:string" />
    <xs:element name="setup" type="xs:string" />
    <xs:element name="dataDefinition"
type="spec:dataDefinition" minOccurs="0"/>

```

```

    <xs:element name="relations" minOccurs="0"
      type="spec:relations" />
    <xs:element name="flow" type="spec:flow"
      maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<!-- end of definition of type useCaseType -->

<!-- beginning of definition of type interruptionType -->
<xs:complexType name="interruptionType">
  <xs:sequence>
    <xs:element name="id" type="spec:useCaseId" />
    <xs:element name="name" type="xs:string" />
    <xs:element name="description" type="xs:string" />
    <xs:element maxOccurs="unbounded" name="flow">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="description" type="xs:string" />
          <xs:element name="fromSteps" type="xs:string" />
          <xs:element name="toSteps" type="xs:string" />
          <!-- beginning of definition of element step -->
          <xs:element maxOccurs="unbounded" name="step">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="stepId"
                  type="spec:stepId" />
                <xs:element name="action"
                  type="xs:string" />
                <xs:element name="condition"
                  type="xs:string" />
                <xs:element name="response"
                  type="xs:string" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<!-- end of definition of type interruptionType -->

```

```

<xs:complexType name="featureType">
  <xs:sequence>
    <xs:element name="featureId" type="spec:featureId" />
    <xs:element name="name" type="xs:string" />
    <xs:element name="dataDefinition"
      type="spec:dataDefinition" minOccurs="0"/>
    <!-- beginning of definition of element useCase -->
    <xs:element maxOccurs="unbounded" name="useCase"
      type="spec:useCaseType" />
    <!-- end of definition of element useCase -->
  </xs:sequence>
</xs:complexType>

<xs:complexType name="phoneType">
  <xs:sequence>
    <xs:element name="feature" minOccurs="0"
      maxOccurs="unbounded" type="spec:featureType" />
    <xs:element name="interruption" minOccurs="0"
      maxOccurs="unbounded" type="spec:interruptionType" />
  </xs:sequence>
</xs:complexType>

<!-- definition of system phone -->
<xs:element name="phone" type="spec:phoneType" />
</xs:schema>

```

APPENDIX C

CSPm generated from *Important Messages* feature

```
transparent sbdia

-- composes the memory and the flow and compress the composition
-- special event is used to force successful termination of
-- composition whenever the left RHS does

channel success
FEND(F) = F; success -> SKIP
MEND(Mem) = Mem /\ (success -> SKIP)

-- the channels used for memory read/writte
-- the values transmitted by the memory will be defined further

channel get,set : Var.Type

-- yields the range of a var which type us tn
values(tn) = { val | v<-Var, val<-extensions(get.v.tn) }

-- the memory process

Mcel(v,val) = get!v!val -> Mcel(v,val)
             [] let t = tag(val) within
                set!v!t?val_:values(t) -> Mcel(v,t.val_)

Memory(binding) = ||| (v,val) : binding @ Mcel(v,val)

-- given a set of variables mVars, this function yields the
-- alphabet of get/set events over the given variables

pick({x}) = x
varType(v,b) = pick( { t | (v2,t)<-b, v == v2 } )
dom(m) = { x | (x,y)<-m }

alphaMem(b)={g,s | vname<-dom(b), tname<-{tag(varType(vname,b))},
                g<-productions(get.vname.tname),
                s<-productions(set.vname.tname) }
```

```

-- given a memory process MEM, this function creates a copy get2 of
-- the channel get

dupGet (MEM)=MEM[[get.x<-get.x,get.x <- get2.x|x <- extensions(get)]]

-- channel dedicated to test selection, used to read the value
-- of the variables

channel get2 : Var.Type

-- given a set of variables mVars, this function yields the
-- alphabet of get2 events over the given variables

alphaGet2(b) = { get2.v.t.val | get.v.t.val <- alphaMem(b) }

-- control event that makes the test selection more efficient
-- we will only verify the system state after steps that change the
-- state mem_update is an auxiliary event to mark mem_update of a
-- sequence of variables update after the test purpose synchronizes
-- in this event it verifies the boolean guards

channel mem_update

-- a subset of the naturals set which will be used in every
-- occurrence of declared variables of such type
nametype Number = {0..5}

--11169
datatype Var = f11169_inbox | f11169_UC03_selected | f11169_selected
              | f11169_important
datatype Type = t11169_1.Set(t11169_Message)

tag(t11169_1.v) = t11169_1

-- 11169 data

-- types
nametype t11169_Natural = {0..2}
datatype t11169_Message = M.t11169_Natural

-- constants
f11169_MAX = 2

-- memory
b11169_MEM = {

```



```

(f11169_inbox,t11169_1.{(M.0),(M.1)}),
(f11169_selected,t11169_1.{}),
(f11169_important,t11169_1.{(M.2)})
}

a11169_MEM = alphaMem(b11169_MEM)

f11169_MEMORY = dupGet( Memory(b11169_MEM) )

