Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Formal Specification Generation from Requirement Documents

Gustavo da Fonseca Limaverde Cabral

DISSERTAÇÃO DE MESTRADO

Recife
27 de setembro de 2006

Acknowledgements

This work has been developed in the context of a research cooperation between Motorola Inc. and CIn-UFPE. Thanks the entire group for all the support, criticisms and suggestions throughout the development of this research. Thanks Daniel Leitão, Dante Torres, and Prof. Flávia Barros for defining and implementing the Controlled Natural Language translator to CSP. Thanks Prof. Paulo Borba, Prof. Alexandre Mota for the suggestions and ideas. And a special thanks to Prof. Augusto Sampaio for the patience and bright advices.

Resumo

A escrita de requisitos, dentro do processo de desenvolvimento de sistemas, está sujeita a falhas, uma vez que os requisitos são escritos em Linguagem Natural, como Inglês, que pode conter definições ambíguas ou de difícil entendimento. Por outro lado, Linguagem Natural é a opção mais simples e flexível para se especificar um sistema, e é a linguagem de entendimento comum entre clientes e contratados. Desta forma, para minimizar a existência de erros nos documentos de requisitos, técnicas de validação com inspeção ou revisão de documentos são utilizadas. Entretanto, o custo de se realizar este tipo de validação é alto e sua eficácia é questionável; erros podem persistir. Além disso, requisitos escritos usando de linguagem natural são de difícil processamento, dificultando a geração de outros artefatos a partir do mesmo.

Esta dissertação define uma estratégia que utiliza *templates* de especificação de casos de use e uma Linguagem Natural Controlada (LNC) para descrever requisitos. Os *templates* de casos de uso asseguram a estruturação correta do documento de requisitos e a LNC garante a exatidão da gramática do texto que especifica o comportamento do sistema. Foram criados dois *templates* de casos de uso, cada um com uma visão diferente do sistema. A visão mais abstrata se chama visão do usuário e a visão mais detalhada se chama visão de componentes. A partir dessa estruturação dos requisitos torna-se, possível definir uma estratégia de geração automática de uma especificação formal da aplicação em questão.

A geração automática de especificação formal de sistemas reduz custo e necessidade de mão de obra especializada em projetos de desenvolvimento de software. Ou seja, uma vez que é possível realizar a geração automática do modelo formal de sistemas podemos fazer uso do mesmo na validação de propriedades do sistema. Além disso, artefatos como casos de teste e diagramas UML podem ser gerados a partir deste. Em particular, esta dissertação define uma estratégia para gerar modelos formais na álgebra de processo CSP a partir das duas visões de caso de uso, mantendo a consistência entre os artefatos. Também foi definida uma relação de refinamento entre os modelos gerados garantindo a consistência entre as visões.

Finalmente, todo o processo foi automatizado através de ferramentas. Estas foram validadas através de experimentos realizados no contexto de aplicação para celulares da Motorola, empresa parceira e financiadora do projeto de pesquisa com o CIn/UFPE.

Palavras-chave: Especificação de Caso de Uso, Linguagem Natural Controlada, Geração de Modelos Formais, Refinamento Formal de Sistemas e CSP

Abstract

Specifying requirements, in the software development process, is subject to incorrect definitions once these documents are usually written in Natural Language, such as English, which tends to induce ambiguities and unclear interpretations. Nevertheless, Natural Language is a simple and flexible alternative for specifying a system behavior; it is a notation that can be understood by both customer and service supplier. Thus, techniques such as document inspection and revision are employed to minimize the occurrence of errors and therefore validate documents. However, the document revision task cost is high and its effectiveness is questionable; errors may persist after document revision. Furthermore, the automatic processing of requirements written in Natural Language is a delicate task, making other artifact generation from requirements impracticable.

This dissertation defines a strategy that includes use case specification *templates* and a Controlled Natural Language (CNL) to describe system requirements. The use case *templates* assure a well-structured requirement document and the CNL guarantees the accuracy of grammar for the text, which defines the system behavior. Two types of use case *templates* were created; each one defines a different system view. The most abstract view is called user view and the more detailed view is called component view. This proposed requirement pattern enables the definition of a strategy to automatically generate a formal specification from the system under development.

The automatic generation of formal models reduces cost and the need for specialized staff in software development projects. In other words, once system formal models can be automatically generated, system properties can be validated using the generated models. Furthermore, artifacts such as test cases and UML diagrams can be generated from the model. Particularly, this dissertation defines a strategy to generate formal specifications in the CSP process algebra notation based on the two use cases views; preserving the consistence between the artifacts. In addition, a relation between these two models assuring the consistence between the views has been defined.

Finally, this entire strategy was mechanized through the development of tools. These tools were validated through experiments accomplished in the Motorola's mobile phone systems specification context. Motorola funded this research project in partnership with CIn/UFPE.

Keywords: Use Case Specification, Controlled Natural Language, Formal Specification Generation, Formal Models Refinement and CSP

Contents

1 Introduction **1**
1.1 Objects and Context 4
1.2 Proposed Approach 6
1.3 Dissertation Organization 7

2 Background and Related Work **8**
2.1 Background 8
2.1.1 Requirements Specification 9
2.1.2 Use Case Diagrams 9
2.1.3 Use Case Specification 10
2.1.4 Use Case Specification Abstraction Levels 10
2.2 Formal Methods 10
2.2.1 CSP 11
2.3 Specification Refinement in CSP 12
2.4 Related Work 13
2.4.1 Reasoning and Natural Language Requirements 13
2.4.2 First-Order Logic Specification Generation 15
2.4.3 Pro-case Specification Generation 16
2.4.4 Policies based CSP Specification Generation 18

3 Use Case Creation **21**
3.1 Use Case Creation Process 21
3.2 User View Use Case Template 21
3.2.1 Feature 22
3.2.2 Requirement 22
3.2.3 Description 23
3.2.4 Execution Flow 23
3.2.4.1 Step 23
3.2.4.2 Flow Types 23
3.2.4.3 Reference between Execution Flows 23
3.2.5 User View Use Case Example 24
3.3 Component View Use Case Template 24
3.3.1 Component View Use Case Example 26
3.4 Controlled Natural Language 28
3.4.1 Lexicon 28

3.4.1.1 Verb 28
3.4.1.2 Term 29
3.4.1.3 Modifier 30
3.4.2 Ontology 31
3.4.3 Case Frame 31
3.4.4 Case Frame Restriction 33
3.5 Some Considerations 34
3.6 Tool Support 35

4 CSP Specification Generation 38
4.1 CNL Translation to CSP 38
4.1.1 CSP Alphabet 38
4.1.2 CSP Events Generation 40
4.2 User View Model Generation 41
4.3 Component View Model Generation 43
4.4 Some Considerations 46
4.5 Tool Support 47

5 Model Refinement 48
5.1 Abstraction Levels 48
5.2 Refinement Mapping 49
5.3 Example Refinement 50
5.4 Component View as a Refinement of the User View 52
5.5 Some Considerations 53
5.6 Tool Support 53

6 Some Experiments in the Motorola Context 55
6.1 Methodology 56
6.2 Context 57
6.2.1 Mobile Application Specification 57
6.3 Use Case Specification process 59
6.4 Accomplished Experiments 61
6.4.1 Experiment 1 61
6.4.1.1 User View Use Cases Creation 62
6.4.1.2 Component View Use Cases Creation 62
6.4.1.3 Views Refinement 62
6.4.2 Experiment 2 63
6.4.2.1 User View Use Cases Creation 63
6.4.2.2 Component View Use Cases Creation 64
6.4.2.3 Views Refinement 64
6.4.3 Experiment 3 64
6.4.3.1 User View Use Cases Creation 65
6.5 Some Considerations 65
6.5.1 Use Case Creation 65

6.5.2 Use Model Generation 66
6.5.3 Models Refinement 66

7 Conclusion 67
7.1 Related Work 68
7.2 Future Work 69
7.2.1 CNL knowledge bases dynamically defined 69
7.2.2 Dynamic Use Case Specification 70
7.2.3 Automatic Test Execution from Generated Test Cases 70
7.2.4 Formal Model Animation to Validate System Implementation 71

List of Figures

1.1 Defect fix cost 1
1.2 Research project initiatives overview 5
1.3 Proposed strategy overall process 6

2.1 Requirements reasoning in order to find inconsistencies 14
2.2 Attempto Controlled English Specification Organization 15
2.3 Example of ACE sentence typesetting 16
2.4 Example of textual use case 16
2.5 Parse tree from one use case sentence 17
2.6 Generated Pro-case specification from use case in Figure 2.4 17
2.7 The entire [SHR$^+$05] process: from specification to code generation 18
2.8 CSP generated from the presented policies for the Pager agent 19

3.1 User view use case template 22
3.2 Example of a user view use case 25
3.3 Component view use case template 26
3.4 Example of a component view use case 27
3.5 Verb definition template 28
3.6 Verb definition examples 29
3.7 Term definition template 29
3.8 Term definition examples 29
3.9 Modifier definition template 30
3.10 Modifier definition examples 30
3.11 Ontology fragment 31
3.12 Case frame definition template 32
3.13 Case frame example 33
3.14 Case frame restriction definition template 33
3.15 Case frame and respective case frame restriction example 34
3.16 CNL grammar as HTML pages 35
3.17 Microsoft Word 2003 plug-in to validate CNL sentences 36
3.18 Java version of **Use Case Validator** 37

4.1 CSP generated for SetItem case frame (Figure 3.15) 39
4.2 Example of datatypes generated based on the Lexicon and the on Ontology 39
4.3 Example of a CNL sentence and its translation to a CSP event 40
4.4 CSP specification generated from the main flow of Figure 3.2 41

4.5 CSP specification generated from the main flow of Figure 3.2 use case 42
4.6 CSP specification generated from the exception flow of Figure 3.2 use case 43
4.7 Part of the component view model alphabet including the `setComp` channel as example 43
4.8 Part of the main component process (Composition between components) 44
4.9 `USER_P` process exchanging messages with other components 45
4.10 `MESSAGE_APP` process 46

5.1 Events decomposition at different abstraction levels 48
5.2 Mapping of sequence of events in different views 49
5.3 Mapping function 50
5.4 Abstract View composition with Mapping Process and events hiding 50
5.5 CSP process examples 51
5.6 Mapping function application to the example from Figure 5.5 51
5.7 `View_1` composition with the mapping process to accomplish refinement 51
5.8 Illustrative usage example of the mapping function 52
5.9 Mapping between abstract and concrete views 52
5.10 Mapping process specification based on the map 53
5.11 Assertions verified by FDR tool 54
5.12 Properties verification with FDR 54

6.1 Development process and affected phases 55
6.2 First and second experiments 56
6.3 Motorola applications screenshots 58
6.4 Solution steps 59
6.5 Execution flow definition after requirements analysis 59
6.6 Execution Flow graph made from requirements analysis 60
6.7 Tools involved in the process 60

CHAPTER 1

Introduction

The use of formal models, which are an abstract way to specify computer systems, is an industrial reality. Initially, one can realize that the benefits regarding the use of abstract notation, before starting the system implementation, are only related to a better understanding of the problem. What has become increasingly evident is that the use of abstract formal representation combined with models refinement can even promote the decrement of implementation time. One of the possible applications would be the automatic generation of source code from formal models [KWB03]. The testing phase could be also positively impacted by the use of models for test cases generation [DJK+99].

As it was shown, requirements come up to be the starting point for software development and therefore needs to be specially treated in order to produce high quality requirements documents. These documents are the input to the formal specification activity and no uncertainties should remain concerning its contents. There is a variety of requirement specification methodologies, such as TROPOS project [BGG+04]; a software development methodology founded on concepts used to model early requirements. Investing on good requirement methodologies is an effective way to reduce cost. It has been shown [BBL76] that, the sooner a problem is found during the software development cycle the least expensive is to fix it. Figure 1.1 presents this idea.

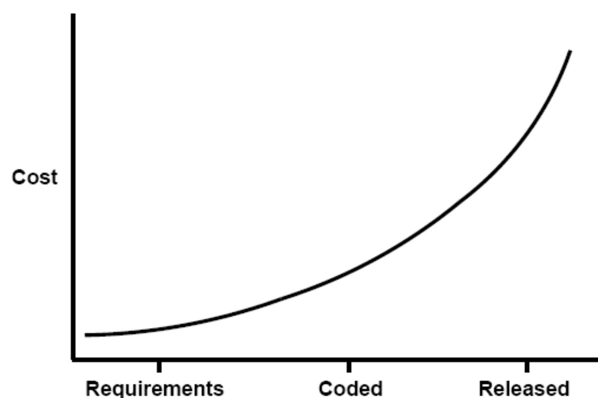

Figure 1.1 Defect fix cost

Once a project has precise requirement documents it is possible to create a formal specification in order to validate system properties. This task is commonly accomplished by software engineers experienced with formal languages and specifications. He or she would read and understand requirement documents, or any other equivalent artifact, and use his or her own creativity to specify a formal model. This is a manual process and the final specification may

not cover all specified requirements or contain inconsistencies regarding requirements. It may be misunderstood by the engineer or even be written in an ambiguous or unclear way. This manual process introduces the possibility of a problematic formal model creation.

This problem is minimized through the adoption of techniques and tools [SFGP05] that propose a better arrangement of requirements, making them unambiguous, uniform and categorized. These approaches smoother the understanding of requirements but does not solve the problem. The ideal picture is requirement documents that can be automatically processed by a program that would generate the formal model. The formal model creation is a creative task and therefore cannot be easily achieved trough a program routine. Additional information is necessary to automatically create a formal specification. The most likely to succeed option would be simplifying requirements, making them simple, direct, unambiguous, and uniform. This approach would not only impact in requirement's structure but in the way they should be written. Therefore, Natural Language Processing techniques should be applied in order to validate the requirement semantics.

Natural languages processing can be impracticable if textual requirements understanding depends on previous knowledge about the application domain. This information gap brings the possibility of non-understanding of requirements [SS97] by a program, or even by a person. This is a great challenge in the linguistics field of study.

An alternative is the use of simpler languages to describe requirements, not a natural language. These languages are called Controlled Natural Languages (CNL) [SLH03]. They contain a smaller and restricted grammar that prevents the writer from introducing ambiguous and non-uniform sentences. Depending on the number of restrictions of the CNL it may be impossible to use it without tool assistance to validate sentences. There are editors, such as MS Word [LLM04], that execute simple grammatical and vocabulary validation. More complex editors execute logical analysis of texts to find semantic problems [FSS90, FST99, GZ05].

There are several ways to formally specify systems. Some specification languages are more flexible than others. Formal specification languages use mathematical approaches to validate specification properties. Examples of formal languages are Z [Spi92], CSP [RHB97], and Circus [Oli05]. In spite of the existence of formal proof benefits, the incidence of formal methods use in real project is small. This fact is related to the demand for highly specialized personnel. Semi-formal languages, such as UML [Sel04], use has increased. These languages simplicity and flexibility enable its wide use and understanding. In addition to that, tools such as Rational Rose [Qua98] enable fast design. UML 2.0, which includes UML-RT [HYfCl05] and OCL [WK99], enables precise systems specification. The development of tools for formal languages support is a good approach to make its use more feasible. The benefits of formal languages use are beyond formal properties verifications (livelock, deadlock, determinism). It can be used to generate system test cases [dCN06, Car06], UML artifacts [Mun06], or even implementation in languages like Java [FC06, WAF02].

Nowadays, the use of CNL for requirements specification has not been wildly explored. One popular language processing initiative is the ACE project [FSS90]. It focuses on document validation and information retrieving through predicate logic analysis. The effort to process text documents to assist software implementation has been small. The ideal scenario would be using natural languages for systems implementation purposes. The information gap between these

two levels of specification makes it necessary to adopt an interactive and incremental process for specification enrichment [SMSB05].

In the past, the use of formal methods was restricted to critical systems [Kni02], where software failure could provoke life loss, for instance. This kind of systems needs to be checked so unexpected behaviors can be avoided; possible errors must be predicted and treated. The effort to formally specify system compensates future issues. The advance of formal languages brought new prospects for its use in everyday projects. The abstraction level increased and it is now possible to perform model transformations and refinements. Tools have been developed for specification validation and code generation. The cost of formally specifying systems got lower and its use may even decrease the cost of following software development phases.

Natural languages are very flexible and enable high level definitions creation; however, they are complex to process. Their use is actually the new bound in system specification research domain. More robust and complex compilers shall be created in order to process natural languages and generate abstract formal models or even systems implementation.

Requirements specification is a complex and expensive activity. Any mistake during its definition may propitiate problems in subsequent development phases. Because Motorola operates on the telecommunication business the presence of volatile and unstable requirement is a very common scenario. Keeping requirements coherent with implementation and test artifacts has proven to be a difficult task. The CIn/BTC research project has been conceived to address problems, researching ways to improve and automate Motorola's activities through scientific studies. It turns out that these solutions can be applied to other branches of software development, not just mobile system, reducing project costs and development time.

The automatic generation of project artifacts from requirements seems to be an interesting route to explore. This approach would prevent the existence of out-of-date requirements because they would be the starting point to generate other artifacts. It would not be necessary to maintain other artifacts like UML diagrams or test cases, once they can be automatically generated. There would be a synchronism between requirements and generated artifacts.

A direct benefit would be the increase of productivity, once the artifacts generation is automatic. A second benefit is related to quality of the generated artifacts. Once the initial requirements, input to the proposed process, are validated, all generated artifacts would be consistent with the specified requirements. In this context, it would be probably necessary to increase the requirements specification phase time in order to define more detailed and complete requirements. This would bring the system specification phase to a greater level of importance during the software development process. Whenever requirement inspectors [AS02] would attend to document validation meetings, they would not worry about the syntactical content but to their meaning, the proposed system behavior.

Mobile applications may contain complex features, which include concurrent behavior and message exchanging. To specify requirements for system of this nature can be a hard task, especially if it is necessary to validate the captured behavior. It may be toilsome to analyze concurrent elements, running in parallel, and validate their behavior. In this case, a formal specification, through a language such as the process algebra CSP that contains parallelism operators, can be very useful to validate the system requirements, particularly using the FDR model checker [Ros95, fdr97].

The formal model specification activity is complex and it requires creativity. For one simple system it may be created several equivalent specifications. Because the proposed approaches accomplish this talk automatically based on requirements, all generated models will follow a standard format which is likely to simplify the specification understanding. The intention is that the generated model should not be directly manipulated by the developer.

These are some of the expected results of this dissertation; it involves requirements engineering, natural language processing, and formal methods concepts, techniques and tools. It is an example of multidisciplinary approach to solve real software engineering problems.

1.1 Objects and Context

We propose a strategy that automatically translates Use Cases, written in Natural Language (English), into specifications in the CSP process algebra [RHB97]. For obvious reasons, it is not possible to allow a full natural language as a source. We define an English CNL with a fixed grammar, in order to allow an automatic and mechanized translation into CSP.

As the context of this work is a research cooperation between CIn/UFPE and Motorola (known as CIn/BTC), related to testing mobile applications, the proposed CNL reflects this domain. The formal specification generated in CSP is used in the project as an internal model to the automatic generation of test cases, both in Java (for automated test cases) and in CNL itself (for manual tests). Figure 1.2 contains the possible outputs.

Unlike the cited approaches, which focus on the translation at a single level, we consider use case views possibly reflecting different levels of abstraction of the application specification. This is illustrated in this dissertation through a user and a component view. We also explore a refinement relation between these views; the use of CSP is particularly relevant in this context: its semantic models and refinement notions allow to precisely capture a formal relation between the use and the component views. The approach is entirely supported by tools. A *plug-in* to the Microsoft Word 2003 [LLM04] has been implemented 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. Finally, the FDR [Ros95], a CSP model checker, is used to check the refinement between the user and the component views.

This work has been developed in the context of the CIn/BTC research project, which is sponsored by Motorola Inc. [Inc06] in cooperation with CIn/UFPE [CIn06]. This cooperation started in 2003 and, initially, aimed at the creation of human resource specialist in the Software Testing area. This project has grown significantly and today it is formed of about 200 people, including software quality and test engineers, managers, and researchers.

The CIn/BTC is divided in three areas: formal education and hands-on training, operation, and research. This dissertation is one of the research team results. The research activities are motivated by the operation team existing issues and detected improvements possibilities.

The main goal of this project is to analyze Motorola's artifacts and process, search for possible improvements and propose a viable solution that shall automate some of the Motorola's activities, such as requirement specification, test design, and design artifacts creation. This project also proposes ways to evaluate test cases coverage and to improve test case selection and execution plan guided by estimative.

Figure 1.2 shows the research project initiatives that aim to improve the software development process through automation. This dissertation defines the emphasized activities: software requirements description in Controlled Natural Language (CNL) and Intermediate Representation generation. CNL is a constrained version of English, which enables precise text processing. This standardization makes it possible to translate requirements into a formal specification.

The activity *Software Requirements in CNL* defines use case specification templates and the CNL. Use case specifications are used to define the system behavior. They define sequences of events, or steps, that the user may accomplish when using the system. The CNL is used to write these steps, which may is basically composed by the user action and the system response.

The *Intermediate Representation* is the formal representation of requirements in the CSP [RHB97] notation. The main idea here is to translate requirements to a well defined formal language in order to accomplish its manipulation, validation, analysis and refinements. This intermediary representation is not directly accessed by the final user, when using the tools that implement the strategy. This allows the use of formal methods techniques excluding the necessity of formal methods specialized knowledge.

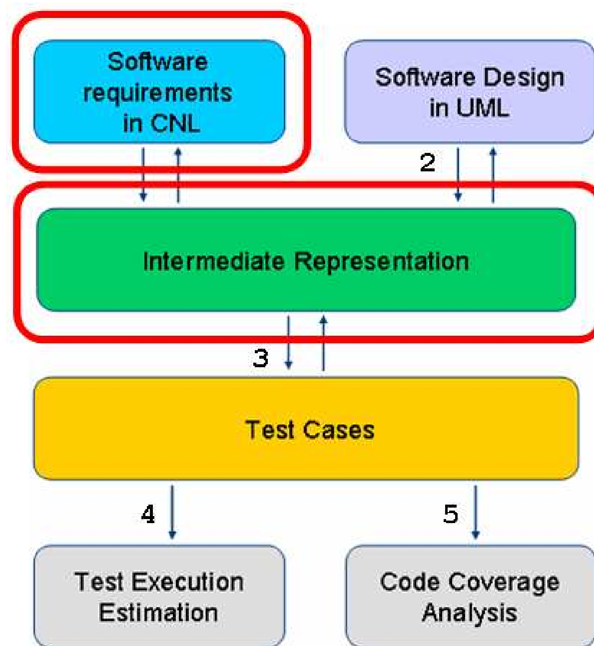

Figure 1.2 Research project initiatives overview

The others activitie presented in Figure 1.2 are:

- Software Design in UML (2): this initiative uses the generated intermediate representation, in CSP, to generate UML Realtime (UML-RT) [HYfCl05] diagrams such as Class, State and Structure diagrams [Mun06]. UML-RT diagrams are useful to the development team, describing implementation details about the system. The reverse flow, generation of the Intermediary Representation from legacy UML-RT diagrams can be more useful for test case generation purposes. UML-RT diagrams contain information about user interaction with the system that may be used for testing purposes.

