Federal University of Pernambuco
Informatics Center

Post Graduation in Computer Science

Modelling and Integrating Formal Models: from Test Cases and Requirements Models

Clélio Feitosa de Souza

MASTER THESIS

Recife
April, 26 2007

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CLÉLIO FEITOSA DE SOUZA

Modelling and Integrating Formal Models: from Test Cases and Requirements Models

Dissertação apresentada ao Curso de Mestrado em Ciência da Computação como requisito parcial à obtenção do grau de Mestre em Ciência da Computação

ORIENTADOR: Prof. Alexandre Cabral Mota

RECIFE, ABRIL/2007

Souza, Clélio Feitosa de
Modelling and Integrating Formal Models: from Test Cases and Requirements Models / Clélio Feitosa de Souza. – Recife: O autor, 2007.
xi, 106 folhas: il., fig.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIN. Ciência da Computação, 2007.

Inclui bibliografia.

1. Métodos. 2. Modelos Formais. I.Título.

005.14 CDD (22.ed.) MEI2007-065

Acknowledgements

First, I would like to thank God for being the source of all accomplishments in my life.

I would also like to thank my family for all support and love. A special thanks goes to Josirene, my fiancee and future wife, for all love, patient and support in all moments of my life.

This work has been developed in the context of a research cooperation between Motorola Inc. and CIn-UFPE. Thus, thanks to the entire group for all the support, criticisms and suggestions throughout the development of this research.

Thanks to Gustavo *"San"*, Euclides, Cristiano *"Galego"* and Rafael Marques for friendship and support.

Thanks to Prof. Paulo Borba and Prof. Augusto Sampaio for great suggestions and ideas.

Finally, a special thanks goes to Prof. Alexandre Mota for his contributions in the whole work.

Resumo

A especificação formal de um sistema ou seu modelo formal é uma forma abstrata de representar suas propriedades (características). Métodos formais é um ramo da Engenharia de Software com foco no desenvolvimento de sistemas tendo uma especificação formal do mesmo como ponto de partida. Inicialmente, as vantagens de usar notações abstratas antes da implementação do sistema estavam apenas relacionadas a um melhor entendimento do problema. Depois, tornou-se evidente que o uso de notações formais abstratas combinadas com técnicas de refinamentos de modelos reduzem o tempo de desenvolvimento e aumentam a qualidade do produto de sistema.

A fase de testes é positivamente influenciada pelo uso de métodos formais. Pesquisas têm sido desenvolvidas para melhorar a qualidade do sistema usando modelos formais e casos de teste. Uma vez verificadas as propriedades do sistema através de uma investigação dos modelos formais, é possível gerar casos de testes confiáveis do sistema que serão colocados em ação para verificar a implementação do sistema posteriormente. O campo de pesquisa que explora métodos formais aplicados com testes de software é chamada de Testes Baseados em Modelos, ou simplesmente *MBT*, do inglês *Model-Based Testing*.

Porém, há situações onde não é possível possuir o modelo abstrato definido a priori. Para superar tal restrição outras técnicas surgiram para sintetizar um modelo abstrato seguindo apenas execuções do sistema. As execuções de um sistema contêm comportamentos necessários para construir um modelo abstrato desse sistema. Na literatura atual, tais técnicas usadas para construir representações abstratas seguindo execuções do sistema são chamadas de *Anti-Model-Based Testing* ou simplesmented *Anti-MBT*. Então, depois de construir um modelo abstrato, técnicas de verificação de modelos e geração de casos de teste seguindo modelos formais podem ser aplicadas normalmente.

O propósito desse trabalho é dar suporte a algumas técnicas de *MBT* usadas no contexto da Motorola (CIn/BTC). Em tais técnicas, as especificações usadas para gerar casos de testes são geralmente incompletas, inconsistentes, e às vezes não existem. Portanto, usando casos de testes reais do sistema é possível criar novas especificações e atualizar especificações originais do sistema, e posteriormente gerar novos casos de teste usando comportamentos válidos do sistema. Um outro problema detectado em nosso contexto é a distância existente entre as representações abstratas e reais. Um caso de teste abstrato, por exemplo, é útil em técnicas formais, mas não é possível executar um caso de teste diretamente no sistema. Dessa forma, para executar (manualmente ou automaticamente) os casos de teste gerados pelas técnicas de *MBT* é necessário primeiro traduzi-los em uma representação real.

Como resultado desse trabalho nós desenvolvemos técnicas formais de modelagem do comportamento do sistema usando casos de teste. Os resultados das técnicas de modelagem são

modelos formais especificados nos formalismos de LTS ou CSP. Além disso, nós definimos uma técnica de unificação que une modelos formais gerados a partir de diferentes artefatos do sistema (requisitos e casos de teste). O resultado da técnica de unificação é um completo e unificado modelo do sistema, que contém informações providas de diferentes artefatos. Nós também definimos uma técnica para traduzir casos de teste abstratos em representações reais. Os casos de teste reais gerados por nossa técnica de tradução são usados no contexto do time de automação de testes da Motorola, onde esse trabalho está inserido. Finalmente, nós automatizamos as técnicas desenvolvidas usando linguagens de programação e especificações formais. O resultado é a ferramenta TCRev que é capaz de modelar, unificar e traduzir modelos do sistema.

A ferramenta TCRev interage com o outras ferramentas externas, tais como *FDR* e *FDR Explorer*. Todos os resultados foram validados em estudos de casos reais executados no contexto da Motorola. Nessa dissertação nós apresentamos um destes estudo de casos.

Palavras-chave: métodos e modelos formais, teste baseado em modelos (*MBT*), *Anti-MBT*, CSP, LTS, modelagem, unificação e tradução de modelos

Abstract

A formal specification of a system or its system formal model is an abstract way to capture its properties (characteristics). Formal Methods is an area of Software Engineering which aims to develop systems using formal specifications as basis. Initially, the benefits of using abstract notations, before starting the system implementation, were only related to a better understanding of the problem. Later, it has become evident that the use of abstract formal representations combined with refinement model approaches reduces development time as well as increases system quality.

The testing phase is positively affected by the use of formal models. Researches have been developed to improve the system quality by using formal models and test cases. Once we have verified the system properties investigating its formal models, we can generate sound system test cases, which will be used to check the system implementation later. The research field which explores the use of formal approaches with software testing is called Model-Based Testing, or simply MBT.

Nevertheless, there are situations which it is not possible to have the abstract model defined a-priori. To overcome such a challenge other approaches came out to synthesise an abstract model following system executions. System executions provide behaviours necessary for building an abstract model of the system. In the current literature such approaches used for synthesizing an abstract representation following system executions are named Anti-Model-Based Testing or Anti-MBT. Thus, after having an abstract model, approaches for model-checking and generating test cases from formal models can be applied.

The purpose of this work is to underlie some MBT approaches used in the context of Motorola (CIn/BTC). In such approaches, the specifications used for generating test cases are usually incomplete, inconsistent, and sometimes they do not exist. Thus, using real system test cases it is possible to create new specifications and update original system specifications, and afterwards generate new test cases with sound behaviours of the system. Another issue detected in our context is the gap between abstract and real specifications. An abstract test case, for example, is useful for formal analysis, but it is not possible to execute an abstract test case directly in the system. Then, to manually or automatically execute test cases generated by MBT approaches (abstracts) it is first necessary to translate them into a real representation.

As result of this work we developed formal approaches for modelling system behaviours using system test cases. The results of the modelling approach are formal models specified in LTS and CSP formalisms. Beyond, we defined a unification approach which combines formal models generated from different system artefacts (requirements and test cases). The result of the unification approach is a complete and unified model of the system, which contains information provided from different artefacts. We also defined an approach for translating

abstract test cases (CSP and LTS) into real representations. The real test cases generated by using our translating approach are used in the test automation team of Motorola, by which this work is inserted. Finally, we automatized the developed approaches using programming languages and formal specifications. The result is the TCRev tool, which is able to model, unify and translate models of the system. TCRev tool interacts with external tools, such as FDR and FDR Explorer tools. All results were applied in real case studies performed in the context of Motorola. In this dissertation we show one of those case studies.

Keywords: formal methods and models, model-based testing (MBT), anti-model-based testing (Anti-MBT), CSP, LTS, modelling, unifying and translating models

Contents

List of Figures

CHAPTER 1

Introduction

Computing systems are increasingly more complex. One reason associated to this growth in complexity concerns the multidisciplinary aspect of the current computing systems. Moreover, in the traditional software development, it is common due to human limitation to build software with inconsistences. Sometimes, such inconsistences can result in several impacts to the whole system where the software is embedded. In critical systems, which constantly deal with human lives, that may lead to catastrophic events.

Certainly, to build trustful systems it is first necessary to have a sound system representation. Usually, in the traditional software development, requirement documents are used to describe the system. Many of those documents are written using some kind of natural language, such as English. A natural language to describe the behaviour of a system can lead to misunderstanding. It is difficult to specify system requirements free of misinterpretation. Moreover, there is no guarantee that the requirement documents are complete and do not have contradictory information. Beyond such limitations, requirements are the starting point in any software development and therefore need to be well specified in order to produce a final software product with high quality.

To avoid inconsistences and misunderstanding in the requirement documents, several approaches have been proposed to specify systems formally [CS06, HMBH94, SG03]. With the support provided by such approaches developers can have a higher level of understanding about the system and its properties (functionalities), as well as avoid ambiguities and inconsistences. One of those works is related to Motorola's context, where our work is also inserted. More details about the Motorola's context is described in Section 1.2.

There are several ways to specify systems formally. The choice of formalism is guided by which aspects of the system we must describe as well as what properties we need to prove correct. Well-know examples of formal languages are Z [WD96], specifically designed to describe structural aspects of systems, and CSP [Hoa83, Sch00, Ros97], well suited to characterise behavioural aspects of systems.

Although the literature [FHP02, D. 01, KVZ98] has many examples of the successful use of formal methods in industrial problems, its use still is restricted particularly because of the cost usually associated to its application. The development of supporting tools is a first step towards turning formal methods more feasible and common in industry [L. 06, For97]. In general, formal approaches are used to verify system properties (for instance, using theorem proving and model-checking), such as livelock, deadlock and determinism. However, researches have shown that formal approaches can be used to generate sound system test cases [DJK+99], UML artifacts [MS06], or even implementations in programming languages.

A less formal, and more traditional, approach to find bugs or inconsistences in software is

based on software testing, where test cases are the main artifact. Test cases are widely used in real projects. A test case is a finite sequence of input and expected output behaviour. A test case can be specified either using a manual or automatical approach. In the automatic approach, the terms automatic test cases or test scripts are used interchangeably. A test script is a test case that specifies a testing scenarios using automatic steps (having a programming language as basis). Test scripts are useful when human interactions are infeasible. For instance sending several messages to another phone, or inserting one hundred contacts in the phone book.

A formal specification of the system is also named a system formal model, or simply a system model. A system model is an abstract way to specify computer systems. Initially, the benefits of using abstract notations before starting the system implementation were only related to a better understanding of the problem. Later, it has became evident that the use of abstract formal representations combined with a theory of refinement reduces the time in the software development and increases the quality of the final software product [BSS95, Sha85]. The testing phase was also positively affected by the use of formal models. Researches [J. 03a, FHP02] have been developed to improve the quality of the system by using formal models and test cases. Once we have verified the system properties through formal models, we can generate sound system test cases, which will be put in action to verify the system implementation later. The research field which explores the use of formal approaches with software testing is called Model-Based Testing, or simply MBT [DJK$^+$99].

Nevertheless, there are situations where it is not possible to have the abstract model defined a-priori. To overcome a limitation, other approaches came out to synthesise an abstract model following system executions [BIMP04, UKM03a]. In the current literature such approaches used for synthesising an abstract representation following system executions are named Anti-Model-Based Testing or Anti-MBT [BIMP04]. Thus, after having a formal abstract model, techniques such as model-checking and testing can be applied as usual.

1.1 Goals and Contributions

In this work we have four major complementary purposes:

1. Apply formal methods in an industry real case;
2. Provide a technique to generate formal abstract models of the system using system test cases;
3. Define a strategy to unify different formal models generated from different artefacts (requirements and test cases);
4. Translate formal models (formal test cases) into real models (real tests).

Currently, the applicability of formal methods in industry real contexts has been questionable. Many works propose innovative formal techniques, but without focus on a real context. Thus, the use of formal methods in an industry real context is considered as a priority contribution of our work. In this work, all artefacts (requirements and test cases) used as input are provided by CIn/BTC Motorola's test teams.

With this work we aim to underlie some MBT approaches used in the Motorola's context. In such approaches, the specifications used for generating test cases are usually incomplete, inconsistences, and sometimes they do not exist. Thus, using real system test cases it is possible to create new specifications and update older ones, and afterwards generate new test cases with sound behaviours of the system. Another issue detected in our context is the gap between the abstract and real specifications. An abstract test case, for example, is useful for formal issues, but it is not possible to execute an abstract test case directly in the system. Then, to manually or automatically execute the test cases generated by the MBT approaches (abstracts) it is first necessary to translate them into a real representation, either manual or automatic representations.

In this work, we show two important approaches used to assist the current MBT techniques developed in the Motorola's context. Such approaches deal with modelling system artefacts (requirements and test cases) and unifying system abstract representations (formal models).

The modelling approach aims to specify a system formal model through system executions described by test cases. This approach is similar to the Anti-MBT approach, where a-priori there are not system models and the main goal is to create a system formal representation. The idea of the modelling approach is to interpret specification documents of the system (real models) written in natural language (English) and translates them into formal models (abstract). One essential component used by the modelling approach is the model of processing natural languages described by the works [Lei06, Tor06]. The processing of natural language is responsible for interpreting English sentences given as input and generating associated CSP events as output.

The result of the modelling approach is a formal model specified in a CSP process or LTS diagram. A CSP process is a entity where its actions are described through events and channels able to communicate values. An LTS diagram is a graphic representation of a CSP process. The diagram contains states which defines system situations and transitions are used to permute among states.

Beyond the generated formal model, the modelling approach is also responsible for assisting the approach of translating abstract representations into concrete ones. The translation approach is another contribution of our work. In such an approach we use mappings table generated by the modelling approach to perform the translation. The table maps abstract actions into real ones. Thus, using the mappings table it is possible to convert an abstract representation into a real one. In our context, abstract representations are specified in CSP or LTS, whereas real representations are sentences in natural language (English) or commands used in automatic test cases.

Unifying formal models also named *merging* or *integrating* formal models aims to take several formal models and unify them into a single one. In some situations where there are different views of the system it is essential to have a way to integrate all views into a unique and consistent one. In our case, for instance, several formal models specified either in CSP or LTS are generated from a system behaviour. System behavioural models are generated following several artefacts, such as requirements, manual and automatic test cases. Thus, a unifying approach is necessary to integrate all models.

Our purpose with the formal models unification is to build a unique and complete system formal model. With a unified formal model it is possible to check properties (model-checking) using a more detailed specification of the system. Beyond checking-model properties, we can also use a unified model to generate automatic and manual complementary test cases. Moreover, we update original requirements documents with information provided from other system artefacts, such as system test cases. Such a situation is common to occur in a real software development, because test cases are modified and created daily by testing teams, whereas requirements documents are in general out of date. In other situations, we can combine information from requirements and architecture documents to generate design artefacts and UML diagrams, such as, structure and sequence diagrams. Summarizing, the unification aims at combining different artefacts with complementary information about system behaviour to generate specific results.

For this unifying approach we developed two algorithms able to unify abstract models either in CSP or LTS. The first attempt uses an algorithm implemented in Java able to integrate LTS models. The result is another LTS model which contains combined information of initial LTS models. The other attempt uses CSP specification able to generate a unified CSP process which represents the initial models. The initial models are also specified in the CSP formalism.

1.2 Context

This work has been developed in the context of the CIn/BTC research project, which is sponsored by Motorola in cooperation with CIn/UFPE. This cooperation started in 2003 and, initially, aimed at creating human resource specialists in the area of software testing. This project has grown significantly and today it has 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 operation team existing issues and detected improvements motivate the research activities. The main goal of the Research 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 1.1 shows the research project initiatives that aim to improve the software development process through automation. This dissertation defines the emphasized activities: a unifying approach for system models, a modelling approach for generating system models following test cases, and a translating approach of abstract representation into a real one.

The *Intermediate Representation* is the formal representation of requirements in the CSP 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. When using the tools that implement this strategy, the final user does not directly access this intermediary representation. This allows the use of formal methods techniques excluding the necessity of formal methods specialized knowledge. The formal specification generated in CSP is used in

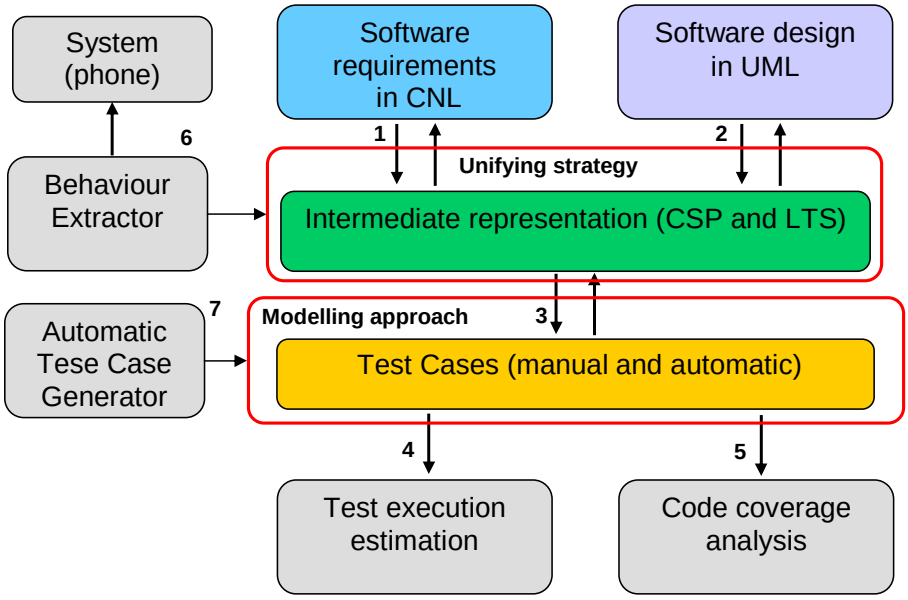


Figure 1.1 Research Team's activities overview

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).

The other activities shown in Figure 1.1 are:

- Software Requirements in CNL (1): The activity defines use case specification templates and the CNL or Controlled Natural Language. 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. This task is very close to our unifying approach, because the formal model generated from use case specifications is used as input to our unifying approach.

- Software Design in UML (2): this initiative uses the generated intermediate representation, in CSP, to generate UML Realtime (UML-RT) [MS06] 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 tests 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 [Nog06, Car06].

- 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.

- Behaviour Extractor (6): extract a formal model following the own system; differently from our modelling approach which uses the system test cases to generate a formal model, this attempt aims to generate a formal model using directly the own system. An automatic mechanism for scanning all system paths and extracting all system behaviour is been developed to assist this task.

- Automatic Test Case Generator (7): generate automatic test cases following abstract events (CSP events). This task uses the system behaviour (paths) extracted directly from the system by extractor. The goal is to generate a new implementation for abstract actions until not registered in the mappings table between abstract and real actions. This attempt comes to complete our translating approach.

1.3 Dissertation Organization

Besides this introductory chapter, this thesis has other seven chapters.

Chapter 2 summarizes the area of software testing. It explains the main concepts used in this work, and shows where we are inserted in terms of software testing area.

Chapter 3 outlines the formalism (CSP and LTS) used in this work. Basically, in our work, CSP is used for modelling the abstract representations (requirements and test cases), unifying the formal models, checking system properties, performing processes refinements. We use the LTS formalism to manipulate the CSP processes described by our unifying algorithms. The unifying algorithms which manipulates LTS structures are based on the Java language.

Chapter 4 presents our first contribution. It describes our modelling approach used to generate abstract representations from real test cases. Moreover, it discusses the translating approach used to convert the abstract test cases into test scripts (real test cases).

Chapter 5 presents our second contribution. It discusses our unifying approaches. The approaches are presented based on the used formalism (CSP or LTS).

Chapter 6 also represents one of our contributions; it shows the TCRev tool. The TCRev tool is responsible for modelling test cases, unifying formal models and translating abstract tests into test scripts. In this chapter, we also discuss the TaRGet tool. The TaRGet tool is an

attempt of the Research Team to build a complete test toolbox, which contains all endeavors of the team. The result is a test toolbox that have all functionalities described in Figure 1.1.

Chapter 7 shows a real case study performed in the Motorola's context. In this case study we selected a set of test cases of messaging feature, and some use case specifications written in CNL generated specifically for this case study. The use case specifications also describe the messaging feature.

Chapter 8 summarizes the main conclusions of this work. It presents the important points and the found limitations. Moreover, initial discussions about interesting future works are shown.

CHAPTER 2

Software Testing

Software testing is a well-defined activity in many traditional software development processes. Many researches related to software testing have been done since the 80's. Myers in his book [Mye04] defines software testing as the process of executing a program or system with the intention of finding bugs. Myers completes such a definition saying that software testing is any activity that aims to evaluate an attribute or capability of a program or system and determines whether it meets its required results. Myers' definition is complete, because he defines software testing as a process inside a bigger software development process. Thus, the process of testing involves a set of activities with the purpose of finding inconsistences. An inconsistence can be either software bugs or disagreements with the initial specification.

This chapter is divided into two main sections. Section 2.1 discusses general issues about software testing, such as definitions and notations. Section 2.2 is concerned with aspects of formal methods applied to software testing. When testing is applied together with formal methods we have a formal testing approach, and that is the main focus of this work.

2.1 Software Testing

Software testing activities are not easy tasks. The main challenge is related to software issues. Some researches [Het88] say that software can be defined like other physical processes where inputs are received and outputs are produced; software just differs in the manner in which it fails. Most physical systems fail in a fixed set of ways, whereas software can fail in many several ways. Detecting all different software failure modes is an infeasible task. The tasks performed inside the software testing process are just those concerned with the maximum quantity of bugs before the software is delivered to the final user.

Software bugs will always exist. This is not related directly to careless or irresponsible programmers, but due to software complexity. The software testing problem is not easy to solve. We can never be totally sure about the correctness of a piece of software using testing. We can use methods to give us support to verify the correctness of system specifications. These methods are involved in formal verification and validation methods, or V&V methods, where the aim is to perform validations on system models to guarantee the system specification is free of undesirable aspects.

The notation V&V has been widely used in software testing. As suggested in [And86]:

- **Verification** refers to a set of activities that assures that the software correctly implements the specified functions.

- **Validation** refers to activities that assure that the software was built attending to all costumers requirements.

There are many different verification techniques but they all fall basically into 2 major categories - *dynamic testing* and *static testing* [And86]. Dynamic testing involves the execution of a system. So, test cases are chosen and performed in the system to verify its behaviour. Dynamic testing can be further divided into three subcategories - functional testing, structural testing, and random testing.