-- 11169_UC01

-- types

-- constants

-- memory

-- input values
channel in_11169_UC01_2M_x : Set(t11169_Message)

a11169_UC01in = {|in_11169_UC01_2M_x|}

-- output values

-- channels
channel action11169_UC01_1M, response11169_UC01_1M,
action11169_UC01_2M, response11169_UC01_2M

--uc alphabets
a11169_UC01i = Union({action11169_UC01_1M, action11169_UC01_2M,
a11169_UC01in})
a11169_UC01o = Union({response11169_UC01_1M, response11169_UC01_2M})
a11169_UC01 = Union({a11169_UC01i, a11169_UC01o})

f11169_UC01_FLOW = let

f11169_UC01_START =
SKIP;
f11169_UC01_1M

f11169_UC01_1M =
action11169_UC01_1M ->
response11169_UC01_1M ->
SKIP;
f11169_UC01_2M

```

```

f11169_UC01_2M =
  get!f11169_inbox.t11169_1?inbox ->
  in_11169_UC01_2M_x?x : { v | v<-Set(inbox), (not empty(v)) } ->
  action11169_UC01_2M ->
  response11169_UC01_2M ->
  set!f11169_selected!t11169_1.x -> mem_update ->
  SKIP

within f11169_UC01_START

f11169_UC01 = f11169_UC01_FLOW

-- 11169_UC02

-- types

-- constants

-- memory

-- input values

-- output values
channel out1_11169_UC02_1A : Number

a11169_UC02out = {|out1_11169_UC02_1A|}

-- channels
channel action11169_UC02_1M, response11169_UC02_1M,
action11169_UC02_1A, response11169_UC02_1A

--uc alphabets
a11169_UC02i = Union({action11169_UC02_1M, action11169_UC02_1A})
a11169_UC02o = Union({response11169_UC02_1M, response11169_UC02_1A,
a11169_UC02out})
a11169_UC02 = Union({a11169_UC02i, a11169_UC02o})

f11169_UC02_FLOW(f11169_UC02_Cleanup) = let

f11169_UC02_START =
  f11169_UC01;
  SKIP;
  f11169_UC02_1M [] f11169_UC02_1A

f11169_UC02_1M =
  get!f11169_important.t11169_1?important ->

```

```

get!f11169_selected.t11169_1?selected ->
get!f11169_inbox.t11169_1?inbox ->
card(union(important,selected)) <= f11169_MAX &
action11169_UC02_1M ->
response11169_UC02_1M ->
set!f11169_inbox!t11169_1.diff(inbox,selected) ->
set!f11169_important!t11169_1.union(important,selected) ->
mem_update ->
SKIP

f11169_UC02_1A =
get!f11169_important.t11169_1?important ->
get!f11169_selected.t11169_1?selected ->
card(union(important,selected)) > f11169_MAX &
action11169_UC02_1A ->
response11169_UC02_1A ->
out1_11169_UC02_1A!(card(union(important,selected))-f11169_MAX) ->
SKIP;
(f11169_UC02_Cleanup [] SKIP);
f11169_UC02_START

within f11169_UC02_START

f11169_UC02(f11169_UC02_Cleanup) = f11169_UC02_FLOW(
  f11169_UC02_Cleanup)

-- 11169_UC03

-- types

-- constants

-- memory
b11169_UC03_MEM = {
  (f11169_UC03_selected,t11169_1.{})
}

a11169_UC03_MEM = alphaMem(b11169_UC03_MEM)

f11169_UC03_MEMORY = dupGet( Memory(b11169_UC03_MEM) )

-- input values
channel in_11169_UC03_1M_x : Set(t11169_Message)

a11169_UC03in = {|in_11169_UC03_1M_x|}

```

```

-- output values

-- channels
channel action11169_UC03_1M, response11169_UC03_1M,
action11169_UC03_2M, response11169_UC03_2M

--uc alphabets
a11169_UC03i = Union({action11169_UC03_1M, action11169_UC03_2M,
a11169_UC03in})
a11169_UC03o = Union({response11169_UC03_1M, response11169_UC03_2M})
a11169_UC03 = Union({a11169_UC03i, a11169_UC03o})

f11169_UC03_Ext_11169_UC02_Cleanup =
get!f11169_important.t11169_1?important ->
card(important) > 0 & f11169_UC03

f11169_UC03_FLOW = let

f11169_UC03_START =
SKIP;
f11169_UC03_1M

f11169_UC03_1M =
get!f11169_important.t11169_1?important ->
in_11169_UC03_1M_x?x : { v | v<-Set(important), (not empty(v)) } ->
action11169_UC03_1M ->
response11169_UC03_1M ->
set!f11169_UC03_selected!t11169_1.x -> mem_update ->
SKIP;
f11169_UC03_2M

f11169_UC03_2M =
get!f11169_important.t11169_1?important ->
get!f11169_UC03_selected.t11169_1?selected ->
action11169_UC03_2M ->
response11169_UC03_2M ->
set!f11169_important!t11169_1.diff(important,selected) ->
mem_update ->
SKIP

within f11169_UC03_START

(
FEND(f11169_UC03_FLOW)
[|union(a11169_UC03_MEM, {success})|]
MEND(f11169_UC03_MEMORY)

```