- Test Cases (3): test generation from model; also know as Model Based Testing (MBT) is an important aspect of the proposed approach. The system is implemented based on its requirements, so it is necessary to define testes that verify the implementation. The Intermediary Representation is an equivalent representation of requirements and test cases generated from it will cover the respective requirements. A strategy to accomplish test case generation is define in [dCN06].
- Test Execution Estimation (4): once a set of test cases is automatically generated it is necessary to choose what test cases will be executed based on the execution team information, its size, experience, etc. The goal is to maximize requirements and code coverage based on a fixed number of tester and time (deadline). To accomplish this task estimation techniques are used to analysis test cases and all variables involved in the test case planning and execution process.
- Code Coverage (5): one important metric to verify test cases quality is code coverage. It is not sufficient to know that a certain test case covers a set of requirements, it is necessary to know how well it covers it. This information can be retrieved through code instrumentation and test case execution. Using tools, this process can be fulfilled and reports will show how well test cased cover the code.

1.2 Proposed Approach

In our approach, each use case is specified using a template. It is structured to hold information concerning traceability with requirements, a brief description and the way actor interacts with the system. There are two use case templates: the **user view** and the **component view**.

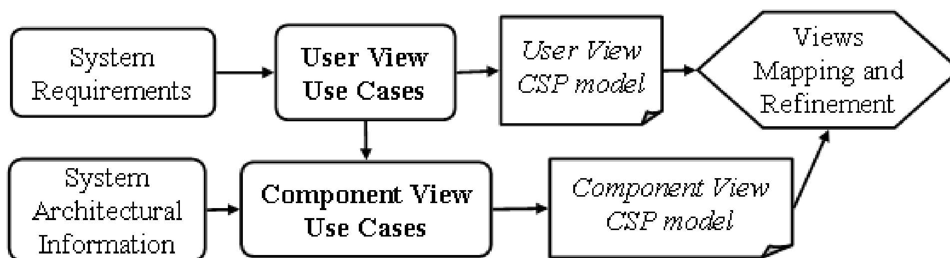

Figure 1.3 Proposed strategy overall process

As shown in Figure 1.3, after `System Requirements` are analyzed, the **user view use cases** are created. Later, **component view use cases** are created based on the **user view use cases** and the adopted `System Architectural Information`.

The language used to write these use cases is a CNL. Using CNL it is possible to write imperative and affirmative sentences. An imperative sentence describes actor actions and an affirmative sentence describes system characteristics, such as a GUI description. CNL grammatical rules are defined through knowledge bases that map verbs to CSP channels and verb complements to values of CSP datatypes.

Each use case sentence is translated into a CSP event, and a sequence of sentences produces a sequence of CSP events, combined with the CSP *prefix* operator, which gives rise to a CSP process. Each use case defines part of system formal specification. The presence of alternative or exception execution flows in use cases is captured by the CSP *choice* operator, thus allowing processes to be combined. Hence, 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*.

Finally, the relation between user and component use cases is established by a mapping from the more abstract to the more concrete model. This event mapping relation is used to prove that the component view model is a refinement of the user view model.

1.3 Dissertation Organization

The current chapter is the dissertation introduction and presents its context, including: problems, main goals, state of the art, and an overview of the solution.

Chapter 2 analyses related researches in order to justify the need to define a new formal specification generation strategy. It also gives an overview of use case specification techniques and the CSP formal notation, which are fundamental concepts for the proposed solution.

Chapter 3 contains the use case template definitions and explanations about its usage, which includes the use of the CNL. This chapter also contains a brief presentation of the tools that were implemented to support this strategy.

Chapter 4 defines the CSP use model generation approach based on the presented use case templates and the CNL. This strategy is mechanized by a tool that is detailed in Section 4.5.

Chapter 5 explores a refinement relation between the generated CSP models, which describes the user and the component views, and how it is mechanically checked using FDR.

Chapter 6 discusses the results obtained after experiments using the proposed approach. These experiments were executed in a real process environment, following a software development process, which includes artifacts validation through inspections.

Finally, Chapter 7 summarizes our contributions, contrast the proposed solution with related work, and suggest topics for further research.

CHAPTER 2

Background and Related Work

The use of formal models to specify systems is an industrial reality. The use of such abstract and concise notations improves the problem understanding process. On the other hand, requirement documents, which specifies systems in an informal way, are created in the beginning of software development [Kru00]. Because system requirements are written in natural language, the chance of wrong or unclear definitions to occur is high. The existing validation techniques includes documents inspection [AS02] and logical analysis of system definition [GZ05, FST99].

Section 2.1 show the system specification and formal method background to assist the strategy understanding. Section 2.4 presents some existing work related to formal models generation.

2.1 Background

Before any product is manufactured, generally, it needs to be specified. In our case the product is software. The software specification works as a contract between client and contractor [BE98]. It is a document that describes what the client is requesting, and therefor it defines what should be delivered by the contractor. It should describe the system's behavior under various conditions as it is used by possible customers [Pre00].

This specifying system is complex, it is necessary to know if the specification is correct [Som01]. Both client and contractor need to be able to read the specification document and understand its content in order to seal a treat. However, client and contractor are usually different people with different backgrounds and do not necessarily have the same understanding of system specification definitions. Thus, it is necessary to use a suitable language and document templates to minimize misunderstandings when analyzing such specification documents.

There is a variety of specification languages that can be used to describe systems. English is one of them; probably the most simple and error prone. However it is not possible to describe a system using a complex, hard to understand, abstract, language; we should not expect that the client would understand such language, such as formal notations. Whence, it is necessary to adopt a more concise, well-defined, and less abstract notation.

The use of such notation enables the creation of specification documents that can be validated by both client and contractor. Notice that this initial phase is error prone; both client and contractor can neglect the document validation and let wrong definitions pass by. The execution of documents inspections [AS02] is widely used to improve artifacts quality. In addition to that, such documents need to be understood by technical staff in the subsequent phase. These initial errors propitiate the creation of wrong project artifacts such as diagrams and source code.

This dissertation focuses exactly on this point. It define means to minimize the presence of errors in specification documents enabling the translation of initial specifications to a formal specification, which is a more unambiguous and precise notation.

Hence, this chapter presents the most common notations used to describe system specifications and indorse the use of a CNL, along with use case templates, to specify system's behavior. Moreover the CSP notation is introduced and recommended as the most appropriate formal target notation. Base on these definitions, it should be possible to define a strategy to generate the system formal model.

Further, it is introduced the refinement concept, which is necessary to relate models proving that one model, more concrete, holds the same behavior of another model, more abstract. This notion is essential to demonstrate the equivalence between specification; it ensures the possibility to accomplish models improvements without the anxiety of being producing a disconfirm model. The improved model behavior should not differ from the previous model.

The existence of a variety of specification notations enables the creation of model in different abstraction levels and following different paradigms. The choice of which notation to use is basically related to abstraction power.

2.1.1 Requirements Specification

A system can be specified through a set of requirements. During meeting with customer, each requirement [Som01] depict objects and goals stated for a system. It is defined in an informal way, using natural language. I must be clear, correct, unambiguous, specific, and verifiable. Requirements should also be uniquely identified through a sequence number or a meaningful tag of some kind, for instance.

Usually, requirements are classified as functional and non-functional. Functional requirements describe the practical system behavior, specifying what tasks it should accomplish; what operations are to be automated. Functional requirements define the internal workings of the software, that is, the calculations, technical details, data manipulation and processing, and other specific functionalities. They are supported by non-functional requirements, which impose constraints on the design or implementation. Non-functional requirements specify criteria that can be used to judge the operation of a system, rather than specific behaviors. Typical non-functional requirements are reliability, scalability, and cost.

Both types of requirements are freely specified using verbal expressions, English for instance. Hence, these definitions can contain ambiguous and conflicting ideas. This technique is very flexible and powerful, nevertheless error tending. Thereby, requirement specification is not the most viable input to be processed to generate the formal specification.

2.1.2 Use Case Diagrams

A use case diagram [BRJ99] is a visual illustration of the different scenarios of interaction between an actor and a use case. The usefulness of use case diagrams is more as a tool to determine use cases relations (*include*, *extend*, and *generalization*) and possible actor interactions with the system [DP02]. The next step after defining use case diagrams is to document the business functionality as use case specifications.

2.1.3 Use Case Specification

Use cases specification captures the system behavior under various situations as the system responds to a request from one of its stakeholders [Coc00]. It's a story about how a user interacts with the system under many circumstances. The use case specification format may vary; it can be a narrative test, an outline of steps, or a template-based description.

Before specifying use cases, it is necessary to define the actors that will be participating of the use cases. These actors are different people (or other systems) that are interacting with the system in order to accomplish some goal.

Because use cases specifications are used as an input to other project phases such as design, development, and testing, we need to ensure that the visual depiction of the requirements is translated into clear and well-defined use case specifications [DP06]. Use case specifications are used as input for design and development and for writing test cases (unit, system, and regression tests, as the case may be).

2.1.4 Use Case Specification Abstraction Levels

Use cases specifications give a time perspective to the specification. Therefor, it is important to firm level of details to specify use case steps; the user viewpoint for example.

Use case diagrams are one of the most popular specification techniques. Its initial importance were brought with UML [BRJ99] popularization. Use case flexibility makes it possible to specify virtually any type of activity using actors and functions. Unlike RUP [Kru00], UML is a language, not a process that defines how use cases should be specified. Consequently, use cases diagrams interpretation gets confuse when a large number of use cases are specified and related to each other [Fow98]. It is necessary to define a reference abstraction level to employ use cases correctly.

To enable the generation of formal specification from use cases it is necessary to determine fixed abstraction levels. Therefore this topic is discussed in Chapter 3 to establish the way the proposed use cases templates should be used.

2.2 Formal Methods

All the practical experiences and research involving formal methods show that the use of formal models is not a viable answer to eliminate all software failures, but neither are they beyond the budget constraints of software developers. When well used, formal methods are a practical way to demonstrate the absence of undesired behavior, an essential critical systems property.

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.

The choice of which notation to use is application dependent. Indeed, a number of complementary methods may often be required for a single application. When specifying state based aspects of systems it is best to use a notation such as Z [Spi92] or VDM [Jon90]. Moreover, if the specified system contains distributed concurrent characteristics, it is most likely to use a

process algebra, which provides formalisms for modeling parallelism behavior. The following are examples of notations for such objective: CCS (Communication and Concurrency), LOTOS (Language Of Temporal Ordering Specification) or CSP.

At Motorola, it is desired to specify mobile phone applications that contain concurrence definitions. In this case the use of process algebra notations is considered. It promotes understanding of issues related to concurrent components execution.

Therefor, CSP, the Hoare's language of Communicating Sequential Processes [RHB97] [BHR84], was selected as the most suitable notation to specify mobile phone applications. The following section gives an overview about the CSP language.

2.2.1 CSP

The CSP process algebra [RHB97, BHR84] was chosen as the target formalism of our strategy. CSP allows the description of systems in terms of processes that operate independently, and interact with each other through message-passing communication. The relationship between processes is described using process algebraic operators. These few primitive constructors allow the construction of complex specifications.

The main characteristic of components is their behavior and interaction (communication) with other components. Each component runs independently and communicates with others when necessary. CSP process algebra was the formalism adopted in our approach because it can express concurrence and parallelism between the components in an effective way.

CSP use events to define the behavior patterns of the real world objects (processes). First, we must define which events are relevant in the process behavior; each event must be identified with a unique name. One event may occur many times in the process behavior. Event occurrences are considered instantaneous or atomic. The set of process events is called alphabet.

The behavior of a CSP process is described in terms of events, which are atomic and instantaneous operations, like *open* or *close*, that may transmit information, like in the output communication *open!door*. There are two primitive processes STOP and SKIP: STOP communicates nothing and it stands for a canonical deadlock; SKIP represents successful termination.

Some of the CSP operators are the prefix ($a \rightarrow P$), deterministic choice ($P \Box Q$), nondeterministic choice ($P \sqcap Q$), interleaving ($P ||| Q$), the parallel composition ($P \,|[s]|\, Q$, where s is the set of events in which P and Q synchronize), and hiding ($P \setminus s$, where s is the set of events to be hidden). The prefix operator combines an event and a process to produce a new process. The deterministic (or external) choice operator allows the future evolution of a process to be defined as a choice between two component processes. The nondeterministic (or internal) choice operator allows the future evolution of a process to be defined as a choice between two component processes, but does not give the environment any control over which of the component processes will be selected. The interleaving operator represents completely independent concurrent activity. The parallel composition (interface parallel) operator represents concurrent activity that requires synchronization between the component processes. The hiding operator provides a way to abstract processes, by making some events unobservable.

Processes

Processes are the basic unit to capture behavior. Each process is defined by equations and, in general, a set of process is used to get modularity. Processes communicate with each other

through synchronization in their events. Communications can or not carry data.

The primitive processes *SKIP* and *STOP* are terminal process (no communications happen). *SKIP* represents a successful termination through the only communication of the special event, while *STOP* represents a deadlock situation.

Prefix

The most basic construct to model behavior is the prefix operator ($\rightarrow$). Let x 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. This operator can be used to model recursive processes. The behavior of the processes $P = x \rightarrow P$ is the indefinite repetition of the event x.

Sequential Composition

The sequential composition operator (;) allows a process to be initialized after the successful termination of other one. The process $P = A;B$ initially behaves like A, and then like B.

Internal and External Choices

To represent alternative behavior we can use the operator $\Box$ (external choice, deterministic, this operator is mapped to CSPm) or the operator $\sqcap$ (internal choice, non deterministic, this operator is mapped to $\sqcap$ in CSPm). The first one allows the environment control the choice between the options of behavior. While with the later, the environment has no influence about the selection between the behaviors. Thus, consider that the environment make available the event x, if x is the first event of the process P, then $P \Box Q$ behaves like P, else if x is a first event of Q, the $P \Box Q$ behaves like Q. If x is a first event of both P and Q, the choice between them is non-deterministically defined. The process $P \sqcap Q$ behaviors like P or Q, arbitrarily.

Process Composition

When two processes are put in concurrent execution, in general, the desire is that one interacts with other. The interactions can be viewed as events that require the simultaneous participation of both processes. Let P and Q process with the same alphabet, $P \parallel Q$ represents a processes in which P and Q must synchronized in all events. So an event x only occur when both processes are ready to accept it. The process $P\ [|X|]\ Q$ synchronize 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 $P\ |||\ Q$ (P interleaving Q) allows that P and Q execute concurrently without synchronization between them. Each event, offered to interleave of two processes, occur only in one of them. If both are ready to accept that event, the choice between the processes is non-deterministic.

Process 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 massage value sent by the channel. The process $P = c1!v1 \rightarrow c2?v2 \rightarrow Q$ initially sent the value $v1$ through the channel $c1$, receive the value $v2$ through the channel $v2$ and then behaves like Q.

2.3 Specification Refinement in CSP

The CSP notation is defined by three different formal semantics: denotational semantics, algebraic semantics, and operational semantics. CSP uses semantic models to establish relation

between specifications. The semantic models are *Traces*, *Failures* and *Failures-Divergence*. The refinement concept is related to the idea of implementation; one specification refines other if it satisfies its specification requirements.

The *Traces* model is defined as the set of possible sequence of events one specification can execute. One trace from a specification defines a possible finite sequence of events that a process executed until them. However, it does not define the set of traces a specification cannot execute. Therefore, the process $P \,\Box\, Q$ and $P \sqcap Q$ are equivalent in the *Traces* model.

However, the *Failures* model differentiates these two processes. It contains all possible sequence of events a specification can execute associated to a set of events that cannot be executed afterwards. Therefor, it is possible to identify non determinisms.

The *Failures-Divergences* is the most powerful model. It includes the *Traces* and the *Failures* models. It is specially used to evaluate the presence of *livelock* and prove the equivalence between specifications. This dissertation presents a refinement strategy based on *Failures-Divergences* models.

After a wide analysis of different specification techniques, use case specification was chosen as the most adhesive specification style. It captures the user interaction with the system in details. The definition of use cases templates enables the creation of precise CSP formal models.

The use of a CNL guaranties the use case's textual correctness enabling the generation of the related CSP formal model. The use case template defines the model structure and its text denotes its events definition.

Moreover, the CSP refinement paradigm is concise enough to establish a relation between use cases that describes the system behavior in different levels of abstraction. The concepts presented here are used in the strategy definition in the following chapters.

2.4 Related Work

Formal specification generation techniques aim to validate requirements and to ensure consistence between artifacts. The following subsections present the analysis of some existing formal model generation works. This analysis aims to exemplify the current state of the field and motivate the approach proposed in this work.

2.4.1 Reasoning and Natural Language Requirements

The use of first-order logic [Apt96] to identify and remove inconsistencies in requirements is an effective strategy [GZ05]. Logics is a well-founded formalism suitable for requirements incremental validation. However, direct use of logics for expressing requirements and discussing them with clients poses serious usability problems, since clients may not be fluent with formal notation. Therefore, the use of natural language parsing techniques with formal reasoning tries to overcome this problem. The approach presented by [GZ05] defines means to discover requirements inconsistencies using both theorem proving and model-checking techniques.

During the requirements definition, requirements are incrementally described by a number of different users (stakeholders). They are expressed as natural language sentences, and each

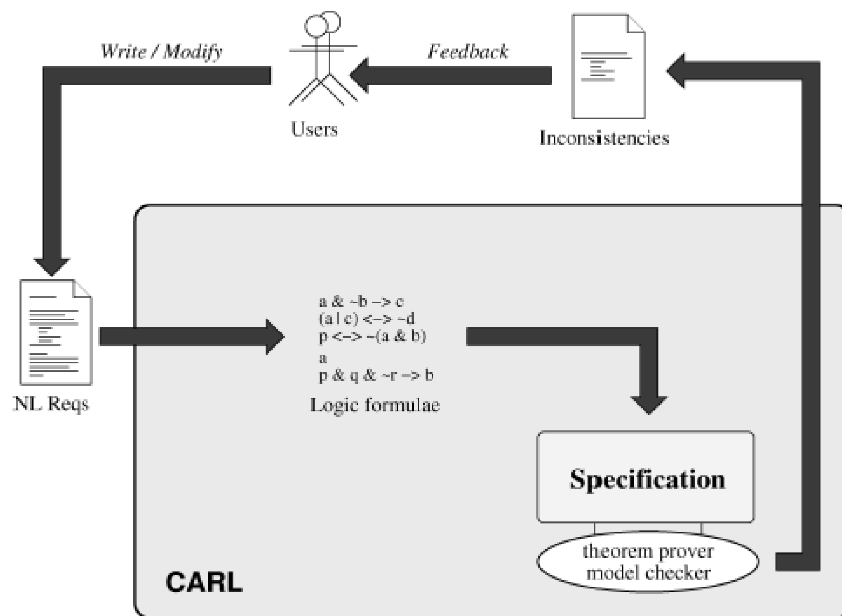

Figure 2.1 Requirements reasoning in order to find inconsistencies

stakeholder may name requirements that are more important from his particular viewpoint. Figure 2.1 presents this strategy. Here, requirements, written in natural language, are processed by a tool called CARL, and subjected to a succession of transformations.

The generated logic formulas are then automatically analyzed as requirements or as system constraints. These analyses output a set of inconsistencies in the specification; the existence of such inconsistencies is checked by using a theorem prover. If any inconsistency is found, a detailed report and the various alternative interpretations are presented to the stakeholders, providing an opportunity to edit the document.

There are various works related to the translation of natural language into logics of different kinds, including [Ali94, FGR^{+}94, FS95, RP92, Web83]. These studies have targeted special-purpose logics, which have resulted in very restrictive Controlled Natural Languages (CNL). In contrast, [GZ05] accepts a less restricted language enabling a more flexible environment.

However, the inconsistencies rate increases along with the language flexibility level. This fact forces the execution of sequential inspections once CARL analyzes the requirements and yields validation reports. In opposition to this fact, the adoption of a CNL enables earlier document validation based on a predefined grammar, which can be used to train designers and reduce errors.

The propositional logic notation is not considered suitable to model design aspects of the system. The generated model should contain information about possible user interaction scenarios with the system. This type of specification employees the idea of sequences of events (user actions and system responses). The generated model also needs to contain information about system components and detail their interaction. A more structured strategy which would hold the temporal concept of the system behavior is necessary. The language used to define system behavior needs to identify domain specific terms related to the system, such as GUI

elements and system component names. The use of propositional logic is more adequate to express high-level requirements.

2.4.2 First-Order Logic Specification Generation

Attempto Controlled English (ACE) [FSS90, FST99] is similar to Section 2.4.1 approach. It defines means to translate requirements into first-order predicate logic specification.

This time requirements are written in a CNL to prevent the presence of inconsistencies. However, the generated logical specifications can grow considerably and its understanding, without tools support, is impracticable. This way, a tool is provided to assist the organization and view of the generated logical model. ACE describes an environment, called OACES (Organized Attempto Controlled English Specifications), for writing large specifications in ACE with means to organize and manage them.

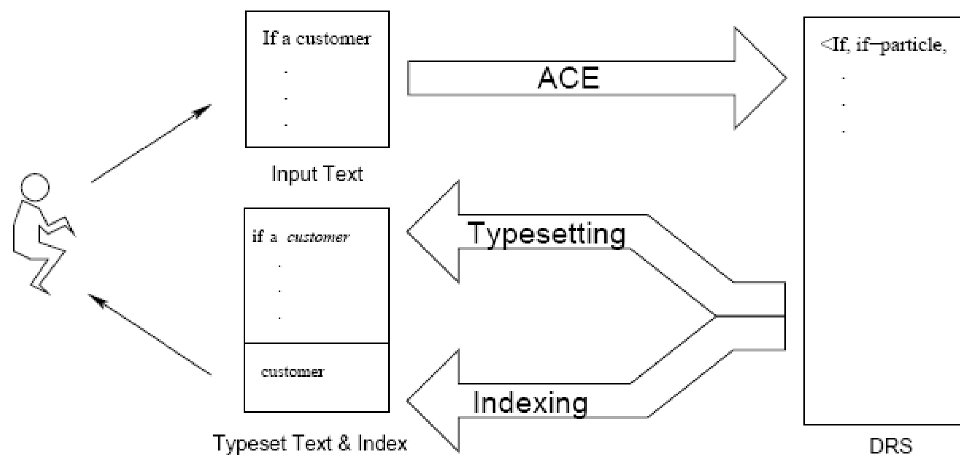

Figure 2.2 Attempto Controlled English Specification Organization

The tool offers alternative ways for typesetting ACE specifications. There are different colors, sizes and fonts of letters to differentiate words with distinct roles and may thus help humans read and understand longer pieces of text. ACE text components are divided into three categories: coordination-subordination constructs, nouns, and verbs; this enables typesetting. Each type is displayed in a different typesetting format. The coordination-subordination constructs are "if", "then", "and", "or", "who", "which", "that" and "not"; these words are shown in boldface. Nouns are composed by substantives, which are either sentence's subjects or objects; these words are displayed in italics. Verbs are shown underlined. Figure 2.3 shows example of ACE sentences and the respective typesetting. This approach aims to improve the requirement edition environment to enable the use of ACE language so the designer will not experience difficulties to use a more restricted language.