Functional testing is responsible for testing the functions of the system as defined in the requirements. This form of testing is also called black-box testing because it does not involve knowledge of the implementation of the system. Structural testing or white-box testing has full knowledge of the implementation of the system. It uses the information from the internal structures of a system to derive tests to check the system behaviour. Random testing is a test approach that freely chooses test cases among the set of all possible test cases. Exhaustive tests is a kind of random testing, where the test cases given as input cover the whole possible set of input values. Random and structural testing are out of the scope of our work. In our work we are interested only in functional testing (black-box). Black-box testing is discussed in Section 2.1.1.

Differently from dynamic testing, static testing does not involve the execution of the system. They are concerned with techniques that analyse consistency and measure some program property. Static tests are out of our scope.

Validation usually takes place at the end of the development cycle and looks at the complete system as opposed to verification, which focuses on smaller subsystems. Validation techniques refer to approaches such as formal methods, fault injection, and dependability analysis. Techniques that use formal specification applied to software testing are the main aim in this work.

A *Test case* is one of the main concepts involved in software testing. A test case describes one or more testing scenarios. Each test scenario is responsible for testing and performing a sequence of system actions. Thus, the behaviour of an *implementation under test* (IUT, for short) is investigated by performing experiments on a IUT and observing the reactions it produces for those experiments. The specification of such an experiment is called a test case, and the process of applying a test on a IUT is called *test execution*. Observations are taken during test executions. Chapter 4 shows more explanations about test cases and important issues related to them.

A test execution can be either manual or automatic. The difference between them is that for an automatic execution the test is described using some programming language as basis (test case scripts), whereas manual tests use descriptions in natural language, such as English (manual test cases). A test case script is defined by a series of automatic steps where each step performs the testing actions. Each step is also able to verify whether such performed actions conform to the *system under testing* (SUT).

Only to make clearer, SUT differs from IUT by having others elements that compose the whole system, such as, communications channels, hardware devices, and others. So, IUT contains only the implementation for testing. It does not represent the whole system under testing.

Both test cases (manual and automatic) describe the behaviour of the system. Usually, test scripts are used to specify testing actions where the human interaction is onerous, for instance,

sending one hundred messages to another cell-phone. Testing is very expensive. It needs a lot of time, cost and human resources, thus an obvious improvement is test automation. Test cases, either manual or automatic, can be used for different kinds of tests, such as functional, performance, maintainability, usability, integrity, and others. In our work we are concerned with automatic test cases.

2.1.1 Black-Box Testing

Contrary to white-box, the black-box approach is a testing approach in which test cases are derived from functional requirements without regarding to code structure. That is why black-box testing is also named functional testing. In the literature, other definitions such as input/output driven [Mye04] or requirements based testing [Het88] can also be seen. Functional testing is a testing method based on the execution of functions and evaluation of their input and output data. The tester treats the SUT as a black-box [How86], so only the inputs and outputs are visible. In black-box testing, various inputs are exercised and the outputs are compared against the specification to check conformance.

In our work, we are concerned with black-box testing. Thus, the test cases we are concerned with do not bear in mind the software code. All test cases are based on requirements and input/output data. In our case, the software functionalities are either described by software requirements or direct through system.

Differently from white-box approaches, in black-box approaches all test cases are directly derived from the specification. No implementation details of the code are considered. Figure 2.1 gives us an overview of the black-box approach. The functionality of each module is tested with regards to its specifications (requirements) and its context (events). Only the correct input/output relationship is investigated.

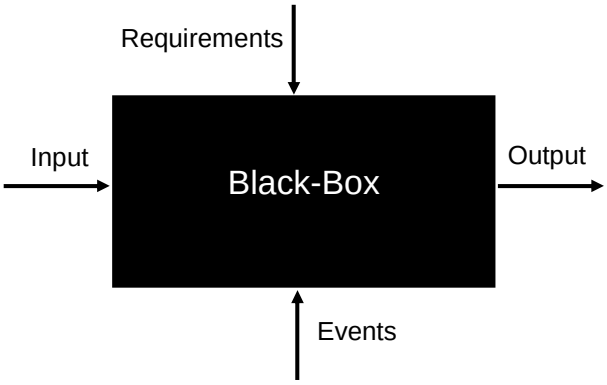


Figure 2.1 Black-box testing overview

As suggested in Figure 2.1, the effectiveness of the black-box approaches is directly related to the input data. Thus, to perform test scenarios it is necessary to have a large set of input data, which is able to cover the software functionalities. Ideally we would be tempted to exhaustively test the input data. However, testing exhaustively all valid combinations would be impossible in a complex context. In the literature, it is usual to find methods to define good input data

for performing black-box testing, such as equivalence partitioning, boundary-value testing and special-values testing [Mye04]. Many of those methods are based on the type of the software under testing.

2.2 Formal Testing

The field of software testing is more than simply applying input data in the software to find bugs. Some approaches attempt to abstract the software originating a kind of model, such as a graph that describes the control structure of the program [DJK+99]. Using such abstract models it is possible to analyse and determine the testing adequacy. For instance, in some situations an adequate test suite can test all possible paths of the model. Such testing approaches have widely been increased in past years.

Although many system specifications are written in natural languages, such as English, and thus are easily accessible, they are often incomplete and liable to different and possibly inconsistent interpretations [TB99]. Such ambiguities are not a good basis for testing: if it is not clear what a system should do. Therefore, it is difficult to test whether it does what it should do.

With formal methods systems are specified and modelled by applying techniques from mathematics and logic. Such formal specifications and models have a precise, unambiguous semantics, which enables the analysis of systems and the reasoning about them with mathematical precision and rigour. Until recently formal methods were a merely academic topic, but now their use in industrial software development is increasing, in particular for safety critical systems and for telecommunication software where this work is inserted.

Testing with formal methods [TB99] also called formal testing has been increased the testing area by introducing system models in earlier life cycle. A formal specification is a precise, complete, consistent and unambiguous basis for design and code development as well as for testing. Such system models have formal descriptions about system behaviour. This is a first big advantage in contrast with traditional testing processes where such a basis for testing is often lacking. A second advantage of the use of formal specifications for testing is their suitability to automatic processing by tools. Algorithms have been developed to derive tests from a formal specification. Many of those algorithms have been implemented resulting in automatic, faster and less error-prone test generation tools. This opens the way towards automatic testing where the SUT and its formal specification are the only required prerequisites. Thus, formal methods provide a rigorous and sound basis for automatic generation of tests. Tests can be formally proved to be valid, that is, they test what should be tested, and only that.

In formal testing if the IUT and its model are put into two black boxes and we perform all possible test cases, then it is not possible to distinguish between the IUT and its model. This is formally represented by testing equivalences, as originated by [TB99]. Based on the observations, it is decided whether the IUT is correct (verdict *pass*) or not (verdict *fail*).

Ideally, a set of test cases is exhaustive when all nonconforming implementations are detected. This notion is too strong in practice, because exhaustiveness requires infinite test cases. Thus, an important requirement for test generation algorithms is that they should produce good test cases. A good test case is a test that has a great potential to produce execution failures.

As previously discussed, model-based specifications not only provide a rigorous method to verify and validate the properties of the system, but also provide a useful basis for undertaking software testing by generating good test cases through system specification. Moreover, a precise description of the system can give us support to many others different tasks int the process of testing. For instance, using a precise specification of the SUT, it is possible:

- Automatize the testing process;
- Formalize testing strategies: approaches and frameworks;
- Integrate and unify other precise specifications (formal models);
- Reformulate or update original system requirement documents.

As formal specifications precisely describe the system requirements, they are useful to guide functional tests in an automatic way. Thus, approaches that use formal specifications can improve and automatize the testing process [TB99] by formalizing the testing approaches. An automatic support for test cases generation is essential in real situations (industry), so it will drastically reduce the overall time spent on testing and maintenance.

Particulary, in our work, we are concerned with integrating (unifying) formal models into a unique and unified system formal specification. We start by specifying initial artefacts (requirements documents, test cases, or design diagrams). Each initial artefact provides different system behaviour representations. Thus, unifying formal models it is possible to have a more complete system representation. With a unified formal model of the system it is also possible to reformulate original requirement documents. Usually, in real situations, during the testing process, requirement specifications become out of date in later phases. Often, that is related to market pressures to delivery a product as soon as possible. Thus, a lot of effort is spent to develop and test products, but less is done to update the requirement documents with new information about the system. Requirements do not receive such an attention as they should in later phases of the testing process. More details about how to formally specify initial artefacts (particulary, test cases and requirements documents) and integrate (unify) formal models can be found in subsequent chapters of this work.

2.2.1 Model-Based Testing

Model-Based Testing (MBT) is concerned with automatic generation of test procedures using system models [DJK$^+$99]. The idea of MBT is to have a model of the system, or specification, and use such a model to generate sequences of inputs and expected outputs (test procedures). The input is applied to the SUT and the system's output is compared with the model's output (black-box view). The model's output can be viewed through model's execution traces after executing the input procedures. This implies the model must be valid, in other words, the model must faithfully represents the system behaviour.

A system model can be built either through an automatic or manual way. In an automatic way tools are used to automatically extract the system model. Requirement documents, test cases or even the own system can be to extract the system model. In a manual way of generating

a system formal model, software engineers design the system behaviour following a formalism. In principle, after having built the model the test cases derivation can be done automatically.

There are many different ways to derive tests from a model. Because testing is usually experimental and based on heuristics, there is no best way to do this. Then, several approaches have been recently proposed to automatically generate test cases from models in different formalisms [Car06, Nog06, RG00]. The works presented in [Nog06, Car06] are classified on generation based on test purposes and are also inserted in the Motorola's context. They use the MBT approach for generating test cases based on models specified in labelled transition systems (LTS) and CSP, respectively. Chapter 3 gives an overview about CSP and LTS formalisms. A MBT approach overview is shown in Figure 2.2.

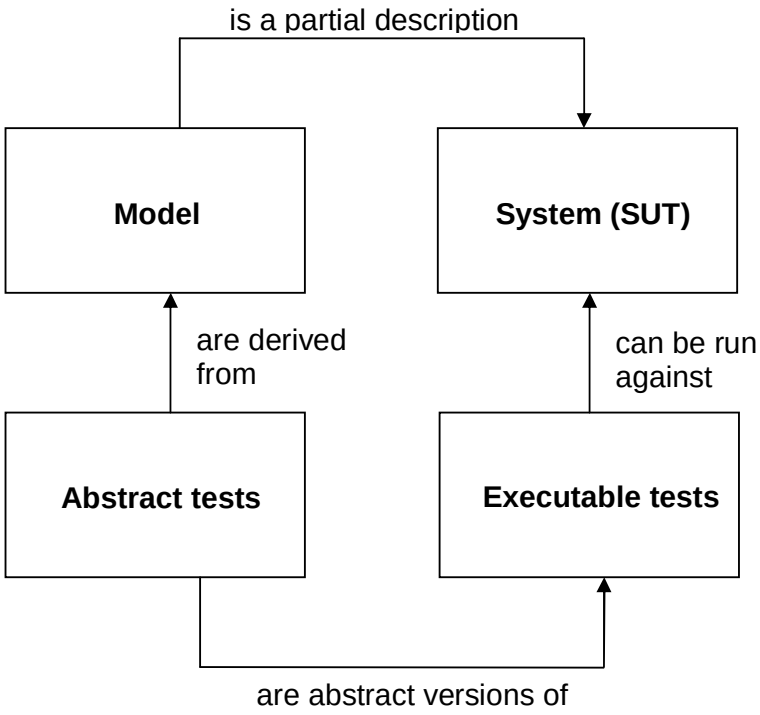


Figure 2.2 Model-based testing overview

Figure 2.2 suggests that a **Model**, also called test model, is usually an abstract and partial representation of the desired behaviour of **System (SUT)**. As previously said, MBT assumes that the system model is sound. The test cases derived from this approach are functional tests (black-box) in the same abstraction level of the model. These test cases are known as abstract test suite (**Abstract tests**). The abstract test suite cannot be directly executed against the SUT because it is in the wrong abstraction level. Therefore an executable test suite (**Executable tests**) must be derived from the abstract test suite. Executable tests can communicate with the SUT and perform the test. This is done by mapping the abstract test procedures to executable test procedures. Thus, the derivation takes each abstract step and looks for the corresponding concrete (executable) ones.

In our work, you will see that abstract test procedures are specified either using CSP or LTS structures. Thus, abstract test procedures (in CSP or LTS) are mapped into executable test

procedures. In our case, executable test procedures correspond to a set of test script commands able to be automatically executed on SUT. The CSP and LTS formalisms are presented in Chapter 3. More details about our approach for modelling and mapping abstract test procedures to test scripts can be found in Chapter 4.

The MBT approach can be applicable in several domains, since from protocol-based applications to state-rich systems, such as mobile telephony systems. All those applications have shown MBT is a useful and effective approach. Test cases execution through models of the system provides efficient ways to guarantee the conformance of the system implementation with its desirable behaviour.

Nevertheless, there are situations where MBT cannot be applied. Some reasons for that can be related to strong technical restrictions, which make MBT infeasible. An example of one strong restriction is when we do not have, or is not possible to have, the system specification available a-priori, so MBT cannot be applied. To overcome such challenges, solutions have been as additional approaches to give support to classic MBT. Such approaches intend to built an abstract model of the SUT following other sources that describe system behaviours. One attempting is to perform reverse engineering of specifications through testing. Using an approach like reverse engineering the abstract model of the SUT can be directly built from test cases or by their execution traces. In literature, it is possible to find other definitions, such as synthesis of test model through test scenarios [J. 05b] and anti-model-based testing - Anti-MBT [BIMP04]. In this work, we will use the last approach.

2.2.2 Anti-Model-Based Testing

Anti-MBT or anti-model-based testing [BIMP04] is an approach for building abstract models (test models) of the SUT using a reverse engineering process. As previously discussed, Anti-MBT is useful when it is not possible to have the test model of the SUT a-priori. Then, Anti-MBT approach builds the test model of the SUT by following its execution traces. The execution traces are described through test cases execution. This approach is concerned to perform or foresee the test cases and, by following the execution traces, be able to synthesize an abstract model of the system. This technique is deeply related to a reverse engineering process, usually applied on a traditional software process.

Differently from traditional approaches of building models, the Anti-MBT approach is based on reverse engineering of system behaviour described through execution traces. Such step can be performed in a high automation level by monitoring directly the execution traces on SUT. The work reported in [BIMP04] suggests to apply the Anti-MBT approach following three steps.

As shown in Figure 2.3, the first step consists in test cases selection. This selection is based on a high-level specification, which usually is a partial description of system. In this approach, a good strategy for selecting test cases can be through feature descriptions of the system. A feature description is a partial specification that is concerned only with a specific group of functionalities.

After selecting a set of test cases, the second step is to execute all test cases on the SUT. During the execution, execution traces are monitored and captured. After executing all test cases, we have a set of execution traces, which describes dynamic behaviours of the SUT. The

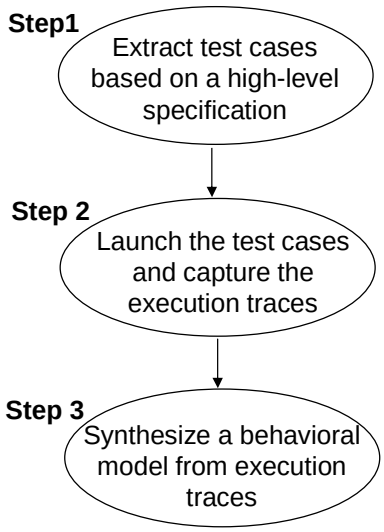


Figure 2.3 Anti-Model-Based Testing approach.

approach finalizes synthesizing all behavior extracted from the set of execution traces into a unique behavioural unit of the system. This last step is shown in Figure 2.3 (the third step).

It is worth observing that Anti-MBT does not replace MBT. The purpose of doing an approach like Anti-MBT is to build the test model (abstract model) of the SUT. Thus, after generating the test model, MBT can be normally performed.

CHAPTER 3

CSP

This chapter introduces the language CSP using its machine readable version CSP_M [Sca98]. CSP_M is an ASCII version of CSP used in practise to analyse real systems. A short introduction to its basic concepts is given in Section 3.1. Its syntax (in terms of CSP_M constructs) is presented in Section 3.2. Finally, its semantical models are showed in Section 3.3.

3.1 An Introduction to CSP

CSP (*Communicating Sequential Processes*) is a formal specification language widely used for specifying concurrent and distributed systems. In CSP, each system can be seen as a process. Processes communicate with each other through communication channels (or events). The behaviour of a process is described by possible sequence of events that the process is able to perform. The set of all events used to model a system (a collection of processes) is denoted by Σ; it is known as the alphabet of the whole specification. Events are atomic and their occurrences are instantaneous; they take no time to happen after they are allowed to occur. Sequences of events, named *traces*, are used to specify a possible behaviour that a process can perform. For instance, a trace of a process that can perform events `a`, `b` and `c` is denoted by `<a, b, c>`. The set of all possible sequences of events based on the alphabet Σ is denoted Σ^*.

A simultaneous production of events occurs when two processes are interacting through the same event. This type of communication occurs in parallel systems. In that case, the communication only occurs when all involved subsystems are ready to perform the common event. If only one process cannot engage in such common event, the whole system is forbidden to perform this communication. This type of communication is often referred as *handshake communication*.

Communications in CSP operate inside an *environment*. The environment always functions like a supplier of events. The environment may select one visible event at a time to be executed.

Concurrent systems can introduce *non-determinism*, even if the subsystems are completely *deterministic*. A system is called deterministic, if it always behaves the same after entering the same inputs. Other common problem that occurs in concurrent systems is *deadlock*. A system is in deadlock if and only if it cannot evolve externally or internally. In parallel systems, such a situation can occur when these systems compete for shared resources. For instance, one system acquires a certain resource and another system acquires another. If the former needs the resource of the latter to complete its task, and vice-versa. Thus, the interaction between these systems block forever (deadlock).

Finally, a system has a *livelock* if and only if it is performing an infinite loop of invisible actions (τ-events). Differently from a visible action, a τ-event in a system describes a internal action (invisible), so the environment is not concerned about occurrences of invisible actions. From the user viewpoint, who cannot observe internal activities, it is not clear whether the system has a livelock or not because the simply freezes externally.

The successful termination of concurrent systems is specified in CSP as *distributed termination*. In a successful distributed termination, the whole system only terminates when all concurrent subsystems also terminate. Once a subsystem is able to terminate, it cannot be stopped either by the environment or other parallel subsystem. The termination event in CSP is defined by $\surd$. Thus, whenever a system executes the $\surd$-event it goes to termination state (Ω). Once in Ω-state, a process is not able to perform any other action.

3.2 CSP Syntax

CSP was not originally designed to be a language supported by machines. When research teams began [For97] to develop tools for CSP, they realised it was necessary to define a version of CSP able to be machine processable. Thus, a machine-readable dialect of CSP (CSP_M) was defined. CSP_M has ASCII representations for all operators of CSP (for instance, the CSP operator $\rightarrow$ in CSP becomes the CSP_M operator `->`). Beyond the traditional CSP operators, CSP_M has another interesting feature: it is equipped with a functional programming language fragment based on Haskell [Hut07] to capture the data structure aspects of CSP processes. In this section the syntax of CSP_M is presented.

3.2.1 Expressions

CSP_M provides a wide variety of mathematical expressions, for example sequences, sets, boolean values, tuples, and others. The simplest kind of expression is an identifier which is evaluated in the context it occurs.

In CSP_M, the usual arithmetic operations are defined. They are sum (+), difference (−), multiplication (*), integer division (`/`) and remainder operations (%). Sequences may be written as literals by enclosing the comma-separated (extension form) list of events with the characters < and >, such as <> (the empty sequence), and `<1, 2, 3>` (the sequence containing the elements 1, 2, and 3 in this order). Another way to define sequences is informing the range of elements (intension form), so the sequence `<1..3>` means the same defined former one. As predefined operations on sequences we have:

- concatenation (`<1, 2, 3> ^ <4, 5, 6>`);
- length (`#<1, 2, 3>` or `length(<1, 2, 3>)`);
- test if a sequence is empty (`null(<>)`);
- get the first element of a non-empty sequence (`head(<1, ,2, 3>)`);
- get the tail of a non-empty sequence (`tail(<1, 2, 3>)`);

- join together a sequence of sequences (`concat(<1>, <2, 3>)`);
- test if an element occurs in a sequence (`eleme(1, <1, 2, 3>)`);
- convert a sequence to a set (`set(<1, 2, 3>)`;
- sequence comprehension (`< x | x <- <1, 2, 3> >`).

Similar to sequences, sets also can be defined by extension (`{1, 2, 3}`) or intension (`{x | x <- {1..3}}`). Traditional set of operators applied to sets are also predefined such as:

- set union (`union({1}, {2, 3})`);
- set intersection (`inter({1}, {2, 3})`);
- set difference (`diff(s1, s2)`);
- distributed union (`Union({{1}, {2, 3}})`);
- distributed intersection (`Inter({{1}, {2, 3}})`);
- membership tests (`member(1, {1, 2, 3})`);
- cardinality (`card({1, 2, 3})`;
- check for empty set (`empty({})`;
- powerset construction (`Set({1, 2, 3})`);
- set comprehension (`{ x | x <- {1, 2, 3} }`).

Finally, beyond sequences and sets we also have tuples as predefined elements. A tuple is composed by predefined or defined types enclosed by a pair of parenthesis `(.)`. Differently from sequences and sets it is only possible to define a tuple as an extension of its elements (`((1, 2, 3))`). It is not possible to define a tuple using a range such as (`(1..2)`).

All predefined data structures (sequences, sets and tuples) are weakly typed, in others words it is possible to have in the same structure elements of different types such as the sequence `<1, 2, 3, true>`, which contains integers and one boolean. The boolean constants `true` and `false` are also predefined in CSP$_M$. Comparisons like equality (==) and ordering (<, >, <=, >=) operators may take values of any type as argument, but always deliver a boolean value as result. Finally, there is conditional expressions (`if b then x1 else x2`) that delivers a result based on a boolean value.

3.2.2 Pattern Matching

As mentioned above, CSP$_M$ provides functional based features such as *pattern matching* and *recursion*. Pattern matching is usually used to simplify function definitions, yielding a more elegant presentation of the function. For example, consider a function that takes a sequence as input and returns another sequence as output with all its elements in reverse order. A definition without pattern matching can be given as follows:

```
reverse(s) =
    if null(s) then
       <>
    else
       reverse(tail(s)) ^ <head(s)>
```

By using pattern matching we avoid the use of functions `null(.)`, `head(.)` and `tail(.)`, because their functionalities are captured by the patterns `<>` (empty sequence) and `<x>^s` (a sequence whose head is `x` and tails is `s`). Thus, the new definition becomes simply:

```
reverse(<>) = <>
reverse(<x>^s) = reverse(s) ^ <x>
```

Pattern matching is used beyond function definitions. Any CSP data structure can make use of pattern matching. For example, the set `{x+y | (x, y) <- {(1, 2),(2, 3)}}` means the sum of elements contained on tuples given as input, its results is similar to `{3, 5}`. Others examples of pattern matching are:

- indexed forms of operators (`||(x, y):{(1, 2),(2, 3)} @ c!x+y -> STOP`);
- item communications (`d?(x, y) -> c!x+y -> STOP`)

3.2.3 Types

In CSP$_M$, there are two predefined types: booleans (`Bool`) and integers (`Int`). The former provides the usual values `true` and `false`. The latter contains all integer values. It is worth noting that in practise, as integers are an infinite set, a subset is used instead. Otherwise the specification becomes non-analyzable.