```
) \ union(a11169_UC03_MEM, {success})

-- 11169 behaviour
aget2_11169 = Union({alphaGet2(b11169_MEM),
alphaGet2(b11169_UC03_MEM)})

a11169 = Union({a11169_UC01, a11169_UC02, a11169_UC03})

f11169 =
(
  FEND(f11169_UC02(f11169_UC03_Ext_11169_UC02_Cleanup) []f11169_UC03)
  [|union(a11169_MEM, {success})|]
  MEND(f11169_MEMORY)
) \ union(a11169_MEM, {success})

-- system
aget2 = Union({aget2_11169})
acontrol = union(aget2, {mem_update})
aS = Union({a11169, acontrol})

S = f11169
```

APPENDIX D

CSPm generated from *Sending and Receiving* SMS/MMS feature

```
transparent sbdia

-- composes the memory and the flow and compress the composition
-- special event is used to force successful termination of
-- composition whenever the left RHS does

channel success
FEND(F) = F; success -> SKIP
MEND(Mem) = Mem /\ (success -> SKIP)

-- the channels used for memory read/writte
-- the values transmitted by the memory will be defined further

channel get,set : Var.Type

-- yields the range of a var which type us tn
values(tn) = { val | v<-Var, val<-extensions(get.v.tn) }

-- the memory process

Mcel(v,val) = get!v!val -> Mcel(v,val)
             [] let t = tag(val) within
                set!v!t?val_:values(t) -> Mcel(v,t.val_)

Memory(binding) = ||| (v,val) : binding @ Mcel(v,val)

-- given a set of variables mVars, this function yields the
-- alphabet of get/set events over the given variables

pick({x}) = x
varType(v,b) = pick( { t | (v2,t)<-b, v == v2 } )
dom(m) = { x | (x,y)<-m }

alphaMem(b)={g,s | vname<-dom(b), tname<-{tag(varType(vname,b))},
                g<-productions(get.vname.tname),
```

```

s<-productions(set.vname.tname) }

-- given a memory process MEM, this function creates a copy get2 of
-- the channel get

dupGet(MEM)=MEM[[get.x<-get.x,get.x <- get2.x|x <- extensions(get)]]

-- channel dedicated to test selection, used to read the value
-- of the variables

channel get2 : Var.Type

-- given a set of variables mVars, this function yields the
-- alphabet of get2 events over the given variables

alphaGet2(b) = { get2.v.t.val | get.v.t.val <- alphaMem(b) }

-- control event that makes the test selection more efficient
-- we will only verify the system state after steps that change the
-- state mem_update is an auxiliary event to mark mem_update of a
-- sequence of variables update after the test purpose synchronizes
-- in this event it verifies the boolean guards

channel mem_update

-- a subset of the naturals set which will be used in every
-- occurrence of declared variables of such type
nametype Number = {0..5}

--33629
datatype Var = f33629_selected | f33629_UC7_current_type
              | f33629_UC3_current_mode
datatype Type = t33629_1.Set(t33629_contact)
              | t33629_2.t33629_UC7_address_type
              | t33629_3.t33629_UC3_mode

tag(t33629_1.v) = t33629_1
tag(t33629_2.v) = t33629_2
tag(t33629_3.v) = t33629_3

-- 33629 data

-- types
datatype t33629_number = n_99887766 | n_88446622
datatype t33629_address = js_AT_gmail_DOT_com | ps_AT_ufpe_DOT_br
datatype t33629_contact = Phone.t33629_number | Email.t33629_address

```

```

datatype t33629_interaction = dialog_call | SMS_notification

-- constants
f33629_max_selection = 10
f33629_all_contacts = {(Phone.n_99887766), (Phone.n_88446622),
                      (Email.js_AT_gmail_DOT_com),
                      (Email.ps_AT_ufpe_DOT_br)}
f33629_JohnSmith_info = {(Phone.n_99887766),
                        (Email.js_AT_gmail_DOT_com)}
f33629_UFPE_group = {(Phone.n_88446622), (Email.ps_AT_ufpe_DOT_br)}

-- memory
b33629_MEM = {
  (f33629_selected, t33629_1.{})
}

a33629_MEM = alphaMem(b33629_MEM)

f33629_MEMORY = dupGet( Memory(b33629_MEM) )

-- 33629_UC1

-- types

-- constants

-- memory

-- input values

-- output values

-- channels
channel action33629_UC1_1M, response33629_UC1_1M,
       action33629_UC1_2M, response33629_UC1_2M