This strategy is valid in the logic preposition generation scope. It assures that requirements are written according to the ACE language enabling the formal model generation. Nevertheless, the use of the logic paradigm is not the most indicated in the Motorola's context. The logical notation is mostly suitable to verify inconsistencies between requirements. The use of logical

If a *customer* enters a valid *card*
and a correct *code*
then the *automated teller* accepts the *card*
If a *card* is not readable
then the *automated teller* rejects the *card*

Figure 2.3 Example of ACE sentence typesetting

models is quite popular in order to validate requirements. Similar approaches to ACE are [GN00, GN02].

2.4.3 Pro-case Specification Generation

In [Men04] a strategy is defined to convert textual use case to Pro-case (Protocol use case). Textual use cases describe in natural language how actors cooperate with the system by communicating and performing actions to achieve a particular goal. Pro-case is a notation for specifying component behavior proposed in [PM03] which is based on protocols [PV02]. The Pro-case notation captures the overall behavior of the system and allows details to be hidden. Figure 2.4 is an example of textual use case that contains main, alternative (variations) and exception (extension) execution flows.

Use Case: M1 Seller submits an offer
Scope: Marketplace
SuD: Marketplace Information System
Primary Actor: Seller
Supporting Actor: Trade Commission
Main success scenario specification:
1. Seller submits item description.
2. System validates the description.
3. Seller adjusts/enters price and enters contact and billing information.
4. System validates the seller's contact information.
5. System verifies the seller's history
6. System validates the whole offer with the Trade Commission
7. System lists the offer in published offers.
8. System responds with a uniquely identified authorization number.

Extensions:
2a Item not valid
2a1 Use case aborts
5a Seller's history inappropriate
5a1 Use case aborts
6a Trade commission rejects the offer
6a1 Use case aborts
Variations:
2b Price assessment available
2b1 System provides the seller with a price assessment.

Figure 2.4 Example of textual use case

This strategy employees linguistic tools to obtain a parse tree for each use case step. Each parse tree defined an action and its principal attributes. Figure 2.5 is the parse tree generated from the phrase "Seller submits item description". The parse tree defines the syntax role of each sentence vocable. Finally, all acquired actions are converted into a pro-case specification.

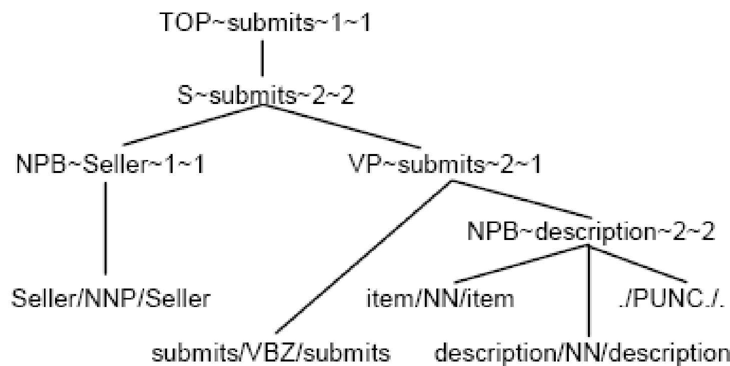

Figure 2.5 Parse tree from one use case sentence

Most steps in a use case describe communication or internal actions. However, this strategy uses special actions to change the flow of control in the use case. These special actions are typically used to define alternative (variation) or exception (extensions) flows. Each exception has a textually-specified triggering condition.

Alike CSP, Pro-case specify behavior in terms of atomic events. These events are emitted (!), or absorbed (?), and internally processed by entities. They are associated with operators in order to define the entities behavior. Among the operator are (in priority order): $*$ for repetition, ; for sequencing, and $+$ for alternative.

```
?SL.submitItem ; #validateDescription ;
(  #cond2a
+  ( NULL + #cond2b ; !SL.providePriceAssessment ) ;
   ?SL.adjustPrice ; #validateSeller ; #verifySeller ;
   (  #cond5a
   +  !TC.validateOffer ;
      (  #cond6a
      +  #listOffer ; #respondAuthorizationNumber )
   )
)
```

Figure 2.6 Generated Pro-case specification from use case in Figure 2.4

Figure 2.6 shows the specification automatically generated from the Figure 2.4 use case. Each token in this definition is associated to a sentence or term from the use case. This specification also models alternative behaviors through the $+$ operator.

This strategy directly process Natural Language, therefor problems may occur if the use case sentences are not clear and the parser tree can not be obtained. The *ad hoc* solution is to learn to write machine-processable textual use cases. Doing so will result into more readable, clear and less ambiguous use cases, which is actually defining a CNL.

The use of Pro-case as target notation is a questionable option. Some Pro-case notations [PV02] are very similar to CSP's and Pro-case's expressiveness does not support the definition of the non-deterministic choice.

2.4.4 Policies based CSP Specification Generation

The work reported in [SHR+05] proposes a strategy to automatically translate Agent-Oriented (AO) system's requirements into a formal specification in CSP (see Section 2.2.1). Thus, the generated CSP model is used as the basis for code generation and other transformations.

The requirements in [SHR+05] are defined as Policy definitions. Policies are sets of system considerations specified to guide decisions of courses of action. [SHR+05] presents some Policy examples related to business objectives:

1. The customer database must be backed up nightly between 1 a.m. and 4 a.m.
2. Platinum customers are to receive no worse than 1-second average response time on all purchase transactions.
3. Only management and the HR senior staff can access personnel records.
4. The number of connections requested by the Web application server cannot exceed the number of connections supported by the associated database.

Along with this CSP generation strategy, an Agent-Oriented Software Engineering (AOSE) methodology [PHS06] was defined to assist system Policies specification. In addition, a tool called R2D2C (Requirements-to-Design-to-Code), which is a NASA patent-pending approach, provides a strategy to generate CSP models from requirements and produce code from it.

In R2D2C, engineers may write specifications as scenarios in the UML use case format using a CNL. Figure 2.7 shows this approach phases, from scenarios capturing to code generation.

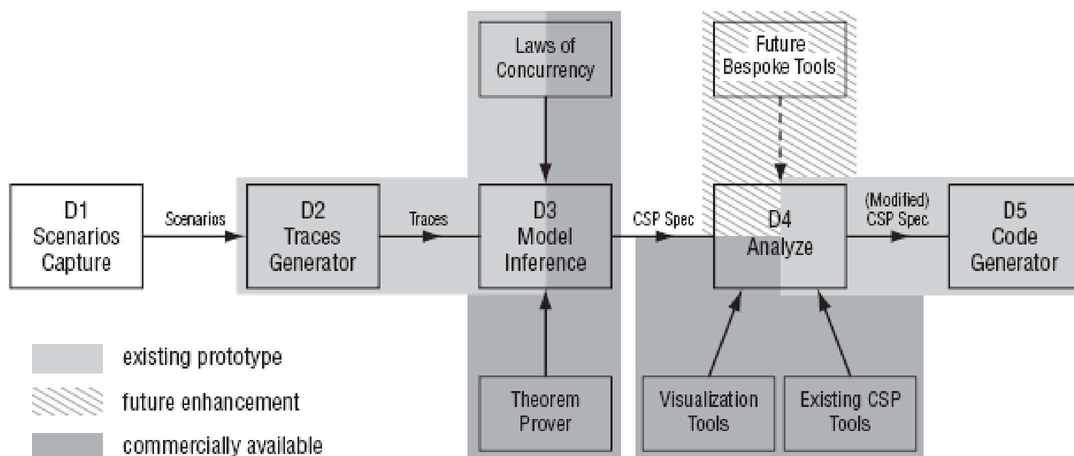

Figure 2.7 The entire [SHR+05] process: from specification to code generation

In order to understand this strategy it is presented an example of AO Policies and the respective generated CSP model. These policies define the operation of a system where a Pager agent sends pages to engineers and controllers when there is a spacecraft anomaly. The Pager agent receives requests from the user interface agent informing that no analyst is logged on, so it gets paging information from the Database agent and pages an appropriate analyst. When instructed by the user interface, the agent stops paging the analyst. These policies are stated as follows:

- When the Pager agent receives a request from the User Interface agent; the Pager agent sends a request to the Database agent for an analyst's pager information and puts the message in a list of requests to the Database agent.
- When the Pager agent receives a pager number from the Database agent, then the Pager agent removes the message from the paging queue and sends a message to the analyst's pager and adds the analyst to the list of paged people.
- When the Pager agent receives a message from the user interface agent to stop paging a particular analyst; the Pager agent sends a stop-paging command to the analyst's pager and removes the analyst from the paged list.
- When the Pager agent receives another kind of message, reply to the sender that the message was not recognized.

The following is a partial CSP description of the Pager agent based on the above policies:

$$
\begin{array}{l}
PAGER_BUS_{db_waiting,paged} = pager.Iin?msg \rightarrow \\
\quad case \\
\qquad GET_USER_INFO_{db_waiting,paged,pagee,text} \\
\qquad\quad if\ msg = (START_PAGING, specialist, text) \\
\\
\qquad BEGIN_PAGING_{db_waiting,paged,in_reply_to_id(msg),pager_num} \\
\qquad\quad if\ msg = (RETURN_DATA.pager_num) \\
\\
\qquad STOP_CONTACT_{db_waiting,paged,pagee} \\
\qquad\quad if\ msg = (STOP_PAGING, pagee) \\
\\
\qquad pager.Iout!(head(msg), UNRECOGNIZED) \\
\qquad\qquad \rightarrow PAGER_BUS_{db_waiting,paged} \\
\quad\quad otherwise
\end{array}
$$

Figure 2.8 CSP generated from the presented policies for the Pager agent

This specification presents the process *PAGER_BUS* that receives a message on its *Iin* channel and stores it in a variable called *msg*. If the message is of type *START_PAGING*, then the *GET_USER_INFO* process is called with parameters of the specialist to page (pagee) and the text to send. If the message is of type *RETURN_DATA* with a pagee's pager number, then the database returns a pager number and the *BEGIN_PAGING* process is executed with a parameter containing the original message *id* and the passed pager number. The third type of message that the Pager agent might receive is one of type *STOP_PAGING*. This message contains a request to stop paging a particular specialist. When this message is received, the *STOP_PAGING* process is executed with the parameter of the specialist type. If the Pager agent receives any other message than the above three messages, an error message is returned to the sender of the message stating that the message is *UNRECOGNIZED*. After this, the *PAGER_BUS* process is again executed.

The formal model derived (in CSP) embodies the policy for anomaly resolution that was specified in the scenarios. However the way policies are structured is very confusing. Each

policy definition contains information about the involved agents and their respective behaviors. The behavioral information is spread and the time (sequence of events) notion is hard to visualize. Besides that, the generated CSP model is quite complete. [SHR+05] did not present details about the policies text processing and the formal model generation. Nevertheless, the presented idea serves as inspiration to define this dissertation approach.

Agents from the AO paradigm can be seen as Components that exchange messages to implement use case functionality. These components are usually presented in distributed systems. Therefor the use of CSP notation to specify such system is reinforced.

CHAPTER 3

Use Case Creation

Use case specifications [RA98] capture system behavior, possibly at different levels of abstraction. Therefore, depending on the developer's need, use cases are created for different purposes. In this chapter we present use case specification templates to document systems from the perspective of the user and of the system components. Both templates define execution flows that determine the interaction between the user and the system. The Controlled Natural Language (CNL), which can be seen as a processable version of English, is used to write use case steps making it possible to accomplish validations and transformations.

3.1 Use Case Creation Process

The way designers choose to specify use cases depends on their experience in the application domain. There are several techniques available to aid identifying possible use cases. Some techniques involve early requirement analysis [FLM^{+}04] or organizational modeling [SC02] in order to identify possible use cases. However, the Rational Unified Process (RUP) [Kru00] seems to be the most popular approach concerning the development of industrial applications.

Basically RUP defines a methodology that aims to capture use cases based on customers' needs. This, in principle, helps to ensures that the use case model will capture the expected functionalities of the system being developed. Use cases can be found by analyzing the system requirements and interviewing potential users as information source.

As use cases are discovered, they should be described briefly. Next, the use case model should be reviewed by the customer to verify whether all found use cases can together provide what the customer wants. Use case models, such as Use Case Diagram [DP02], can be used for this purpose. In an iterative development, a subset of use cases is selected to be detailed using use case specification templates, such as the ones presented in Sections 3.2 and 3.3.

3.2 User View Use Case Template

The user view use cases specify system behavior when one single user executes it. It specifies user operations and expected system responses. Figure 3.1 presents a use case template. Its fields contain comments explaining how the template should be filled. The subsequent subsections explain this template in some details.

Feature <id>

UC <id> - <Use Case Name>

Related requirement(s)

Requirements Ids
<Include here the requirements, using their id, associated with this use case>

Description

<Write here a brief use case description>

Main Flow
From Steps: <from where this flow starts (step ids)>
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 Steps: <from where this flow starts (step ids)>
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>

Exceptions Flows
From Steps: <from where this flow starts (step ids)>
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>

Figure 3.1 User view use case template

3.2.1 Feature

Use cases are initially grouped to form a feature. Each feature contains an identification number. This grouping is convenient for organization purposes; it is not obligatory for the application of the proposed use case template. The use case itself includes a `requirement` list, a brief `description`, and `execution flows`.

3.2.2 Requirement

The requirement list is used for traceability purposes, thus it is possible to check the use case origin. This information is also used to group use cases by requirements. If a certain set of requirements change, it is possible to know which use cases might be impacted and, if it is the case, update them. Thus, test cases related to these use cases can also be updated or regenerated (assuming an automatic approach).

3.2.3 Description

The description gives a general idea about the use case. Because each use case usually considers several requirements, its description should explain the use case main purpose.

3.2.4 Execution Flow

Usually one use case can be used to specify different usage scenarios, depending on user inputs and actions. Hence, each execution flow represents a possible path (a sequence of steps) that the user can take. The following subsections describe the execution flow components.

3.2.4.1 Step

The tuple `(user action, system state, system response)` is called a **step**. Every step is identified through an identifier, an `id`. The user action describes an operation accomplished by the user; depending on the system nature it may be as simple as pressing some button or a more complex operation, such as printing a report. The system state is a condition on the actual system configuration 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.

3.2.4.2 Flow Types

Execution flows are categorized as `main`, `alternative` or `exception` flows. Usually, main execution flows represent the use cases' *happy path*, which is a sequence of steps where everything works as expected. Alternative execution flows represent a choice situation. During the execution of the main flow, it may be possible to execute a different action from the one specified in the main flow and continue the execution of the use case through a different path. Beyond that, it is even possible to define alternative flows beginning at alternative flow steps.

Exception execution flows specify error scenarios caused by invalid input data or critical system states. Alternative and exception flows are strictly related to system state conditions. The system may happen to respond differently given the same user action.

3.2.4.3 Reference between Execution Flows

As presented in the previous section, there are situations when a user can choose between more than one path. When this happens it is necessary to define one execution flow for each path. Each execution flow has a starting point, or *initial state*, and a *final state*. The starting point is represented by the `from steps` field and the final state by the `to step` field.

The `from steps` field can actually assume more than one value, meaning that this flow can be *trigged* from different sources. When this happens, the specified execution flow can be executed after one of the steps is executed. Furthermore, the `to step` field reference only one execution flow step, making it possible to reuse execution flow steps or even define loops.

In the main flow, whenever the `from steps` field is defined as `START` it means that this

use case does not depend on any other, so it can be the starting point of the system usage. Alternatively, the use case main flow may refer to other use case steps, meaning that it can be executed after a sequence of events has occurred in the corresponding use case.

Yet, when the `to step` field from any execution flow is set to `END`, this flow shall terminate successfully after all steps from the flow are executed. After that, the user can execute another use case that has the `from steps` field set to `START`.

Originally, the user view use case template in Figure 3.1 was defined to specify Motorola's mobile phone functionalities. Nevertheless, this template is generic enough to permit the specification of any application, not only mobile phone ones.

The user view use case template holds the main characteristics of other use case definitions, such as UML use cases [RJB99]. However, the flexibility is augmented here. The existence of execution flows starting and ending according to other execution flows makes it possible to associate use cases not only through regular UML associations such as extend, generalization, and include. It enables the reuse of part of other use cases execution flows with the possibility of defining loops so use cases can collaborate to design even more complex tasks.

3.2.5 User View Use Case Example

In order to better visualize the user view use case, this subsection brings an use case example using the template in Figure 3.1. This research was done in partnership with Motorola, therefore, the examples presented in Figure 3.2 is an use case example that specifies a functionality presented in mobile phones.

This use case specifies that text messages received by a mobile phone can be moved from the inbox folder to a special folder, called `Important Messages` folder. This user view use case, in particular, includes a list of related requirements, a brief description, and two execution flows: the main and the exception flow. The `from steps` field, in the main flow, is defined as `START` so this flow does not depend on any other flow, and it is one of the possible starting points to navigate through these application functionalities. The `to step` field is set as `END` so once the four steps from the main flow are executed the flow terminates successfully and the user can execute any use case that have the `from steps` field set to `START`.

The system state column is mainly used to specify conditional situations. Note that this example captures one exception flow. The normal execution of the main flow would pass through the step `2M`, and go on until the end of the main flow. The exception execution goes from step `2M` to step `1E`, when the `message storage is full` (system state). In this case, given the same user action, `Select Move to Important Messages option`, depending on the system state a different system response is presented.

3.3 Component View Use Case Template

A component view use case specifies the system behavior based on the user interaction with system components. In this view, the system is decomposed into components that concurrently process user requests and communicate among themselves. Figure 3.3 shows the component view use case template. It is used to specify use cases in the architectural level of abstraction,

Feature 12896

UC 02 - Incoming message moved to the Important Messages folder

Related requirement(s)

Requirements Ids
REQ_1302, REQ_1326

Description

User accepts message and moves it to the Important Messages folder.

Main Flow
From Steps: START
To Step: END

Step Id	User Action	System State	System Response
1M	Read incoming message.		Message content is displayed.
2M	Open the menu.	"Important Messages" feature is on.	"Move to Important Messages" option is displayed.
3M	Select "Move to Important Messages" option.	Message storage is not full.	"Message moved to Important Messages folder" is displayed.
4M	Wait for at most 2 seconds.		The next message is highlighted.

Exceptions Flows
From Steps: 2M
To Step: END

Step Id	User Action	System State	System Response
1E	Select "Move to Important Messages" option.	Message storage is full.	"Memory required" dialog is displayed.
2E	Confirm memory information dialog.		Message content is displayed.

Figure 3.2 Example of a user view use case

therefore refining the user view use cases specified with the template in Figure 3.1. In other words, for each user view use case, it can be defined a related component view use case and user view steps are decomposed into component messages exchange.

Normally use cases describe system functionalities without revealing the internal structure of the system [RJB99]. However, the proposed component view use cases break this convention and it is actually used to details user view use cases, which follows the regular use case idea; creates an interface between the actor and the system.

In the component view it is necessary to define the component that is invoking an action and the one that is providing the service. It is a message exchange process composed by a `sender`, a `receiver` and a `message`. The user (from the user view) is viewed here as a component, and can either send or receive messages to or from other components, respectively. A component can also send a message to itself. These particularities enable the definition of concurrent scenarios, which is a non-functional requirement. Thus, components can share resources and exchange messages, which is not possible in regular use case models [RJB99].

Feature <id>

UC <id> - <Use Case Name>

Main Flow

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

Step Id	Sender	Message	System State	Receiver
<Step id>	<Component that sends message>	<Message sent from sender component to receiver component>	<system state related to the specified message>	<Component that receives message>

Alternative Flows

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

Step Id	Sender	Message	System State	Receiver
<Step id>	<Component that sends message>	<Message sent from sender component to receiver component>	<system state related to the specified message>	<Component that receives message>

Exception Flows

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

Step Id	Sender	Message	System State	Receiver
<Step id>	<Component that sends message>	<Message sent from sender component to receiver component>	<system state related to the specified message>	<Component that receives message>

Figure 3.3 Component view use case template

The execution flow idea (`main, alternative, and exception`) is the same as in the user view. The system state column plays the same role as previously described in the user view (see Section 3.2.4.3).

A quite similar idea is presented by [Buh98] where it develops a notation called Use Case Maps (UCMs) that allows the design of scenarios at a more abstract level in terms of sequences of responsibilities over a set of components, just as in the component view. Alike, UCMs do not model explicit inter-component communication such as sequence charts do. Yet, UCMs can be translated to Message State Chart (MSC) [HT03] specifications [Bor99]. MSC is also supported by Model Checkers [AY99], allowing properties verification.

3.3.1 Component View Use Case Example

In this section, the component view use case template is presented through an example, so that its construction can be better understood. The example in Figure 3.4 also covers a mobile phone application domain, and it is in fact a refinement of the user view use case example from

Figure 3.2. A formal refinement is later detailed in Chapter 5. As mentioned before, user view steps are decomposed into component messages exchange. Thus, components (including the user) concurrently communicate among themselves.

Feature 12896

UC 02 - Incoming message moved to the Important Messages folder

Main Flow

From Steps: START
To Step: END

Step Id	Sender	Message	System State	Receiver
1M	User	Read incoming message.		Message App
2M	Message App	Open incoming message.		Message Viewer
3M	User	Open the Menu.		Message App
4M	Message App	Display Menu.	"Important Messages" feature is on.	Menu Controller
5M	Menu Controller	"Move to Important Messages" option is displayed.		User
6M	User	Select the "Move to Important Messages" option.		Message App
7M	Message App	Select"Move to Important Messages" option.		Menu Controller
8M	Menu Controller	Save message at "Important Messages" folder.	Message storage is not full.	Message Storage App
9M	Message Storage App	"Message moved to Important Messages folder" is displayed.		User
10M	User	Wait for at most 2 seconds.		User
11M	Message App	The next inbox message is highlighted.		List App
12M	List App	Available message is selected.		User

Exception Flows

From Steps: 7M
To Step: END

Step Id	Sender	Message	System State	Receiver
1E	Menu Controller	Save message at "Important Message" folder.	Message storage is full.	Message Storage App
2E	Message Storage App	"Memory required" message is displayed.		Display App
3E	User	Confirm memory information dialog.		Message App
4E	Message App	Message content is displayed.		User

Figure 3.4 Example of a component view use case

Figure 3.4 is the component view of the Figure 3.2 user view use case. In Figure 3.4, there is one main and one exception flow. The execution of the main flow can be deviated to an exception path after step 7M, when the Message App sends a message to the Menu Controller component. Here, the next message to be exchanged depends on the current system state. Just like in the user view example, the Message Storage state (full or not full) determines the next message to be exchanged between the components. Note that the exception flow step 1E is activated after the step 7M, when the condition fails. The to step field, in the exception flow, states that after the execution flow occurs the execution of the use case terminates (END).

3.4 Controlled Natural Language

Use case fields (user action, system state, system response, and message) are written in a Controlled Natural Language (CNL) with a fixed grammar, defined by knowledge bases. CNL usage does not only make use case text clear and uniform but also makes it possible to process it in order to generate CSP constructions.

The CNL grammar is basically a subset of English grammar. Its sentences construction contains domain specific verbs, terms, and modifiers. The phrases construction is centered on the verb. Domain terms and modifiers are combined in order to take thematic roles around the verb [Fil76]. This strategy is detailed in [Lei06] where it has been used to translate test cases sentences into CSP constructions. The following subsections describe knowledge bases used to store these vocables involved in the definition of the CNL. Besides serving as a reference to understand CNL sentences construction, these knowledge bases are used by the system (also specified in [Lei06]) that implements the translation from CNL sentences to CSP constructions.