It is also possible to introduce new types based on the previously defined ones. These *Named Types* (or abbreviations) associate a name with a type expression that can be used in other name type definitions. A name type is similar to label a type expression. Thus, we can define a name type `EVEN_NUMBERS` as follows:

```
nametype EVEN_NUMBERS = { x | x <- {1..10}, (x%2)==0 }
```

Then, whenever `EVEN_NUMBERS` is used that means the set of ten first even numbers `{2, 4, 6, 8, 10}`.

Another possibility of introducing types is by enumeration using the keyword `datatype`. A `datatype` allows to introduce composite types. For example, consider a simple named type `MessageType` which is simply an enumeration of the tags SMS, MMS and EMS.

```
datatype MessageType = SMS | MMS | EMS
```

Using type `MessageType`, we can define the composite type `Message` that combines the type of a message and a value, for example. That is possible using a `subtype` definition, such as:

```
datatype Value = {0..10}
subtype Message = MessageType.Value
```

A `subtype` definition associates a name (`Message`) with a subset (`{0..10}`) of an existing type (`MessageType`). The value concerns the amount of messages to be sent, for instance.

Possible values of composite `Message` are `SMS.1`, `MMS.3`, and `EMS.10`. Thus, if the value `MMS.3` occurs, it means there are three messages of type MMS being communicated.

3.2.4 Channels and Events

For communications it is necessary to define channels with the keyword `channel`. Channels can be typed and untyped. Untyped channels are simply events. They can be used to signal that a special state has been reached inside a process. Thus, to specify a state the phone has just received a message, we can model as:

```
channel ReceivedMessage
```

On the other hand, typed channels are usually used when it is necessary to exchange data between two or more processes. For instance:

```
channel SendMessage : Message
```

The previous channel definition allows channel `SendMessage` communicates values of type `Message`. That is, we can observe the communications `SendMessage.SMS.1` or `SendMessage.MMS.3`.

Every channel definition defines a set of events. The definition channel `channel a` for instance defines the single event `a`, whereas `channel b : {1..3}` defines the set of events `{b.1, b.2, b.3}`.

As in a data based communication (a communication using typed channels), data is exchanged, we have the notions of input and output communication. That is, an input communication is allowed to occur when its expected output communication can happen. Thus, an output communication is given by `a!x` which requires a value for the variable `x`. Thus, if the context associates `x` to `3` then `a!x` becomes `a.3`. An input communication, denoted as `a?x`, creates a binding between the variable `x` and the value communicated through channel `a`. Thus, if the value `2` passes through channel `a` then `x` assumes the value `2` in this scope. As soon as the environment selects one of the offered events the value `x` is set to the value, which is laid down by the event.

Summarising the general forms of communicating in CSP_M are:

`!x` Output communication;

`?x:A` Constrained input communication;

`?x` Unconstrained input communication.

3.2.5 Process

A CSP process is defined by means of equations. The equation `P = T` defines the process `P` using the valid CSP basic processes and operators described in `T`; `T` is known as the body of process `P`. CSP processes can be parameterised to handle state based information. Process parameters can be used in the body of the process to describe more rich behaviours.

In CSP_M we have two basic processes:

`STOP` A process that cannot perform any actions. It is used to denote a deadlock situation explicitly.

`SKIP` A process that terminates successfully.

Each operator and its semantics will be explained as follows:

`c->P` The prefix operator offers an event (`c`) to its environment and waits indefinitely for its acceptance. If the environment allows the event, it occurs and the original process now behaves like `P`.

`P;Q` The sequential composition of the processes `P` and `Q`. This process behaves like `P` until `P` terminates successfully. When this happens the behaviour is given by `Q`.

`P/\Q` The interrupt process behaves like `P` until any event of `Q` happens. In this case, the behaviour is given by `Q`.

`P\A` The hiding process behaves like `P` except that all events in `A` are not visible to the environment.

`P[]Q` The external choice process offers the initial events of the processes `P` and `Q` to the environment which may choose which events should be produced.

`P|~|Q` The internal choice process chooses non-deterministically which of the two processes `P` and `Q` will be executed. In this case, the environment has no influence over the choice between `P` or `Q`.

`P[>Q` The timeout process initially offers all events that the process `P` may perform and then opts for offering the events of `Q`.

`b & P` The boolean guard behaves like the process `P` if the boolean condition `b` evaluates to `true`, otherwise it stops. This operator is a shorthand for `if b then P else STOP`.

`P[[a<-b]]` The renaming process maps all events `a` into the events `b`.

`P|||Q` The processes `P` and `Q` are running in parallel, without synchronizing over any events.

`P[|a|]Q` The processes `P` and `Q` are running in parallel. All events in `a` can only be performed by both processes simultaneously.

`P[a||a']Q` The processes `P` and `Q` are running in parallel. All events in conjunct of `a` and `a'` can only be performed by both processes simultaneously.

`;x:s@P` The replicated sequential composition executes the process `P` for each element in `s` sequentially, at which every occurrence of `x` in `P` is replaced by the actual element of `s`.

`[]x:s@P` The replicated external choice offers the initial events of all processes `P`, where `x` is replaced by one element of `s`.

`|~|x:s@P` The replicated internal choice offers the initial events of one of the processes `P`, where `x` is replaced by one element of `s`. Therefore it is necessary that `s` must not be empty.

`|||x:s@P` The replicated interleave runs all processes `P` in parallel without synchronization over any events, where every occurrence of `x` in `P` is replaced by one element of `s`.

`[|a'|]x:s@P` The replicated parallel (sharing) runs all processes `P` in parallel, where every occurrence of `x` in `P` is replaced by one element of `s`. These processes have to synchronize over all events that are in the communicating set `a`.

3.3 The semantics of CSP

In the previous section we defined the syntax of the CSP (indeed the CSP_M) language, accompanied by an informal description (semantics) of the corresponding constructs. In this section we briefly present the semantics of CSP using the three semantic styles: operational, denotational and algebraic.

Our work uses the operational style to propose the merging operator in Chapter 5, although the denotational and algebraic styles are also useful to analyse properties, via refinement (see Section 3.4), of our CSP processes as well as the properties of the merging operator, respectively.

3.3.1 The operational semantics of CSP

The operational semantics interprets CSP specifications as transition systems (LTS – Labelled Transition Systems). This style presents the semantics using production rules called *firing rules* ([BHR84]).

3.3.1.1 Labelled Transition System

A labelled transition system (LTS) is a tuple S=(Q, A, T, q_0), where:

- Q is a finite set of states;
- A is a finite set of labels;
- T is a transition relation ($Q \times A \times Q$);
- q_0 is the initial state, where $q_0 \in$ Q.

An LTS can also be seen graphically (see Figure 3.1), where some conventions are assumed:

- A state is a circle that represents a system state. In Figure 3.1 the state after performing the action `a` is the state `1`;
- An initial state is marked as `0` and represents the system initial state. In Figure 3.1 the initial state is represented by label `0`;
- A labelled transition is a directed arrow with a label over the arrow. It represents an action that occurs and changes the system state. In Figure 3.1 the letters that label the arrows mean the system transitions.

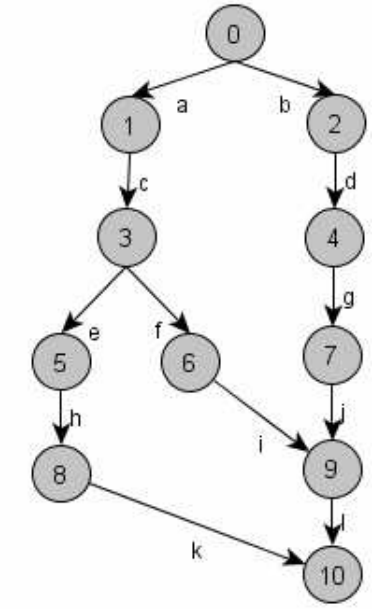


Figure 3.1 A sample of an LTS model

3.3.1.2 Transition rules

As it is well-known, the operational semantics of a language is presented by means of transition (or firing) rules. These rules are used to build a model (LTS) from each term of a language; each basic process and operator originates a possible CSP process and thus for each of these there is one or more firing rules to describe its meaning.

To ease the understanding of each firing rule presented in this section, we first briefly explain what means by a firing rule. A firing rule, presented in what follows, is built by three distinct parts: a hypothesis, a condition, and a conclusion.

$$\frac{X}{Y \xrightarrow{a} Z} \quad (condition) \tag{3.1}$$

X is a precondition that is required to hold for the rule to be applicable. If X is empty the rule is always applicable for a process of the form Y. In CSP the parameter Y always represents a process term (pattern). If for a given state a process Y is found and the precondition X is valid, then Y produces an event a (building a labelled transition) and thus behaves like Z.

The following sections present some firing rules for CSP starting with the basic processes SKIP and STOP, followed by the sequential and parallel composition. Those firing rules will be important in Chapter 5 when we define the semantic of the CSP merge operator.

3.3.1.3 CSP Operators

Basic Processes

The process STOP does not evolve (there is no interaction), so no inference rules can be defined for this process. On the other hand, the process SKIP can perform the special event $\surd$. After such an event has been produced SKIP is behaves like the special state Ω, where no further actions can be performed (at this point, it is similar to STOP).

$$\frac{}{SKIP \xrightarrow{\surd} \Omega} \tag{3.2}$$

Prefix

The prefixed operator $e \rightarrow P$ can produce the e, which can be a complex expression, and behaves like P.

As the event e can be complex, that is involve variables, the firing rule for $e \rightarrow P$ uses two auxiliary functions: *commos(e)* which results the set of all events from the expression e, and *subs(a, e, P)* which substitutes in the scope defined by P all occurrences of e for a. If for example e is the expression *c?x?y* and the generated event a is *c.1.2*, subs(c.1.2, c?x?y, P) substitutes all further occurrences of x and y for the values 1 and 2, respectively. And more explicitly, we can consider the process term $c?x?y \rightarrow d.x \rightarrow P(x,y)$, resulting in the following resolved process.

$$subs(c.1.2,\ c?x?y \rightarrow d.x \rightarrow P(x,y)) = d.1 \rightarrow P(1,2)$$

Based on the previous discussion the inference rule for the prefix operator is given by:

$$\frac{}{e \rightarrow P \xrightarrow{a} subs(a,e,P)}\ (a \in commos(e)) \tag{3.3}$$

Sequential Composition

Another important operator is the sequential composition. The process $P;Q$ is a process built by the process P followed by the process Q. The first process P can produce events until it terminates and emits the $\surd$-event. When this can happen, the sequential composition operator consumes the $\surd$-event, transforming it into a τ-event. It is worth noting that in the left hand-side rule the event a must be distinct from $\surd$ ($a \neq \surd$). Otherwise this rule would coincide with the right hand-side rule, allowing a non-deterministic behavior which is not the case for this operator.

$$\frac{P \xrightarrow{a} P'}{P;Q \xrightarrow{a} P';Q}\,(a \neq \surd) \qquad \frac{P \xrightarrow{\surd} P'}{P;Q \xrightarrow{\tau} Q} \tag{3.4}$$

External Choice

The external choice operator allows the environment to choose from a number of alternative events to be produced. It is necessary to produce a visible event to make a selection. After the event is emitted, the process that produced that event can continue to produce events. If both processes can produce the same event, one of them is chosen non-deterministically.

$$\frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'}\,(a \neq \tau) \qquad \frac{Q \xrightarrow{a} Q'}{P \,\Box\, Q \xrightarrow{a} Q'}\,(a \neq \tau) \tag{3.5}$$

If one of the processes offers a τ-event, this τ-action is consumed immediately without changing the state of the original choice.

$$\frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \,\Box\, Q \xrightarrow{\tau} P \,\Box\, Q'} \tag{3.6}$$

Hiding

The hiding operator leaves all events untouched that are produced by the subordinate process, as long as they are not in the set of events that are to be hidden.

$$\frac{P \xrightarrow{a} P'}{P \setminus B \xrightarrow{a} P' \setminus B}\,(a \notin (B \cup \surd)) \tag{3.7}$$

Events that are in the set of hidden events are changed into τ-events by this operator which makes them invisible for the environment.

$$\frac{P \xrightarrow{a} P'}{P \setminus B \xrightarrow{\tau} P' \setminus B}\,(a \in B) \tag{3.8}$$

If the subordinate process offers to terminate, the whole process terminates and evolves to Ω.

$$\frac{P \xrightarrow{\surd} P'}{P \setminus B \xrightarrow{\surd} \Omega} \tag{3.9}$$

Renaming

The renaming operator itself produces no events, but transforms events from the subordinate process into other events. It is necessary to define a relation R that specifies which event should be changed into a another event, to perform a transition. The default rule is the identity relation, where each event is in relation with itself. If $a, b, \in \Sigma$ are in the relation R the following rule applies:

$$\frac{P \xrightarrow{a} P'}{P\,[[\,R\,]] \xrightarrow{b} P'\,[[\,R\,]]}\,(aRb) \tag{3.10}$$

The events $\surd$ and τ are not affected by this operator:

$$\frac{P \xrightarrow{\tau} P'}{P\,[[\,R\,]] \xrightarrow{\tau} P'\,[[\,R\,]]} \qquad \frac{P \xrightarrow{\surd} P'}{P\,[[\,R\,]] \xrightarrow{\surd} \Omega} \tag{3.11}$$

Parallelism

If one of the processes can produce a τ-event, the behavior is similar to the choice operator where the corresponding process evolve but the original composition remains.

$$\frac{P \xrightarrow{\tau} P'}{P \overset{\|}{X} Q \xrightarrow{\tau} P' \overset{\|}{X} Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \overset{\|}{X} Q \xrightarrow{\tau} P \overset{\|}{X} Q'} \tag{3.12}$$

All events outside the set X can be produced without the participation of both processes.

$$\frac{P \xrightarrow{a} P'}{P \underset{X}{\|} Q \xrightarrow{a} P' \underset{X}{\|} Q}\ (a \in \Sigma \setminus X) \qquad \frac{Q \xrightarrow{a} Q'}{P \underset{X}{\|} Q \xrightarrow{a} P \underset{X}{\|} Q'}\ (a \in \Sigma \setminus X) \tag{3.13}$$

If the event belongs to the event set X then both processes must participate simultaneously.

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \underset{X}{\|} Q \xrightarrow{a} P' \underset{X}{\|} Q'}\ (a \in X) \tag{3.14}$$

Finally, we have firing rules to deal with termination. If one of the processes can terminate, the operator will produce a τ action and the process evolves to Ω.

$$\frac{P \xrightarrow{\surd} P'}{P \underset{X}{\|} Q \xrightarrow{\tau} \Omega \underset{X}{\|} Q} \qquad \frac{Q \xrightarrow{\surd} Q'}{P \underset{X}{\|} Q \xrightarrow{\tau} P \underset{X}{\|} \Omega} \tag{3.15}$$

If both processes have already terminated the whole process terminates and evolves to Ω.

$$\frac{}{\Omega \underset{X}{\|} \Omega \xrightarrow{\surd} \Omega} \tag{3.16}$$

Conditions

Sometimes it is necessary to check some conditions in specifications and to act differently depending on the result of the condition. In CSP the *if-then-else* statement enables the developer to specify such conditions. If the condition in the statement evaluates to *true* the *then* case, otherwise the *else* case is executed.

In the following rules *C[[b]]* stands for the result of the evaluation of the boolean expression b. The resulting operational semantics for *if-then-else* statements is as follows:

$$\frac{C\,[[\,b\,]] = true, \quad P \xrightarrow{a} P'}{(if\ b\ then\ P\ else\ Q) \xrightarrow{a} P'} \qquad \frac{C\,[[\,b\,]] = false, \quad Q \xrightarrow{a} Q'}{(if\ b\ then\ P\ else\ Q) \xrightarrow{a} Q'} \tag{3.17}$$

3.3.2 The denotational semantics of CSP

The denotational semantics of CSP gives another way of describing the behaviour of a process. A denotational semantic describes a language in terms of functions. It main use is to provide a formal basis to state the concept of refinement.

CSP can be described using three complementary semantic models: traces, failures, and failures-divergences. The traces model characterises what a process can do. The failures model complements the traces model by also capturing what a process cannot do. Finally, the failures-divergences model complements the failures model by also describing if divergent behaviors can occur.

In our work, we are specifically concerned with the traces model. As we will see in Chapters 4, and 5, the reason to deal with traces model is test cases are described following execution traces of the system, moreover the implementation of the CSP merge operator is based on functions that operate in sequence of events (traces) of the system.

3.3.2.1 The Traces Model

The traces model of the denotational semantics describes the sequences of events that a process can perform. To understand this model we take a look at the following two processes:

$$P = (a \rightarrow P) \ \sqcap \ (b \rightarrow P)$$

$$R = (a \rightarrow R) \ \Box \ (b \rightarrow R)$$

The traces which these two processes can produce are identical. In each step of execution the processes can perform either an event a or an event b. The difference is that a process $P \ \Box \ Q$ can simultaneously offer all events to the environment that P and Q can produce. From the viewpoint of producing events, the non-deterministic process can also produce an event a or an event b. Consequently the traces of the two processes are identical.

The traces model defines a set of sequences of events for each CSP process. As in the operational semantics, we present here the traces for the basic processes and thus for operator-based processes.

Basic Processes

As process STOP cannot produce any events, its set of possible traces contains only the empty trace.

$$traces(\text{STOP}) = \{\langle\rangle\}$$

The process SKIP, however, also contains the trace $\langle\surd\rangle$.

$$traces(\text{SKIP}) = \{\langle\rangle, \langle\surd\rangle\}$$

Prefix

The prefixed process $a \rightarrow P$ can produce an event a and then behaves like process P. Thus, its traces contains the usual empty trace and all traces from P whose prefix is the event a.

$$traces(\mathrm{a} \rightarrow \mathrm{P}) = \{\langle\rangle\} \cup \{\langle a\rangle \hat{}\, \mathrm{s} \mid \mathrm{s} \in traces(\mathrm{P})\}$$

The term Σ defines all visible events of a CSP specification. Σ^{τ} defines the alphabet extended by the internal event τ, which is invisible to the environment. $\Sigma^{\surd}$ denotes the alphabet extended be the event $\surd$ that indicates successful termination. Finally, $\Sigma^{\tau,\surd}$ refers to the alphabet extended by τ and $\surd$. Both events τ and $\surd$ are not part of the alphabet ($\Sigma \cap \{\tau, \surd\} = 0$). The term Σ^* describes a set of all finite sequences of the events in the alphabet Σ.

Sequential Composition

Let *P;Q* be a sequential composition between processes P and Q. Its traces are formed by those of P without the special event $\surd$ ($traces(P) \cap \Sigma^*$) plus the events starting with P and terminating successfully followed by the traces of Q. Note that the special event $\surd$ is suppressed in the concatenation between the traces of P and of Q.

$$\begin{aligned} traces(P;Q) = {} & (traces(P) \cap \Sigma^*) \\ & \cup \{s \frown t \mid s\langle\surd\rangle \in traces(P) \wedge t \in traces(Q)\} \end{aligned}$$

Internal and External Choice

The traces model does not allow to distinguish external and internal choice. For both operators the observed behaviour either corresponds to P or Q. Hence the traces of $P \sqcap Q$ and $P \,\Box\, Q$ are the union of the traces of the subprocesses P and Q.

$$traces(P \sqcap Q) = traces(P) \cup traces(Q)$$

$$traces(P \,\Box\, Q) = traces(P) \cup traces(Q)$$

The complete reference about the traces semantics model and the others semantics model (Stable Failure and Failure-Divergences Models) can be found in [BHR84].

Hiding

In CSP, the hiding operator transforms visible events to τ-events, which are not observed in the traces model. Given a trace $\mathrm{s} \in \Sigma^*$ and a set of events $X \subseteq \Sigma$ hiding on traces can

be defined by restricting the elements of the trace to elements of the specified alphabet without those elements in X.

$$s \setminus X = s \upharpoonright (\Sigma \setminus X)$$

In the resulting traces $s \setminus X$ all occurrences of event, which are in the set X are removed. Using this definition, the traces of the hiding operator can easily be defined on the traces of the original process.

$$traces(P \setminus X) = s \setminus X \mid s \in traces(P)$$

Renaming

The renaming operator requires a relation R, which maps certain events to other specified events. Implicity this relation is an identity for all events that are not specified to be renamed. Like the hiding operator this operator can best be defined by describing the effects of the renaming on the traces of the original process. Therefore R^* is to be defined as an extension of R to traces, which maps traces of elements of *domain(R)* to traces consisting of elements of *range(R)*.

$$\langle a_1, ..., a_n \rangle R^* \langle b_1, ..., b_m \rangle \Leftrightarrow n = m \wedge \forall\, i \leq n \;\; . \; a_i \; R \; b_i$$

The elements of the traces of *P[[R]]* can now be defined using the definition of R^*, since all occurrences of events of *domain(R)* must be replaced by the specified events of *range(R)*.

$$traces(P[[R]]) = t \mid \exists s \in traces(P) \; . \; sR^*t$$

Parallelism

The parallel operator in CSP can be used to model parallel processes, which must synchronise over a specified set of events, which can only be performed simultaneously. All other events of the parallel processes may be produced independently from each other, like with the interleaving operator.