--uc alphabets
a33629_UC1i = Union({action33629_UC1_1M, action33629_UC1_2M})
a33629_UC1o = Union({response33629_UC1_1M, response33629_UC1_2M})
a33629_UC1 = Union({a33629_UC1i, a33629_UC1o})

f33629_UC1_FLOW = let

f33629_UC1_START =
  SKIP;
  f33629_UC1_1M

```



```

f33629_UC1_1M =
  action33629_UC1_1M ->
  response33629_UC1_1M ->
  SKIP;
  f33629_UC1_2M

f33629_UC1_2M =
  action33629_UC1_2M ->
  response33629_UC1_2M ->
  SKIP

within f33629_UC1_START

f33629_UC1 = f33629_UC1_FLOW

-- 33629_UC2

-- types

-- constants

-- memory

-- input values
channel in_33629_UC2_1M_x : Set (t33629_contact)
channel in_33629_UC2_1A_x : Set (t33629_contact)
channel in_33629_UC2_1B_x : Set (t33629_contact)
channel in_33629_UC2_1C_x : Set (t33629_contact)
channel in_33629_UC2_1D_x : Set (t33629_contact)
channel in_33629_UC2_1E_x : Set (t33629_contact)

a33629_UC2in = {|in_33629_UC2_1M_x, in_33629_UC2_1A_x,
                in_33629_UC2_1B_x, in_33629_UC2_1C_x,
                in_33629_UC2_1D_x, in_33629_UC2_1E_x|}

-- output values
channel out1_33629_UC2_1A : Number
channel out1_33629_UC2_1C : Number
channel out1_33629_UC2_1E : Number

a33629_UC2out = {|out1_33629_UC2_1A, out1_33629_UC2_1C,
                 out1_33629_UC2_1E|}

-- channels
channel action33629_UC2_1M, response33629_UC2_1M,

```

```

    action33629_UC2_2M, response33629_UC2_2M,
    action33629_UC2_1A, response33629_UC2_1A,
    action33629_UC2_1B, response33629_UC2_1B,
    action33629_UC2_1C, response33629_UC2_1C,
    action33629_UC2_1D, response33629_UC2_1D,
    action33629_UC2_1E, response33629_UC2_1E

--uc alphabets
a33629_UC2i = Union({action33629_UC2_1M, action33629_UC2_2M,
                    action33629_UC2_1A, action33629_UC2_1B,
                    action33629_UC2_1C, action33629_UC2_1D,
                    action33629_UC2_1E, a33629_UC2in})
a33629_UC2o = Union({response33629_UC2_1M, response33629_UC2_2M,
                    response33629_UC2_1A, response33629_UC2_1B,
                    response33629_UC2_1C, response33629_UC2_1D,
                    response33629_UC2_1E, a33629_UC2out})
a33629_UC2 = Union({a33629_UC2i, a33629_UC2o})

f33629_UC2_FLOW(f33629_UC2_Interruption1, f33629_UC2_Interruption2,
                f33629_UC2_Interruption3, f33629_UC2_Interruption4,
                f33629_UC2_Interruption5, f33629_UC2_Interruption6)
= let

f33629_UC2_START =
  f33629_UC1;
  SKIP;
  f33629_UC2_1M [] f33629_UC2_1A [] f33629_UC2_1E [] f33629_UC2_1D
  [] f33629_UC2_1C [] f33629_UC2_1B

f33629_UC2_1M =
  in_33629_UC2_1M_x?x : {v | v<-Set(f33629_all_contacts),
                        (not empty(v))} ->
  card(x) <= f33629_MAX_SELECTION &
  action33629_UC2_1M ->
  response33629_UC2_1M ->
  set!f33629_selected!t33629_1.x -> mem_update ->
  SKIP;
  (f33629_UC2_Interruption1 [] SKIP);
  f33629_UC2_2M

f33629_UC2_2M =
  action33629_UC2_2M ->
  response33629_UC2_2M ->
  SKIP

f33629_UC2_1A =

```

```

in_33629_UC2_1A_x?x : { v | v<-Set(f33629_all_contacts),
                      (not empty(v)) } ->
card(x) > f33629_max_selection &
action33629_UC2_1A ->
response33629_UC2_1A ->
out1_33629_UC2_1A!(card(x) - f33629_max_selection) ->
SKIP;
(f33629_UC2_Interruption2 [] SKIP);
f33629_UC2_START

```

```

f33629_UC2_1B =
in_33629_UC2_1B_x?x : { v | v<-Set(f33629_UFPE_group),
                      (not empty(v)) } ->
card(x) <= f33629_max_selection &
action33629_UC2_1B ->
response33629_UC2_1B ->
set!f33629_selected!t33629_1.x -> mem_update ->
SKIP;
(f33629_UC2_Interruption3 [] SKIP);
f33629_UC2_2M

```

```

f33629_UC2_1C =
in_33629_UC2_1C_x?x : { v | v<-Set(f33629_UFPE_group),
                      (not empty(v)) } ->
card(x) > f33629_max_selection &
action33629_UC2_1C ->
response33629_UC2_1C ->
out1_33629_UC2_1C!(card(x) - f33629_max_selection) ->
SKIP;
(f33629_UC2_Interruption4 [] SKIP);
f33629_UC2_START

```

```

f33629_UC2_1D =
in_33629_UC2_1D_x?x : { v | v<-Set(f33629_JohnSmith_info),
                      (not empty(v)) } ->
card(x) <= f33629_max_selection &
action33629_UC2_1D ->
response33629_UC2_1D ->
set!f33629_selected!t33629_1.x -> mem_update ->
SKIP;
(f33629_UC2_Interruption5 [] SKIP);
f33629_UC2_2M

```