3.4.1 Lexicon

The Lexicon stores vocables that may appear in CNL sentences. Each vocable may be a **verb**, a **term**, or a **modifier**. The following subsections describe each one of these vocables.

3.4.1.1 Verb

A verb is used to define an action accomplished by the user, to give a description of the system state, or to specify a message. Actions are described as imperative commands, such as a statement to the user or component to accomplish some operation. In the case of the user view, the verb usually acts as a command in the user action column sentences, which are operations that the user executes in order to obtain a system response.

```
<verb>
   <name>           </name>
   <third-person>   </third-person>
   <past>           </past>
   <participle>     </participle>
   <gerund>         </gerund>
</verb>
```

Figure 3.5 Verb definition template

Figure 3.5 is the XML [RM01] structure used to define CNL verbs. The `verb` definition contains possible verb forms, such as present or past tense. The `name` tag contains the verb in the infinitive form. The infinitive form is used in present tense sentence constructions. Because the singular third person form may vary, the `thirdperson` tag defines it. The `gerund` tag contains the gerund form of the verb (*-ing*). Moreover, the `past` tag defines the past tense form of the verb and the `participle` tag its participle form. Figure 3.6 contains some examples of verb definitions.

```
<verb>
   <name>perform</name>
   <third-person>performs</third-person>
   <gerund>performing</gerund>
   <past>performed</past>
   <participle>performed</participle>
</verb>
<verb>
   <name>set</name>
   <third-person>sets</third-person>
   <gerund>setting</gerund>
   <past>set</past>
   <participle>set</participle>
</verb>
```

```
<verb>
   <name>accept</name>
   <third-person>accepts</third-person>
   <gerund>accepting</gerund>
   <past>accepted</past>
   <participle>accepted</participle>
</verb>
<verb>
   <name>have</name>
   <third-person>has</third-person>
   <gerund>having</gerund>
   <past>had</past>
   <participle>had</participle>
</verb>
```

Figure 3.6 Verb definition examples

3.4.1.2 Term

A term is an element, or entity, from the application domain. It may be just a noun or a noun combined with adjectives or other nouns. It is seen as an application domain object that is manipulated somehow by user or by components throughout the use case execution.

```
<term>
   <name>     </name>
   <plural>   </plural>
   <class>    </class>
   <model>    </model>
</term>
```

Figure 3.7 Term definition template

Figure 3.7 is the XML structure used to define a term. The `name` tag is the term name itself. It defines the singular form of the term. The `plural` tag contains the plural form of the term. Furthermore, the `class` tag defines the Ontology class it belongs to, which determines how the term is related to other terms. Finally, the `model` tag contains the CSP code representation of the term. This representation is used to define *CSP datatype* values.

```
<term>
  <name>conversation history</name>
  <plural>conversation histories</plural>
  <class>list</class>
  <model>CONVERSATION_HISTORY</model>
</term>
<term>
  <name>clean up request</name>
  <plural/>
  <class>dialog</class>
  <model>CLEAN_UP_REQUEST</model>
</term>
```

```
<term>
  <name>voice mail</name>
  <plural/>
  <class>application</class>
  <model>VOICE_MAIL</model>
</term>
<term>
  <name>message center folder</name>
  <plural/>
  <class>menu</class>
  <model>MESSAGE_CENTER_FOLDER</model>
</term>
```

Figure 3.8 Term definition examples

Figure 3.8 gives four examples of term definition. The term conversation history, for instance, has the plural defined as conversation histories. It belongs to the list class of the Ontology, which means a conversation history is treated as a list by the verbs that refer to this class. Further, its CSP code is defined as CONVERSATION_HISTORY.

3.4.1.3 Modifier

A modifier can be anything that qualifies a term, such as an adjective or an adverb. It may even be a noun, once nouns can detail terms characteristics.

```
<modifier>
    <name>           </name>
    <position>       </position>
    <precedence>     </precedence>
    <number>         </number>
    <article>        </article>
    <model>          </model>
</modifier>
```

Figure 3.9 Modifier definition template

Figure 3.9 is the XML structure used to define modifiers. The name tag specifies the name of the modifier. Next, the position tag defines whether the modifier goes before or after the term. The tag precedence specifies the priority order among the modifiers. The number tag is used to determine whether the modifier agrees with a singular or a plural term. The article tag defines whether the modifier can be preceded by a definite or an undefined article. The model tag contains the CSP code representation of the modifier, thus it is used to define *CSP datatype* values.

```
<modifier>
    <name>some</name>
    <position>before</position>
    <precedence>0</precedence>
    <number>plural</number>
    <article>no</article>
    <model>SOME</model>
</modifier>
<modifier>
    <name>next</name>
    <position>before</position>
    <precedence>0</precedence>
    <number>plural</number>
    <article>no</article>
    <model>NEXT</model>
</modifier>
```

```
<modifier>
    <name>with <int/> <term/></name>
    <position>after</position>
    <precedence>0</precedence>
    <number>singular</number>
    <article>no</article>
    <model>WITH_N_NOUN.Int.Item</model>
</modifier>
<modifier>
    <name>at most <int/></name>
    <position>before</position>
    <precedence>0</precedence>
    <number>singular</number>
    <article>no</article>
    <model>AT_MOST.Int</model>
</modifier>
```

Figure 3.10 Modifier definition examples

Figure 3.10 illustrates four modifier definitions. The with <int/> <term/> modifier

is used along with an integer and a term. To better understand this modifier, consider as an example the sentence `Create a message with 3 images`. The number `3` is the integer that goes after `with` and `images` is the other term required by the modifier construction.

3.4.2 Ontology

As already mentioned, each application domain has specific elements and entities. Here, they are called terms and are grouped into classes according to their characteristics. These classes can also be related by inheritance.

```
<ontology>
    <class>
       <description>Generic Class</description>
       <name>Object</name>
       <code>object</code>
       <subclasses>
          <class>
             <description>Represents a generic value</description>
             <name>Value</name>
             <code>value</code>
             <subclasses>
                <class>
                   <description>Represents a state value, e. g.,
                                "enabled", "ON", "high". </description>
                   <name>State Value</name>
                   <code>state_value</code>
                   <subclasses />
                </class>
...
```

Figure 3.11 Ontology fragment

Figure 3.11 presents a small fragment of the Ontology that defines the `Object`, `Value`, and `State Value` classes. The `State Value` class inherits the `Value` class, and the `Value` class inherits the `Object` class. In Figure 3.8, the term `clean up request` is a dialog due to the fact that it belongs to the `dialog` class of the Ontology. This class aims to restrict the way terms are combined with verbs to avoid inconsistent sentences in the use cases. Section 3.4.3 explains how Ontology classes are combined with verb complement definitions.

3.4.3 Case Frame

The case frame defines the relation between verbs and terms. Each case frame determines how a verb can be used to instantiate a sentence. The case grammar formalism [Fil76] is used in order to define how verbs are associated with terms, which can be detailed by modifiers. Each term takes a certain thematic role around the verb; it can be an agent or theme of the sentence, for instance. Each case frame can also be associated to more than one verb, all of them assuming the same meaning (*synonymous*).

```
<frame>
  <description>   </description>
  <name>   </name>
  <verblist>
    <verb>   </verb>
    <verb>   </verb>
  </verblist>
  <roles>
    <role mandatory="   ">agent</role>
    <role mandatory="   ">theme</role>

    <role mandatory="   ">from-value</role>
    <role mandatory="   ">to-value</role>

    <role mandatory="   ">from-loc</role>
    <role mandatory="   ">to-loc</role>
    <role mandatory="   ">at-loc</role>

    <role mandatory="   ">instrument</role>
  </roles>
</frame>
```

Figure 3.12 Case frame definition template

Figure 3.12 shows the case frame definition template. Each `frame` tag contains a case frame definition. The `description` tag gives a brief explanation about how the case frames can se used, possibly including examples. The `name` tag identifies the case frame. The `verblist` tag contains a set of `verb` tags that refer to the verbs related to this frame case. The verbs from this list should have the same semantic meaning. Consequently, they own the same arguments defined in the `roles` tags. Each `role` tag determines a possible verb argument and its type. It has the `mandatory` attribute, which determines whether the argument is obligatory or not. The following are the possible role types:

- agent: a term that executes the verb action.
- theme: a term that suffers the verb action.
- from-value: a term that determines the theme past value.
- to-value: a term that determines the theme future value.
- from-loc: a term that determines the theme past location.
- to-loc: a term that determines the theme future location.
- at-loc: a term that determines the theme current location.
- instrument: a term that determines the way the agent executes the verb action.

Figure 3.13 is an example of case frame definition. In this case, the `SelectItem` identifies the case frame defined by the `select` and the `choose` verbs. The `agent` and the `theme` are mandatory, thus they need to be specified when these verbs are used. The `from-loc` is not mandatory. As a result, it is not necessary to specify this argument. Finally, Table 3.1 presents examples of CNL sentences and illustrate how the verbs are associated with their arguments.

```
<frame>
  <description>Select an item from location. Example: Select
              the send message option from menu</description>
  <name>SelectItem</name>
  <verblist>
    <verb>select</verb>
    <verb>choose</verb>
  </verblist>
  <roles>
    <role mandatory="True">agent</role>
    <role mandatory="True">theme</role>
    <role mandatory="false">from-loc</role>
  </roles>
</frame>
```

Figure 3.13 Case frame example

E-mail **is** available.	`<agent>` is `<to-value>`
Delete the selected message.	Delete `<theme>`
Open the URL link with browser.	Open `<theme> <instrument>`
Exit from the message menu.	Exit `<from-loc>`
Return to saved messages folder.	Return `<to-loc>`
Phone **is** in message inbox screen.	`<theme>` is `<at-loc>`

Table 3.1 CNL sentence examples

3.4.4 Case Frame Restriction

The case frame restriction defines the relation between verb's arguments and Ontology classes. Each verb's argument belongs to an Ontology class in order to restrict the way phrases are written. This minimizes the possibility of writing semantically wrong sentences.

```
<frame>
  <name> </name>
  <restrictions>
     <restriction name=" ">
        <class role=" "> </class>
     </restriction>
     <restriction name=" ">
        <class role=" "> </class>
     </restriction>
  </restrictions>
</frame>
```

Figure 3.14 Case frame restriction definition template

In Figure 3.14, the `frame` tag defines a case frame restriction and its identification is captured by the `name` tag. It contains the `restrictions` tag that holds all possible restrictions. Each `restriction` tag contains an attribute `name` for identifies and a list of `class` tags; each one defines the Ontology class associated with the verb's argument role.

```xml
<frame>
  <description>Set the value of an item. Ex.:
  Set the Fix Dialing to on</description>
  <name>SetItem</name>
  <verblist>
     <verb>set</verb>
     <verb>check</verb>
  </verblist>
  <roles>
     <role mandatory="True">agent</role>
     <role mandatory="True">theme</role>
     <role mandatory="false">to-value</role>
  </roles>
</frame>
```

```xml
<frame>
  <name>SetItem</name>
  <restrictions>
     <restriction name="DTSET_FIELDVALUE_FIELD">
        <class role="theme">field</class>
        <class role="to-value">field_value</class>
     </restriction>
     <restriction name="DTSET_STATEVALUE_SENDABLEITEM">
        <class role="theme">sendable_item</class>
        <class role="to-value">state_value</class>
     </restriction>
     <restriction name="DTSET_STATEVALUE_ITEM">
        <class role="theme">item</class>
        <class role="to-value">state_value</class>
     </restriction>
     <restriction name="DTSET_ITEM">
        <class role="theme">item</class>
     </restriction>
  </restrictions>
</frame>
```

Figure 3.15 Case frame and respective case frame restriction example

Figure 3.15 contains the case frame definition `SetItem` for the verbs `set` and `check`, and its respective case frame restriction. Observed that this case frame contains the following roles: `agent`, `theme`, and `to-value`. Based on theses roles, there are four defined restrictions: the three first restrict the `theme` and the `to-value` arguments, and the last one restricts only the `theme` argument, once the `to-value` argument is not mandatory. Each restriction has a name; this name is used to define a *CSP datatype* (See Section 4.1 for more details). To conclude the restriction definition, it is necessary to associate every role to an Ontology class. This association restricts verb's arguments, for example: the `DTSET_FIELDVALUE_FIELD` restriction defines that the *theme* is a term from the `field` class and the `to-value` argument belongs to the `field_value` class.

3.5 Some Considerations

The definition of user view and component view use cases involves previous knowledge of the application requirements and architectural definitions, such as design patterns. Only when the designer is aware of these definitions and have defined which use cases are to be created (as suggested in Section 3.1), the use case creation should start.

During their creation, It is defined a relation between requirements and use cases. This relation is detailed enough to point which use cases should be verified whenever requirements changes. An alternative approach would be mapping requirements and steps. This would enable verifying what steps are impacted if requirements happen to change.

Another important observation concerns references between execution flows from the same use case and even from different use cases. This relation is originated by the `from steps` and `to step` fields. This relation is neither a UML generalization, nor inclusion, nor extension of a use case [BRJ99]. It is a *new* way to relate sequence of steps in order to reuse steps.

Analyzing the component view use case, it is easy to verify that it is actually a textual way to specify UML Sequence diagrams [BRJ99]. The columns sender and receiver define actors

involved in the communication and the message is the service request itself. The message order determines the Sequence diagram arrangement. Besides one component sending a request to another document, the receiver component can respond this request through another message dispatch. This time the receiver acts as the sender, and vice-versa. Moreover, Collaboration diagrams can also be retrieved from component view use cases.

3.6 Tool Support

Use case sentences must be adherent to the CNL grammar, so designers have to know the CNL grammar. Thus, it was implemented a tool that automatically generated the CNL grammar documentation from the presented CNL knowledge bases. The CNL grammar is generated as HTML pages (Figure 3.16) so it is possible to learn the CNL syntax navigating through the grammar definitions.

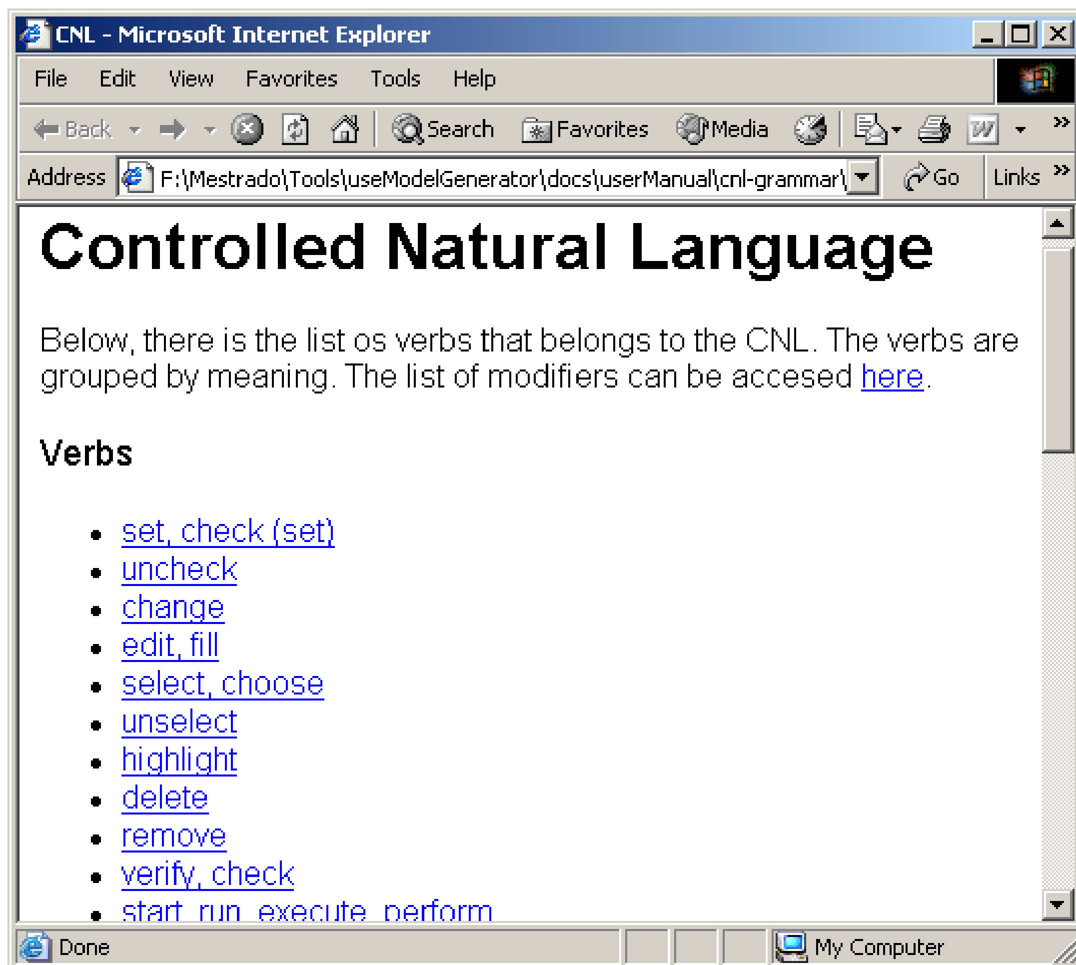

Figure 3.16 CNL grammar as HTML pages

Learning the CNL may be a complex task, once the CNL domain specific terms and ex-

pressions may be constantly updated each time a new set of requirements is considered. Thus, it is recommended that the designer do not waste much time trying to figure out a way to write sentences adherent to the CNL. He should focus attention on use case meaning and complexity.

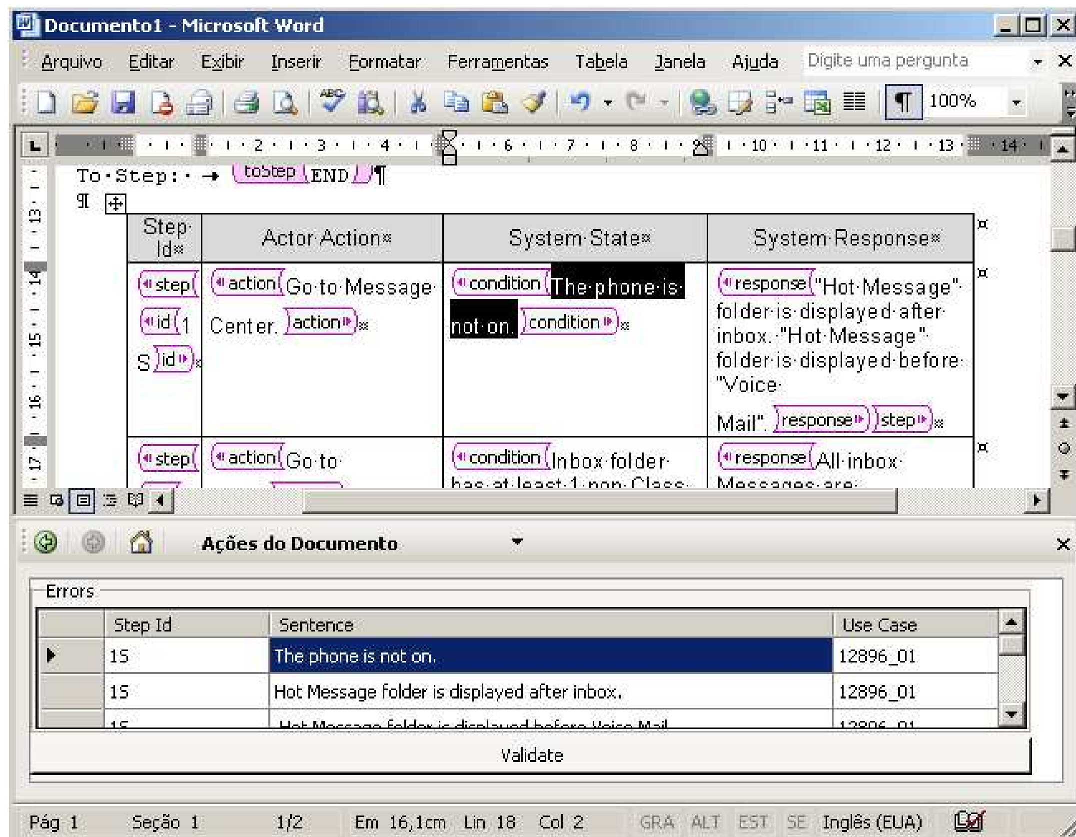

Figure 3.17 Microsoft Word 2003 plug-in to validate CNL sentences

Therefore, it was implemented a tool to automatically validate the use case sentences and report all found inconsistencies. This tool is called **Use Case Validator** and it was a Microsoft Word 2003 [LLM04] *plug-in*. It ensures use cases are written according to use case templates and the CNL syntax. MS Word 2003 is capable to structure the use case's content through XML schemas definitions. The plug-in processes the use case sentences to find inconsistencies (phrases not according to the CNL grammar). Two modules compose the plug-in. One is implemented using the .NET Platform [JYS02] and the other is implemented in Java [Gra97]. The .NET module is a GUI program that accomplishes the CNL validation within Word. The Java module is the Natural Language Processing (NLP) unit responsible to verify whether sentences are written according to the CNL rules. More details about the NLP module implementation can be found at [Lei06].

It was also implemented a Java version of the **Use Case Validator** (Figure 3.18). It reads a word file path and reports all inconsistencies, such as sentences not according to CNL.

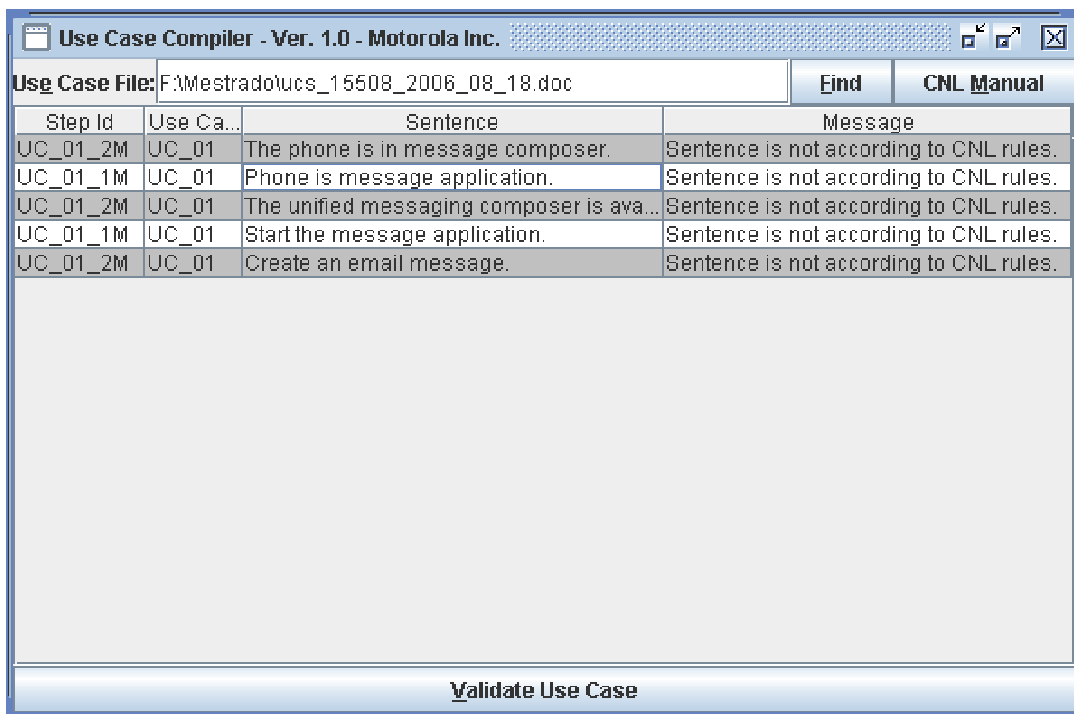

Figure 3.18 Java version of **Use Case Validator**

CHAPTER 4

CSP Specification Generation

Once the use cases are created, as explained in the previous chapter, following the use case templates and CNL grammar, it is possible to generate formal specifications from it. Particularly, our target is a model in the CSP process algebra.