The following recursive definition can be used to compute all possible traces of two traces $s,t \in \Sigma^*$ of parallel processes. All occurrences of x and x' are representing events, which are in the synchronisation set X ($x, x' \in X$), whereas $y, y' \in \Sigma \setminus X$ are events that are not in

the synchronisation set.

$$
\begin{aligned}
s \underset{X}{||} t &= t \underset{X}{||} s \\
\langle\rangle \underset{X}{||} \langle\rangle &= \{\langle\rangle\} \\
\langle\rangle \underset{X}{||} \langle x\rangle &= \{\} \\
\langle\rangle \underset{X}{||} \langle y\rangle &= \{\langle y\rangle\} \\
\langle x\rangle \frown s \underset{X}{||} \langle y\rangle \frown t &= \{\langle y\rangle \frown u \mid u \in \langle x\rangle \frown s \underset{X}{||} t\} \\
\langle x\rangle \frown s \underset{X}{||} \langle y\rangle \frown t &= \{\langle x\rangle \frown u \mid u \in s \underset{X}{||} t\} \\
\langle x\rangle \frown s \underset{X}{||} \langle y\rangle \frown t &= \{\} \ if x \neq x' \\
\langle x\rangle \frown s \underset{X}{||} \langle y\rangle \frown t &= \{\langle y\rangle \frown u \mid u \in s \underset{X}{||} \langle y'\rangle \frown t\} \cup \{\langle y'\rangle \frown u \mid u \in \langle y\rangle \frown s \underset{X}{||} t\}
\end{aligned}
$$

Applying these rules to two given traces express that events, which are in the synchronisation set, can only be produced, if both traces are starting with this event. On the contrary events of $\Sigma \setminus X$ can be produced independently of each other. Using this definition of parallel traces the definition of the traces of two parallel processes can be described by froming the union of all possible parallel traces of the subprocesses P and Q.

$$traces(P \underset{X}{||} Q) = \bigcup\{s \underset{X}{||} t \mid s \in traces(P) \wedge t \in traces(Q)\}$$

Conditions and Guarded Commands

The if-then-else operator chooses, depending on the value of a boolean expression b, whether the traces of the process P or Q must be evaluated for the computation of the traces of the process.

$$traces(\textit{if } b \textit{ then } P \textit{ else } Q) = \begin{cases} traces(P), & \text{if } b \text{ evaluates to } true \\ traces(Q), & \text{if } b \text{ evaluates to } false \end{cases}$$

Guarded commands are defined analogous to the if-then-else operator, where the else-case refers to the process STOP.

$$traces(b\&P) = \begin{cases} traces(P), & \text{if } b \text{ evaluates to } true \\ traces(STOP), & \text{if } b \text{ evaluates to } false \end{cases}$$

3.4 Refinement Relation

Refinement for CSP processes mean a relation between two processes stating whether one is better than the other in the sense of the semantical models [BHR84]. Therefore, the refinement concept can be applied to different semantics models, which each semantics model drives the refinement properties. Thus, to define that a process Q *refines* another process P, we written the refinement relation $P \sqsubseteq Q$. This same relation interpreted over the traces model ($P \sqsubseteq_T = Q$ said as *Q trace-refines P*) leads to the slightly weaker concept of traces refinement:

$$P \sqsubseteq_T Q \equiv P =_T Q \sqcap P \equiv traces(Q) \in traces(P)$$

When the FDR performs the traces refinement it tries to verify the refinement relation $traces(Q) \in traces(P)$. The above equivalence to the refinement definition is easily checked, though the interpretation of this characterization is a little different. It states that if a process P is indistinguishable from $P \sqcap Q$, then any situation where P is suitable must allow that $P \sqcap Q$ is suitable (since this is equal to P), and so Q must also be suitable since the internal choice could always be resolved in favour of Q. The process Q is a refinement of P because it will be appropriate in any environment which will find P acceptable. An alternative way of thinking about the equivalence is that all of Q´s behaviours must already be allowed by P, since the introduction of Q does not introduce any new behaviours. This algebraic characterization of refinement is also appropriate for other semantic models, as will be discussed in later chapters. If the model is clear from the context then the subscript to the refinement symbol will be dropped.

Refinement satisfies a number of laws. Such laws refer to properties like reflexive, transitive, and anti-symmetric in all models. Figure 3.2 shows some refinement properties.

$P \sqsubseteq P$	⟨$\sqsubseteq$-reflex⟩
$P_0 \sqsubseteq P_1 \wedge P_1 \sqsubseteq P_2 \Rightarrow P_0 \sqsubseteq P_2$	⟨$\sqsubseteq$-trans⟩
$P_0 \sqsubseteq P_1 \wedge P_1 \sqsubseteq P_0 \Rightarrow P_0 = P_1$	⟨$\sqsubseteq$-anti-sym⟩
$RUN \sqsubseteq_T P$	⟨$\sqsubseteq_T$-bottom⟩
$P \sqsubseteq_T STOP$	⟨$\sqsubseteq_T$-top⟩

Figure 3.2 Refinement laws

In our work we are concerned with trace refinement relations. We combine sub-processes (see Chapter 5 about merging processes) and verify if the former processes is refined by the combined process. Moreover, the relation $traces(P) \in traces(Q)$ is key for implementing the merging operator, either based on LTS or CSP models. If the merging operator is applied to two sub-processes P and Q, and, moreover, we have $traces(P) \in traces(Q)$ then the following

relation must be preserved:

$$traces(P) \in traces(Q) \in traces(PQ)$$

In the above relation, the process PQ means the combined process generated from merging of P and Q. The entire trace refinement relation for merging operator can be formulated as:

$$PQ \sqsubseteq_T P \sqsubseteq_T Q \equiv PQ =_T (Q \sqcap P) \sqcap PQ \equiv traces(Q) \in traces(P) \in traces(PQ)$$

3.5 CSP Tool Support

CSP is a successful specification language with available tool support. The most known tool is FDR or Failures and Divergences Refinement Tool [For97]. The other tool is the ProBE graphical animator [For98]. Both tools use the machine-readable of CSP (CSP_M) as previously discussed.

3.5.1 The FDR Model-Checker

The FDR tool, which has been developed by Formal System Europe, is based on the theory of CSP [BHR84]. FDR employs the operational semantics of CSP to generate a labelled transition system for a CSP specification. From such a transition system FDR can perform analysis of deadlock or livelock, by analysing the structure of the graph and identifying states with no transition or states that can be reached by a loop of τ-labelled transitions. Additionally FDR allows to check, if a specification meets certain properties, by performing refinement checks on the transition system of the specification.

The concept of generating transition graphs or automata out of formal specifications is well known and often used for testing and model checking. Thus, using the operation semantics of CSP it is possible to generate a LTS from a CSP specification. For all sequential subprocesses a transition system representing its behaviour can easily be determined by applying the rules of the semantics for each operator, as shown on Section 3.3. For instance, the parallel operators generate a synchronisation tree referencing the sequential graphs, which allows to completely compute the product graph of the specified system. As soon as the LTS is completely generated, standard techniques like normalisation can be applied to create a smaller graph representation. Refinement relations can also be checked by comparing the LTS of two processes. The next section shows a tool that interacts with the FDR and is able to generate the LTS diagrams.

3.5.2 The FDR *Explorer*

By reusing FDR as a back-end tool support, it is also possible to develop another tools as front-ends. The FDR manual [For97] describes how one can use Tcl/Tk scripts [WJH03] as direct interaction with FDR engine (server). The FDR *Explorer* is an extended API as Tcl/Tk script

files used to work together with FDR server [L. 06].

The FDR *Explorer* was proposed in the work reported in [L. 06]. Summarizing, the FDR *Explorer* builds a file representing the LTS of a given CSP specification using the FDR server. This file contains all the data of an LTS in mathematical form (sets, relations, etc.). From this file it is possible to implement a set of new functionalities. For instance, consider to call the FDR *Explorer* as:

```
bash$ FDRExplorer abstract_use_model.csp USE_MODEL_normalize
```

The first parameter (*abstract_use_model.csp*) means the CSP specification given as input and the second one (*USE_MODEL_normalise*) represents the target process of specification which is given to compilation step. In that case the process *USE_MODEL_normalise* uses the transparent function *normalise* of FDR. The generated file (*abstract_use_model.csp.leo*) is similar to:

```
Starting CSP compiler

Ready

Started: 56 Finished: 46 Transitions: 100
Started: 105 Finished: 89 Transitions: 200
Started: 144 Finished: 128 Transitions: 300
Started: 185 Finished: 174 Transitions: 400
Started: 225 Finished: 213 Transitions: 500

...

Ready

BEGGING FDR OBJECT MODEL EXPLORATION

Loading selected file abstract_use_model.csp

    known processes = USE_MODEL_normalize ... CHAOS(1)

Compiling only the given target processes {USE_MODEL_normalize}

Starting compilation of { USE_MODEL_normalize } processes

    Compiling USE_MODEL_normalize now...
    Finished USE_MODEL_normalize...
Compilation finished with { ism__2 } ISMs


<<<<START LTS INFO OF USE_MODEL (ism__2)>>>>
    name        = USE_MODEL
```

```
    transitions = {1 1 0} {2 88 5} {3 89 10} ... {11 95 9}
    Event Encoding for USE_MODEL (ism__2)
         event(0) = _tau
         event(1) = _tick
         event(2) = chM.3.tick
         event(3) = chM.3.evC
         event(4) = chM.3.evH
        ...

<<<<END LTS INFO OF UNIFIED_USE_MODEL (ism__2)>>>>


DELETING ELEMENTS IN { ism__2 }
END FDR EXPLORATION FOR abstract_use_model.csp

SEE file abstract_use_model.csp.leo
```

The generated file contains information about the processes compilation step. If a target process is given as input (as above) the compilation step is performed considering only the target process, otherwise all processes are compiled and analysed automatically. Beyond compilation information it is also possible to get the alphabet and transition system from processes.

Currently, we are using FDR *Explorer* to compile the CSP processes, get their alphabets and generate theirs transition diagrams. Chapter 5 shows how to use the LTS diagrams to generate a unified model from target processes. The unified model is a LTS diagram that contains all behaviour from initial LTS diagrams.

CHAPTER 4

Modelling Test Cases as Formal Models

As previously discussed in Chapter 2, test cases are central artefacts in a software testing process. They specify testing scenarios of the SUT. In a real testing process, a considerable amount of resources (human and time) are allocated to deal with design and specification of test cases [Mye04].

Initially, the test case design process starts by specifying the possible testing scenarios. Thus, the test designer models several desirable testing scenarios. Depending on the SUT, the number of testing scenarios can be enormous. The mobile applications context is an example of complex testing scenario. Usually, due to a large variety of testing scenarios, the test designer divides a wider scenario into thin ones. Each thinner scenario is responsible for testing a specific functionality of the SUT. Their grouping allows feature testing. So, for example, all testing scenarios related to messaging, which describes actions as sending, receiving, saving, or deleting messages are grouped into the messaging feature. In that way, several features testing can be specified, such as phone-book, email, multimedia feature and others.

Beyond design and specification tasks, the testing process can contain inspection procedures. An inspection procedure validates whether test cases were specified correctly. Basically, it consists in analysing testing scenarios and verifying whether the specified test cases indeed cover whole scenarios. Others issues related to test cases are also verified, such as writing standard, language coherence, implementation logic in test scripts, and others.

In Section 4.1 we will show how testing scenarios are specified in real test cases; the design task is not covered here. Section 4.2 shows the proposed approach for modelling automatic test cases in formal models. As an essential part of our modelling approach, in Section 4.2.2.1 we briefly discuss how to translate sentences written in natural language (English) into CSP events. Finally, Section 4.4 shows the related works involving modelling or synthesis of formal models from system scenarios, as well approaches for generating test scripts.

4.1 Test Case

Following the testing process, after having modelled all testing scenarios and defined the test features for the SUT, the next step is to specify the testing scenarios as test cases. Like testing scenarios, test cases are also grouped into features test. A test case is composed of [Mye04]:

- **Inputs:**

  – Initial condition: assures the test case can be executed. It sets all necessary conditions (initial state) for performing the test case;

– Steps: define the sequence of actions to be executed by either testers or automatic testing procedure;

- **Outputs:**

  – Expected results: this field specifies the expected results of SUT after giving a specific input or performing a step;

  – Post-condition: specifies the final state of SUT after performing all test steps.

It is worth noting that in the Motorola's automation context, the term post-condition is used in a different way. They say post-condition is responsible for setting the SUT (phone) in a *safety state*. The safety state is a state which guarantees the correct execution of the next test case. Thus, the post-condition sets the phone to its initial state before starting the test. The reason for that is because in a test suite, usually, many automatic test cases (test scripts) are sequentially executed. In our work, as we are concerned with Motorola's test scripts, we follow such a definition for the post-condition term.

In our context, either automatic or manual test cases follow the same structure. But, differently of manual test cases, all parts of a test script are performed automatically. The validation steps, expected results field, are internally performed by an *automatic test framework* [Ran02]. The automatic test framework is responsible for performing and validating the actions, so in a test script the expected results field is not visible by testers. An automatic test framework is simply a software layer that provides functions to codify test scripts [Ran02]. Many test functionalities, such as screen navigation, fields checking, step validations and others, are directly supported by the framework. Test scripts are directly executed over automatic test framework.

Beyond traditional test cases, as previously discussed, there are other test cases whose purpose is not to find bugs. Such test cases are the configuration test cases. In most cases, a configuration test case is an automatic test case responsible for setting up all SUT's configurations before executing a usual test (either manual or automatic). Configuration test cases define configuration procedures. They do not specify behaviours of the SUT. In our work, we are concerned with capturing system behaviours through testing scenarios (test cases), thus configuration tests are out of our scope.

4.1.1 Manual Test Cases

Manual test cases are described following some natural language (English) as basis. Thus, they are used by testers for manual executions. Figure 4.1 shows a traditional manual test case.

The objective of the test case in Figure 4.1 is to test the system behaviour when the user is composing a message using the messaging composer and the flip of the phone is closed. The test steps are specified from one to five steps. Notice that, before executing the test, all initial conditions need to be verified. The test's initial conditions are: first the application for composing a message has to be available on the phone, second the phone needs to have a flip (hardware), finally phone's memory cannot be full. As observed in the test steps (specifically in step 4), when the user closes the flip while composing a message, the unsaved message is automatically saved in a specific folder (the draft messages folder). That is the reason by which

Case	Case Description	Procedure	Expected Results
1	TC_MSG_CLOSE_FLIP	The objective is to test the system behaviour when the user's composing a message and the flip is closed.	
	Initial Conditions:	The messaging composer application is available. The phone is a flip phone. The message memory is not full.	
	Test Procedure (Step Number):		
	1	Start the message center.	The phone is message center.
	2	Create a message.	The phone is message composer.
	3	Insert a recipient address into the recipients field.	The recipients field is filled.
	4	Close the flip.	The saving message transient is displayed in display.
	5	Open the flip and wait for the transient screen.	The message saved transient is displayed in display informing that the message was saved in the drafts folder.
	Final Conditions:	Delete the saved message on drafts folder.	The saved message on drafts folder is deleted.

Figure 4.1 An instance of a manual test case

the phone's memory must be available (not full). In *Final Conditions:* the original phone state is restored (delete the saved message).

In the testing process, after specifying the test case as shown in Figure 4.1, the next step is the test execution. Initially, for executing, the tester sets up the SUT following the procedures contained in the initial conditions. In order to do that, the tester can use a configuration test case, rather than configuring manually. After configuring the SUT for testing, the tester follows the described steps and executes them directly on the SUT. The result of each step is verified against the expected results defined in the test. If all steps, after being performed, agree with their expected results the test verdict is *pass*, otherwise the verdict is *fail*. If some steps, by some reason, could not be executed the tester reports the test verdict as *inconclusive*. In automatic execution (for test scripts), the procedure is the same. It differs only in the way of execution, instead of being executed by a tester, the test script is directly executed by the automatic test framework. The automation test framework is responsible for performing all steps, validating them and giving the test verdict.

4.1.2 Automatic Test Cases (Test Scripts)

First, it is important to note that not every test scenario can be automated, in others words, there are test scenarios that are not possible to specify an automatic test case (test script) for testing them. Such test scenarios are related to situations that involve user actions, such as change

phone battery, insert a phone charger or close phone flip as shown in test of Figure 4.1. Thus, a test designer cannot specify an automatic test case for the previous situations.

A test script is a test case that specifies a testing scenario through automatic steps (having a programming language as basis). Test scripts are useful when human interactions are tedious and onerous, for instance sending several messages to another phone, or inserting one hundred contacts in the phone book. Figure 4.2 shows a test script that specifies a test scenario able to verify if the messages are shown correctly in the outbox folder.

```
TestCase TC_VIEW_AND_HIGHLIGHT_IN_OUTBOX_FLD:

  Setup:

    description("Delete all messages.");
      phone.goToIdle();
      phone.startApp(MESSAGING);
      phone.delete(ALL_MESSAGES);

    description("Set the phone to show the messages in the outbox folder by Subject.");
      phone.goToIdle();
      phone.startApp(MESSAGING);
      phone.goToAndSelectMenuItem(MESSAGE_SETUP);
      phone.scrollToAndSelect(FOLDER_VIEW);
      phone.scrollToAndSelect(SUBJECT);
      phone.waitForTransientScreen(Notice.CHANGED_FOLDER_VIEW_SUBJECT);

    description("Compose and send two messages to a valid phone number.");
     ...

    description("Go to the Outbox folder and verify that this screen is displayed.");
     ...

  Steps:

    description("Highlight the first message.");
      phone.scrollToMessage(Content.SMS_ONE);
      phone.verifyHighlight(Content.SMS_ONE);

    description("Highlight the second message.");
      phone.scrollToMessage(Content.SMS_TWO);
      phone.verifyHighlight(Content.SMS_TWO);

    description("Go to the Message Center and verify that this screen is displayed.");
      phone.goToAndSelectMenuItem(BACK);
      phone.checkScreen(MESSAGE_CENTER);

  PostCondition:

    description("Delete all messages.");
      ...

    description("Set the phone to show the messages in the folder by Address.");
      ...
```

Figure 4.2 An instance of an automatic test case (test script)

Differently of the former scenario (Figure 4.1) it is possible to design a test script to automatically test the system behaviour. Note that, the test script shown in Figure 4.2 contains the same parts as manual test cases: initial conditions, steps and final conditions. It just differs in

two points: first, it uses different naming to initial and final conditions (**Setup:** for initial conditions and **PostCondition:** for final conditions), then the expected results (validation steps) are not shown in the test case. Nevertheless, the validation steps are internally executed by the automation test framework.

Other important aspect of test scripts is that all test steps are first specified in a higher level (**description** command) and then described in programming steps (script commands). The step in a higher level is specified using natural language (English), as shown by description commands. The description command defines the step goal, whereas the following script commands describe the automatic actions need to be achieved that goal.

As previously shown, both test cases (manual and automatic) specify system behaviour, but, particulary, we are concerned with automatic test cases. The effort of modelling manual test cases is covered in the work [Lei06]. Such a work defines a strategy for reading manual test cases and translating them into CSP specifications. The translation process is based on natural language processing techniques [Bar88]. The process takes as input English sentences and generates corresponding CSP events as output. The work presented in [Lei06] is strictly related to the Motorola's context and our work. We use natural language processing to translate the English sentences contained in *description* commands into CSP events. Such a translation step is essential to our approach of modelling test scripts using formal models. For that reason, in Section 4.2.2.1 we will give further explanations about the natural language processing and how it is currently related to our work.

4.2 Modelling Test Cases as Abstract Formal Models

As discussed in Chapter 2, there are some situations where MBT is not feasible. One of those situations is when it is not possible to have the abstract model defined a-priori. To overcome such a challenge, others techniques, such as Anti-MBT [BIMP04], came out to synthesise an abstract model following system executions. The system executions provide system behaviours necessary for building an abstract model of the SUT. So, after having an abstract model MBT can be applied as usual.

In our work, we also apply techniques similar to Anti-MBT for building a formal system specification. As we have already known (Section 4.1), system test cases (either manual or automatic) describe system behaviours through testing scenarios. Thus, we propose to apply reverse engineering from the system test cases and build a formal representation of the SUT based on extracted behaviours. However, before creating a whole system model, the first step is to model each system test case as a single formal model.

The strategy of using the own system test cases to generate an abstract model of the SUT is interesting, because test cases mean real behaviours (actions) of the SUT. Test cases are specified and executed directly using the system. Furthermore, in a conventional testing process, test cases are continuously created and modified to adapt to new or modified behaviour of the SUT. Testers are frequently specifying or modifying test cases. Thus, it was observed in real situations that many updated information about the SUT comes from the testers' experiences. Testers know well the feature under testing, so they add new information when are specifying new test cases or modifying older ones. Usually, such information is not contained in the

original requirements documents.

Using test cases to build system models it is easy to develop a way to extract system behaviours automatically. Once many test cases are specified following a sequential flow, it is easy to extract each testing scenario separately. That not happen when we have others kind of documents which describe the system behaviour in a complex way, using scenarios and different flows. In such situation it is difficult to extract the system behaviour properly.

4.2.1 Feature Tests *versus* Interaction Tests

Test cases can have several denominations. Such denominations are based on their execution flows. Test cases that have a sequential flow are commonly named as *feature testing* or *unit testing*. They have such a definition because are concerned with testing a specific feature of the SUT at a time. As all test suites are grouped by features and we have a feature test sequentially specified in a test case, it is easy to define an automatic approach for extracting system actions by features.

There are others kind of test cases, such as interaction tests, which do not necessarily specify the system actions in a sequential way. *Interaction* test cases [Coh04] describe more elaborated testing scenarios. As its name suggests, it is concerned with test situations where several features are tested simultaneously, in other words, we have testing scenarios involving integrated features that interact with each other. For instance, consider a testing scenario in which a user is composing a message (messaging feature) and a phone call (calling feature) occurs unexpectedly. Such test cases are used in real situations because they are strong candidates to find out a system bug. However, for such test cases it is not so easy to take all action sequences of the SUT automatically.

Interaction test cases are out of our scope. In this work, we are concerned with test cases that test only one feature at a time (feature test). Interaction test cases have been currently explored within our Research Team [AAM06]. Such test cases are a good scope for further extensions of our modelling approach, as discussed in Chapter 8.

4.2.2 Modelling Test Cases Formally

The modelling approach receives a set of test scripts and returns abstract models related to each test script given as input. Beyond generating abstract models, the approach also performs a mapping between the CSP events generated from natural language processing and the corresponding script commands. Such a mapping is stored in a database and it is useful for generating concrete test scripts. Recall from Chapter 2 that the MBT approach uses two representations for test cases: one is an abstract representation and another is a concrete one. In our approach abstract test cases (also named formal test cases) are represented in CSP or LTS notation. Differently from abstract test cases, concrete test cases can be executed (manual or automatic) following test steps. Figure 4.3 gives us an overview of our approach of modelling test cases.

Following Figure 4.3, the first step of our approach is to parse all test scripts given as input. The parsing step is responsible for reading each test script and extracting all contents of the test script (setup, steps and postcondition). With all extracted information, the contents of the

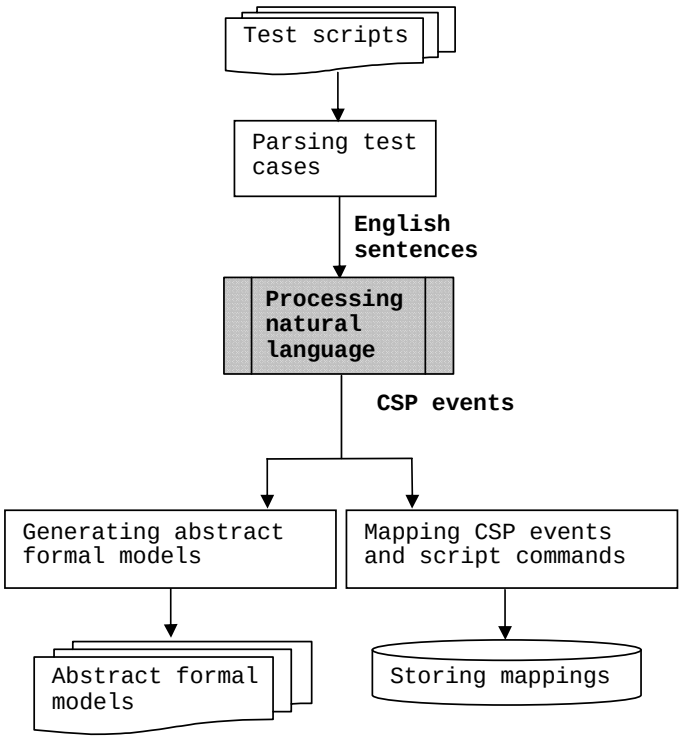


Figure 4.3 Modelling approach of test cases

description command (English sentences) are sent to the system of natural language processing. The natural language processing step (in gray) is out of our scope. We interact with it to translate high level descriptions into CSP events used to be composed in abstract test cases. For further details about the natural language processing, please consider [Lei06]. As previously shown, the description command describes in high level the goal of the action sequences writing in programming language (see Figure 4.4). The result of processing of an English sentence is a corresponding CSP event. All translated CSP events are sent to the modules of generating and mapping. Finally, after performing the modelling approach, abstract models (CSP or LTS) and a set of mappings between CSP events and script commands is built.