```

f33629_UC2_1E =
in_33629_UC2_1E_x?x : { v | v<-Set(f33629_JohnSmith_info),
                      (not empty(v)) } ->

```

```

card(x) > f33629_max_selection &
action33629_UC2_1E ->
response33629_UC2_1E ->
out1_33629_UC2_1E!(card(x) - f33629_max_selection) ->
SKIP;
(f33629_UC2_Interruption6 [] SKIP);
f33629_UC2_START

within f33629_UC2_START

f33629_UC2(f33629_UC2_Interruption1, f33629_UC2_Interruption2,
           f33629_UC2_Interruption3, f33629_UC2_Interruption4,
           f33629_UC2_Interruption5, f33629_UC2_Interruption6) =
f33629_UC2_FLOW(f33629_UC2_Interruption1, f33629_UC2_Interruption2,
                f33629_UC2_Interruption3, f33629_UC2_Interruption4,
                f33629_UC2_Interruption5, f33629_UC2_Interruption6)

-- 33629_UC3

-- types
datatype t33629_UC3_mode = selection_mode | call_mode

-- constants

-- memory
b33629_UC3_MEM = {
(f33629_UC3_current_mode,t33629_3.(selection_mode))
}

a33629_UC3_MEM = alphaMem(b33629_UC3_MEM)

f33629_UC3_MEMORY = dupGet( Memory(b33629_UC3_MEM) )

-- input values

-- output values
channel out1_33629_UC3_1M : t33629_interaction

a33629_UC3out = {|out1_33629_UC3_1M|}

-- channels
channel action33629_UC3_1M, response33629_UC3_1M,
       action33629_UC3_2M, response33629_UC3_2M

--uc alphabets
a33629_UC3i = Union({action33629_UC3_1M, action33629_UC3_2M})

```

```

a33629_UC3o = Union({response33629_UC3_1M, response33629_UC3_2M,
                    a33629_UC3out})
a33629_UC3 = Union({a33629_UC3i, a33629_UC3o})

f33629_UC3_Ext_33629_UC2_Interruption1 =
    true & f33629_UC3

f33629_UC3_Ext_33629_UC2_Interruption2 =
    true & f33629_UC3

f33629_UC3_Ext_33629_UC2_Interruption3 =
    true & f33629_UC3

f33629_UC3_Ext_33629_UC2_Interruption4 =
    true & f33629_UC3

f33629_UC3_Ext_33629_UC2_Interruption5 =
    true & f33629_UC3

f33629_UC3_Ext_33629_UC2_Interruption6 =
    true & f33629_UC3

f33629_UC3_FLOW = let

f33629_UC3_START =
    SKIP;
    f33629_UC3_1M

f33629_UC3_1M =
    action33629_UC3_1M ->
    response33629_UC3_1M ->
    out1_33629_UC3_1M!(dialog_call) ->
    set!f33629_UC3_current_mode!t33629_3.(call_mode) ->
    mem_update ->
    SKIP;
    f33629_UC3_2M

f33629_UC3_2M =
    action33629_UC3_2M ->
    response33629_UC3_2M ->
    set!f33629_UC3_current_mode!t33629_3.(selection_mode) ->
    mem_update ->
    SKIP

within f33629_UC3_START

```

```

f33629_UC3 =
(
  FEND(f33629_UC3_FLOW)
  [|union(a33629_UC3_MEM, {success})|]
  MEND(f33629_UC3_MEMORY)
) \ union(a33629_UC3_MEM, {success})

-- 33629_UC4

-- types

-- constants

-- memory

-- input values

-- output values
channel out1_33629_UC4_1M : t33629_interaction

a33629_UC4out = {|out1_33629_UC4_1M|}

-- channels
channel action33629_UC4_1M, response33629_UC4_1M

  --uc alphabets
a33629_UC4i = Union({action33629_UC4_1M})
a33629_UC4o = Union({response33629_UC4_1M, a33629_UC4out})
a33629_UC4 = Union({a33629_UC4i, a33629_UC4o})

f33629_UC4_Ext_33629_UC2_Interruption1 =
  true & f33629_UC4

f33629_UC4_Ext_33629_UC2_Interruption2 =
  true & f33629_UC4

f33629_UC4_Ext_33629_UC2_Interruption3 =
  true & f33629_UC4

f33629_UC4_Ext_33629_UC2_Interruption4 =
  true & f33629_UC4

f33629_UC4_Ext_33629_UC2_Interruption5 =
  true & f33629_UC4

f33629_UC4_Ext_33629_UC2_Interruption6 =

```

```

true & f33629_UC4

f33629_UC4_FLOW = let

f33629_UC4_START =
  SKIP;
  f33629_UC4_1M

f33629_UC4_1M =
  action33629_UC4_1M ->
  response33629_UC4_1M ->
  out1_33629_UC4_1M!(SMS_notification) ->
  SKIP