Because each template holds a different structure, the generated CSP specifications format differs. Despite the fact that they have different formats, the language used to define their content is the same, CNL. Consequently the translation between CNL sentences to CSP elements is basically the same. Section 4.1 explains the strategy used to translate CNL sentences to CSP events, and Sections 4.2 and 4.3 present the proposed approach to generate the user and the component view specifications, respectively, from use cases.

4.1 CNL Translation to CSP

As it was presented in Section 3.4, use case sentences are written according to CNL grammar, which is defined by knowledge bases. The following subsections define the CSP alphabet and the method used to translate CNL sentences to CSP events, based on this alphabet.

4.1.1 CSP Alphabet

A set of CSP channels and datatypes represent a CSP alphabet, which can be seen as a specialized language. Using the presented CNL knowledge bases, as a dictionary, it is possible to define the CSP alphabet that is related to the CNL grammar. Verbs, terms, and modifiers are translated into possible CSP events and datatypes that are further used to generate the CSP model. To understand the CSP alphabet creation, part of it is presented below.

As mentioned, basically a *case frame* and a *case frame restriction* (see Sections 3.4.3 and 3.4.4 for details) are mapped to CSP channel names and datatypes, respectively. Figure 4.1 shows the CSP channel and datatype generated from the `SetItem` case frame definition in Figure 3.15. It defines the channel `set` and its type, the `DTSet` datatype. `DTSet` datatype is constructed based on four possible subtypes:

- `DTSET_FIELDVALUE_FIELD`
- `DTSET_STATEVALUE_ITEM`
- `DTSET_ITEM`
- `DTSET_STATEVALUE_SENDABLEITEM`

Each of these subtypes is originated from the `SetItem` case frame restriction. Hence, the case frame restricts possible verb arguments and correspondingly it defines the channel's datatype, which restricts the possible messages it can transmit.

```
channel set : DTSet
datatype DTSet =
 DTSET_FIELDVALUE_FIELD.(FieldValue, Set(Modifier)).(Field, Set(Modifier))
 | DTSET_STATEVALUE_ITEM.(StateValue, Set(Modifier)).(Item, Set(Modifier))
 | DTSET_ITEM.(Item, Set(Modifier))
 | DTSET_STATEVALUE_SENDABLEITEM.(StateValue, Set(Modifier)).
                                  (SendableItem, Set(Modifier))
```

Figure 4.1 CSP generated for SetItem case frame (Figure 3.15)

The `DTSET_ITEM` subtype, for instance, is originated from the `DTSET_ITEM` case frame restriction. It contains the `(Item, Set(Modifier))` argument, where `Item` is characterized by a set of `Modifier`. `Item` can be any Lexicon term that belongs to the `Item` class of the Ontology, and the `Modifier` is simply a modifier defined in the Lexicon. Figure 4.2 contains some `Item` terms, such as `ALARM_CLOCK` and `SHORTCUT`. It also contains modifiers, such as `SOME` and `NEXT`.

```
datatype FieldValue =
 CHARACTER
 | WORD_VALUE
 | TEXT_VALUE
 | PHONE_NUMBER_VALUE
 | REPLY_ADDRESS
 | DIGIT_VALUE
 | ADDRESS_VALUE
 | URGENT_VALUE
 | NO_CONTENT_VALUE
 | NUMBER_VALUE
 | INVALID_NUMBER_VALUE
 | EMPTY_CONTENT_VALUE
 | PREDEFINED_TEXT_VALUE
 | SIMPLE_CHARACTER
 ...

datatype Modifier =
 SOME
 | NEXT
 | WITH_N_NOUN.Int.Item
 | AT_MOST.Int
 | STORED_IN.Hardware
 | PREVIOUS
 | ANY
 | RIGHT
 | SEPARATED_BY_SPACE
 | WHICH_IS_NOT_ON.Screen
 | INTEGER.Int
 | VALID
 | BLANK
 | CREATED
 ...

datatype StateValue =
 FAVORITE_MESSAGE_STORAGE
 | FULL_STATE_VALUE
 | HIGHLIGHTED_VALUE
 | HIGH_VALUE
 | NORMAL_VALUE
 | LOW_VALUE
 | EMPTY_STATE
 | YES_VALUE
 | ACTIVE_CALL_STATE_VALUE
 | POPULATED_VALUE
 | DISPLAYED_VALUE
 | AVAILABLE_VALUE
 | WITHOUT_CONTENT_VALUE
 | POWERED_ON_VALUE
 ...

datatype Item =
 MENUITEM.MenuItem
 | VARIABLEITEM.VariableItem
 | FIELD.Field
 | LISTITEM.ListItem
 | KEY.Key
 | MESSAGE_STORAGE
 | SECOND
 | ALARM_CLOCK
 | INCOMING_CALL_ALERT
 | INCOMING_MESSAGE_ALERT
 | OPERATION
 | GENERAL_ITEM
 | SHORTCUT
 | SIGNATURE_ITEM
 ...

datatype SendableItem =
 INCOMING_MESSAGE
 | INBOX_MESSAGE
 | QUICK_NOTE
 | MESSAGE
 | EMAIL_ENTRY
 | SMS_MESSAGE
 | SENT_MESSAGE
 | SMS_LONG_MESSAGE
 | EMS_MESSAGE
 | PROTECTED_CONTENT_MESSAGE
 | DELIVERY_REPORTS_MESSAGE
 | MMS_MESSAGE
 | VOICE_MESSAGE
 | VOICE_MMS_MESSAGE
 ...

datatype Field =
 EDITOR_FIELD
 | PHONE_NUMBER_FIELD
 | MESSAGE_FIELD
 | TO_FIELD
 | MSG_FIELD
 | READ_REPORT_FIELD
 | SUBJECT_FIELD
 | CALL_BACK_NUMBER_FIELD
 | PRIORITY_FIELD
 | CALL_BACK_FIELD
 | DELIVERY_REPORT_FIELD
 | SUBJET_FIELD
 | SERVICE_CENTER_NUMBER_FIELD
 | CC_FIELD
 ...
```

Figure 4.2 Example of datatypes generated based on the Lexicon and the on Ontology

Analyzing the DTSet datatype, it is possible to identify the following datatypes:

- FieldValue
- Modifier
- Field
- StateValue
- Item
- SendableItem

Figure 4.2 contains part of these datatype definitions. The Modifier is produced directly from modifier definitions in the Lexicon (see Section 3.4.1.3). All other datatypes are created based on the Ontology hierarchy; each term from the Lexicon (see Sections 3.4.1.2) belongs to an Ontology class (see Section 3.4.2). The Item datatype for instance is composed by other constituent types, such as MENUITEM, VARIABLEITEM, FIELD, LISTITEM, and KEY. These sub items come from the Ontology hierarchical definition; they are Item specializations. All other values from the Item datatype belong directly to the Item class.

4.1.2 CSP Events Generation

As it was presented in the previous section, the CNL knowledge bases are employed in the CSP alphabet definition. Using this alphabet it is possible to translate each sentence from the use case templates into CSP events, which are used to define use models from the use cases.

Figure 4.3 presents CNL sentences that are according to the SetItem case frame. The sentence Set the current to field to an invalid email address structure is based on the DTSET_FIELDVALUE_FIELD case frame restriction. This restriction allows the verb set to accept two arguments: the value TO_FIELD from the field class and the value EMAIL_ADDRESS_VALUE from the field_value class. They are characterized by a set of Modifiers; the field TO_FIELD is changed by CURRENT_MODIFIER and the field value EMAIL_ADDRESS_VALUE by the INVALID modifier.

```
-- Set the current "to field" to an invalid email address.
set.DTSET_FIELDVALUE_FIELD.(EMAIL_ADDRESS_VALUE, {INVALID}).
                           (TO_FIELD, {CURRENT_MODIFIER})

-- Set at least 3 sms message to read.
set.DTSET_STATEVALUE_SENDABLEITEM.(READ_VALUE, {}).(SMS_MESSAGE, {AT_LEAST.3})

-- Set all flex to on.
set.DTSET_STATEVALUE_ITEM.(ON_VALUE, {}).(FLEX_ITEM, {ALL})

-- Set the primary setup.
set.DTSET_ITEM.(PRIMARY_SETUP_ITEM, {})
```

Figure 4.3 Example of a CNL sentence and its translation to a CSP event

Mapping CNL sentences to CSP events is just the first step to create the CSP model. As already mentioned, the specification generation depends on the use case template structure and fields. The following sections explain the strategy used to generate the user and the component view use models.

4.2 User View Model Generation

The user view model generation is accomplished through the translation of each use case. Each use case should contain at least one execution flow, which is the main flow; alternative and exception flows are not mandatory. As explained in Section 3.2, each flow contains a list of steps, and each of these steps is mapped to a CSP process. The process name is defined by the step id, which is a unique identifier among all steps. The process body is defined by events generated from the user action, system state, and system response fields. Each one of these fields is described through one or more sentences in the CNL format.

To delimitate the events generated from the user action, system state, and system response fields it is used control events (**steps**, **conditions**, and **expectedResults**). Once each sentence from these fields is translated to CSP events, they are arranged in sequence separated by the CSP prefix operator.

The final task is to link these step processes altogether. This link is defined depending on the step position. If it is *not* the last step from the flow, it should refer to all steps that are listed in the `from steps` field from other flows, and to the next step from the flow. This means, after this step is executed, the next step from the flow or another flow step shall execute. If the step is the last step from the flow, the same procedure should be performed; however, once the last step does not have a next step, it should be also added a link to the step defined at the `to step` field of the flow. Notice that the `from steps` and `to step` fields are of major importance to the model construction. They determine when the flow starts and ends, and the steps links. Remember that the `from steps` field can contain more than one step.

If a flow `from steps` field is set to the `START` keyword, the system can start from this flow. In addition, the process should stop, after the last step from the flow is executed, if the `to step` field is set to the `END` keyword. This is represented in CSP through the `SKIP` process.

```
include "CSP_HEADER_USER.csp"

System =
  UC_02_1M ; System
  [] ...
```

Figure 4.4 CSP specification generated from the main flow of Figure 3.2

Figure 4.4 contains the `System` process, which is the main process of the specified system. The `System` process refers to the process `UC_02_1M` and all other execution flows with the `from step` defined as `START`. The statement `include "CSP_HEADER_USER.csp"` imports the CSP alphabet used in the use model specification.

Figure 4.5 shows four processes, which are the translation to CSP of the steps from the use case main flow in Figure 3.2. Each one of these processes is generated based on the previous presented procedure. The process `UC_02_2M`, for instance, contains one user action, one system state and one system response. These three sentences are translated to CSP events, and moreover the processes `UC_02_3M` and `UC_02_1E` are referenced. As previously explained, the process `UC_02_3M` represents the next step from the flow, after `UC_02_2M`, and the process `UC_02_1E` is referenced once the exception flow has the step `UC_02_2M` set at

```
UC_02_1M =
  steps ->
  -- Read incoming message.
  read.DTREA_SENDABLEITEM.(INCOMING_MESSAGE, {}) ->
  expectedResults ->
  -- Message content is displayed.
  display.DTDIS_FIELDVALUE.(MESSAGE_CONTENT_FIELD_VALUE, {}) ->
  UC_02_2M

UC_02_2M =
  steps ->
  -- Open the menu.
  open.DTOPE_MENU.(CSM_MENU_LIST, {}) ->
  conditions ->
  -- "Important Message" feature is on.
  isstate.DTISL_LIST.(FEATURE,{IMPORTANT_MESSAGE_FOLDER}),(ON_VALUE) ->
  expectedResults ->
  --  "Move to Important Messages" option is displayed.
  isstate.DTISS_MENUITEM_STATEVALUE.(MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}).
                                   (DISPLAYED_VALUE, {}) ->
  (UC_02_3M [] UC_02_1E)

UC_02_3M =
  steps ->
  -- Select the "Move to Important Messages" option.
  select.DTSEL_MENUITEM.(MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}) ->
  conditions ->
  -- Message storage is not full.
  isstate.DTISS_ITEM_STATEVALUE.(MESSAGE_STORAGE,{}).(FULL_STATE_VALUE,{NOT})->
  expectedResults ->
  -- "Message moved to Important Message folder" is displayed.
  isstate.DTISS_DIALOG_STATEVALUE.
          (MESSAGE_MOVED_TO_IMPORTANT_MESSAGE_FOLDER,{}).
          (DISPLAYED_VALUE, {}) ->
  UC_02_4M

UC_02_4M =
  steps ->
  -- Wait for at most 2 seconds.
  wait.DTWAI_ITEM.(SECOND, {AT_MOST.2}) ->
  expectedResults ->
  -- The next message is highlighted.
  isstate.DTISS_SENDABLEITEM_STATEVALUE.(MESSAGE,{NEXT}).
                                       (HIGHLIGHTED_VALUE,{})->
  SKIP
```

Figure 4.5 CSP specification generated from the main flow of Figure 3.2 use case

its `from steps` field. Furthermore, notice that the last step from the main flow generates the process `UC_02_4M` that is finalized with the `SKIP` process, once the `to step` field of the main flow is set to `END`.

Figure 4.6 refers to the exception flow translation to CSP. There are two steps; the first one contains a condition, based on the system state column, since this exception occurs when the `Message storage is full` statement is true. Finally the process `UC_02_2E` is finalized with the `SKIP` process, once the `to step` field of the exception flow is set to `END`.

```
UC_02_1E =
  steps ->
  -- Select "Move to Important Messages" option.
  select.DTSEL_MENUITEM.(MOVE_TO_IMPORTANT_MESSAGES_OPTION, {}) ->
  conditions ->
    -- Message storage is full.
  isstate.DTISS_ITEM_STATEVALUE.(MESSAGE_STORAGE,{}).(FULL_STATE_VALUE,{})->
  expectedResults ->
    -- "Memory required" dialog is displayed.
  isstate.DTISS_DIALOG_STATEVALUE.(MEMORY_REQUIRED_DIALOG,{}).
                                  (DISPLAYED_VALUE,{})->
  UC_02_2E

UC_02_2E =
  steps ->
  -- Confirm memory information dialog.
  confirm.DTCON_DIALOG.(MEMORY_INFORMATION_DIALOG, {}) ->
  expectedResults ->
  -- Message content is displayed.
  display.DTDIS_FIELDVALUE.(MESSAGE_CONTENT_FIELD_VALUE, {}) ->
  SKIP
```

Figure 4.6 CSP specification generated from the exception flow of Figure 3.2 use case

4.3 Component View Model Generation

The way events from the user and the component view are translated is the same. However, the component view model structure is quite different. Channels in the component view contain information about the components involved in the message exchange and their names are suffixed by `Comp`, making the user and component view CSP alphabets different. In this particular example, the datatypes used in both views are the same; since both use cases refer to elements from the same application domain. However, it is possible to generate the user or the component view models based on different CNL knowledge bases, which is the source for the CSP alphabet creation.

In Figure 4.7, the user view CSP alphabet is imported through the statement `include "CSP_HEADER_USER.csp"`, and it is defined the datatype `ComponentElement` that is a possible component from the component view. It also defines the `ComponentView` *nametype*, which is the composition of two `ComponentElement`, the **sender** and the **receiver** in the message exchange. The channel `setComp` is an example of a component view channel definition; all other events are defined in a similar manner.

```
include "CSP_HEADER_USER.csp"

datatype ComponentElement = USER | MESSAGE_APP | MESSAGE_VIEWER |
         MENU_CONTROLLER | MESSAGE_STORAGE_APP | LIST_APP | DISPLAY_APP

nametype ComponentView = ComponentElement.ComponentElement

channel setComp : ComponentView.DTSet
```

Figure 4.7 Part of the component view model alphabet including the `setComp` channel as example

As presented in Figure 4.8, the main system process in the component view is defined

by the *parallel* execution every components, including the user. They are composed pairwise using the CSP **alphabetized parallel** operator $[|s|]$, presented in Section 2.2.1. The USER_P and the MESSAGE_APP_P processes are composed to define the process SubSystem1. The composition is accomplished based on events from both components, (User_Channels and Message_App_Channels), to accomplish the alphabetized parallelism.

```
SubSystem1_events = union(User_Channels,Message_App_Channels)
SubSystem1 = USER_P [ User_Channels || Message_App_Channels] MESSAGE_APP_P

SubSystem2_events = union(SubSystem1_events,Message_View_Channels)
SubSystem2 = SubSystem1 [ SubSystem1_events ||
                          Message_View_Channels] MESSAGE_VIEWER_P
User_Channels = {|
 readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{CLASS_2,NON}),
 openComp.USER.MESSAGE_APP.DTOPE_MENU.(CSM_MENU_LIST, {}),
 isstateComp.MENU_CONTROLLER.USER.DTISS_MENUITEM_STATEVALUE.
            (MOVE_TO_IMPORTANT_MESSAGES_OPTION, {}).(DISPLAYED_VALUE, {}),
 selectComp.USER.MESSAGE_APP.DTSEL_MENUITEM.
           (MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}),
 isstateComp.MESSAGE_STORAGE_APP.USER.DTISS_DIALOG_STATEVALUE.
            (MESSAGE_MOVED_TO_IMPORTANT_MESAGE_FOLDER,{}).(DISPLAYED_VALUE,{}),
 waitComp.USER.USER.DTWAI_ITEM.(SECOND, {AT_MOST.2}),
 isstateComp.LIST_APP.USER.DTISS_SENDABLEITEM_STATEVALUE.(MESSAGE, {}).
            (AVAILABLE_VALUE, {}),
 confirmComp.USER.MESSAGE_APP.DTCON_DIALOG.(MEMORY_INFORMATION_DIALOG, {}),
 isstateComp.USER.LIST_APP.DTISS_SENDABLEITEM_STATEVALUE.
            (INBOX_MESSAGE,{NEXT}).(HIGHLIGHTED_VALUE, {})
|}
Message_App_Channels = {|
 readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{}),
 openComp.MESSAGE_APP.MESSAGE_VIEWER.DTOPE_SENDABLE_ITEM.(INCOMING_MESSAGE,{}),
 openComp.USER.MESSAGE_APP.DTOPE_MENU.(CSM_MENU_LIST,{}),
 displayComp.MESSAGE_APP.MENU_CONTROLLER.DTDIS_MENU.(CSM_MENU_LIST,{}),
 isstateComp.MESSAGE_APP.MENU_CONTROLLER.DTISL_LIST.
            (FEATURE,{IMPORTANT_MESSAGE_FOLDER}),(ON_VALUE)
 selectComp.USER.MESSAGE_APP.DTSEL_MENUITEM.
           (MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}),
 selectComp.MESSAGE_APP.MENU_CONTROLLER.DTSEL_MENUITEM.
           (MOVE_TO_IMPORTANT_MESSAGES_OPTION, {}),
 isstateComp.MESSAGE_APP.LIST_APP.DTISS_SENDABLEITEM_STATEVALUE.
            (INBOX_MESSAGE, {NEXT}).(HIGHLIGHTED_VALUE, {}),
 confirmComp.USER.MESSAGE_APP.DTCON_DIALOG.(MEMORY_INFORMATION_DIALOG,{}),
 displayComp.MESSAGE_APP.USER.DTDIS_FIELDVALUE.(MESSAGE_CONTENT_FIELD_VALUE,{})
|}
```

Figure 4.8 Part of the main component process (Composition between components)

Similarly to the user view, each component process is defined as the CSP *external choice* between all the first steps from execution flow with the from steps defined as START. Each use case usually has only one starting point, therefore each component has a subprocess for each use case.

Unlike the user view, here each step is mapped into two CSP events, one for each component that takes part in the communication. Each component process is defined as a sequence of messages communicated with other component. The system state description, if any, is

also exchanged as a message between the involved components. After each message there is a CSP *prefix* to the next step that involves the component. In Figure 4.9, it is defined the USER_P process for the presented use case. Events readComp.USER.MESSAGE_APP and isstateComp.MENU_CONTROLLER.USER are examples of the communication between the user and system components, such as MESSAGE_APP and MENU_CONTROLLER.

```
USER_P =
  -- Scenario Case: Incoming message is moved to the Important Messages folder
  USER_UC_02
  [] ...

USER_UC_02 =
  -- Message: Read incoming message.
  readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{})->
  -- Message: Open the menu.
  openComp.USER.MESSAGE_APP.DTOPE_MENU.(CSM_MENU_LIST,{})->
  -- Message: "Move to Important Messages" option is displayed.
  isstateComp.MENU_CONTROLLER.USER.DTISS_MENUITEM_STATEVALUE.
            (MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}).(DISPLAYED_VALUE, {}) ->
  -- Message: Select the "Move to Important Messages" option.
  selectComp.USER.MESSAGE_APP.DTSEL_MENUITEM.
            (MOVE_TO_IMPORTANT_MESSAGES_OPTION,{})->
  (USER_UC_02_9M [] USER_UC_02_3E)

USER_UC_02_9M =
  -- Message: "Message moved to Important Message folder" is displayed.
  isstateComp.MESSAGE_STORAGE_APP.USER.DTISS_DIALOG_STATEVALUE.
            (MESSAGE_MOVED_TO_IMPORTANT_MESSAGE_FOLDER,{}).
            (DISPLAYED_VALUE, {}) ->
  -- Message: Wait for at most 2 seconds.
  waitComp.USER.USER.DTWAI_ITEM.(SECOND, {AT_MOST.2}) ->
  -- Message: Available message is selected.
  isstateComp.LIST_APP.USER.DTISS_SENDABLEITEM_STATEVALUE.(MESSAGE, {}).
                           (AVAILABLE_VALUE, {}) ->
  USER_P

USER_UC_02_3E =
  -- Message: Confirm memory information dialog.
  confirmComp.USER.MESSAGE_APP.DTCON_DIALOG.(MEMORY_INFORMATION_DIALOG,{})->
  -- Message: Message content is displayed.
  displayComp.MESSAGE_APP.USER.DTDIS_FIELDVALUE.
            (MESSAGE_CONTENT_FIELD_VALUE, {}) ->
  USER_P
```

Figure 4.9 USER_P process exchanging messages with other components

As in the user view, if there are an alternative or an exception flow in the use case, the steps referred in the from steps are used to identify from where the alternative or exception flow originates. The resultant effect in the component process is the addition of the CSP *external choice* operator, after the step specified in the from steps field, referring to the alternative or exception flow first step. In Figure 4.9, at the end of the main flow process, there is a reference to processes USER_UC_02_9M and USER_UC_02_3E in order to link it to the main and to the exception flow. Figure 4.10 contains the MESSAGE_APP component process. Observe that it exchanges messages with the previously presented USER_P component process and others.

```
MESSAGE_APP_P =
  -- Scenario Case: Incoming message is moved to the Important Messages folder.
  MESSAGE_APP_UC_02

MESSAGE_APP_UC_02 =
  -- Message: Read incoming message.
  readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{})->
  -- Message: Open incoming message.
  openComp.MESSAGE_APP.MESSAGE_VIEWER.DTOPE_SENDABLE_ITEM.
          (INCOMING_MESSAGE,{})->
  -- Message: Open the menu.
  openComp.USER.MESSAGE_APP.DTOPE_MENU.(CSM_MENU_LIST, {}) ->
  -- Message: Display menu.
  displayComp.MESSAGE_APP.MENU_CONTROLLER.DTDIS_MENU.(CSM_MENU_LIST,{}) ->
  -- Message: "Important Message" feature is on.
  isstateComp.MESSAGE_APP.MENU_CONTROLLER.DTISL_LIST.
            (FEATURE,{IMPORTANT_MESSAGE_FOLDER}),(ON_VALUE) ->
  -- Message: Select the "Move to Important Messages" option.
  selectComp.USER.MESSAGE_APP.DTSEL_MENUITEM.
            (MOVE_TO_IMPORTANT_MESSAGES_OPTION, {}) ->
  -- Message: Select the "Move to Important Messages" option.
  selectComp.MESSAGE_APP.MENU_CONTROLLER.DTSEL_MENUITEM.
            (MOVE_TO_IMPORTANT_MESSAGES_OPTION, {}) ->
  (MESSAGE_APP_UC_01_1OM [] MESSAGE_APP_UC_02_3E)

MESSAGE_APP_UC_01_1OM =
  -- Message: The next inbox message is highlighted.
  isstateComp.MESSAGE_APP.LIST_APP.DTISS_SENDABLEITEM_STATEVALUE.
            (INBOX_MESSAGE, {NEXT}).(HIGHLIGHTED_VALUE, {}) ->
  MESSAGE_APP_P

MESSAGE_APP_UC_02_3E =
  -- Message: Confirm memory information dialog.
  confirmComp.USER.MESSAGE_APP.DTCON_DIALOG.(MEMORY_INFORMATION_DIALOG, {}) ->
  -- Message: Message content is displayed.
  displayComp.MESSAGE_APP.USER.DTDIS_FIELDVALUE.
            (MESSAGE_CONTENT_FIELD_VALUE,{})->
  MESSAGE_APP_P
```

Figure 4.10 `MESSAGE_APP` process

4.4 Some Considerations

The user view model is much simpler to generate than the component view model. This is due to the fact that it is does not invoke parallelism in its definition. Moreover, the user view use case template is simpler to use. On the other hand, using the component view use case template, besides involving architectural knowledge by the designer, requires an abstract view of the system; it must be viewed as a set of components.