Figure 4.4 High level goal *versus* action sequences

4.2.2.1 Processing Natural Language

As shown in Figure 4.3 the step of processing natural language it is very important to our modelling approach. The natural language processing aims to translate sentences in natural language (English) into CSP events or LTS transitions. Therefore, to do this it is necessary to set structures used for performing the translation. Such structures are grammatical terms and classes, knowledge databases, syntactic and semantic parsers.

The work detailed in [Lei06] shows a strategy for processing manual test cases specified in English and translating them into test cases in CSP notation. It uses a strategy based on domain ontology, in others words the terms contain semantic values defined by an specific ontology. Taking the ontology as basis, it uses syntactic and semantic parsers to capture and interpret the sentences. As our application domain is mobile systems, the semantic value of terms defined in the knowledge databases are strictly related to that domain. So, terms such as "phone", "message", "SMS", "phonebook", "contact number" are commonly presented.

4.2.2.1.1 CSP alphabet definition: CSPHeader To handle abstract test cases (in CSP notation) it is necessary to specify a set of CSP definitions (datatypes, channels, tuples and others) able to recognize and manipulate a CSP event. As previously discussed, each CSP event (abstract) has a corresponding real action, and such relation is stored in our mappings table.

Real actions are described in natural language (English) and specify what the testers must do in a test case. For instance, the event `delete.DTDEL_ITEM.(MESSAGES, {ALL})` presented in abstract test cases mean the real action *Delete all messages* in a real test. In this example, the step of deleting all messages was specified in CSP by a channel `delete`, which means the deletion action, and a tuple preceded by a datatype, `DTDEL_ITEM.(MESSAGES, {ALL})`. Such tuple construction says to channel `delete` that all message items should be deleted of the phone.

The result of structuring all CSP elements able to recognize the abstract test cases in CSP is a file named *CSPHeader.csp*. The CSP header file contains all definitions of datatyes, tuples and channels used for translating natural sentences in high-level into CSP_M events.

For further explanations about the CSP events definition, using tuples, datatypes and channels, and how they were built using English sentences, please refer to work [Lei06, Tor06].

4.2.2.2 Generating Abstract Formal Models

The generation module is responsible for building abstract models from a set of CSP events received from test cases. For each test case it is generated one corresponding model. An abstract model generated from a single test case is also called abstract or formal test case. Abstract test cases are represented either using CSP or the LTS formalism. Using the CSP formalism, an abstract model is viewed as a finite CSP process that terminates with `SKIP`. All events in the CSP process occur sequentially using the prefix operator (`->`). Figure 4.5 shows an abstract test case in the CSP notation generated from test script of Figure 4.2.

In Figure 4.5 the CSP events **setup**, **test**, and **postcondition** were put only to analyze the CSP events generated from each part of test case. They are not generated from natural language processing step, and, consequently are not used for unifying process, as explained in Chapter 5.

```
TC_VIEW_AND_HIGHLIGHT_IN_OUTBOX =

 setup ->
   delete.DTDEL_ITEM.(MESSAGES, {ALL}) ->
   set.DTSET_ITEM.(VIEW_MESSAGES, {}).(BY_SUBJECT, {}) ->
   compose.DTWRITE_ITEM.(MESSAGES, {TWO}) ->
   send.DTSEND_ITEM.(MESSAGES, {TWO}).(NUMBER, {VALID}) ->
   goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}) ->
   verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {}) ->

 test ->
   highlight.DTSELECT_ITEM.(MESSAGE, {FIRST}) ->
   highlight.DTSELECT_ITEM.(MESSAGE, {SECOND}) ->
   goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}) ->
   verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {}) ->

 postcondition ->
   delete.DTDEL_ITEM.(MESSAGES, {ALL}) ->
   set.DTSET_ITEM.(VIEW_MESSAGES, {}).(BY_ADDRESS, {}) ->

 SKIP
```

Figure 4.5 An abstract test case (CSP notation)

Another way to represent an abstract test case is using the LTS formalism. Recall from Chapter 3 that an abstract model in LTS is a directed graph where the states are connected through a set of generated CSP events. Figure 4.6 shows an abstract test case in LTS that corresponds to the same abstract test of Figure 4.5.

The purpose of generating abstract test cases in both notations (CSP and LTS) is because after unifying all abstract test cases (formal tests) in a unique formal model we can perform two kinds of guided generation of test cases using contributions of the Motorola Research Team. One is based on CSP [Nog06] and another on LTS [Car06]. Thus, to be able to use both generation approaches we perform the modelling and unifying following both notations. In Chapter 5 we will see that we defined two unifying approaches, again one based on CSP and another in LTS.

4.2.2.3 Associating CSP Events with Script Commands

The mapping module takes all CSP events and maps them on corresponding script commands. The goal of mapping CSP events to script commands is the possibility of capturing an abstract test case (test case in CSP or LTS) as a real test script. Figure 4.7 shows the mapping process. First, the content of the description command (English sentence) is translated into a CSP event. After translating into a CSP event, the set of corresponding script commands is captured to compose a new mapping with the CSP event. This new mapping is stored as a new entry data in a mappings table. We use a table of a relational database to store all generated mappings.

Based on the mappings table it is possible to generate new test scripts. The mappings table is the basis of our strategy for generating new test scripts. The generation step follows a translation of abstract test cases into concrete test scripts. So, it is only necessary to have the mappings table and an abstracted test case (in CSP or LTS). In Chapter 7 we show some results of modelling and generating new test scripts.

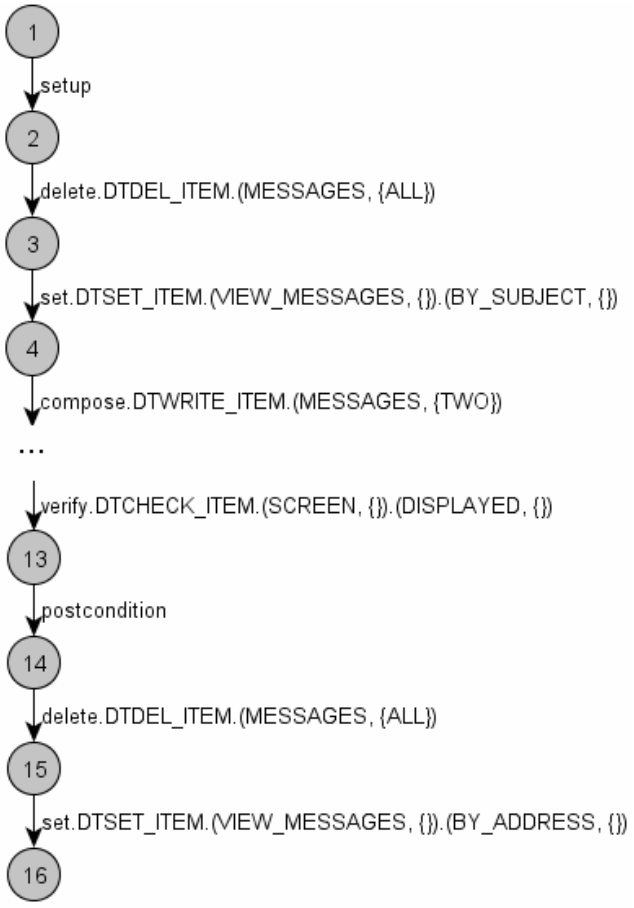


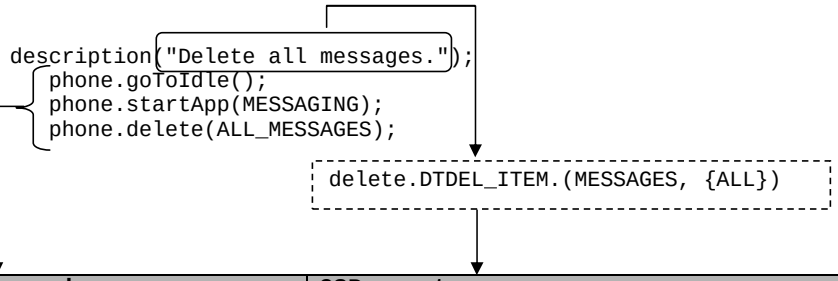
Figure 4.6 An abstract test case (LTS notation)

4.3 Translating Abstract Test Cases to Test Scripts

The translating step from abstract test cases into test scripts uses the mappings table generated by the modelling step (Figure 4.7). The idea is to take the abstract test cases and extract all CSP events. For the extraction step we use a parser which captures all CSP events and sent to the lookup mechanism. With all CSP events extracted from the abstract test case we search on the mappings table for all corresponding script commands. After extracting all script commands associated with each given CSP event we organize the commands and a separate file. The result is an automatic test case as shown in Figure 4.2.

It is important to note that there are situations when not every CSP events are registered in the mappings table. In such cases, we generate test scripts with special elements. These elements are identified in the test case by the tag **prototype**. Figure 4.8 shows part of test case generated with one unregistered CSP event.

As shown in Figure 4.8 the not-found CSP event (`PLAY.(MULTIMEDIA_FILE, {ONE})`) in the mappings table is put in the test script with a prototype tag (**prototype**). The sequence of script commands are commented and must be manually coded. Once the user coded manually, this new entry is stored in the mappings table. Inside our Research Team there are other works



ID	Script commands	CSP event

1	phone.goToIdle();
	phone.startApp(MESSAGING);
	phone.delete(ALL_MESSAGES);
		delete.DTDEL_ITEM.(MESSAGES, {ALL})

2	phone.goToIdle();
	phone.startApp(MESSAGING);
	phone.scrollToAndSelect(OUTBOX);
	phone.checkScreen(OUTBOX);
		verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {})

3	...	...

Figure 4.7 Mapping of CSP event and script commands

```
prototype("PLAY.(MULTIMEDIA_FILE, {ONE})");
  /*  (“prototype” tag)
   *  This code is intended to be empty.
   *  A new code implementation must be put here.
   */
```

Figure 4.8 Example of prototype tag

which intend to generate the sequence of script commands for unregistered CSP events. Such works will complement our translation approach.

4.4 Related Works and Important Considerations

In this section we will analyse other works related to our modelling approach. First, in Section 4.4.1, we will discuss other techniques used for abstracting a test model of the SUT. Obviously, abstraction methods vary depending on whatever system specifications (requirements) are available. In Section 4.4.2, we discuss methods for translating abstract test cases into concrete ones.

4.4.1 Other Modelling Approaches

In the following, we divided the approaches into two categories. The first part (Section 4.4.1.1) includes the approach for modelling test models from implementation descriptions of the SUT. The second (Section 4.4.1.2) includes works that build the test model deriving from natural language specifications of the SUT, which is more similar to our modelling approach.

4.4.1.1 Modelling from Implementation Descriptions

Works such as [DBG01, SA99] use system descriptions to extract manually or semi-automatically a test model that describes processor architectures. In [DBG01], the MμALT design is translated manually to an FSM in the GOTCHA Definition Language (GDL) which formed the test model [HN99]. GOTCHA (an acronym for Generation of Test Cases for Hardware Architectures) is a tool for generating an abstract test suite from a test model of the SUT and a set of testing directives.

In [KVZ98], the test model is a formal specification in LOTOS (Language of Temporal ordering Specifications [ED89]) of system protocols. The LOTOS specification was manually produced and was checked using the CADP (CAESAR/ALDEBARAN Development Package) verification tools, a toolbox with formal verification capabilities. This model is then used as input to the TGV (Test Generation with Verification technology [BMJ]). TGV is a prototype for the generation of conformance test suite for protocols. TGV translates the LOTOS test model into an Input Output Labelled Transition System (IOLTS format).

In [SA99], a test model of the microprocessor was built semi-automatically. The control part in the description initial design is generally implemented using an FSM that encapsules the description of control behaviour. The states of this FSM are naturally selected as candidates of the test model states. The authors developed an algorithm to extract these states automatically from the microprocessor RTL Verilog or VHDL [Ash98] design.

4.4.1.2 Modelling from Natural Language Specifications

Modelling approaches that use natural language specification to build a test model are closer to our approach. In that cases where the SUT was specified in natural language with no formal specification or implementation descriptions, hand-built test models often need to be constructed. A common difference of the following works to ours is that our test model is built automatically, whereas many other approaches are based on manual efforts. In such approaches, the advantage for following natural language descriptions is a higher-level of abstraction contained in natural sentences (black-box), whereas implementation details are in lower-level and, consequently, harder to understand and abstract the system behaviour.

In [FHP02], this was done for parts of POSIX (Portable Operating System Interface [Lew91]) standard and Java [Gra97] exception handling facility. In both cases, the model was crafted as an FSM in GDL. At this stage, specification defects and inconsistencies were discovered.

The work proposed in [A. 99] aims to study the feasibility of automatic test derivation and execution from a number of formal specifications and different test execution approaches. Test generation was done with a tool named TorX, a generic model-based testing environment. TorX allows plugging in different test generation tools, which accept models of the SUT in different formal languages. In such work, three formal models of the same SUT were built for that study in LOTOS, SDL [M. 91], and PROMELA [Hol03], each to use with a different test tool.

Finally, the work [J. 03b] used the modelling language of AutoFocus to build a test model for the WAP Identity Module (WIM). AutoFocus is atool for developing graphical specifications for embedded systems based on concise description techniques and simple, formally defined clock-synchronous semantics. A model in AutoFocus is a hierarchically organized set

of time-synchronous communicating EFSMs that use functional programs for its guards and assignments.

After describing other modelling approaches, we could detach strong and weak points in our modelling approach. First, detaching the strong points, we have the same achieved points of modelling following natural language (high-level abstraction). The actions described in our formal models have a high level of abstraction. They define the actions a user-level interactions.

Other strong point of our modelling approach is an automatical mechanism for extracting and interpreting the natural sentences contained on test cases in CSP structures, whereas, usually, the others approach are fundamentally based on manual effort. A weak point in our modelling approach is built test models do not have explicit states notion. Some test models on above works contain explicit states. Explicit states are used here like processes parameters in test model. So, in a model that follows processes algebra, such as CSP, we have parameters that define the process states.

Although we do not have explicit states in our test models, we can characterize a state as a model execution trace. Thus, starting from initial point it is possible to define the state of the model by following the execution trace on model. We named such state notation as implicit because of our test models do not have the states explicitly. In our unifying approach we had to consider such implicit states on test models. Thus, as we will see on Chapter 5, the unified model needs to preserve all internal states of initial models.

A good extension for our modelling approach is to expand the test model to consider explicit states. By introducing explicit states notion we can inform test directives to guide the process of test cases generation. Such test directives are used for analysing the test execution after giving such directives as parameters for test cases. In literature, test directives is commonly used as *test points*. A test point is a specific value for test case input and state variables [Bin99]. Others further extensions for our modelling approach are summarized on Chapter 8.

4.4.2 Other Translating Approaches

Recalling from previous sections, the generation process of test cases from test model generates abstract test cases, so such test cases cannot be executed on SUT directly. Then, first, they need to be translated into concrete test cases. Due to this fact MBT approaches use a component (usually called *tc-translator* [PERH05]) that translates the abstract test cases to concrete ones. Thus, concrete test cases are applicable to the test platform of the SUT which can be a simulation environment, a test framework or directly the SUT itself.

The tc-translator's task is to bridge the abstraction gap between the test model and the SUT by adding missing information (setup information) and translating entities of the abstract test case to concrete constructs of the test platform's input language. For example, if the data type values of an operation's operand are abstracted to equivalence classes in the test model, the tc-translator select a concrete and type-correct operand for that operation [SA99]. In our case, the operation's operand are system events, thus in abstract test cases the events are in CSP notation, whereas in concrete test cases the events are translated into script commands applicable to the test automation framework.

Some MBT approaches use a configuration file or table to configure the tc-translator [J. 03a, FHP02, SA99]. This table contains the translation relation between abstract entities (states,

operations, and others) of the model and concrete instructions for the SUT's test platform. By doing so, the tc-translator can easily be adjusted for different test platforms. Often one operation in the model is a macro for the tc-translator and is substituted by many instructions at concrete level. Furthermore, the table may specify how to determine the precise timing of operations in a concrete test case if the test model abstracts from timing issues. Our modelling approach follows this same idea. First, we take concrete test cases (former test cases), next we translate them into abstract test cases. Thus by translating from concrete test cases to abstract ones in the second step, we store the entities mapping between abstract and concrete events in a table of a relational database.

CHAPTER 5

Unifying Formal Models

Unifying formal models also named *merging* or *integrating* formal models aims to join several formal models into a single unified model. In some situations where there are different views of the system it is essential to have a way to integrate all views into a unique and concise one. In our case, for instance, several formal models specified either in CSP or LTS are generated from system behaviour. System behavioural models are generated following several artefacts, such as requirements, manual and automatic test cases, or even from own system, those artefacts are used to describe the system behaviour. Thus, a unifying approach is necessary to integrate all these models.

Our purpose with the formal models unification is to build a unique and complete system formal model. With a unified formal model it is possible to check properties (model-checking) using a more detailed specification of the system. Beyond checking-model properties, we can also use a unified model to generate automatic and manual test cases. Moreover, we can update the original requirements documents with information provided from other system artefacts, such as system test cases. Such a situation is common to occur in a real software development, because test cases are modified and created daily by testing teams, whereas requirement documents are unmindful in later phases. In other situations, we can combine information from requirements and architecture documents to generate design artefacts and UML diagrams, such as, structure and sequence diagrams. Summarizing, the unification has the purpose of combining different artefacts with complementary information about system behaviour to generate specific results.

In this chapter we will detail our unifying approach. Section 5.1 presents an overview of the proposed unifying approach. Next, in Section 5.2, we detail the unifying strategy by showing the main concept of unifying, the *merging* operator. Finally, Section 5.3 shows some related works concerned with the merging of models.

5.1 Approach Overview

In our approach, we handle initial structures (artefacts) to generate system formal models. The generated models contain information about system behaviour (actions). The idea is to use such formal models to build a unified model with updated and complete information about the system.

The system initial models can be originated from different artefacts in several abstraction levels, so generating structurally different models. Formal models generated from a lower abstraction level will have more detailed structures, whereas models with a higher abstraction

level will have a shallower structure. Such a difference between the models will not affect our unifying strategy, as we will discuss in subsequent sections.

The artefacts have descriptions about system behaviour in different views and abstraction levels. Artefacts can be requirements and architecture documents, design diagrams, test cases, and even the own system application. Each artefact has an abstraction level, for example, test cases and requirements contain *user view* information (higher abstraction), whereas architecture and design artefacts have information in *component view*. Information in component view is more detailed (lower abstraction) and is concerned with system components and their interactions.

Formal models in user view specify actions between the user and system, such as, starting applications, navigating through screens, selecting items in lists and menus, and making phone-calls. Components view models specify actions among components. A component can be either internal or external and an interaction in the component view can be any action (internal or external) able to occur in the cell-phone's system. An internal action can be performed by applications (messaging, phonebook, and browsing), screen controllers, and memory managers. External actions involve the user, mobile telephony operators, or even other systems, and are related to external world.

The key idea is to use either the user or component view models as input to the *merging operator*, and generate a unified formal model with more complete and detailed information about the system. A priori it does not make sense to unify user and component view models together, but our approach does not make such a restriction. In other words it is possible to unify models in both views (user and component). In our case study in Chapter 7 we unify information related to a unique view (user or component) at a time. Figure 5.1 shows a black-box view of our unifying approach.

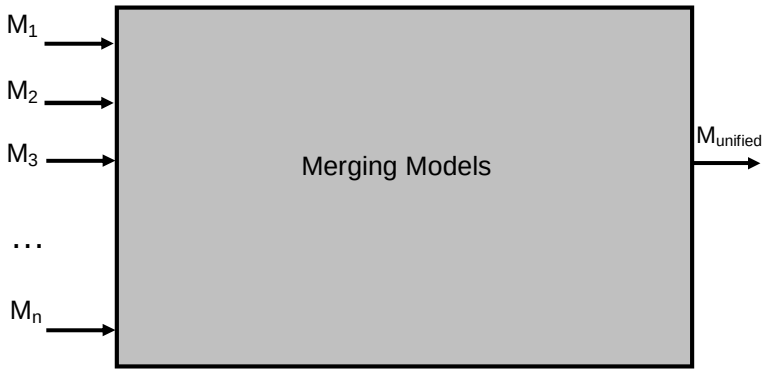


Figure 5.1 Black-box view of unifying models

In Figure 5.1, the symbols M_1 to M_n mean the initial formal models (before unifying) given as input to the merging operator. Thus, we can see the unifying approach as a black-box procedure, where $M_{unified}$ represents the unified model after merging the initial models. Section 5.2.2 shows two white-box views of the merging operator.

5.1.1 Merging Models Outcomes

After merging all initial models we have a unified view of the system. As previously said, with a unified model we verify system properties and generate new artefacts. Figure 5.2 shows the possible outcomes we can get through a unified model.

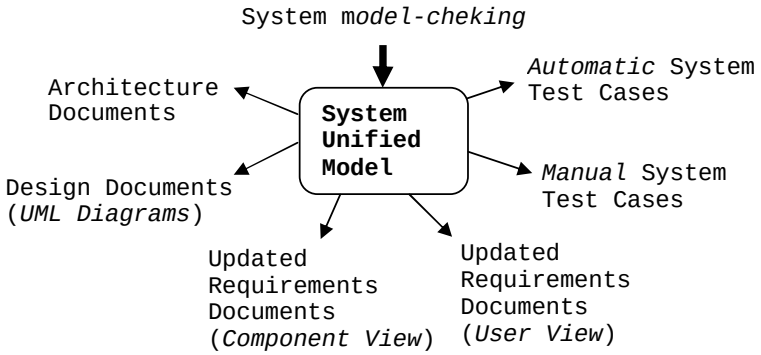


Figure 5.2 Outcomes of unifying strategy

According to Figure 5.2, with a unified formal model it is possible to check system properties using a more detailed specification. Beyond checking-model properties, we can also use it to generate new test cases (automatic and manual) and update original requirement documents. In other situations, we can combine information from requirements (component view) and architecture documents to generate design artefacts and UML diagrams, such as, structure and sequence diagrams.

In a real situation, we have to focus in which outcomes we would like to generate through a system unified model. For instance, to generate test cases we have to consider initial models generated from the user view, such as original test cases and requirements in the user view. This makes reasonable, because test cases essentially have system descriptions in user view. In our context, it does not make sense to unify requirements in components view and design documents, when the purpose is, for instance, generate test cases. We deal only with black-box tests, so they do not have information about interactions among system internal components, but only actions of the user view.

In Chapter 7, we deal with requirement documents in the user view and automatic test cases. The purpose is to show how such artefacts can be used to generate a unified model, and afterwards, which outcomes we can gain by using a unified model.

5.1.2 Merging Operator Overview

Our unification strategy is based on a *merging* operator (symbolised by $\oplus$). The merging operator is responsible for unifying models into a single one. It works by taking two or more models and combining their information. The unified model contains all behaviour presented in the initial models. The unification of the merging operator is based on the transitions of the involved models.