within f33629_UC4_START

f33629_UC4 = f33629_UC4_FLOW

-- 33629_UC5

-- types

-- constants

-- memory

-- input values

-- output values

-- channels
channel action33629_UC5_1M, response33629_UC5_1M

--uc alphabets
a33629_UC5i = Union({action33629_UC5_1M})
a33629_UC5o = Union({response33629_UC5_1M})
a33629_UC5 = Union({a33629_UC5i, a33629_UC5o})

f33629_UC5_Ext_33629_UC2_Interruption1 =
  true & f33629_UC5

f33629_UC5_Ext_33629_UC2_Interruption2 =
  true & f33629_UC5

f33629_UC5_Ext_33629_UC2_Interruption3 =
  true & f33629_UC5

```

```

f33629_UC5_Ext_33629_UC2_Interruption4 =
  true & f33629_UC5

f33629_UC5_Ext_33629_UC2_Interruption5 =
  true & f33629_UC5

f33629_UC5_Ext_33629_UC2_Interruption6 =
  true & f33629_UC5

f33629_UC5_FLOW = let

f33629_UC5_START =
  SKIP;
  f33629_UC5_1M

f33629_UC5_1M =
  action33629_UC5_1M ->
  response33629_UC5_1M ->
  SKIP

within f33629_UC5_START

f33629_UC5 = f33629_UC5_FLOW

-- 33629_UC6

-- types

-- constants

-- memory

-- input values

-- output values

-- channels
channel action33629_UC6_1M, response33629_UC6_1M,
        action33629_UC6_2M, response33629_UC6_2M

--uc alphabets
a33629_UC6i = Union({action33629_UC6_1M, action33629_UC6_2M})
a33629_UC6o = Union({response33629_UC6_1M, response33629_UC6_2M})
a33629_UC6 = Union({a33629_UC6i, a33629_UC6o})

```



```

f33629_UC6_FLOW = let

f33629_UC6_START =
  SKIP;
  f33629_UC6_1M

f33629_UC6_1M =
  action33629_UC6_1M ->
  response33629_UC6_1M ->
  SKIP;
  f33629_UC6_2M

f33629_UC6_2M =
  action33629_UC6_2M ->
  response33629_UC6_2M ->
  SKIP

within f33629_UC6_START

f33629_UC6 = f33629_UC6_FLOW

-- 33629_UC7

-- types
datatype t33629_UC7_address_type = number | email

-- constants

-- memory
b33629_UC7_MEM = {
  (f33629_UC7_current_type, t33629_2.(number))
}

a33629_UC7_MEM = alphaMem(b33629_UC7_MEM)

f33629_UC7_MEMORY = dupGet( Memory(b33629_UC7_MEM) )

-- input values
channel in_33629_UC7_1M_sourceType : t33629_UC7_address_type

a33629_UC7in = {|in_33629_UC7_1M_sourceType|}

-- output values

-- channels
channel action33629_UC7_1M, response33629_UC7_1M,

```

```

        action33629_UC7_2M, response33629_UC7_2M,
        action33629_UC7_1A, response33629_UC7_1A

--uc alphabets
a33629_UC7i = Union({action33629_UC7_1M, action33629_UC7_2M,
                    action33629_UC7_1A, a33629_UC7in})
a33629_UC7o = Union({response33629_UC7_1M, response33629_UC7_2M,
                    response33629_UC7_1A})
a33629_UC7 = Union({a33629_UC7i, a33629_UC7o})

f33629_UC7_FLOW = let

f33629_UC7_START =
    SKIP;
    f33629_UC7_1M

f33629_UC7_1M =
    in_33629_UC7_1M_sourceType?sourceType : { v | v <-{(number),
                                                (email)}} ->

    action33629_UC7_1M ->
    response33629_UC7_1M ->
    SKIP;
    f33629_UC7_2M [] f33629_UC7_1A

f33629_UC7_2M =
    get!f33629_UC7_current_type.t33629_2?current_type ->
    current_type == (number) &
    action33629_UC7_2M ->
    response33629_UC7_2M ->
    SKIP

f33629_UC7_1A =
    get!f33629_UC7_current_type.t33629_2?current_type ->
    current_type == (email) &
    action33629_UC7_1A ->
    response33629_UC7_1A ->
    SKIP

within f33629_UC7_START

f33629_UC7 =
    (
        FEND(f33629_UC7_FLOW)
        [|union(a33629_UC7_MEM, {success})|]
        MEND(f33629_UC7_MEMORY)
    ) \ union(a33629_UC7_MEM, {success})

```