The user view main process, `System` (see Figure 4.4), is defined as the CSP *external choice* among the steps of use case flows that has the `from steps` field set to `START`. In contrast, the component view main process is defined as the parallel composition between system components.

Our model generation strategy is quite similar to [Bor99], which generated to MSC from Use Case Maps [Buh98]. However, the component view template promotes better reuse of

specifications, since it is possible to reuse any sequence of steps. Correspondingly to CSP, MSC holds the concurrent aspect of the specified system. However, CSP is a process algebra that enables the definition of channels and datatypes, along with flexible and elegant parallel operators. There are MSC extensions, such as Extended Message State Chart (EMSC), that enable the definition of datatypes using formal data language (DL) notations [EFM99].

There is a wide discussion about using CSP or MSC to model concurrent systems. Nevertheless, CSP notation is a more refined algebraic notation with a verified refinement theory, which is supported by model checker such as FRD [fdr97, Gar97], animators [pro98], and implementations [WAF02, P.H02, Gal96].

4.5 Tool Support

A Java [Gra97] application was implemented to mechanize the translation of the user and the component views use cases into CSP models. The application reads user and component view use cases as Word 2003 document files, check its content (using the tool presented in Section 3.6), and generates the user and the component models.

Here, the NLP module [Lei06] is once again used to retrieve CSP events from the CNL sentences. The use model generation tool itself implements the strategy presented in this chapter; it structures the events generated by [Lei06] into processes to define the system formal model.

CHAPTER 5
Model Refinement

Modeling systems at different levels of abstraction has the advantage of capturing several architectural views, as illustrated here with the user and the component views. Nevertheless, it is essential that the several architectural views produced are consistent. In general, these views are expressed using different alphabets (event names) so a relation is needed in order to compare them. Thus, one or more events from one model can be related to one or more events of another model. Defining such a relation allows replacing abstract events for more concrete ones, formally keeping track of the relationship between the models.

5.1 Abstraction Levels

This dissertation defines only two abstraction levels: the user and the component views. However, the strategy presented in this section is generic enough and can be applied to an arbitrary number of views. Thus, use case engineers can define new use case templates and propose new ways to map events from use cases written in different levels of abstraction.

The main goal of this approach is to *decompose* events using other events that detail system behavior, in an incremental way. This would enrich the model with more details and eventually the events can be mapped into more concrete constructions, such as programming languages commands (typically method calls).

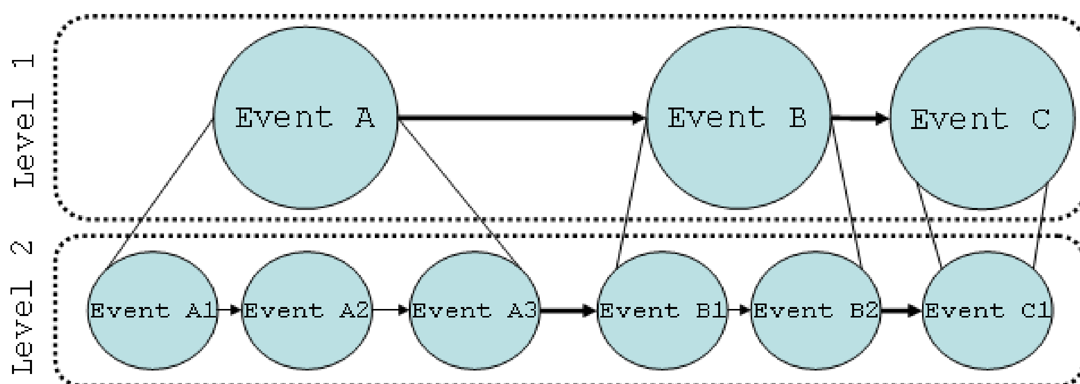

Figure 5.1 Events decomposition at different abstraction levels

Figure 5.1 illustrates a sequence of events `A`, `B`, and `C` that exemplifies part of an execution flow at a certain abstraction level called `Level 1`. These events belong to the `Level 1` alphabet. In addition to that, it is possible to detail these events using a different set of events,

more specifically, another alphabet. In this case, it would be the `Level 2` alphabet, which contains events such as `A1`, `A2`, and `A3`, which are a more concrete abstraction of the `A` event.

Thus, the event decomposition strategy may be applied to map sequences of events at different levels of abstraction, or **views**, as it is called here. This mapping works as a dictionary that translates sequence of events from one model to another.

5.2 Refinement Mapping

As previously described, here we consider that the refinement relation between different views, such as the user and the component views, is a mapping from sequences of events from one view to the other. This provides flexibility, since it allows a *many to many* relationship between events in the two models. Figure 5.2 gives a graphical idea of the mapping specification; each sequence of event from a certain `View_1` is mapped to a sequence of events in `View_2`.

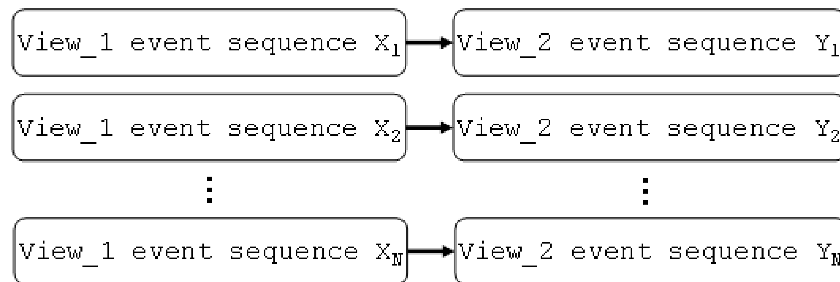

Figure 5.2 Mapping of sequence of events in different views

Once this mapping is defined as a set of pairs of sequences, a function uses it to create a *CSP process* that represents the mapping. In each pair of the mapping, the first sequence represents events from the most abstract view, and the second sequence contains events from the more concrete view. This mapping is an injective relation since one sequence of events from one view is decomposed into only one sequence of events from the other view; otherwise, one sequence of events from one view would have more than one possible translation to the other view alphabet.

Figure 5.3 presents the function that generates the mapping process used in the refinement. The `MAPPING_FUNCTION` receives the mapping between the two views and uses it to create a process using the `MAPPING_PROCESS` function. `MAPPING_PROCESS` is defined as an *indexed external choice* among the processes generated by the `makeProcess` auxiliary function. This function takes each pair from the mapping and forms a sequence initiated by the events from the abstract model followed by corresponding events from the concrete model, terminating by the `SKIP` process. The `head` function returns the first element of a sequence and the `tail` function returns a list without its first element. Finally, the `first` and the `second` functions are projection functions on pairs, with obvious behavior.

As presented in Figure 5.4, the process that represents the mapping is composed, through an *alphabetized parallel composition*, with the abstract model. This composed process contains events from the alphabet of both views. Once the events from the abstract model are hidden,

```
MAPPING_FUNCTION( map) = MAPPING_PROCESS(map); MAPPING_FUNCTION( map)
MAPPING_PROCESS(map) = [] p : map @ makeProcess( first(p)^second(p))

makeProcess(<>) = SKIP
makeProcess(seq) = [] i : {| head(seq) |} @ (i -> makeProcess( tail(seq)))

first(p) = let (m,n) = p within m
second(p) = let (m,n) = p within n
```

Figure 5.3 Mapping function

it must produce a process that is refined by the concrete model. The mapping process works as a trigger from one view to another; occurrences of events in the abstract model force the occurrence of the related concrete events based on the defined map.

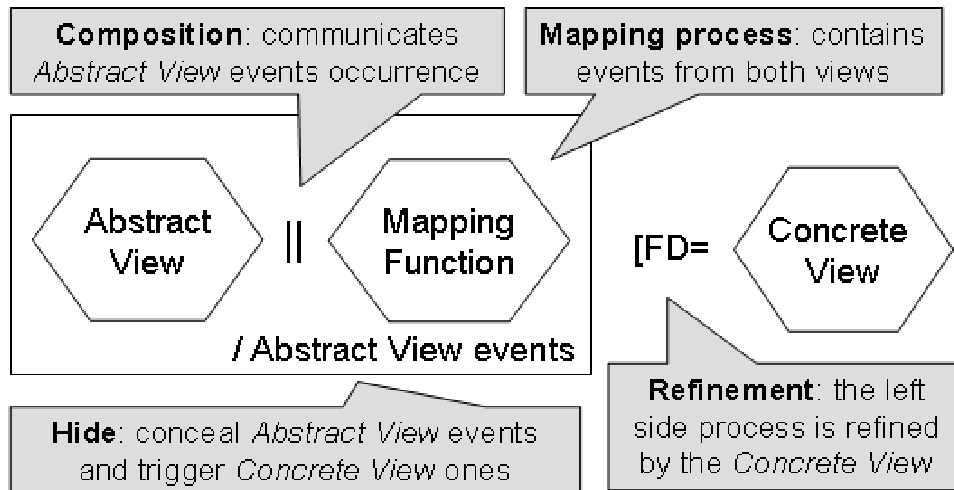

Figure 5.4 Abstract View composition with Mapping Process and events hiding

This presented mapping strategy is based on a framework composition technique [MSM05]. Here we focus on relating events from different models, while the framework composition strategy also aims to accomplish communication between frameworks and the environment.

5.3 Example Refinement

In order to better understand the proposed refinement strategy, it is applied to a model example presented in Figure 5.5. This example contains the `View_1` and the `View_2` processes, which are seen as two models, each one at a different abstraction level. Here, `View_2` is seen as a more concrete view than `View_1`.

The events `a`, `b`, and `c` are used by and constitute the *alphabet* of `View_1`. The channels `a1`, `a2`, `a3`, `b1`, `b2`, and `c1` are the *alphabet* of `View_2`. Both processes, `View_1` and `View_2`, use the *prefix* and the *choice* operators. After engaging in event `a`, `View_1` offers `b`

```
channel a, b, c
events_view_1 = { a, b, c}
View_1 = a -> (  b -> View_1
              [] c -> View_1)
```

```
channel a1, a2, a3, b1, b2, c1
View_2 = a1 -> a2 -> a3 ->
              (  b1 -> b2 -> View_2
              [] c1 -> View_2)
```

Figure 5.5 CSP process examples

and c to the environment, then after engaging on b or c it recurses. Similarly, after events a1, a2, and a3 are committed, View_2 offers b1 and c1. After engaging on b1 and b2, or c1, it recurses.

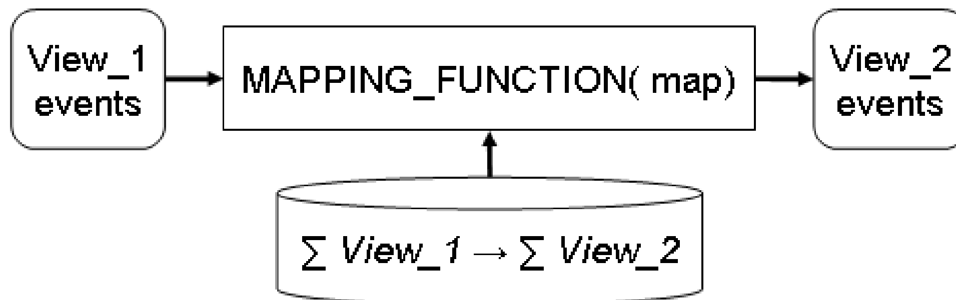

Figure 5.6 Mapping function application to the example from Figure 5.5

As explained, the processes View_1 and View_2 from Figure 5.5 are a simple example of an abstract and a concrete model. It aims to illustrate the proposed strategy before showing the user and the component view mapping. View_1 alphabet, which contains more abstract events, are mapped to View_2 events so the strategy can be used to replace abstract events from View_1 with more concrete ones using the MAPPING_FUNCTION. Observe in Figure 5.6 that MAPPING_FUNCTION uses the mapping between the views as a dictionary in order to translate View_1 events to View_2 ones.

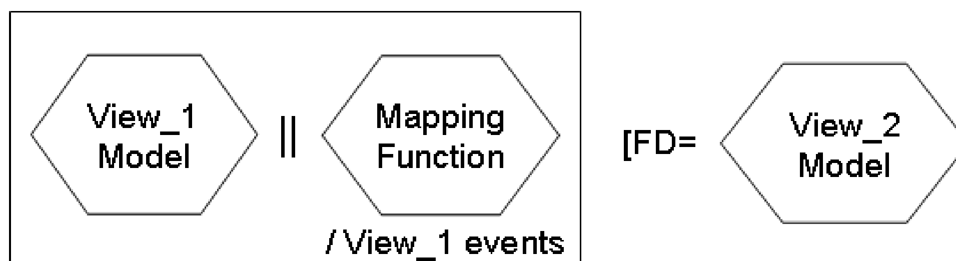

Figure 5.7 View_1 composition with the mapping process to accomplish refinement

Figure 5.7 presents the graphical representation of the CSP code in Figure 5.8. View_1 is composed with MAPPING_FUNCTION generating View_1_with_mapping. This process has the events from View_1 hidden, resulting View_1_mapped that must be refined by the View_2.

```
map = {(<a>,<a1,a2,a3>),(<b>,<b1,b2>),(<c>,<c1>)}

View_1_with_mapping = View_1 [| events_view_1 |] MAPPING_FUNCTION(map)
View_1_mapped = View_1_with_mapping \ events_view_1

View_1_mapped [FD= View_2
```

Figure 5.8 Illustrative usage example of the mapping function

5.4 Component View as a Refinement of the User View

The idea presented in the previous section can be used to relate user and component view models. In this case the component view model must refine the user view through events mapping. The `User View` events are mapped to `Component View` events so abstract events from the user view are replaced with more concrete ones using the `MAPPING_FUNCTION`.

Figure 5.9 presents part of the `map` between the user and the component views generated in Section 4. For example, step `1M` from the user view is mapped to the steps `1M` and `2M` from the component view, and step `2M` is mapped to steps `3M`, `4M`, and `5M`. Notice that the *control events* (See Section 4.2) are used in the mapping definition. Basically, each step of the user view is mapped to one or more steps of the component view.

```
map = < ( < steps, read.DTREA_SENDABLEITEM.(INCOMING_MESSAGE, {}),
expectedResults,display.DTDIS_FIELDVALUE.(MESSAGE_CONTENT_FIELD_VALUE,{})>,

< readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE, {}),
openComp.MESSAGE_APP.MESSAGE_VIEWER.DTOPE_SENDABLE_ITEM.
        (INCOMING_MESSAGE, {}) > ) ,

( < steps,open.DTOPE_MENU.(CSM_MENU_LIST, {}),
conditions,isstate.DTISL_LIST.(IMPORTANT_MESSAGES_FOLDER, {}),
expectedResults,isstate.DTISS_MENUITEM_STATEVALUE.
                       (MOVE_TO_IMPORTANT_MESSAGES_OPTION, {}).
                       (DISPLAYED_VALUE, {}) > ,

< openComp.USER.MESSAGE_APP.DTOPE_MENU.(CSM_MENU_LIST, {}),
displayComp.MESSAGE_APP.MENU_CONTROLLER.DTDIS_MENU.(CSM_MENU_LIST, {}),
isstateComp.MESSAGE_APP.MENU_CONTROLLER.DTISS_FEATURE.(VALUE,{ON}),
isstateComp.MENU_CONTROLLER.USER.DTISS_MENUITEM_STATEVALUE.
           (MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}).(DISPLAYED_VALUE,{}) >
), ...
>
```

Figure 5.9 Mapping between abstract and concrete views

Using the `map` definition from Figure 5.9 it is created a `MAPPING_FUNCTION` instance that is composed with `User_View` producing `User_View_with_mapping`. The user view events are hidden from `User_View_with_mapping`. The resulting process should

be refined by `Component_View` (Figure 5.10).

```
User_View_with_mapping = User_View [| events_user_view |]
                         MAPPING_FUNCTION( map)

User_View_mapped = User_View_with_mapping \ events_user_view

User_View_mapped [FD= Component_View
```

Figure 5.10 Mapping process specification based on the map

5.5 Some Considerations

Notice that the user view model is not constructed using any CSP *parallel composition* operations; it is defined based on CSP *external choices* between use cases specification. However, the component view contains a more complex structure, such as the parallel composition between component processes. Nevertheless, it is possible to map a sequence of events from the user view to the component view, once each user interaction with the system is decomposed into a succession of components communication.

The abstract view composition with the mapping process intends to trigger concrete view events in the composed model; however this behavior is only possible if the abstract view communicates only abstract view events to the mapping process. This is assured once the alphabet from the views are disjoint, guarantying that the synchronization set between abstract and mapping process contains only events from the abstract view.

Analyzing further possible refinements, notice that the equivalence relation between these models, using the proposed approach, can be achieved if an inverse mapping is defined from the component to the user view. In other words, if it is defined a map (dictionary) that translate concrete events to more abstract ones, it would be possible to accomplish the inverse refinement. Therefore, the equivalence between the models would be achieved.

5.6 Tool Support

The definition of the mapping between the user and the component views needs to be manually defined since it is necessary to interpret sequence of events in both views and relate these events. Once this map is defined, it is possible to automatically verify the proposed refinement.

In other to relate the generated models, the use cases from both views, as Word 2003 documents, are read and their use models are generated as defined in Section 4.5. Next, the user view model is composed with the mapping process and the user view events are hidden (see Section 5.4). These are the necessary steps to verify the refinement between user and component views. Hence, the refinement relation can be mechanically checked using FDR [Ros95], a refinement checker for CSP. After loading the two models and the mapping functions, along with the map, the only remaining task is to define *assertions*, such as in Figure 5.11, to check

system properties. The first assert is related to the illustrative example from Figure 5.8 and the second is related to the user and component view refinement from Figure 5.10. The results established that both refinements hold, as expected.

```
assert View_1_mapped [FD= View_2
assert User_View_mapped [FD= Component_View
```

Figure 5.11 Assertions verified by FDR tool

Besides refinement checking, the FDR tool (Figure 5.12) can verify if a model is deadlock, livelock or non-determinism free. Moreover, CSP operators bring the possibility to accomplish quite complex compositions and verify elaborate (possibly domain-specific) system properties.

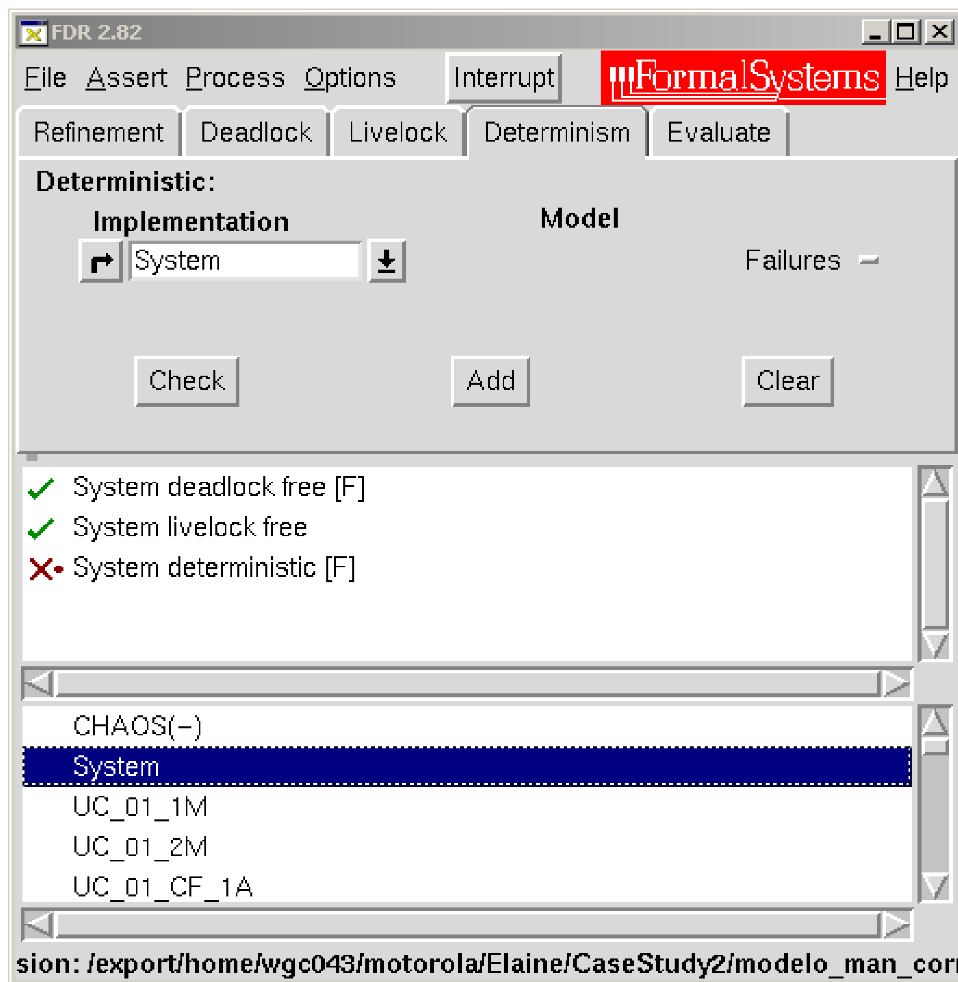

Figure 5.12 Properties verification with FDR

CHAPTER 6

Some Experiments in the Motorola Context

The application of any scientific method in a real project environment is fundamental to verify the strategy's effectiveness and possible benefits. Moreover, the introduction of new techniques in an operational software process, such as that of Motorola, needs to be analyzed in order to estimate the potential impact, involving costs, such as time and risks, and viability in small scope, before the solution can be widely employed.