A transition T in an LTS model is represented by triple $(s \times t \times s')$, where s is the initial state, t is the transition label and s' is the final state after performing the transition T. In CSP,

a transition is characterised by events performed by processes. CSP processes are composed by event alphabets. The alphabet of a process (represent by Σ) contains all events able to be performed by the process. To show how the merging operator analysis the transitions of the models, let's consider two single LTS models, *A* and *B* from Figure 5.3.

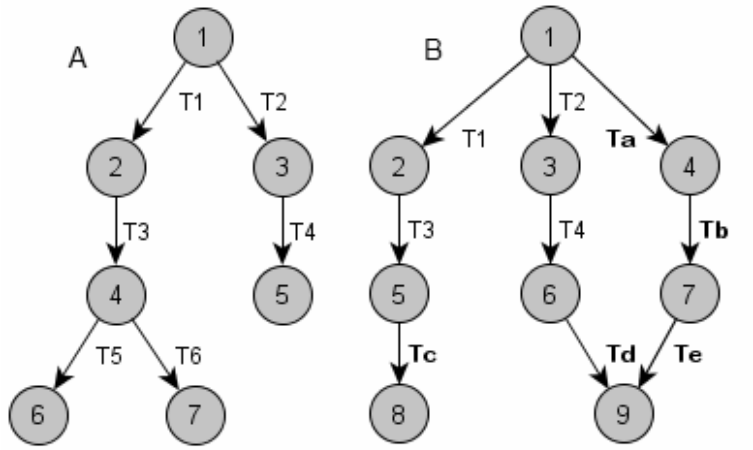


Figure 5.3 Unifying idea: initial models

Figure 5.3 shows two LTS models (*A* and *B*), where their transitions are labelled with a letter *T* follows by a number, for instance T_1, T_2. The model *B* can also have transitions labelled as T_a, T_b, and others. Thus, in Figure 5.3, transition labels terminated with a letter, such as T_a, T_b, mean transitions that occur only in model *B*. Transition labels terminated with a number, such as T_1, T_2, mean transitions that can occur in both models. Then, in Figure 5.3, we have four common transitions (T_1 to T_4) presented in both models, and five transitions that occur only in model *B* (T_a to T_e).

Visually, we can see that both models are distinguished, nevertheless complementary. There is information in model `A` that there is not in `B` and vice-versa. The purpose of the unification is to complement the information and create a unique formal model that contains all information contained in both initial models.

Initially, the unification process takes both initial models (*A* and *B*) and verifies which transitions are common (presented in both models). For those common transitions, it is generated a unique transition that represents them. When different transitions occur, both transitions must be preserved in the unified model. The analysis of the transitions must take in account the order in which the transitions occur. Two transitions are common if they have the same label and occur in the same position of their traces. Finally, after analysing the common transitions we can build the unified model that it will contain all transitions of initial models. Figure 5.4 shows the final model after unifying the models *A* and *B* from Figure 5.3.

Figure 5.4 represents the unified model with only two initial models, but it is possible to apply the unification process with more models compositionally. Initially, two initial models would be used to generate a partial unified model. Using the partial unified model and another initial model we would generate another partial unified model. That process would be performed with all initial models and, after all, we would have the final unified model.

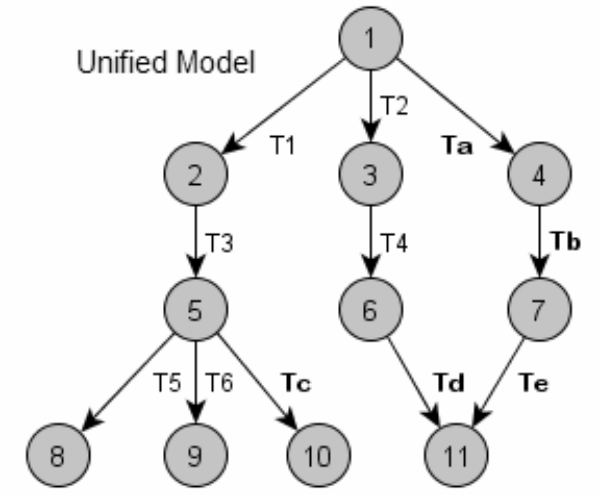


Figure 5.4 Unifying idea: final unified model

Notice that, the order used to unify the models is not relevant. In other words, the merging operator preserves the commutative property. Then, the equation $A \oplus B = B \oplus A$ should be preserved. The analysis of that and other properties of the merging operator will be shown in Section 5.2.4.

5.2 The Merging Operator

As we saw, the unification process is similar to a partner-matching process, which determines whether two transitions match. The merging operator is responsible for matching such transitions to perform the unification. However, it is essential the transitions of the initial models use the same alphabet. It does not make sense to merge models that use different alphabets. The precondition of our merging approach is that all involved models must have the same alphabets.

The natural language processing system [Lei06] is responsible for interpreting sentences in natural language (English) and generating corresponding transitions (events in CSP). Such a system uses knowledge bases and domain ontology to assure that independently of which resources are used to generate models (test cases, requirements, architecture and design documents), all generated models use the same alphabet. This is an important issue, because with that our merging precondition is satisfied. The merging precondition assures that all CSP events in the initial models have the same CSP alphabet.

Another important issue for our merging models is the structure of the initial models. The initial models are expanded models. An expanded model means a CSP process without parameters. In CSP, the state of a process is represented by parameters. For a CSP process with parameters, a state is represented by values of its parameters. However, in our case, it is possible to consider the states of an expanded model are represented by its paths. The definition of

merging operator must considerate this, because we have to generate a sound unified model, in other words with all initial expanded states preserved.

After giving those initial explanations related to merging operator, we can detail how the merge behaves like. In advance, the merging behaves similar to CSP parallelism, but it differs when we must preserve the expanded states of models involved on unification. Recall from Chapter 3 that in a parallel composition common events (synchronization events) are synchronized between involved processes (see firing rules 3.14), but when they differ (different events) only one process is able to performed at a time, as shown in the firing rules 3.13. Thus, in the parallelism the unchosen process can be selected later. The possibility of selecting the unchosen process after being made the decision by the first process does the parallelism impractical for our merging approach. The parallelism does not preserve the expanded states of the initial models.

The purpose is to have a behaviour similar to the syncronization step of the parallelism (as shown in the firing rules 3.14 of Chapter 3), but performing a different behaviour when branches had been selected. In that case, we must discard the unchosen branch. This can be avoiding that the unchosen branch be selected later. Thus, introducing an external choice behaviour between these processes we eliminate the possibility of occurring an event of the unchosen branch at all.

Summarizing, the merging operator behaves like parallelism synchronizing in the common events of the processes involved in the unification (as shown in rules 3.14), and like an external choice eliminating the possibility of generating inconsistent information when different events occurred. Once it behaves like an external choice it can not behave like parallelism at all. The next section we show the modelling (behaviour) of merging operator by its firing rules.

5.2.1 Modelling the Merging Operator

We said that in the operational semantics of CSP [Ros97], each operator is defined using *firing rules* that specify the operators behaviour. For the merging operator we also used a set of firing rules to specify its behaviour.

As discussed in the previous section, our merging operator, symbolized by $\oplus$, first must behave similar to the parallelism synchronizing events as well as to the external choice distinguishing events. Thus, similar to Roscoe [Ros97] definition, we have rules to promote τ actions:

$$\frac{P \xrightarrow{\tau} P'}{P \oplus Q \xrightarrow{\tau} P' \oplus Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \oplus Q \xrightarrow{\tau} P \oplus Q'} \tag{5.1}$$

The merging operator also has to handle distributed termination (see Rules (5.2)), similarly to the rules of parallelism:

$$\frac{P \xrightarrow{\surd} P'}{P \oplus Q \xrightarrow{\tau} \Omega \oplus Q} \qquad \frac{Q \xrightarrow{\surd} Q'}{P \oplus Q \xrightarrow{\tau} P \oplus \Omega} \tag{5.2}$$

Once all arguments have terminated and become Ω (like *STOP* process in CSP$_M$), we can finish the whole process using the following rule:

$$\frac{}{\Omega \oplus \Omega \xrightarrow{\surd} \Omega} \tag{5.3}$$

There are two rules for ordinary visible events: one for common events (a parallel behaviour):

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \oplus Q \xrightarrow{a} P' \oplus Q'} \ (a \neq \tau) \tag{5.4}$$

and two others for distinguishing events (an external choice behaviour):

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{b} Q'}{P \oplus Q \xrightarrow{a} P'} \ (a \neq b) \qquad \frac{P \xrightarrow{a} P', Q \xrightarrow{b} Q'}{P \oplus Q \xrightarrow{b} Q'} \ (a \neq b) \tag{5.5}$$

Notice that, although Rules (5.5) must capture an equivalent pattern as those defined by Roscoe's (Rules (3.5)), we must consider the extra restrictions that we have two events ready to engage and that these events are different from each other. Otherwise, Rules (5.4) and (5.5) could be applied non-deterministically.

Rules (5.4) and (5.5) are the most important. Rule (5.4) captures the behaviour of parallelism, so it describes the synchronization of common events. And Rules (5.5), which have the same conclusion as the rule of external choice determine the occurrence of branches and are responsible for discarding the unchosen branch according to our preceding discussion.

5.2.2 Implementing the Merging Operator

The merging operator was implemented using two approaches: one approach based on LTS and another based on the CSP formalism. The reason for having two different unifying approaches is for underlying two kinds of test cases generation strategies inserted in our context. The first strategy [Car06] uses a formal model based on LTS, whereas the second strategy [Nog06] uses a CSP approach for generating test cases.

Although we have different kinds of implementations of the merging operator, they have a unique behaviour. The unified model generated from the approach based on the LTS formalism is an LTS model, whereas the unified model generated from the merging based on CSP is a CSP process.

The merging based on CSP uses functional aspects of CSP$_M$ and process operators, such as, parallelism, renaming, external choice, and hiding. The merging based on LTS implements algorithms is based on the Java language to perform the unification.

5.2.2.1 Merging Based on LTS

The merging approach based on LTS takes initial models specified in LTS diagrams and generates a unified model also in LTS. This approach can be performed either in one or two steps (see Figure 5.5). The first step is necessary when the initial models are initially specified in

CSP. Thus, this step is responsible for translating CSP models into LTS diagrams. For the translation step we use functionalities of the FDR *Explorer* 3.5.2. The result of this translation is the generation of LTS models equivalent to original CSP models. When all initial models are previously specified in LTS, this approach is performed by a unique step (second step of Figure 5.5).

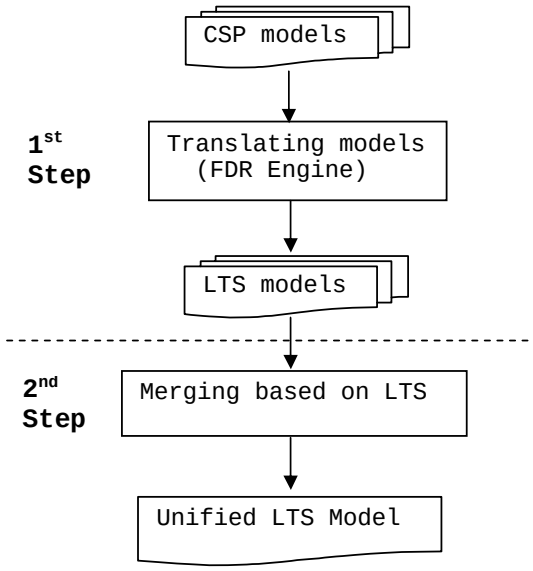


Figure 5.5 Merging approach based on LTS

For the second step of the merging based on LTS, we developed a tool in the Java language that implements the merging operator behaviour. The tool uses a graph model that represents the LTS models. The tool also implements procedures to handle graph operations, such as reading, modifying, searching, and writing. A white-box view of merging based on LTS can be shown in Figure 5.6.

In the next sections we describe the algorithms used to implement the merging based on LTS. Basically, we have three algorithms: *merging*, *split*, and *join*.

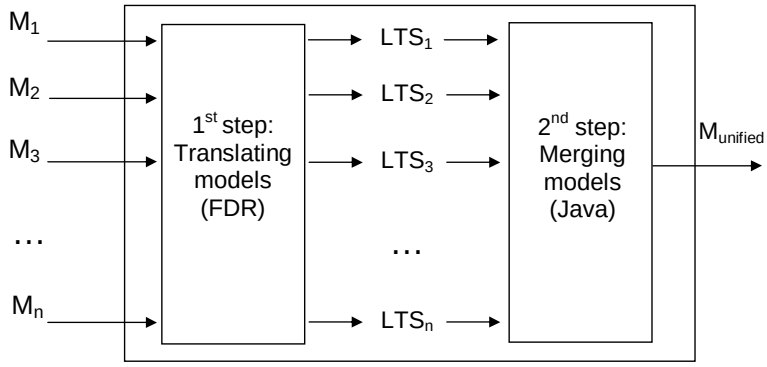


Figure 5.6 White-box view of merging models based on LTS approach

5.2.2.1.1 Merging Algorithm The main algorithm of merging based on LTS is the merging algorithm. As shown in Algorithm 1, the merging algorithm receives a list of LTS models ([$\mathtt{LTS}_1$, $\mathtt{LTS}_2$, ..., $\mathtt{LTS}_n$]) and returns a unified LTS model (variable $\mathtt{LTS}_{unified}$).

Algorithm 1: Merging Algorithm

Data: $LTS = (S, L, \triangle, q)$ is a LTS model, where S is a finite set of states, L is a set of labels that denotes the communicating alphabet, $\triangle \subseteq (S \times L \times S)$ defines the labelled transitions between states, and $q \in S$ is the initial state.

Input: List of LTS models, such as [LTS_1, LTS_2, ..., LTS_n]

Result: $LTS_{unified} = (S', L', \triangle, q')$, where $LTS_{unified} = LTS_1 \oplus LTS_2 \oplus ... \oplus LTS_n$

```
1  begin
2      LTS_unified ← LTS_1
3      for i ← 2 to n do
4          T ← transitions of LTS_i where q is start point
5          foreach t ∈ T do
6              s' ← new state
7              l ← label of t in LTS_i
8              t' ← new transition with (q × l × s')
9              add t' to root element (start point) of ExtractedLTSs
10         ExtractedLTSs ← split(LTS_i, ExtractedLTSs )
11     foreach lts ∈ ExtractedLTSs do
12         LTS_unified ← join(lts, LTS_unified)
13 end
```

The algorithm starts by initializing the variable $\mathtt{LTS}_{unified}$ with $\mathtt{LTS}_1$ (line 2). Next, for the other LTS models ([$\mathtt{LTS}_2$, ..., $\mathtt{LTS}_n$]) it extracts all possible start points and stores them in the variable `ExtractedLTSs`. The inner-most loop (lines 5 to 9) is responsible for extracting all start points of a given LTS ($\mathtt{LTS}_i$) and adding to `ExtractedLTSs`. A starting point is the root element of a LTS model, so following the root element it is possible to reach any other node on the model. Following the LTS notation, the starting points are all nodes which have the initial state as `q` (line 4). Figure 5.7 shows the variable `ExtractedLTSs` in line 10, after performing the inner-most loop to extract all starting points.

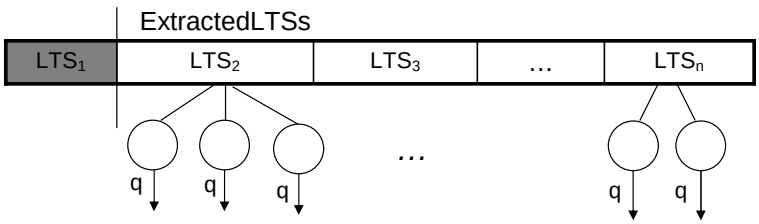


Figure 5.7 Variable *ExtractedLTSs* on line 10

As shown in Figure 5.7, for all LTS models it was created their starting point (root elements). For instance, for the LTS_2 was created three starting points, which mean that from the root element of LTS_2 it is, initially, possible to take three different paths. Notice that, the model LTS_1 was not considered, because it means the unified model a-priori.

5.2.2.1.2 Splitting paths Based on the original model and starting points, the variables $\texttt{LTS}_i$ and `ExtractedLTSs` respectively, the function `split` performs the paths splitting. To do this the function `split` follows the starting points on original model and builds all possible paths. All paths returned by `split` are recursively added in the variable `ExtractedLTS`, as shown in Algorithm 2.

Algorithm 2: Split Function

Input: A $LTS = (S, L, \triangle, q)$ model. A List of LTSs ($ListLTS = [LTS_1, LTS_2, ..., LTS_n]$) to be created recursively.

Result: A list of LTS models $ListLTS = [LTS_1, LTS_2, ..., LTS_n]$. For all $LTS = (S', L', \triangle', q') \in ListLTS$ we have $S' \subseteq S, L' \subseteq L, \triangle' \subseteq \triangle$, and for each $s \in S'$ there is only one transition $t \in \triangle'$ that involves $(s \times l \times s')$, where $s' \in S'$, $l \in L'$ and $q' = q$.

```
1  begin
2      foreach lts ∈ ListLTS do
3          s ← last state of lts
4          T ← transitions in LTS where s is start point
5          foreach t ∈ T do
6              s' ← new state
7              l ← label of t in LTS
8              t' ← new transition with (s × l × s')
9              add t' to ListLTS
10     if all paths of LTS were not visited then
11         recursive call in split(ListLTS)
12 end
```

As previously said, the function `split` receives an LTS model stored in the variable `LTS` and its starting points stored on `ListLTS` as parameter. As result it returns a list of paths of model `LTS` (`ListLTS` variable). The function `split` is a recursive algorithm for visiting all paths on model. Basically, for each starting point it adds the next transition in the variable `ListLTS`. The transitions are extracted from the original variable `LTS`. Each transition of `LTS` is recursively added to the list of simple LTS models (`ListLTS`). After the recursion, when all paths of LTS had been visited, the variable `ListLTS` is returned. Such a variable contains all paths extracted from `LTS`.

Notice that, for each original model ([LTS_2, ..., LTS_n]) it was create one or more paths. If there are loops in paths they are extracted and preserved. Figure 5.8 shows the variable `ExtractedLTSs` after splitting the paths of all original models ($\texttt{LTS}_i$).

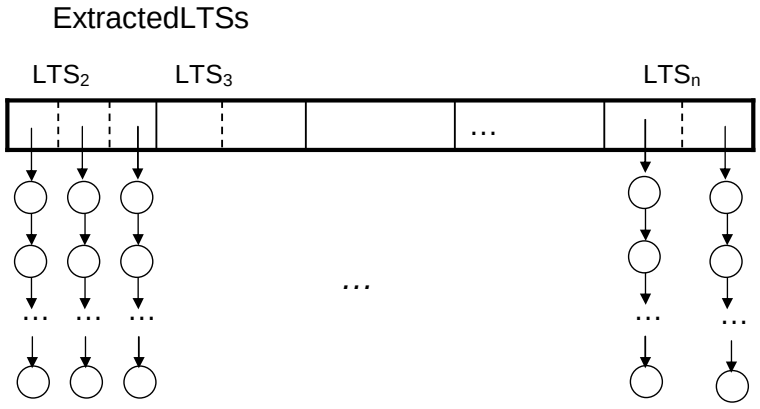


Figure 5.8 Variable *ExtractedLTSs* after splitting all paths

5.2.2.1.3 Joining paths After extracting all paths using the function `split`, the merging algorithm finishes taking each path stored in `ExtractedLTS` and joining with the $\text{LTS}_{unified}$ (lines 11 to 12 in Algorithm 1). The result of joining paths in the original model is the final unified model. Algorithm 3 shows the step of joining paths.

The function `join` receives two parameters, $\texttt{LTS}_1$ and $\texttt{LTS}_2$. Model $\texttt{LTS}_2$ represents the current unified model, whereas $\texttt{LTS}_1$ means a behaviour that must be added to the unified model. The purpose of function `join` is to compare transitions of $\texttt{LTS}_1$ and $\texttt{LTS}_2$. As previously discussed, following the transitions comparison, two possible situations can occur. First, both transition labels are similar ($l_1 = l_2$). In such a case, the current transition should be preserved. The second situation occurs when two transition labels are different. In the last case, a new branch with the tail of LTS_1 must be introduced in LTS_2. After all, we have a new LTS_2 with all transitions added from LTS_1.

5.2.2.2 Merging Based on CSP

The merging based on CSP is performed in a unique step. It takes each CSP initial models and uses them to compose the unified model. Initial models are represented as CSP processes. The composition strategy combines parallelism and functional aspects of CSP_M. The result is a unified CSP model that contains information provided from different models. Figure 5.9 shows the merging based on CSP.

The white-box view of merging based on CSP is shown on Figure 5.10. In this approach initial CSP models are given as input (for instance, M_1, M_2, ..., M_n). Each CSP model uses a specific channel to send its events (chM_1, ..., chM_n). In the merging based on CSP, all events of an input process are able to be sent through channels. For instance, the model M_1 sends all its events through channel chM_1.

Figure 5.10 shows the whole unification process. Notice that, we also have all initial models on the left side and the final result on the right side, moreover all intermediate unifications are being sequentially composed. That composition is done using parallel compositions with all involved processes: initial models and unification components (`UC`s). For each initial model there is a `UC` responsible for receiving its input events and performing the unification.

Algorithm 3: Join Function

Input: A $LTS_1 = (S, L, \triangle, q)$ model, where $s \in S$. A simple $LTS_2 = (S', L', \triangle', q')$ model, where $s' \in S'$.

Result: A unified LTS model $LTS_{unified}$ that contains all behaviour of LTS_1 and LTS_2.

```
1  begin
2      t2 ← transition in LTS2 where s' is start point
3      l2 ← label of t2 in LTS2
4      T ← transitions in LTS1 where s is start point
5      while there are transitions of LTS2 not visited do
6          foreach t ∈ T do
7              l1 ← label of t in LTS1
8              if l1 = l2 then
9                  keep transition t in LTS1
10                 sNext ← next state of transition t
11                 T ← transitions in LTS1 where sNext is start point
12             else
13                 create a new branch in LTS1 with the rest of LTS2
14                 return LTS1
15 end
```

A unification component can be internal ($\texttt{UC}_{INNER}$) or external ($\texttt{UC}_{OUTER}$). The $\texttt{UC}_{INNER}$ performs the unification and forwards its result to the next `UC` through an instance of channel `inner`. The last `UC` is always a $\texttt{UC}_{OUTER}$ that means its result will not be forwarded to another `UC`. The result of $\texttt{UC}_{OUTER}$ will represent the final unification of all initial models and the events are directly sent to the external environment. In the next sections, we detail each element contained in Figure 5.10.

5.2.2.2.1 List of models and unified model As Figure 5.10 suggests the merging based on CSP receives a list of CSP models and performs the unification. The list of models is given as a sequence of theirs indexes, so the list `Models = <1..n>` means a list of models M_1 to M_n. Each model is represented by an integer (1 to n), thus it is possible to specify instances of channels to each model.