```

-- 33629 behaviour
aget2_33629 = Union({alphaGet2(b33629_MEM),
                    alphaGet2(b33629_UC3_MEM),
                    alphaGet2(b33629_UC7_MEM)})

a33629 = Union({a33629_UC1, a33629_UC2, a33629_UC3, a33629_UC4,
              a33629_UC5, a33629_UC6, a33629_UC7})

f33629 =
(
  FEND(f33629_UC3 [] f33629_UC4 [] f33629_UC5 [] f33629_UC6 []
       f33629_UC7 [] f33629_UC2(f33629_UC3_Ext_33629_UC2_Interruption1 []
       f33629_UC4_Ext_33629_UC2_Interruption1 []
       f33629_UC5_Ext_33629_UC2_Interruption1,
       f33629_UC3_Ext_33629_UC2_Interruption2 []
       f33629_UC4_Ext_33629_UC2_Interruption2 []
       f33629_UC5_Ext_33629_UC2_Interruption2,
       f33629_UC3_Ext_33629_UC2_Interruption3 []
       f33629_UC4_Ext_33629_UC2_Interruption3 []
       f33629_UC5_Ext_33629_UC2_Interruption3,
       f33629_UC3_Ext_33629_UC2_Interruption4 []
       f33629_UC4_Ext_33629_UC2_Interruption4 []
       f33629_UC5_Ext_33629_UC2_Interruption4,
       f33629_UC3_Ext_33629_UC2_Interruption5 []
       f33629_UC4_Ext_33629_UC2_Interruption5 []
       f33629_UC5_Ext_33629_UC2_Interruption5,
       f33629_UC3_Ext_33629_UC2_Interruption6 []
       f33629_UC4_Ext_33629_UC2_Interruption6 []
       f33629_UC5_Ext_33629_UC2_Interruption6))
  [|union(a33629_MEM, {success})|]
  MEND(f33629_MEMORY)
) \ union(a33629_MEM, {success})

-- system
aget2 = Union({aget2_33629})
aconrol = union(aget2, {mem_update})
aS = Union({a33629, acontrol})