Hence, to validate the presented strategy and the related implemented tools, three experiments have been accomplished. These experiments involved the adoption of the proposed use case templates and CNL, and its translation to a CSP formal model. This procedure served to evaluate the use case template structure and the generated use model. Further analyses supplied enough information to improve the templates structure, the CNL knowledge bases, and the formal model generation algorithm performance.

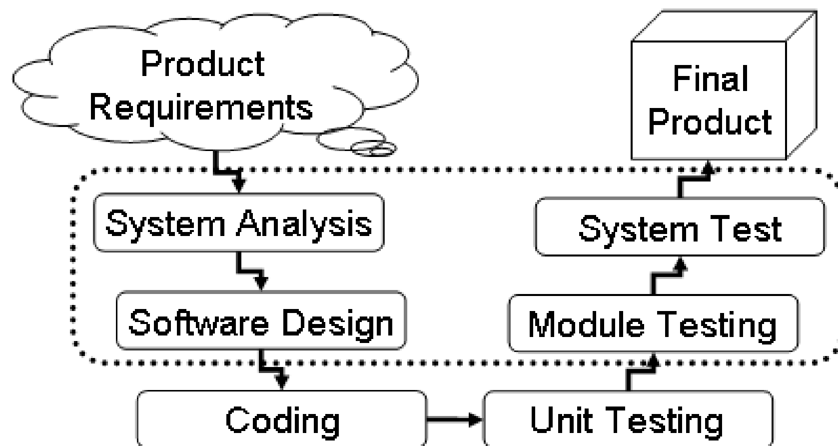

Figure 6.1 Development process and affected phases

Figure 6.1 shows the well-known software development **V-model** [SR00], which relates specification and testing activities. The activities from the **V-model** structure occurs pairwise, simultaneously, during the development cycle. The tasks arrangement strategy aims to engage both specification and test teams to cooperate. Thus, for each design artifact produced, yielded after each phase, it is produced a related validation artifact.

Because the strategy proposed in this dissertation defines two use case templates, user and component view (see Section 3), these new standards are introduced in the `System Analysis` and `Software Design` phases. Beyond the new template usage implications, team members need to be familiar with the CNL syntax. Thus, this chapter shows details

about experiments execution and results, and an overall analysis about the proposed solution regarding its impact in a real project.

6.1 Methodology

Figure 6.2 shows inputs and outputs produced by the two first accomplished experiments. The first experiment was accomplished without any tool support. The use case templates were used to specify the system behavior during the experiment, and the related CSP formal model was manually generated from it. This approach intended to verify the use case templates flexibility concerning specification of a large number of different requirements. The CNL power was also evaluated during the sentences validation stage.

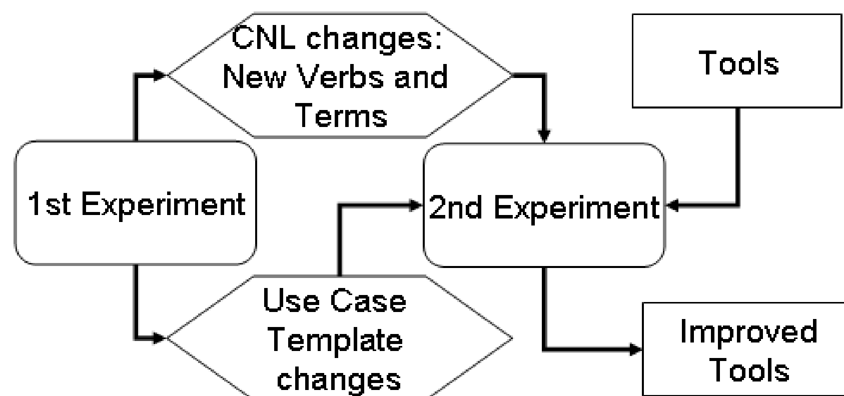

Figure 6.2 First and second experiments

After this first experiment was finished, it was possible to define a list of necessary changes in the proposed strategy. These changes were related to problems in the definition of use case flows relationship, in the use case template and to the correct generation of the CSP models. It was also found some improvement opportunities that would make the use case template more didactic and easy to use.

The second experiment was larger than the first one; tools were used to assist in the use case creation and translation to CSP models. Moreover, it was possible to refine even more the use case specification template once the requirements specified in this second experiment had certain characteristics not yet considered, such as non-functional requirements. This type of requirements is not directly captured by the proposed templates.

Next, the experiment results were analyzed and metrics [Kru00] such as training and use case creation time were collected. This information helps to identify how the proposed solution may impact Motorola's development process.

The third was executed to verify how test case generation tools [dCN06, Car06], also produced in the CIn/BTC research project, behaved when processing the generated CSP formal model. Thus, the model was used as input to these tools to generate a set of test cases that cover the system requirements.

6.2 Context

These three experiments were accomplished in the CIn/BTC research project by three different teams, formed of 5 people each. All involved personnel had basic knowledge about system specification and attended a 2 hour training, which included:

- Understanding Motorola's requirement document templates
- Use case templates usage
- Writing use cases following the CNL grammar
- MS Word 2003 usage with XML schema support
- CNLValidator tool usage [1]
- CSPGenerator tool usage [2]

The participation of people with Motorola process knowledge during these experiments helped to validate the use case templates and CNL according to Motorola specification patterns, since CNL contains domain specific terms. Their experience with Motorola's environment was fundamental to guarantee a template that would satisfy both requirements: the formal specification generation strategy and Motorola's quality assurance (QA) standards.

The existence of tools that accomplish requirements and use case specification management were known by Motorola team. An example of such a tool is DOORS [NE00], which defines standards in order to structure requirement data and store them in a controlled environment (database). However, a tool to analyze requirement document for inconsistencies and grammar correctness had not been used until then.

6.2.1 Mobile Application Specification

The proposed experiment focuses on mobile phone applications, based on Motorola phone requirements. However, the proposed strategy pattern is general enough to specify other types of systems, such as desktop or web applications.

Generally speaking, mobile phone applications contain menus, icons and dialogs boxes just like desktop applications. Their format and size (visual perspective), may differ but the idea is similar. The definition of the application navigation paths is very common in mobile applications, since these applications contain virtually one new screen to each user action that occurs.

Figure 6.3 contains screenshots from Motorola phone applications. Notice that the top of the screen contains some icons that provide system information such as battery status. Each screen also has a title at the top, and at the bottom there are possible operations the user can accomplish. Usually the elements inside the menu or dialog are items that can be selected or fields that can have their values edited. There may exist unusual screens, which hold a more specialized design, for example, the camera application on the right bottom in Figure 6.3. This type of screen holds different buttons or elements; however the CNL sentences can describe them without distinction.

[1] Further information about this tool usage can be found in Section 3.6
[2] Further information about this tool usage can be found in Section 4.5

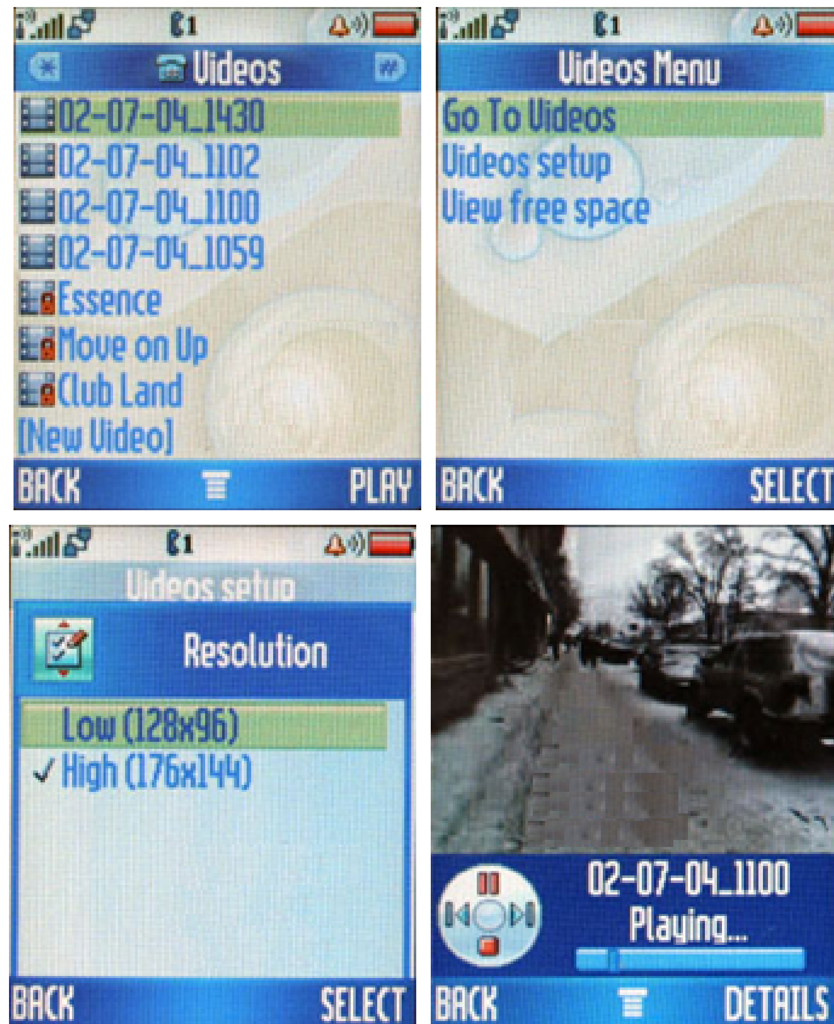

Figure 6.3 Motorola applications screenshots

Whenever certain software functionality (feature) is implemented for more than one product, in a product line for instance, it is common to group these functionalities in product families. Each family of products has a certain set of features implemented. The number of features presented in the product usually defines its price. Another managerial benefit of feature definitions is the possibility to reuse their specification whenever it is created a new product that is a variation of a previous designed one.

The existence of interruptions is an additional aspect that needs to be treated in mobile applications. The existence of multiple applications running in parallel is a common reality. A phone call is an example of a popular interruption event. A User may be listening to music or taking pictures with their cell phone and an incoming call may emerge. Despite its importance, interruption is not been considered here, and it is a possible future work opportunity (see Section 7.2).

6.3 Use Case Specification process

As presented in Figure 6.4, the generation of the system formal model is preceded by several other activities. The process starts with the analysis of existing early requirement documents. These documents are usually written in an informal way, determining what the system should do, but not how. These requirements are not written and structured in a fixed way; they define the client needs in a marketing fashion.

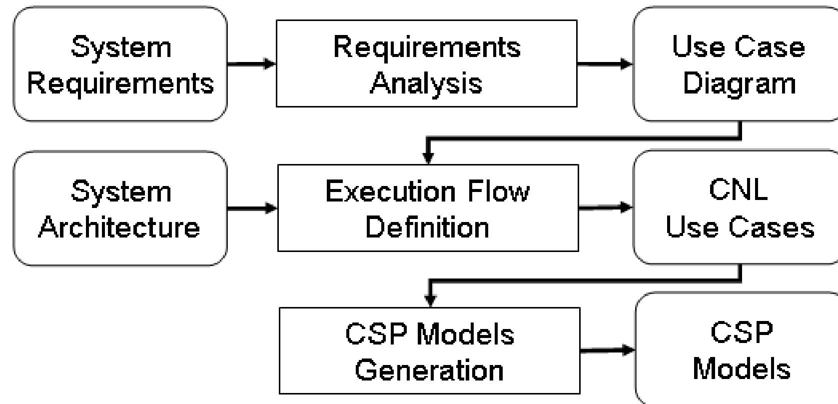

Figure 6.4 Solution steps

After these high level abstraction requirements are analyzed, use cases are defined as UML use case diagram (see Section 2.1.2). This type of diagrams gives a general idea about use cases relation and complexity. In order to achieve total requirements coverage, use cases are defined after requirements are grouped by similarity.

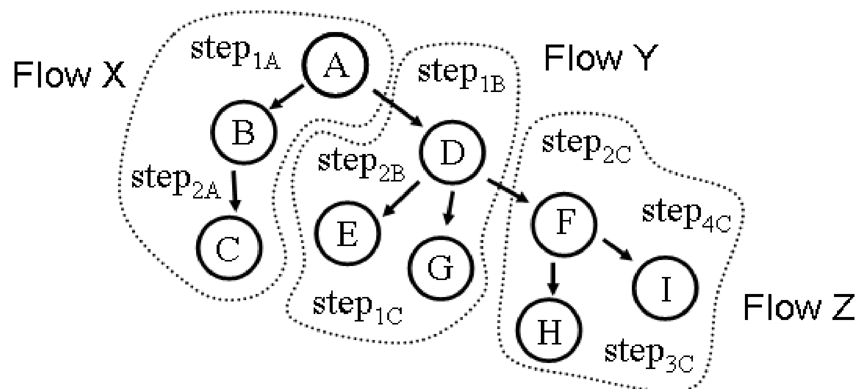

Figure 6.5 Execution flow definition after requirements analysis

Once the use case diagram is modeled, a *graph-formatted* sketch (Figure 6.5) is created for each use case in order to determine execution flows: paths the user can cover through the system execution. These flows are then classified as main, alternative and exception flows. In the Figure 6.5 example, once composed, `Flows X`, `Y` and `Z` form a use case. This composition is accomplished using execution flows information (*from steps* and *to step*).

The graph-formatted sketch creation can be assisted through a graph modeling tool. In the accomplished experiments, it was used *yED* tool [yB06] that allows the creation of graphs with labeled transitions. Figure 6.6 is part of a graph created during one of the experiments. It is actually an LTS [MK99] that assists the use cases creation.

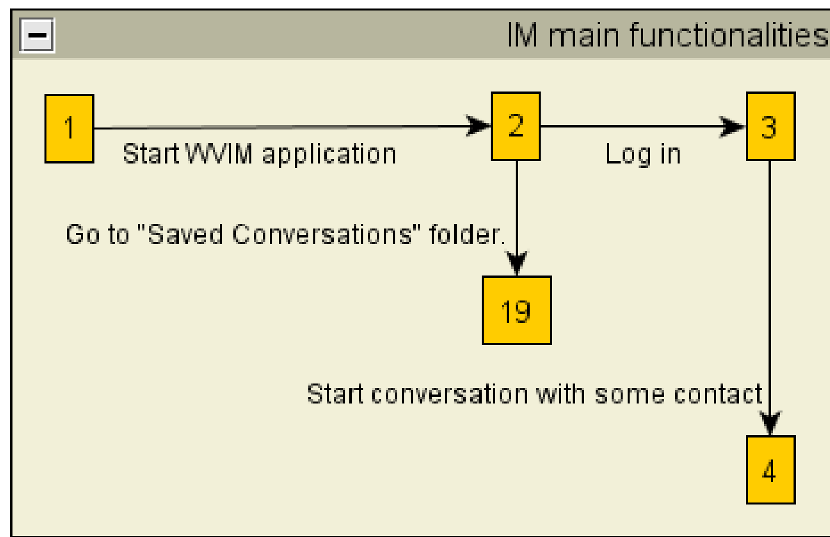

Figure 6.6 Execution Flow graph made from requirements analysis

Based on these sketches, the user view use cases are created first, since they are more abstract than the component view use cases. Both user and component view use cases are described according to the templates presented in Section 3. The component view use cases design requires extra information such as system's architectural information.

As already mentioned, the component view is created based on user view use cases. Each step from the user view is decomposed into one or more steps involving message exchange between system components (see Section 3.3). The component view represents how the user view is implemented, based on the particular system architecture.

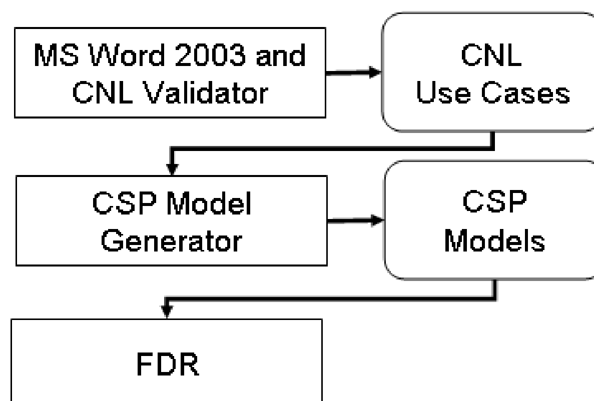

Figure 6.7 Tools involved in the process

6.4 Accomplished Experiments

As already mentioned, three experiments were performed. Besides important feedback concerning the proposed strategy, these experiments produced information such as the time involved to create the proposed use cases. This time is considered in order to evaluate the impact of the strategy adoption in a real project. Thus, metrics (such as training time) have been collected in order to prospect the strategy use by real Motorola projects.

As previously mentioned, these experiments resulted in some changes in the initial method. Figure 6.2 presents the experiments inputs and outputs, which were essential to the validation and improvement of the strategy and implemented tools. Changes such as CNL grammar adaptations and use case structure improvements were the main necessary changes.

6.4.1 Experiment 1

Use Case template adjustments to the Motorola context were the main achieved feedbacks after the first experiment execution. This information was used to validate the strategy and promote the implementation of better tool prototypes.

Activity	Time
Use Case templates training	1 hrs
CNL training	1 hrs
Total	**2** hrs

Table 6.1 Training time

The first experiment execution included an initial training period in order to familiarize the designers with the new use case templates and the CNL grammar. The total training took only 2 hours; designers had previous experience with Motorola's standards and have easily understood the new templates.

This first set of requirements (feature) selected to apply the strategy was a simple one, with straightforward definitions. Table 6.2 shows the time spent during this feature analysis.

Activity	Time
Read and understand Requirements	4 hrs
Define Use Case Diagram	2 hrs
Define Graph-formatted sketch	4 hrs
Total	**10** hrs

Table 6.2 Experiment 1 requirements analysis

6.4.1.1 User View Use Cases Creation

Because the user view use case templates are to be used by Motorola staff, it was defined based on preexistent Motorola use case templates. Hence, this experimental usage of the template by personnel, who knows Motorola's standards, produced an important feedback about the template format. The total time to create 4 use cases and its formal model was 41 hours; it corresponds to around 10 hours per use case.

Activity	**Time**
User view Use Case writing	20 hrs
CNL grammar validation	6 hrs
CSP formal model generation	5 hrs
Total	**31** hrs

Table 6.3 Four user view use cases creation time

6.4.1.2 Component View Use Cases Creation

The creation of the component view use cases required further understanding of the specified system architecture. This task took an extra time once the architectural documentation of the feature was used as reference during the use case specification phase.

The number of new terms to be added in the CNL grammar was only 13. These terms were generally related to entities manipulated by the system components.

Activity	**Time**
Component View Use Case writing	10 hrs
CNL grammar validation	2 hrs
CSP formal model generation	20 hrs
Total	**32** hrs

Table 6.4 Two component view use cases creation time

The manual creation of the component view use model was quite difficult. Because the component view CSP model is the parallel execution of components, it is necessary to define each component process and make sure it synchronizes with other components. This process is error prone and took an extra effort to properly complete. Two use cases were defined together with their corresponding formal models in 32 hours (not considering the feature analysis); this corresponds to 16 hours per use case.

6.4.1.3 Views Refinement

In this experiment, there were 182 sentences in the 4 user view use cases. Only 2 component view use cases were created, which were related to 2 user view use cases. Hence, it was neces-

sary to map 71 sentences. The mapping process took a total of 18 hours, which corresponds to an average of 15.2 minutes per sentence.

6.4.2 Experiment 2

The second experiment took advantage of the implemented tools in order to decrease the time consumption during the strategy application. In addition, there was no training effort, once the team was already trained.

However, the requirement set to be analyzed changed; therefore, it was necessary to study the new feature before writing new use cases. This second experiment involved a very complex feature, and only 5 use cases were created, which correspond to a modest part of the requirements. Table 6.5 shows some metrics about requirements understanding and initial use cases prospecting.

Activity	**Time**
Read and understand Requirements	4 hrs
Define Use Case Diagram	1 hrs
Define Graph-formatted sketch	3 hrs
Total	**8** hrs

Table 6.5 Experiment 2 Requirements Analysis

6.4.2.1 User View Use Cases Creation

The specified feature contained particular characteristics such as a larger variety of possible phone setup configurations and several possible user input values. The combination of different phone setups and user inputs brought a combinatory explosion problem which has not been expected by the strategy. The different combination of values entered by the user and system configuration would bring different system responses. However, some combinations would produce the same system response. Based on calculations, this feature enabled around 32 possible phone setups and 27 different user inputs, which would generate 864 different scenarios. Thus, to write 864 different execution flows, each one specifying a different scenario, is not a viable solution. The solution was to prioritize the creation of use cases, covering the most frequent usage flows. This problem was further investigated and a proposed solution could be the adoption of a new concept (Parametrised Use Case), which is one possible future works suggested in Section 7.2.

Since the CSP formal model generation tool was adopted for the second experiment, the time to generate the formal model is not considered. Unfortunately, there was another problem; the CNL knowledge bases needed to be updated with new terms and modifiers. This task may be very complex if the new feature differs much from features already considered. During this second experiment 45 new terms have been added to the CNL knowledge base. The total use case and formal model creation time was 32 hours, which corresponds to 6.4 hours per use case. The tool usage decreased the time in 36%.

Activity	Time
User view Use Case writing	18 hrs
CNL grammar validation	6 hrs
CSP formal model generation	0 hrs
Total	**24** hrs

Table 6.6 Five user view use cases creation time

6.4.2.2 Component View Use Cases Creation

A simple component view use case was created since the experiment focused on test case generation from the user view. Once the user view use cases were written and the CNL knowledge base was updated, there was a reduced possibility to exist new terms to be added during the component view use cases validation. In this experiment, only 10 new terms had to be added to the CNL knowledge base. Using the tool, it took only 6 hours to create one use case and generate its CSP model, which corresponds to a 62,5% time improvement.

Activity	Time
Component View Use Case writing	4 hrs
CNL grammar validation	2 hrs
CSP formal model generation	0 hrs
Total	**6** hrs

Table 6.7 One component view use cases creation time

6.4.2.3 Views Refinement

The dictionary that translates events from the user to the component view had, once more, to be updated with new mappings. Because almost every sentence from the user view is new it is necessary to define new mapping for each of them.

Currently, this manual mapping procedure is a complex operation and takes a significant amount of time. The user and the component use cases are used as input to this process and it is necessary to go over each user view sentence and verify its translation to the component view. In this experiment, there were 121 sentences in the five user view use cases. Because only one component view use case was created, it was only necessary to translate 26 sentences. The mapping process took a total of 7 hours, 16.5 minutes per sentence.