5.2.2.2.2 MERGING_MODELS function The unified model is simply the result of the function `MERGING_MODELS` with all internal communications hidden (`\{|chM, inner|}`). Internal communications are used to forward events among initial models and intermediate unifications (`UC`).

```
UNIFIED_MODEL = MERGING_MODELS(Models)\{|chM, inner|}
```

The function `MERGING_MODELS` implements a recursive algorithm that receives a sequence of models (`MODELS = <1..n>`) and performs the models merging. As suggested

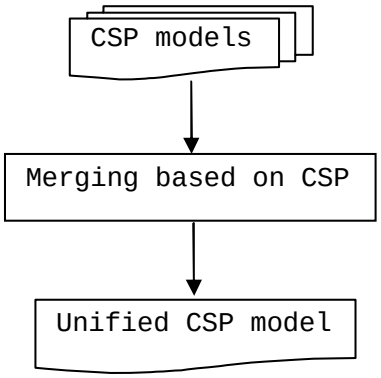


Figure 5.9 Merging based on CSP

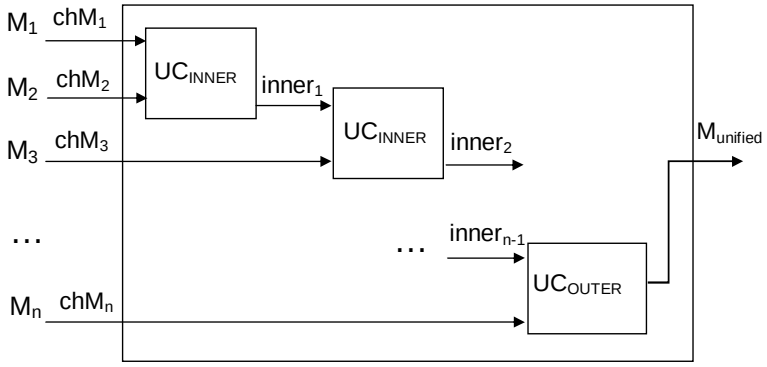


Figure 5.10 White-box view of merging based on CSP

in Figure 5.10, the algorithm is based on parallel communications with initial models (`M(1)`, `M(2)`, ..., `M(n)`) and a unification interface named `UCInterface`. The following specification shows the function `MERGING_MODELS`:

```
MERGING_MODELS(<m1>^<m2>^<>) =
  ( ( M(m1)
        [| {|chM.m1|} |]
      UCInterface(chM.m1, chM.m2, 1)
    )
      [| {|chM.m2|} |]
    M(m2)
  )

MERGING_MODELS(<m>^models) =
  let
    i = #(models)
  within
  ( ( MERGING_MODELS(models)
        [| {|inner.(i-1)|} |]
      UCInterface(inner.(i-1), chM.m, i)
```

```
    )
      [| {|chM.m|} |]
    M(m)
  )
```

It is worth observing that all initial models use instances of the channel `chM` to communicate their events, like `chM.1` to model 1, whereas intermediate unifications (`UCs`) use instances of channel `inner`. To do that we have to specify the id (1 to n) of the model involved in the unification.

5.2.2.2.3 chM and inner channels As previously said, each parallel communication uses a unique instance of channel for communicating, either channels `chM` or `inner`. All communications through `chM` and `inner` follows the syntax `chM.i` and `inner.i` where `i` is an index of the model related with the communication. A-priori, each initial model has an instance of the channel `chM`, for instance, the model 1 uses the instance `chM.1` to communicate its events. The internal channels (`inner`) involve communications between an initial model and an internal unification interface (`UCInterface`). The definition of all channels `chM` and `inner` are as follows:

```
N = #(Models)
channel chM, inner : {1..N}.Alphabet
```

Notice that, each model (1 to N) has an instance of channels `chM` and `inner`. This is necessary to inform the integer that identifies the id of the model. In the above definition, `Alphabet` means the events of all initial models. Here, we simply represent in this way for simplification, but in practical use we define those channels only with real events able to be performed by models. The following specification gives us an idea:

```
Alphabet = union(union(union( ...
    Alpha_MODEL_1,
    Alpha_MODEL_2),
    Alpha_MODEL_3),
    ...
    Alpha_MODEL_N)
```

The set `Alpha_MODEL_1` contains all events performed by `MODEL_1`, for instance. To define the set of events performed by each model we use the FDR *Explorer* tool. The FDR *Explorer* interacts with FDR and extracts all events contained in the CSP processes in a text file. It is only necessary to inform the desired processes and the file that contains the specification. For instance, using the CSP process defined in Figure 4.5, which specifies a test case, we have the set:

```
Alpha_TC_VIEW_AND_HIGHLIGHT_IN_OUTBOX = {
   setup,
   delete.DTDEL_ITEM.(MESSAGES, {ALL}),
```

```
   set.DTSET_ITEM.(VIEW_MESSAGES, {}).(BY_SUBJECT, {}),
   compose.DTWRITE_ITEM.(MESSAGES, {TWO}),
   send.DTSEND_ITEM.(MESSAGES, {TWO}).(NUMBER, {VALID}),
   goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}),
   verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {}),
   test,
   highlight.DTSELECT_ITEM.(MESSAGE, {FIRST}),
   highlight.DTSELECT_ITEM.(MESSAGE, {SECOND}),
   goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}),
   verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {}),
   cleanup,
   delete.DTDEL_ITEM.(MESSAGES, {ALL}),
   set.DTSET_ITEM.(VIEW_MESSAGES, {}).(BY_ADDRESS, {}),
   _tick
}
```

Combining all alphabets we get only the (common) necessary events. That data refinement was used considering performance aspects.

5.2.2.2.4 UCInterface definition The interface component, named `UCInterface`, is responsible for initializing all internal unification components (UC_{INNER} and UC_{OUTER}). It receives three parameters, two channels and one natural number. The two channels given as parameter are used in parallel communications between the processes. The communication can involve initial models through channels `chM` and also involve unification components through channels `inner`. The natural number given as the third parameter is used to identify which initial model is being involved in the communication. The definition of `UCInterface` is as follows:

```
UCInterface(ch1, ch2, 1) = UC(INNER, pNON, ch1, ch2, inner.1)

UCInterface(ch1, ch2, i) =
  if (i == (N-1)) then
     UC(OUTER, pNON, ch1, ch2, inner.i)
  else
     UC(INNER, pNON, ch1, ch2, inner.i)
```

The `UCInterface` function is responsible for initializing the main component, the *unification component (UC)*. `UC` implements the logic of the merging operator. As we saw, an `UC` can be internal or external, and the difference is done by its first parameter. An internal `UC` has its first parameter as type `INNER`. Such distinguishing is very important, because an internal `UC` uses only internal channels (`inner`) to communicate its final events to other `UC`, whereas an external `UC` (type `OUTER`) means the last unification component. The last `UC` provides its events to the next `UC` and also gives its events as final output. The result of the last `UC` means the result of the final unification.

According to the function `MERGING_MODELS`, when `UCInterface` is called by the first time, the value of the third parameter `i` is equal to `#(Models) - 1`. That represents the

index of the last UC (UC_{OUTER}), being initialized. In another case, when the third parameter of UCInterface is 1 we have the first UC initialized (UC_{INNER}). Then, what the functions MERGING_MODELS and UCInterface do is to initialize all unification components sequentially in a reverse way, starting from the last one to the first one. Figure 5.10 shows the result of the MERGING_MODELS function together with the function UCInterface.

5.2.2.2.5 OUTER and INNER datatypes As previously said, the types OUTER and INNER are used only to distinguish between an internal and external UC. Its definition is presented as:

```
datatype TYPE = INNER | OUTER
```

5.2.2.2.6 Unification Component – UC The representation of UC_{INNER} is analogous to UC_{OUTER}, changing only the first parameter, which defines its type. Thus, for simplification, we will only show the specification of UC_{INNER}. The component UC is detailed as:

```
UC(INNER, pNON, chLeft, chRight, chInner) =
 ( chLeft?ev1 -> chRight?ev2 ->
     fr(INNER, pNON, ev1, ev2, chLeft, chRight, chInner)
 []
   chRight?ev2 -> chLeft?ev1 ->
     fr(INNER, pNON, ev1, ev2, chLeft, chRight, chInner)
 )

UC(INNER, pLEFT, chLeft, chRight, chInner) =
 ( chLeft?ev ->
     func_create(INNER, pLEFT, ev, chLeft, chRight, chInner)
 )

UC(INNER, pRIGHT, chLeft, chRight, chInner) =
 ( chRight?ev ->
     func_create(INNER, pRIGHT, ev, chLeft, chRight, chInner)
 )
```

As saw, UC has five parameters. Its logic is directly related to two first parameters. The first parameter means the type of UC. As already explained, the type of UC distinguishes the unification components in internal or external. The second parameter is a type named PATH.

The other parameters of UC are communication channels. Two channels are used to receive events from the left and right sides, channels chLeft and chRight, respectively. The third channel (chInner) is used to send the result of an intermediate unification to the next UC. In this situation when UC is an external component, its state is defined as OUTER, this channel is not used. In such a case, its events are directly sent to the external environment. As we discussed, this occurs because all events generated by the last UC are final events of the unification process.

As shown previously, the action of UC is guided by the firing rules of the merging operator. In CSP, we implemented all firing rules of the merging operator as functions, where each function deals with a specific behaviour of the merging operator.

5.2.2.2.7 PATH datatype The PATH type indicates which path UC is currently handling. The action taken by UC is directly related to the type PATH. The definition of this type is as follows:

```
datatype PATH = pNON | pLEFT | pRIGHT
```

The value pNON, in the first definition of UC, indicates that UC does not did have a defined direction yet. So, UC is receiving events from processes in its left and right side. To determine that UC deals only with a specific process either in its left or right side, we must specify using pLEFT or pRIGHT, respectively. This is necessary because UC needs a direction (left or right) to work properly. The left path means the events sent to the UC by the left process, whereas the right side represents the events sent by the right process.

When UC is in pNON state, that means its actions depend on the firing rules. In other cases when the UC state is pLEFT or pRIGHT the actions are executed by UC. Thus, the state pLEFT indicates that UC must only receive events from its left side, then forward them to the next internal component (channel inner) or send them directly to the external environment if UC is an external component. Another action taken by UC while handling its left side is to ignore all events from the right side. This same analogy is done for state pRIGHT. The last pattern matchings of UC represent such situations.

The action of UC to ignore events from a specific side was a mechanism used to satisfy the distributed termination of a parallel composition. Distributed termination means that the whole process only terminates when all its involved processes also terminate. In later discussions, we show the implementation of distributed terminated that we used.

5.2.2.2.8 Firing rules definition The base case of the merging operator is to handle the distributed termination. This behaviour is described through Rules (5.1), (5.2), and (5.3). In CSP, the termination occurs when the UC receives a termination event (event tick) from any side. When UC must receives an event tick from a specific side, it behaves like process SKIP in that side. Moreover it must generate all other events from the other side, afterwards it behaves like SKIP. To restrict UC generating events from a unique side, it is just necessary to define the path using pRIGHT or pLEFT. The following firing rules describe this situation.

```
fr(type, path, tick, ev2, chLeft, chRight, chInner) =
 (  SKIP
  []
    UC(type, pRIGHT, chLeft, chRight, chInner)
 )

fr(type, path, ev1, tick, chLeft, chRight, chInner) =
 (  UC(type, pLEFT, chLeft, chRight, chInner)
  []
    SKIP
 )
```

Notice that, an event tick is being used to determine that UC received a termination event from a specific side. As we saw, if the path of UC is pNON, all its received events are sent

to firing rules (function `fr`) using the third and forth parameter of function `fr`. The third parameter means the events received from `UC` through its left side. Additionally, the forth parameter of `fr` contains the events received from `UC` through its right side.

When `UC` receives two equal events from two processes it must unify them into a unique event. This behaviour is described by Rule (5.4). The following firing rule implements this idea.

```
fr(type, path, ev, ev, chLeft, chRight, chInner) =
  if (type == INNER) then
    chInner.ev -> UC(INNER, path, chLeft, chRight, chInner)
  else
    ev -> UC(OUTER, path, chLeft, chRight, chInner)
```

The previous pattern matching is used to determine when `fr` receives two equal events (`ev`). Moreover, the condition sentence is necessary to determine which type of unification is being considered, an internal unification (type is equal to `INNER`) or an external one when the type is equal to the `OUTER`. Thus, notice that, when the type of `UC` is `INNER` all events are sent to the next `UC` through channel `inner`, otherwise its events are sent to the external environment directly. The idea is to hide all internal communications (`chM` and `inner`), so only the external events sent by UC_{OUTER} will predominate.

The last firing rule deals with distinguished events. As shown by the behaviour of firing rules 5.5, an external choice must be introduced in this point to create distinguished branches. Again, notice that it is necessary to determine what is the type of unification, and the pattern matching is responsible for determining distinguished events.

```
fr(type, path, ev1, ev2, chLeft, chRight, chInner) =
  if (type == INNER) then
  (
    (  chInner.ev1 -> UC(INNER, pLEFT, chLeft, chRight, chInner)
     []
       chInner.ev2 -> UC(INNER, pRIGHT, chLeft, chRight, chInner)
    )
  ) else (
    (  ev1 -> UC(OUTER, pLEFT, chLeft, chRight, chInner)
     []
       ev2 -> UC(OUTER, pRIGHT, chLeft, chRight, chInner)
    )
  )
```

The previous firing rule determines the branches in the unified model. This is done by introducing an external choice between two distinguished events. Each branch of the external choice is responsible for dealing with only one direction. This is done by calling `UC` with its second parameter specified either as `LEFT` or `RIGHT` path, which it will receive events from a unique side.

5.2.2.2.9 func_create and func_ignore definitions As previously shown, when UC is called to deal with a specific path, it simply receives all events for that path and creates it. The function func_create has two purposes. The first purpose is to create an event to be sent through channel INNER (when UC is an internal component) or create an event to be sent directly to the external environment (when UC is an external component). The event to be created is sent by UC through parameter ev. This first purpose is described in the following specification, particulary by two last matchings of func_create.

```
func_create(INNER, path, ev, chLeft, chRight, chInner) =
  chInner.ev -> UC(INNER, path, chLeft, chRight, chInner)

func_create(OUTER, path, ev, chLeft, chRight, chInner) =
  ev -> UC(OUTER, path, chLeft, chRight, chInner)
```

The other purpose of func_create is to determine the mechanism used to implement the distributed termination in parallel compositions. The distributed termination occurs when UC is handling a specific side and receives an event tick. The idea used to implement the distributed termination is simple. First, UC waits to receive only events from a specific side (either by its left or right side), and when the termination event (tick) is sent from a specific side, by the left side for example, UC sends a tick to the function func_create. In that time, the func_create does not create a new event, it ignores all other events from the other side, by its right side in this example. This is necessary to satisfy the distributed termination. Such a behaviour is described by the two first matchings of function func_create.

```
func_create(type, pLEFT, tick, chLeft, chRight, chInner) =
  func_ignore(chRight, chInner)

func_create(type, pRIGHT, tick, chLeft, chRight, chInner) =
  func_ignore(chLeft, chInner)
```

The other parameters of function func_create are only to keep the state of UC and also inform to the function func_ignore the channel used to receive all events that must be ignored.

The mechanism to implement the distributed termination uses another auxiliary function, the function func_ignore. As suggested, the purpose of function func_ignore is to ignore all events sent by a specific channel (ch parameter). The idea of function func_ignore is to receive all sent event from channel ch (ch?ev) and ignores them. When an event tick is received, the function func_ignore communicates it through a channel inner and behaves like SKIP. At this point, we have all processes as SKIP and the distributed termination satisfied. Rule (5.3) describes this behaviour. The code of func_ignore is shown as following.

```
func_ignore(ch, chInner) = ch?ev ->
  (
    if (ev == tick) then
      chInner.tick -> SKIP
    else
      func_ignore(ch, chInner)
  )
```

5.2.2.2.10 Renaming operations Finally, the last part of the merging based on CSP is the rewriting or renaming of all initial models involved in the unification process. This is used to implement the idea of all processes sending their events to `UC`. The rewriting is done using the CSP renaming operator. All initial models will use a specific instance of channel `chM`. Using the instance of channel `chM` all events of the initial model will be renamed to a pattern like `chM.1.a`, in this case the event `a` belongs to model 1, for instance. In what follows, we have the rewriting of three models (`MODEL_1`, `MODEL_2`, and `MODEL_3`).

```
Alpha_MODEL_1 = {
    delete.DTDEL_ITEM.(MESSAGES, {ALL}),
    set.DTSET_ITEM.(VIEW_MESSAGES, {}).(BY_SUBJECT, {}),
    compose.DTWRITE_ITEM.(MESSAGES, {TWO}),
    send.DTSEND_ITEM.(MESSAGES, {TWO}).(NUMBER, {VALID}),
    goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}),
    verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {}),
  ...
}

M(1) = (MODEL_1 [[ ev <- chM.1.ev | ev <- Alpha_MODEL_1 ]]);
            chM.1.tick -> SKIP

Alpha_MODEL_2 = {
    delete.DTDEL_ITEM.(MESSAGES, {ALL}),
    highlight.DTSELECT_ITEM.(MESSAGE, {FIRST}),
    highlight.DTSELECT_ITEM.(MESSAGE, {SECOND}),
    goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}),
    verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {},
  ...
}

M(2) = (MODEL_2 [[ ev <- chM.2.ev | ev <- Alpha_MODEL_2 ]]);
            chM.2.tick -> SKIP

Alpha_MODEL_3 = {
    delete.DTDEL_ITEM.(MESSAGES, {ALL}),
    set.DTSET_ITEM.(VIEW_MESSAGES, {}).(BY_SUBJECT, {}),
    highlight.DTSELECT_ITEM.(MESSAGE, {SECOND}),
    goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}),
  ...
}

M(3) = (MODEL_3 [[ ev <- chM.3.ev | ev <- Alpha_MODEL_3 ]]);
            chM.3.tick -> SKIP
```

Notice that, the renaming is done using function `M`, so in this way to get the renaming of `MODEL_1` we simple call `M(1)`. Moreover, for each renaming it was necessary to inform the

set of all events of the model, such as `Alpha_MODEL_1`. Recall from the definition of channels `chM` and `inner` that we said that the term `Alphabet` was being used only for representing the alphabet of all involved processes, but in practical terms (performance requirements) we need to inform which specific events the initial models contain.

5.2.3 Validating the merging approaches

To validate both merging approaches (based on LTS and CSP) let's consider a single and abstract example. In this example, we are considering a unification process involving only four models (`MODEL_1`, `MODEL_2`, `MODEL_3`, and `MODEL_4`). All the following initial models are abstract models; their events do not occurs in a real case. Here, they are being used only for demonstration.

```
MODEL_1 = evA -> evB -> evC -> evD -> SKIP

MODEL_2 = evA -> evB -> evE -> evF -> evG -> SKIP

MODEL_3 = evC -> evH -> evI -> SKIP

MODEL_4 = evC ->
  (  evH -> evJ -> evK -> SKIP
   []
     evL -> evM -> SKIP
  )
```

The first step is to determine which merging approach to use: merging based on LTS or CSP. Here, we will show the results in both approaches.

5.2.3.0.11 Using the Merging Based on LTS The merging based on LTS is more direct than the merging based on CSP. As we do not have the initial models in LTS, so what we need is first translating all initial models from CSP to LTS. To do that, we used the FDR *Explorer* tool. The LTS models format is defined following a generic format.

```
is
{is_1 t_1 fs_1} {is_2 t_2 fs_2} {is_3 t_3 fs_3} ... {is_n t_n fs_n}
lt_1=t_1 lt_2=t_2 lt_3=t_3 ... lt_n=t_n
```

Basically, the generic format is composed by three lines. The first line specifies the initial state (`is`). Then, it follows the transitions (`{is_1 t_1 fs_1}`), and finally we have the transition labels. For a transition like `{is_1 t_1 fs_1}`, we have `is_1` that means initial state 1, `t_1` as transition 1, and `fs_1` meaning final state 1. Finally, transition labels are specified as `lt`. Thus, for `MODEL_1` we have the following LTS model in generic format.

```
0
{0 2 1} {1 3 2} {2 4 3} {3 5 4} {4 1 5}
_tau=0 _tick=1 evA=2 evB=3 evC=4 evD=5
```

The generic format have to be translated into a specific format. The literature presents some specific formats to represent LTS models. One of them is the *Trivial Graph Format* or TGF [yWo]. We chose this format to implement our semantic approach. Following the TGF format an LTS model is represented as,

```
s_1 ls_1
s_2 ls_2
s_3 ls_3
...
s_n ls_n
#
is_1 fs_1 lt_1
is_2 fs_2 lt_2
is_3 fs_3 lt_3
...
is_n fs_n lt_n
```

Using the TGF format an LTS model is represented by two sections. The first section gives the states (`s_1` is the state 1, for instance) and state labels (`ls_1` is the label of `s_1`). The second section contains the transitions. Each transition is a format like `is_1 fs_1 lt_1`, where `is_1` is the initial `s_1`, `fs_1` is final `s_1`, and `lt_1` means the label of transition 1. The sections are divided by a special symbol (`#`).

To translate from the generic format into the TGF format we implemented a Java program that takes an LTS model in the generic format and translates it into the TGF format. Then, using this program to translate the LTS from `MODEL_1` we have:

```
0 root
1 1
2 2
3 3
4 4
5 5
#
0 1 evA
1 2 evB
2 3 evC
3 4 evD
```

Using the TGF format it is possible to visualize the LTS model graphically [yWo]. Figure 5.11 shows the LTS in TGF format generated from `MODEL_1`.

After translating all initial models into initial LTS model in TGF format, we can accomplish the merging using the merging algorithm implemented in Java. The result of merging the four initial models is a unified LTS model in TGF format as shown in Figure 5.12.