S = f33629

System = S

assert S :[ deterministic [F] ]
assert S :[ deadlockfree [F] ]
assert S :[ livelockfree [FD] ]

```

Bibliography

- [BBC⁺03] K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, R. M. Hierons, K. Kapoor, P. Krause, G. Luetzgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Working together: Formal methods and testing. Technical report, FORTEST, 2003.
- [BBF97] Mark R. Blackburn, Robert D. Busser, and Joseph S. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *Proceedings of the 12th Annual Conference on Computer Assurance*, pages 54–67, 1997.
- [BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
- [BJ78] Dines Bjorner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, London, UK, 1978.
- [BL02] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. *Journal of Software and Systems Modeling*, pages 10–42, 2002.
- [Cab06] Gustavo Cabral. Geração de especificação formal de sistemas a partir de documento de requisitos. Master's thesis, Universidade Federal de Pernambuco, 2006.
- [Coc97] Alistair Cockburn. Structuring use cases with goals. *Journal of Object-Oriented Programming*, 32(3):35–40, 1997.
- [Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Pub Co, first edition, 2000.
- [Cor03] IBM Rational Software Corporation. Rational unified process. <http://www-130.ibm.com/developerworks/rational/products/rup>, 2003.
- [CS08] Gustavo Cabral and Augusto Sampaio. Automated formal specification generation and refinement from requirement documents. *Journal of the Brazilian Computer Society*, 14:87–106, 2008.
- [CU10] CIn-UFPE. <http://tiny.cc/jufkc>, May 2010.

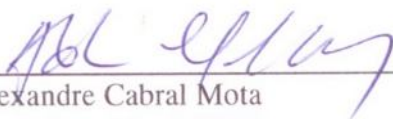
- [DFM09] Adriana Damasceno, Adalberto Farias, and Alexandre Mota. A mechanized strategy for safe abstraction of csp specifications. In *SBMF'09*, pages 118–133, 2009.
- [DM07] Jaroslav Drazan and Vladimir Mencl. Improved processing of textual use cases: Deriving behaviour specifications. In *Proceedings of Software Seminar (SOFSEM)*, 2007.
- [Dra06] Jaroslav Drazan. Natural language processing of textual use cases. Master's thesis, Charles University, February 2006.
- [DRP99] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [DvK92] Eugène Dürr and Jan van Katwijk. VDM++ - a formal specification language for object-oriented designs. *Proceedings of the seventh international conference on Technology of object-oriented languages and systems*, pages 63–77, 1992.
- [Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
- [Gar97] P. Gardiner. *Failures-Divergence Refinement: FDR2 User Manual*. Formal Systems Ltd, 1997.
- [Goe02] M. Goetze. UsageTester: UML-oriented usage testing. Technical report, Ilmenau Technical University, Department of Process Informatics, 2002.
- [Gol04] M. Goldsmith. CSP: The best concurrent-system description language in the world - probably! In *Proceedings of the Communicating Process Architectures (CPA'04)*, pages 227–232, 2004.
- [Gra00] Ian Graham. *Object-Oriented Methods: Principles and Practice*. Addison-Wesley Pub Co, third edition, 2000.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [Jon90] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
- [Lar01] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. , Prentice Hall PTR, second edition, 2001.
- [Lee01] Beum-Seuk Lee. Automatic transformation of natural language requirements into formal specifications. *Proceedings of Doctoral Workshop of the 5th IEEE International Symposium on Requirements Engineering*, 2001.

- [Lei06] Daniel Almeida Leitão. NLFoSpec: Uma ferramenta para geração de especificações formais a partir de casos de teste em linguagem natural. Master's thesis, Universidade Federal de pernambuco, 2006.
- [MBS02] Alexandre Mota, Paulo Borba, and Augusto Sampaio. Mechanical abstraction of CSPZ processes. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right, FME '02*, pages 163–183, London, UK, 2002. Springer-Verlag.
- [Men04] Vladimir Mencl. Deriving behavior specifications from textual use cases. In *Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE04)*, pages 331–341, 2004.
- [Mey92] Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computing Science, 1989.
- [MMC05] Stephen G. MacDonell, Kyongho Min, and Andy Connor. Autonomous requirements specification processing using natural language processing. In *Proceedings of the 14th International Conference on Adaptive Systems and Software Engineering (IASSE05)*, pages 266–270, 2005.
- [Mye04] Glenford J Myers. *The Art of Software Testing*. John Wiley & Sons Inc., New Jersey, second edition, 2004.
- [NCT⁺07] Sidney Nogueira, Emanuela Cartaxo, Dante Torres, Eduardo Aranha, and Rafael Marques. Model based test generation: An industrial experience. In *Proceedings of the First Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, João Pessoa-PB, Brazil, 2007.
- [NFTJ06] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.
- [NSM] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Test generation from state based use case models. To be published.
- [NSM08] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from csp models. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 258–273, Berlin, Heidelberg, 2008. Springer-Verlag.
- [OMG10] OMG. OMG unified modeling language (OMG UML) infrastructure version 2.3. Technical report, 2010.
- [RHB97] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

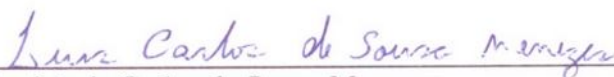
- [Ros95] A. W. Roscoe. Modeling and verifying key-exchange protocols using CSP and FDR. *In Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 98–107, 1995.
- [RPG03] Matthias Riebisch, Ilka Philippow, and Marco Götze. UML-based statistical test case generation. *In Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 25:394–411, 2003.
- [SC08] Stéphane S. Somé and Xu Cheng. An approach for supporting system-level test scenarios generation from textual use cases. *In Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*, pages 724–729, 2008.
- [Sel99] J. M. Selvidge. Statistical usage testing: Expanding the ability of testing. 1999.
- [SF96] Rolf Schwitter and Norbert E. Fuchs. Attempto - from specifications in controlled natural language towards executable specifications. *Proceedings of the 3rd Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1996.
- [SLH03] Rolf Schwitter, A. Ljungberg, and David Hood. ECOLE — a look-ahead editor for a controlled language. *In Proceedings of EAMT-CLAW03*, pages 141–150, 2003.
- [Som03a] Stéphane S. Somé. An approach for the synthesis of state transition graphs from use cases. *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'03)*, 1:456–462, 2003.
- [Som03b] I. Sommerville. *Engenharia de Software (6th edition)*. Addison Wesley, 2003.
- [Som04] Stéphane S. Somé. Supporting use cases based requirements simulation. *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'04)*, 1:381–386, 2004.
- [Som06] Stéphane S. Somé. Supporting use case based requirements engineering. *Information and Software Technology*, 48(1):43–58, 2006.
- [Som07] Stéphane S. Somé. *Use Case Editor (UCed) User Guide version 1.6.2*. School of Information Technology and Engineering (SITE), 2007.
- [Spi92] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992.
- [SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, Cambridge, second edition, 1994.
- [SSLM04] Evan Lenz Simon St. Laurent and Mary McRae. *Office 2003 XML: Integrating Office with the rest of the world*. O'Reilly Media, first edition, 2004.
- [TLB06] Dante Torres, Daniel Leitão, and Flávia Barros. Motorola SpecNL: A hybrid system to generate NL descriptions from test case specifications. *In Proceedings of the Sixth International Conference on Hybrid Intelligent Systems*, page 45, 2006.

- [vW65] A. van Wijngaarden. Orthogonal design and description of a formal language. Technical report, Mathematisch Centrum, Amsterdam, 1965.
- [WPT95] Gwendolyn H. Walton, J. H. Poore, and Carmen J. Trammell. Statistical testing of software based on a usage model. *Software - Practice and Experience*, pages 97–108, 1995.

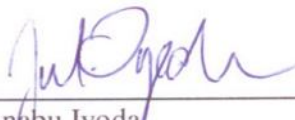
Dissertação de Mestrado apresentada por **Renata Bezerra e Silva de Araújo** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “**Extração Automática de Modelos CSP a Partir de Casos de Uso**”, orientada pelo **Prof. Juliano Manabu Iyoda** aprovada pela Banca Examinadora formada pelos professores:



Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

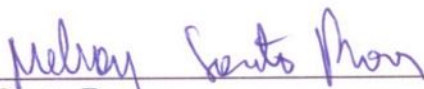


Prof. Luis Carlos de Sousa Menezes
Escola Politécnica de Pernambuco / UPE



Prof. Juliano Manabu Iyoda
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 18 de março de 2011.



Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.