6.4.3 Experiment 3

The third experiment aimed to analyze test case generation algorithms presented in [dCN06, Car06], using the generated CSP models as input. Just like the previous experiments, it was necessary an initial time to understand the feature functionalities. Table 6.8 shows some metrics about requirements understanding and initial use cases prospecting.

Activity	**Time**
Read and understand Requirements	2.5 hrs
Define Use Case Diagram	1 hrs
Define Graph-formatted sketch	1.5 hrs
Total	**5** hrs

Table 6.8 Requirements Analysis

As already mentioned, this time depends on the feature complexity. Thus, this feature can be seen as a moderated complexity level feature. In this experiment, it was possible to specify 2 use cases, which was enough to cover all the feature requirements.

6.4.3.1 User View Use Cases Creation

In this last experiment it was only necessary to create user view use cases since the main goal was test case generation. Table 6.9 shows the time spent in this last experiment. As in the previous experiment, there was no time involved during the CSP formal model generation; it was tool supported.

Activity	**Time**
User view Use Case writing	5.5 hrs
CNL grammar validation	3 hrs
CSP formal model generation	0 hrs
Total	**8.5** hrs

Table 6.9 Two Use Cases creation time

6.5 Some Considerations

The templates use by project engineer was no problem. Some adaptations may be necessary but the use of use case templates is known by most of the engineers. In the other hand, the use of CNL was the biggest barriers. Initially its use may seem restrictive, causing the sensation that it is not powerful enough to specify any time of systems. However, the use case designer should not worry about following CNL rules. They should write use cases freely and after that only the phrases that were not valid according to the CNL knowledge bases must be rewritten, or its terms can be added to tool's bases.

6.5.1 Use Case Creation

Because the use cases are structured through XML schemas [RM01] it is possible to re-adept the use case template format including new fields, for instance. The formal model genera-

tion algorithm may change if the information necessary to generate the model is removed or redefined, though.

Finally, the MS Word 2003 environment is an easy to use, well-kwon tool that is used for a large number of to enterprises. Nevertheless, its use is not obligatory; the use cases can be specified in a XML format and its appearance can be determined by XSL [Paw02] definitions.

6.5.2 Use Model Generation

The usage of formal models may discourage the use of the proposed solution in a real project environment. That is why its generation is hidden from the final user. The main found problem is related to the CNL grammar completeness. There will always be new terms related to new requirements, which will need to be added to the CNL knowledge bases. Beyond that, it is necessary to define the terms class and its relation to the verbs or Case Frames (see Section 3.4), which are actually system events. These problems and improvement opportunities are listed at Section 7.2.

6.5.3 Models Refinement

The experiment of manually defining the relation between events was not considered a feasible task. There might be hundreds of new sentences each time a new set of use cases is written. Howsoever, once the mapping was defined, the refinement between the user and the component view models was verified using FDR.

Therefore, it is necessary to create means to support the events dictionary definition. The solution to this problem requires further analysis. However, the use of Model Learning techniques [MW02, CVV02] seem to be a possible solution. The translation from user view to component view can be seen as a language translation, in this case from English to English. Here, the translation acts as an explanation from a simple specification to a more complex one. Section 7.2 present some related future works.

CHAPTER 7
Conclusion

The use of formal methods frequently brings apprehension to development teams not familiar with abstract specification notation such as the CSP process algebra. However, the benefits of formally specifying systems, especially concerning critical systems [Kni02], are innumerous [LP05, BH95, CMCP+99, CW96]. Therefore, it seems promising to explore alternative means to enable the use of formal specification in real projects, not just academic ones, hiding formalities as much as possible.

The direct use of formal specifications to verify system properties is well-know by the formal methods community. Besides allowing the validation of system properties, the use of formal models enforces a deeper understanding of the system intended behavior and therefore helps to avoid the introduction of bugs during the system implementation. In other words, formal specifications tend to improve the quality of the overall system; the possibility to define successive refinements of the specification, until code, design aids to minimize the presence of errors.

This dissertation indorses the idea of relating requirements, written in CNL, with CSP formal models. The generation of formal specifications from requirements demands the definition of templates that shall mold requirement information. Hence, this structured information works as a requirement meta-data, making it possible to identify requirement elements and definitions.

This dissertation focuses on generating CSP formal specifications through validation and processing of use cases specifications. The sooner requirements are validated, the lower is the risk involved in the system development; problems can be found and analyzed even before system implementation starts. The use of a CNL and use case templates seem relevant to guarantee requirements consistency and correctness.

The use of use case specifications to define system requirements holds once it is necessary to design system's behavior through sequences of steps. The use of such a strategy provides means to define the user possible actions and the respective system response. A sequence of such pairs is fundamental to determine the system behavior. Moreover, it is also possible to describe cases when the system may behave differently from expected; these cases are modeled as alternative or exception execution flows.

The use of CNL ensures that the use cases are written in a non-ambiguous and uniform language. The simplicity of CNL makes it possible to specify use case steps preserving the syntactic correctness of the text and the flexibility of a natural language. Its adoption improves document quality, once its content is written in a standardized way. This fact ensures the documentation understanding no matter who writes it; the CNL ensures the non-personalization of the way sentences are written.

The generated model is a formal representation of the system, which holds the same system

behavior as specified by the use cases. The use of CSP, as the target formal model notation, is justified since CSP is ideal to specify event occurrences between components and the environment. In this case, the user acts as an external element, which belongs to the environment, and interacts with the system exchanging messages. This paradigm makes it possible to separate the system specification from the user behavior. In other words, the user is modeled as an external element that interacts with the system, just as in use case specification. Moreover, the system can be refined once its internal events are detailed.

Initially the generated formal specification considers the user interaction with the system, which is defined by a use case specification template called user view; this use case is used to generate the user view model. Furthermore, the system can be detailed using a second template called component view. It decomposes the system into components refining the user view use model.

This proposed approach covers both generation of formal models and definition of a refinement relation between the generated models. Furthermore, this generation improves requirements quality and uniformity. To complement the presented strategy, tools were implemented to validate the use cases written in CNL and mechanically generate the CSP formal specification. The verification of specification properties and the refinement between the views can be checked using FDR [Ros95, Gar97].

A case study was executed in the Motorola's environment to validate the presented strategy and refine it according to users' feedback. The initial strategy suffered some adjustments to make it more suitable to Motorola's needs. The final solution was more refined and flexible. It enabled the specification of any type of system that involves user interaction, or even systems defined by component interaction. Finally, the case study has explored the use of the generated model to generate test cases and UML diagrams. Both experiments were successfully executed and generated artifacts that could be used by the test design team and the development team from Motorola.

7.1 Related Work

The use of natural or restricted languages to write requirements is approached by various works (see Sections 2.4.1 and 2.4.2). The processing of natural language variation to generate first-order logic models is more suitable for requirements consistence verification. The usage of the logical notation to specify system behavior seems infeasible; the gap between logical propositions and structured design definitions is wide.

The use of CNL to ensure requirements consistence is presented in varios works [FS95, FST99]. The idea of using a CNL to specify systems seems promising. In addition to that, CNL editors [FSS90, SLH03, Sch05] are a viable solution to enable the use of CNL minimizing any negative impact.

The use of use case specification is brightly used in [Men04], however the use of CNL as input language, along with a CNL editor assistant, would be a better combination. This dissertation defines a structured use case specification templates and a fixed CNL grammar; a CNL editor is a possible future work.

In [SHR$^+$05] is defined an approach to generate CSP models from policies. However, the

definition of policies to specify system is quite confusing. The use of the proposed use case specification template enables a better understanding of the system behavior.

Apart from the fact that we use process algebra as formal model, our strategy goes beyond the translation itself: it generates structured models, possibly at different levels of abstraction, and addresses the formal refinement between them. Furthermore, there are tools that mechanize the entire process: from the use case specifications creation to the refinement checking. These tools are essential to the introduction of formal methods in real projects, as in the Motorola environment.

7.2 Future Work

This section contains a list of suggested improvements to make the use of the proposed approach more practical. The use of the presented strategy in real projects is one of the priorities of this dissertation. In addition, the implemented tools aim to assist the use of the proposed strategy even in small-scale projects, contributing to the approach popularization.

7.2.1 CNL knowledge bases dynamically defined

The generation of an abstract and fairly simple CSP formal model was only the first step in order to define a more refined strategy. The generated CSP channels and datatypes are directly related to the CNL knowledge bases; therefore the definition of new channels and datatypes is restricted by CNL base updates. This process is out of the designer specification scope and can be quiet complex.

Thus, an improvement would be defining a complete strategy to enable the definition of the application's domain terms and modifiers along with the use case specifications. These requirements would describe the application elements and their relations. The CNL knowledge bases would contain only a minimal set of objects and operations from which the specified system elements would inherit properties. Similarly, pre-defined operations, which manipulate these primitive objects, would be defined in the CNL bases.

Basically, this idea tries to represent concrete data structures such as lists, sets, or graphical elements in the formal methods universe. This would enable the designer to specify new elements based on this primitive set. Any new term or operation would be specified through a more robust CNL in order to specify all application objects, similar to the analysis classes definition process [Kru00].

Once all application elements and operations are specified, as requirements of the system, it is possible to define the application use cases as is proposed by this dissertation. Then it would not be necessary to worry if the CNL contains the terms used in the use case sentences. The application documentation itself would contain these definitions.

Because large systems are divided into sets of features, it would be possible to reuse objects and terms in new feature requirements and use case specifications. This strategy works as a library importing processes, making previous definitions visible.

Finally, use case steps would manipulate application objects through the same predefined operations, or through new ones defined as requirements. These operations would change the

system state, which would be kept in this new proposed strategy. Here, the generated model, in CSP for instance, would have parameters that represent the specification state. Therefore, this specification would hold information about the application data and each operation, step, of the use case would change the system state.

7.2.2 Dynamic Use Case Specification

Besides making the CNL more extensible, this new approach brings the generated model to a more complex level of detail. It can be seen as a refinement of the current defined approach. The investment of specifying requirements that can be mapped to a formal specification, which contains state and new datatype definitions, is supported by the possibility of generating more complex and refined test cases and UML artifacts (even part of the application code).

The definition of datatypes at the requirement level determines new ways to define test purposes. Existing approaches define test case generation strategies based on part of the model the test should pass through, defining the set of requirements it aims to cover, or by selecting the most critical use cases, for instance. The existence of data and state in the generated model enable the test planner to define the critical set of system inputs and thus generate test cases that shall verify the system behavior on these predefined controlled conditions. It is a clear possibility to generate *test points* based on formal methods.

7.2.3 Automatic Test Execution from Generated Test Cases

The presented refinement strategy can also allow automating test cases execution. The idea is to hierarchically refine each use case sentence using other sentences. If each use case step, written as CNL a sentence, which describes an abstract user action or system response, it should be refined through other sentences, detailing the step definition. The sentence is then decomposed into other sentences until each one can be mapped to a predefined statement. This routine should be executed until every sentence is directly or indirectly associated to a script statement from an automation framework [Liu00].

It is necessary to define a set of script statements of possible primitive operations that would cover possible ways a user can interact with the system. This set of operations would depend on the application nature. However, once this set is defined, any other user operation would be implemented by the composition of these primitive ones. Whenever a sentence is decomposed into other sentences, if any of the decomposed sentences already contains a valid translation to the script language, only the remaining sentences would be decomposed or mapped to primitive operations.

Once the sentences from use case steps are translated to the automation framework commands, directly or indirectly (through sentences decomposition) the generated test cases from the generated model would have its steps already mapped to primitive operations, therefore, defining an automatic test scripts.

7.2.4 Formal Model Animation to Validate System Implementation

Besides automatic execution of test cases, generated from formal models, there are means to animate the formal model [FC06] and trigger the execution of commands that shall execute operations at the real application. These operations would result in system responses, which can be verified using the system response definition from the user view model. In order words, the formal specification would be executed through an animator and the real application would concurrently receive concrete stimuli from the environment.

Bibliography

[Ali94] S. S. Ali. A logical language for natural language processing. In *In Proceedings of the 10th Biennial Canadian Artificial Intelligence Conference*, pages 187–196, Banff, Alberta, Canada, 1994. IEEE Computer Society.

[Apt96] Krzysztof R. Apt. *From logic programming to Prolog*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[AS02] Bente Anda and Dag I. K. Sjøberg. Towards an inspection technique for use case models. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 127–134, New York, NY, USA, 2002. ACM Press.

[AY99] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 114–129, London, UK, 1999. Springer-Verlag.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[BE98] Barry Boehm and Alexander Egyed. Software requirements negotiation: some lessons learned. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 503–506, Washington, DC, USA, 1998. IEEE Computer Society.

[BGG$^+$04] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An agent-oriented software development methodology, 2004.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Softw.*, 12(4):34–41, 1995.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[Bor99] F. Bordeleau. *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines*. PhD thesis, Carleton University - Ottawa,Canada, aug 1999.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[Buh98] R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Trans. Softw. Eng.*, 24(12):1131–1155, 1998.

[Car06] Emanuela Cartaxo. Test case generation by means of UML sequence diagrams and label transition system for mobile phone applications. Master's thesis, Universidade Federal de Campina Grande (UFCG), aug 2006.

[CIn06] CIn/UFPE. www.cin.ufpe.br, 2006.

[CMCP+99] Emanuele Ciapessoni, Piergiorgio Mirandola, Alberto Coen-Porisini, Dino Mandrioli, and Angelo Morzenti. From formal models to formally based methods: an industrial experience. *ACM Trans. Softw. Eng. Methodol.*, 8(1):79–113, 1999.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CVV02] Francisco Casacuberta, Enrique Vidal, and Juan Miguel Vilar. Architectures for speech-to-speech translation using finite-state models. In *Proceedings of the ACL-02 workshop on Speech-to-speech translation: algorithms and systems*, pages 39–44, Morristown, NJ, USA, 2002. Association for Computational Linguistics.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[dCN06] Sidney de Carvalho Nogueira. Geração automática de casos de teste csp dirigida por propósitos. Master's thesis, Universidade Federal de Campina Grande (UFPE), aug 2006.

[DJK+99] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering*, pages 285–294, 1999.

[DP02] Brian Dobing and Jeffrey Parsons. The role of use cases in the uml: a review and research agenda. In *Advanced topics in database research vol. 1*, pages 367–382, Hershey, PA, USA, 2002. Idea Group Publishing.

[DP06] Brian Dobing and Jeffrey Parsons. How uml is used. *Commun. ACM*, 49(5):109–113, 2006.

[EFM99] A.G. Engels, L.M.G. Feijs, and S. Mauw. MSC and data: Dynamic variables. In R. Dsoulli, G. von Bochmann, and Y. Lahav, editors, *SDL'99: The Next Millennium, Proceedings of the 9th SDL Forum*, pages 105–120, Montreal, Canada, jun 1999. Elsevier Science Publishers.

[FC06] Angela Freitas and Ana Cavalcanti. Automatic translation from *ircus* to java. In *FM*, pages 115–130, 2006.

[fdr97] Fdr user's manual version 2.28, 1997.

[FGR+94] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Form. Methods Syst. Des.*, 4(3):243–263, 1994.

[Fil76] C.J. Fillmore. Frame semantics and the nature of language. *In Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech*, 280, 1976.

[FLM+04] Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Pistore, Marco Roveri, and Paolo Traverso. Specifying and analyzing early requirements in tropos. *Requir. Eng.*, 9(2):132–150, 2004.

[Fow98] Martin Fowler. Use and abuse cases, 1998.

[FS95] Norbert E. Fuchs and Rolf Schwitter. Specifying logic programs in controlled natural language. Technical report, University of Zurich, 1995.

[FSS90] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English - not just another logic specification language. In *LOPSTR '98: Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation*, pages 1–20, London, UK, 1990. Springer-Verlag.

[FST99] Norbert E. Fuchs, Uta Schwertel, and Sunna Torge. Controlled natural language can replace first-order logic. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 295, Washington, DC, USA, 1999. IEEE Computer Society.

[Gal96] John Galletly. *Occam-2*. University College London Press, 1996.

[Gar97] P. Gardiner. *Failures-Divergence Refinement, FDR2 User Manual and Tutorial*. Formal Systems Ltd., 1997.

[GN00] Vincenzo Gervasi and Bashar Nuseibeh. Lightweight validation of natural language requirements: A case study. In *ICRE '00: Proceedings of the 4th International Conference on Requirements Engineering (ICRE'00)*, page 140, Washington, DC, USA, 2000. IEEE Computer Society.

[GN02] Vincenzo Gervasi and Bashar Nuseibeh. Lightweight validation of natural language requirements. *Softw. Pract. Exper.*, 32(2):113–133, 2002.

[Gra97] Mark Grand. *Java language reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[GZ05] Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Trans. Softw. Eng. Methodol.*, 14(3):277–330, 2005.

[HT03] David Harel and P. S. Thiagarajan. Message sequence charts. In *UML for real: design of embedded real-time systems*, pages 77–105, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[HYfCl05] He Hai, Zhong Yi-fang, and Cai Chi-lan. Unified modeling of complex real-time control systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 498–499, Washington, DC, USA, 2005. IEEE Computer Society.

[Inc06] Motorola Inc. www.motorola.com.br, 2006.

[Jon90] C. B. Jones. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1990.

[JYS02] Brian Johnson, Marc Young, and Craig Skibo. *Inside Microsoft Visual Studio .NET*. Microsoft Press, Redmond, WA, USA, 2002.

[Kni02] John C. Knight. Safety critical systems: challenges and directions. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 547–550, New York, NY, USA, 2002. ACM Press.

[Kru00] Philippe Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lei06] Daniel Almeida Leitão. *NLFORSPEC: Uma ferramenta para geração de Especificações Formais a partir de Casos de Teste em Linguagem Natural*. PhD thesis, Universidade Federal de Pernambuco, Aug 2006.

[Liu00] Chang Liu. Platform-independent and tool-neutral test descriptions for automated software testing. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 713–715, New York, NY, USA, 2000. ACM Press.

[LLM04] Simon St. Laurent, Evan Lenz, and Mary McRae. *Office 2003 XML: Integrating Office with the rest of the world*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[LP05] Zarrin Langari and Anne Banks Pidduck. Quality, cleanroom and formal methods. In *3-WoSQ: Proceedings of the third workshop on Software quality*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[Men04] Vladimir Mencl. Deriving behavior specifications from textual use cases. In *Workshop on Intelligent Technologies for Software Engineering (WITSE04, Sep 21, 2004, part of ASE 2004)*, pages 331–341, Linz, Austria, 2004.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency : state models & Java programs*. Wiley, Chichester [u.a.], 1999.

[MSM05] Walter Mesquita, Augusto Sampaio, and Ana Melo. A strategy for the formal composition of frameworks. In *SEFM 2005, Third IEEE International Conference on Software Engineering and Formal Methods, 7-9 September 2005, Koblenz, Germany*, pages 404–413, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[Mun06] Patrícia Muniz. Viewing CSP specifications with UML-RT diagrams. Master's thesis, Universidade Federal de Campina Grande (UFPE), aug 2006.

[MW02] Daniel Marcu and William Wong. A phrase-based, joint probability model for statistical machine translation. In *EMNLP '02: Proceedings of the ACL-02 conference on Empirical methods in natural language processing*, pages 133–139, Morristown, NJ, USA, 2002. Association for Computational Linguistics.

[NE00] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[Oli05] M. V. M. Oliveira. *A Refinement Calculus for Circus*. PhD thesis, Department of Computer Science, The University of York, UK, apr 2005.

[Paw02] Dave Pawson. *XSL-FO: Making XML Look Good in Print*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[P.H02] P.H.Welch. Process Oriented Design for Java: Concurrency for All. In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 687–687. Springer-Verlag, April 2002. Keynote Tutorial.

[PHS06] Joaquin Pena, Michael G. Hinchey, and Roy Sterritt. Towards modeling, specifying and deploying policies in autonomous and autonomic systems using an aose methodology. *ease*, 0:37–46, 2006.

[PM03] Frantisek Plasil and Vladimir Mencl. Getting ”whole picture” behavior in a use case model. In *Proceedings of IDPT 2003*, Austin, TX, USA, Dec 2003. SDPS, Society for Design and Process Science, Grandview, Texas, ISSN 1090-9389.

[Pre00] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2000.

[pro98] Probe users manual version 1.25, 1998.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[Qua98] Terry Quatrani. *Visual modeling with Rational Rose and UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[RA98] Colette Rolland and Camille Ben Achour. Guiding the construction of textual use case specifications. *Data Knowl. Eng.*, 25(1-2):125–160, 1998.

[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.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[RM01] Erik T. Ray and Christopher R. Maden. *Learning XML*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[Ros95] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *CSFW '95: Proceedings of the The Eighth IEEE Computer Security Foundations Workshop (CSFW '95)*, page 98, Washington, DC, USA, 1995. IEEE Computer Society.

[RP92] C. Rolland and C. Proix. A natural language approach for requirements engineering. In P. Loucopoulos, editor, *Proceedings of the Fourth International Conference CAiSE'92 on Advanced Information Systems Engineering*, volume 593, pages 257–277, Manchester, United Kingdom, 1992. Springer-Verlag.

[SC02] Victor F. A. Santander and Jaelson Castro. Deriving use cases from organizational modeling. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 32–42, Washington, DC, USA, 2002. IEEE Computer Society.

[Sch05] Rolf Schwitter. http://www.ics.mq.edu.au/ rolfs/peng/, 2005.

[Sel04] Bran Selic. Tutorial: An overview of UML 2.0. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 741–742, Washington, DC, USA, 2004. IEEE Computer Society.

[SFGP05] Bernhard Schätz, Andreas Fleischmann, Eva Geisberger, and Markus Pister. Model-based requirements engineering with autoraid. In *GI Jahrestagung (2)*, pages 511–515, 2005.

[SHR$^+$05] Roy Sterritt, Michael G. Hinchey, James L. Rash, Walt Truszkowski, Christopher Rouff, and Denis Gracanin. Towards formal specification and generation of autonomic policies. In *EUC Workshops*, pages 1245–1254, 2005.

[SLH03] R. Schwitter, A. Ljungberg, and D. Hood. ECOLE - a look-ahead editor for a controlled language, in: Controlled translation. *Joint Conference combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop,Proceedings of EAMT-CLAW03, May 15-17, Dublin City University, Ireland*, 2003.

[SMSB05] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. Xfm: An incremental methodology for developing formal models. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4):589–609, 2005.

[Som01] Ian Sommerville. *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Spi92] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[SR00] Viktor Schuppan and Winfried Rußwurm. A cmm-based evaluation of the v-model 97. In *EWSPT '00: Proceedings of the 7th European Workshop on Software Process Technology*, pages 69–83, London, UK, 2000. Springer-Verlag.

[SS97] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A good practice guide*. John Wiley and Sons Ltd., 1997.

[WAF02] Peter H. Welch, Jo R. Aldous, and Jon Foster. Csp networking for java (jcsp.net). In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 695–708, London, UK, 2002. Springer-Verlag.

[Web83] Bonnie Lynn Webber. So what can we talk about now? In Michael Brady and Robert C. Berwick, editors, *Computational Models of Discourse*, pages 331–371. The MIT Press, Cambridge, MA, 1983. Reprinted in Grosz et al. (1986).

[WK99] Jos Warmer and Anneke Kleppe. *The object constraint language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[yB06] by yWorks yFiles Brochure. www.yworks.com/products/yfiles/doc/yfiles_e.pdf, 2006.