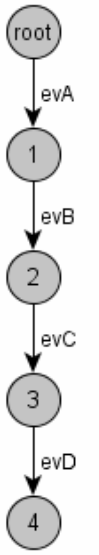


Figure 5.11 Viewing model *MODEL_1* in TGF format

5.2.3.0.12 Using Merging Based on CSP As previously shown, the merging approach based on CSP is performed in a unique step (see Figure 5.9), but it is not so direct as the merging based on LTS. Initially, we have to define the alphabets of initial models. So, using the FDR *Explorer* tool we got four set of alphabets:

```
Alpha_MODEL_1 = {evA, evB, evC, evD}

Alpha_MODEL_2 = {evA, evB, evE, evF, evG}

Alpha_MODEL_3 = {evC, evH, evI}

Alpha_MODEL_4 = {evC, evH, evJ, evK, evL, evM}
```

Using the alphabets of all initial models we can rename the models to allow that they can send their events as data. The renaming of all initial models is:

```
M(1) = (MODEL_1 [[ ev <- chM.1.ev | ev <- Alpha_MODEL_1 ]]);
            chM.1.tick -> SKIP

M(2) = (MODEL_2 [[ ev <- chM.2.ev | ev <- Alpha_MODEL_2 ]]);
            chM.2.tick -> SKIP

M(3) = (MODEL_3 [[ ev <- chM.3.ev | ev <- Alpha_MODEL_3 ]]);
            chM.3.tick -> SKIP

M(4) = (MODEL_4 [[ ev <- chM.4.ev | ev <- Alpha_MODEL_4 ]]);
            chM.4.tick -> SKIP
```

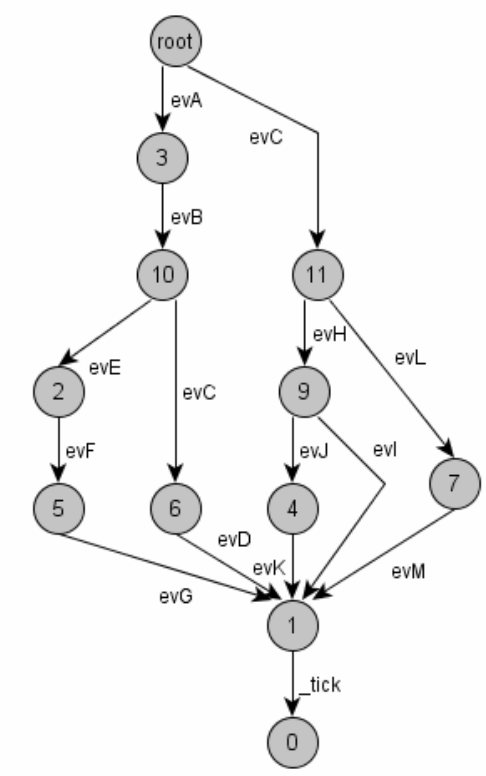


Figure 5.12 Unified model from 3 use cases using merging based on LTS

After renaming all initial models we can configure the other merging components. First we have to define the list of models involved in the unification processes (set `Models`).

```
Models = <1..4>
N = #(Models)
```

Later, we have to define the communication channels. They are the channels `chM` and `inner`.

```
channel chM, inner :
 {1..N}.union(union(union(
    Alpha_MODEL_1,
    Alpha_MODEL_2),
    Alpha_MODEL_3),
    Alpha_MODEL_4)
```

Finally, the unified model is simply the calling of function `MERGING_MODELS` hiding all internal communications (`\{|chM, inner|}`).

```
UNIFIED_USE_MODEL = MERGING_MODELS(Models)\{|chM, inner|}
```

The expansion of function `MERGING_MODELS` applied to the four initial models can be viewed as:

```
USE_MODEL_1 =
 ( ( M(1)
       [|{|chM.1|}|]
     UC(INNER, pNON, chM.1, chM.2, inner.1)
    )
     [|{|chM.2|}|]
    M(2)
  )

USE_MODEL_2 =
  ( ( USE_MODEL_1
       [|{|inner.1|}|]
     UC(INNER, pNON, inner.1, chM.3, inner.2)
    )
     [|{|chM.3|}|]
    M(3)
  )

USE_MODEL_FINAL =
  ( ( USE_MODEL_2
       [|{|inner.2|}|]
    UC(OUTER, pNON, inner.2, chM.4, inner.3)
    )
     [|{|chM.4|}|]
    M(4)
  )

UNIFIED_USE_MODEL = USE_MODEL_FINAL \{|chM, inner|}
```

We can also visualise the LTS generated from process `UNIFIED_USE_MODEL` simply executing the FDR *Explorer*. Figure 5.13 shows the unified model in LTS after translating the from generic format to TGF format.

5.2.4 Analysing the Merging Properties

After modelling and implementing the merging operator, we have to define which properties it must satisfy. Following the basic mathematical properties, such as identity, associativity and commutativity, we analyzed whether the merging operator satisfies these properties. The prove of the merging properties is not described here. As future work we propose the prove of such properties.

5.2.4.1 Idempotent

This property defines a relation where a neutral or identity element is used in a operator, but the result does not have influence. In mathematics, an identity element (or neutral element) is a special type of element of a set with respect to a binary operation on that set. It leaves other

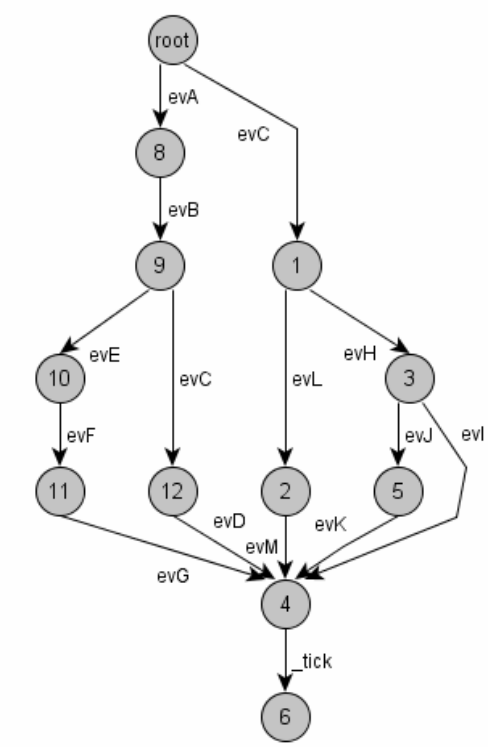


Figure 5.13 Unified model from 3 use cases using merging based on CSP

elements unchanged when combined with them.

Let's consider a par $(S, *)$, where S is a set with a binary operation $*$ on it. Then, an element e of S is called a left identity if:

$$e * a = a \bullet \forall\, a \in S. \tag{5.6}$$

A right identity occurs if:

$$a * e = a \bullet \forall\, a \in S. \tag{5.7}$$

Finally, if e is both a left identity and a right identity, then it is called a two-sided identity, or simply an identity.

Considering the formal definition of identity property applied to a par $(Models, \oplus)$, where $Models$ is a set of models and $\oplus$ a binary merging operation, we have:

$$m \oplus m = m \bullet \forall\, m \in Models. \tag{5.8}$$

The element m of set $Models$ is called a identity (by left and right side). Thus, for the merging operator we can consider that the identity element is the involved own model.

5.2.4.2 Associativity

Associativity means that the order of evaluation is irrelevant if the operation appears more than once in an expression.

Formally, a binary operation $*$ on a set S is called associative if it satisfies the associative law:

$$(x * y) * z = x * (y * z) \bullet \forall\, x, y, z \in S. \tag{5.9}$$

The evaluation order does not affect the value of such expressions, and it can be shown that the same holds for expressions containing any number of $*$ operations. Thus, when $*$ is associative, the evaluation order can therefore be left unspecified without causing ambiguity, by omitting the parentheses and writing simply:

$$x * y * z \tag{5.10}$$

Applying the associativity definition to merging operator we have:

$$(m_1 \oplus m_2) \oplus m_3 = m_1 \oplus (m_2 \oplus m_3) \bullet \forall\, m_1, m_2, m_3 \in Models. \tag{5.11}$$

Although we did not prove the associativity of the merging operator, we argue that it is associative because it is based on the parallelism properties. In the parallelism when the involved processes share the same synchronization set we have the associativity property satisfied. Otherwise, we can not assure that such property is valid. In the merging approach all events sharing the same communication channel (`chM`), so all involved processes share the same alphabet of events. Thus, based on associativity of the parallelism we can assure that the merging operator also satisfies the same property.

Based on associativity definition we can summarize the above equation to:

$$m_1 \oplus m_2 \oplus m_3. \tag{5.12}$$

5.2.4.3 Commutativity

That property says that order of the numbers to be operated does not affect the result. For instance, the addition operation is commutativity because the sum of two numbers instead of numbers order will have the same result, so the equation $2+3=3+2$ is true whenever.

In general, for a binary operation $f : A \times A \rightarrow B$ is said to be commutative, when the equation $f(y,z) = f(z,y)$ is preserved, for any y and z in A. Otherwise, the operation is noncommutative.

Only some binary operations have this property. For instance, it holds for the integers with addition and multiplication, but it does not hold for matrix multiplication. In our case, for the merging operator we have this property accomplished like:

$$m_1 \oplus m_2 = m_2 \oplus m_1 \bullet \forall\, m_1, m_2 \in Models. \tag{5.13}$$

The commutativity property can be assure in the merging operator by taking as basis the parallelism and external choice operators. The commutativity to the parallelism can be shown as:

$$P_1||P_2 = P_2||P_1 \bullet \forall\, P_1, P_2 \;\; as \;\; CSP \;\; processes. \tag{5.14}$$

and for the external choice we have,

$$P_1[]P_2 = P_2[]P_1 \bullet \forall\, P_1, P_2 \;\; as \;\; CSP \;\; processes. \tag{5.15}$$

In those operators we always have the commutativity satisfied, and as the merging operator contains both behaviours (parallelism and choice) we can say that the merging also satisfies the commutativity property.

5.3 Related Works

The notion of unifying system models has been used in other works. Many of them characterise the unification process through a merging operation as well [BCE+06]. Model merging has been studied in many domains. The main works have focused on integration of different database schemes, requirements models, and components of reactive systems. In our work, we deal with models which describe system actions. The actions are captured using requirement documents, system test cases, and even interactions with the own application. In the following sections we will discuss the main approaches which handle the problem of unifying model.

5.3.1 Viewpoints based models

Some approaches for unifying system models are strictly related to the requirements elicitation process [EC01]. They argue that in a distributed modeling task it will frequently be necessary to merge models developed by different teams [BCE+06]. The goal is to describe a complete view of the system using different and complementary information. This information needs to be unified, but first they need to resolve their inconsistencies. These inconsistencies are usually occasioned by stakeholders who often hold different views of how a proposed system should behave. These approaches are based on the concept of *viewpoints-based* [EC01].

Viewpoints-based approaches have been proposed as a way of managing inconsistent and incomplete information gathered from multiple sources [FGH+93]. They use a sort of formalism able to specify different parts of the system separately. Even an earlier behaviour which the team does not have many information can be modelled. Then, using such a formalism it is possible to separate the descriptions provided by different stakeholders, and concentrate on identifying and resolving conflicts between them [EC01]. This kind of formalism is different from classical logic models which usually specify the system as state machine models.

Given an inconsistent set of viewpoints, it would be useful to determine how the inconsistencies affect system properties. Some inconsistencies may have few consequences, and do not need to be resolved, while others may be the result of disagreements about how a system should behave. For example, if the stakeholders' descriptions are modelled as state machines, they may disagree about the details of some states and transitions, without affecting the values of global temporal properties. If the models were consistent and complete, they could be

verified using model-checking. Unfortunately, existing model checkers are based on classical logic, and so cannot cope with inconsistent (and incomplete) models.

The work proposed by [EC01] describes a framework for merging multiple inconsistent state machine models. The framework for merging viewpoints uses multi-valued logics to give support with inconsistent aspect between models. Each logic value has a number of different possibilities, which are true, false, and unknown. The framework includes a multi-valued model checker for reasoning about the temporal properties of inconsistent state machine models.

5.3.2 Partial-behaviour models

Other approaches for merging of requirements models are well similar for view-points based. They have been developed to solve the problem when they do not have a complete comprehension of the system. In such cases, the designers have partial descriptions of the system. Such approaches state classical state-based modeling techniques are generally not suited for providing early feedback, when system descriptions are still partial [UC04]. Following this idea, other notations have been useful for describing partial behavioural descriptions. As an instance we have scenario-based notations, which are partial behavioural descriptions that promote incremental elaboration of system behaviour.

In such works, they state that the modelling and analysis using state-based behaviour descriptions has been shown more successful in tasks of detaching design errors. Lately, researches have been tried to integrate approaches that use partial descriptions and state-based modeling techniques [UC04]. In particular, several approaches of synthesis of state-based models from scenarios-based specifications (e.g. [UKM03b, KRPM99]) have been developed. These approaches aim to combine the benefits of the incremental elaboration in interaction-based specifications with the behavioural analysis in state-based models.

They argue that the best way for modelling is a type of state-based model able to represent unknown aspects of behaviour [UC04]. Using models it is possible to distinguish between positive, negative and unknown behaviours. A positive behaviour refers to a behaviour that the system should perform. The negative behaviour indicates a behaviour that the system is expected to never perform. Even the unknown behaviour could become positive or negative, but the choice has not yet been made.

State-based models that distinguish between these kinds of behaviour are referred to as partial behavioural models, e.g., Partial Labelled Transition Systems (PLTS) [UKM03a], multi-valued state machines [DRPAFV02], Modal Transition Systems (MTSs) [LT88], Mixed Transition Systems [Dam96] and multi-valued Kripke structures [CDEG03].

The work presented in [UC04] introduces the notion of merging in the context of an adaptation of MTSs - Modal Transition Systems [CDEG03]. It argues that the core concept underlying model merging is that of common observational refinement and define consistency as the existence of a common observational refinement.

The notion of adapting MTSs models is based on the idea of eliminating inconsistencies between partial models. Such inconsistencies are characterized by either a couple of transition positive and negative or a unique unknown transition. The idea of observational refinement is the key of such an adaptation. It describes an observational refinement as a refinement that ignores differences in internal transitions (*_tau*), and solving unknown transition to either

positive or negative transition.

5.3.3 State-based models

Other approaches for models merging handle state-based models directly. Such approaches deal with specifications modelled by transition systems. These approaches state that system specifications may be enriched by adding new behaviours required by the user, such as adding new functionality to a given system specification. Then different system specifications may be integrated, but inconsistences are not so easy of detaching.

They argue that the semantic properties and the behaviour of the given system specifications should be preserved. The idea of preserving semantic properties may, for instance, mean that the combined specification exhibits at least the behaviour of each single specification without introducing additional failures for these behaviours. This is captured by formal relation between specifications, called extensions, introduced in [BSS95]. Informally, the behaviour specified by `A` specification extends the behaviour of `B` specification, if and only if, `A` allows any sequence of actions that `B` allows, and `A` can only refuse what `B` refuse, after a given sequence of actions allowed by `A`.

The work presented in [KB95] considers a merging approach for two models of transition systems, the Acceptance Graphs (AG), and the Labelled Transitions Systems (LTS). The AG notation is underlined by notations of Acceptance Tree of Hennessy and Tgraphs. In that approach the merging of behaviour specifications is defined in AG models, which are more tractable mathematically than LTS models. The merging of LTS is based on the merging of AG and relies on a correspondence between LTS and AG.

There are also works which handle the problem of model merging by integrating system components. The work presented in [KB94] proposes an incremental construction approach for distributed system specifications using LTS. These specifications are structured as a parallel composition of subsystem specifications. The approach consists of merging two specifications into a new specification, such that the new specification is a extension of the original ones. This work is complementary to work in [KB95]

Other works use a formalism based on process algebras, such as CCS, CSP, and LOTOS for integrating system components. Models in process algebras are also well characterized by state machine diagrams. The work presented in [Sha85] shows a technique which allows the merging of two processes in CSP specification into a new single process. It states that such a technique can be useful for distributed program, more specifically it can be used to reconfigure a distributed program after a faulty processing element has been detected. In that work, a process merging algorithm which operates on distributed programs using the CSP notation is presented in detail and its operation is discussed.

CHAPTER 6

Test Case Reverse Tool (*TCRev Tool*)

This chapter describes the strategy automation used to implement the previously approaches. Thus, the strategy automation involves the approaches for modelling test cases and unifying formal models.

Section 6.1 introduces the *TCRev* Tool. This tool is responsible for reversing test cases, building and unifying formal models. Section 6.1.1 gives an overview of available functionalities. Next, in Section 6.1.2 we discuss some architecture issues related to tool design. Finally, in Section 6.1.3.1 we summarize another tool *TaRGeT*, which is been developed by the Research Team and it is inserted in our same context for generating and specifying of test cases and requirements.

6.1 The TCRev Tool

The tool TCRev (acronym for Test Case Reverse) is responsible for specifying (modelling) automatic test cases into abstract or formal test cases in LTS and CSP formalisms. Beyond generating abstract tests the tool is also responsible for taking all abstract tests and building a complete formal model. Because of both purposes, the TCRev tool's name makes allusion to a reverse step, which starts from test cases and builds a formal model. Recall from Chapter 2, first we have a built formal model, then abstract test cases are generated, and finally, based on abstract test cases, real test cases are generated and executed in the SUT.

6.1.1 Overview

Basically, the TCRev tool has three functionalities: model real test cases into formal (abstract) ones, unify or compose formal models (abstract test cases), and generate real test cases (automatic test cases) based on abstract ones. All functionalities as described in Figure 6.1.

As represented in Figure 6.1, we have all three basic functionalities related among them by a sequential flow. The flow is represented through direct arrows, where the arrows' source means the input artefacts and arrows' top means the output generated from previous task. The input and output artefacts are also symbolized. Notice that, in Figure 6.1 we made distinction between one artefact or many artefacts.

In the next sections, we will discuss in a more detailed view the above flow by showing each single task and how they are related to each other. In the following tasks we have interactions with other external sub-systems, for instance, the task of modelling test cases interacts with the external sub-system responsible for processing sentences in natural language. Thus, whenever

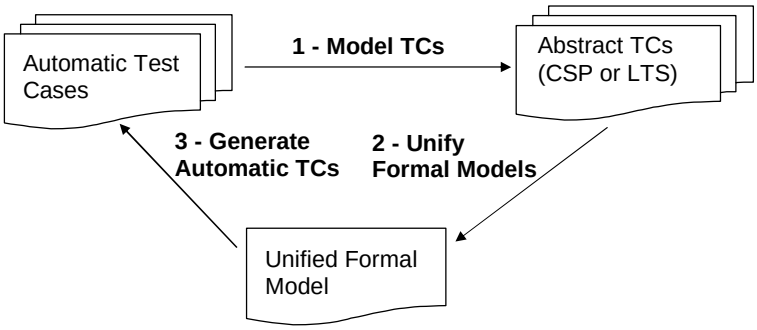


Figure 6.1 TCRev Tool overview

occurs an external interaction we explicitly highlight such an interaction, and, moreover, we inform how such an interaction is related to our system.

6.1.1.1 Task 1 - Model concrete test cases as abstract test cases

The first functionality or task is responsible for modelling real test cases to abstract (formal) ones. Figure 6.2 shows us a more detailed view.

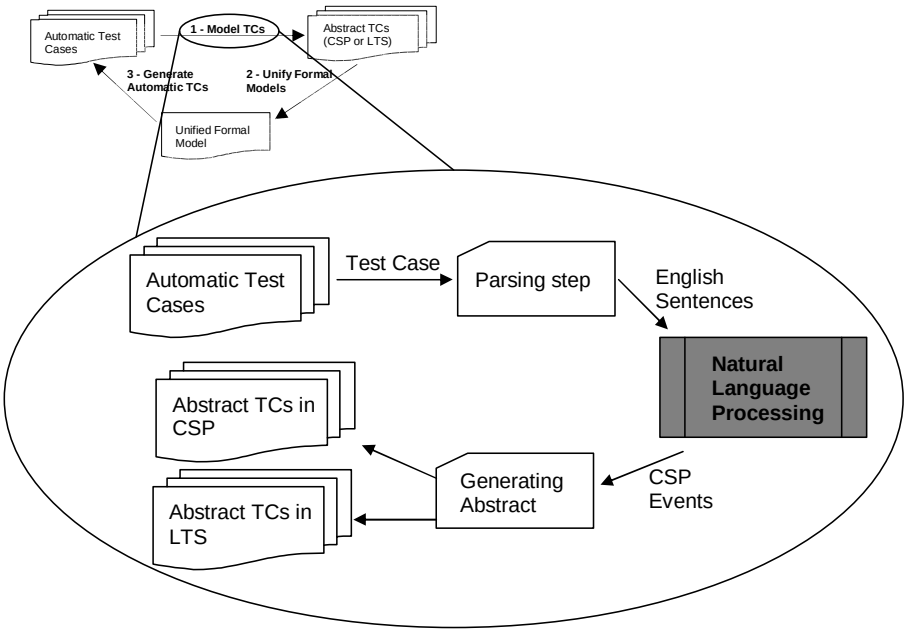


Figure 6.2 Task 1 - Model test cases

As shown in Figure 6.2, task 1 starts by taking each concrete test case (automatic test) from a set of test cases, and supplying to the parsing step. As supposed, the parsing step performs the extraction of all elements from a concrete test case. Recall from Chapter 2 that a test case, more specifically an automatic test case, has three parts: setup, test, and postcondition. Each part of a test cases is composed by a sequence of actions, where an action contains a description in

high-level follows by a set of script commands specified in some programming language. The parser component takes all high-level descriptions (English sentences) and supply them to the natural language processing sub-system [Lei06]. As we previously saw, the natural language processing system translates English sentences into CSP events.

Finally, after translating English sentences into CSP events, the next step is the generation step of abstract test cases. An abstract test case is specified either in CSP or LTS formalisms. Simply, this step takes all CSP events and structures them in a specific output format. The possible output formats are CSP processes or LTS diagrams. The selection of which format use is done by the user, and it depends on the final purpose of generation. For instance, if we are handling test cases and formal models in LTS format, we must use the generation in LTS, otherwise we must take the generation in the CSP formalism.

In Chapter 4, we have some examples of automatic test cases given as input and also formal test cases in CSP or LTS format.

6.1.1.2 Task 2 - Unify formal models

Following by the sequential flow shown in Figure 6.1, the second task deals with the unification step of formal models. The unification task as shown by Figure 6.3 is responsible for implementing the merging operator.

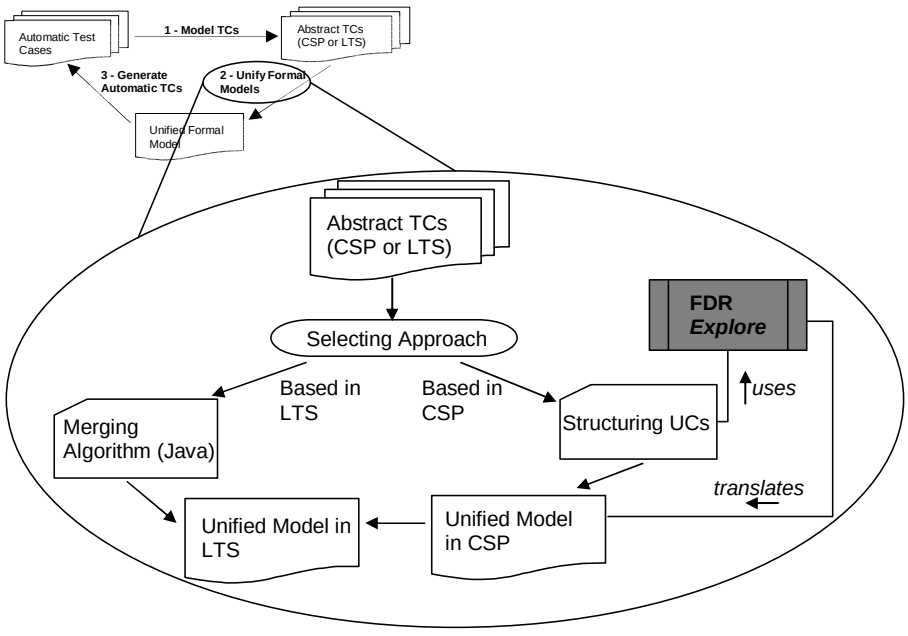


Figure 6.3 Task 2 - Unify formal models

The output from the previous task (model test cases) is the input for unifying formal models. Thus, the unifying starts taking all generated abstract test cases specified either in CSP or LTS. Before performing the unification step it is necessary to inform which unifying approach (based on CSP or LTS) the user wants to accomplish. Based on the selection of each unifying approach we have two different flows. The left flow represented in Figure 6.3 considers the unifying

approach based on LTS, whereas the right flow means the merging based on CSP.

Following the left flow, the tool takes all formal test cases in the LTS notation and performs the merging algorithm. As it was previously shown, the merging based on LTS is based on an algorithm written in Java, which takes a set of LTS models and unifies them into a unique model. In this case, the final model is in LTS notation.

Considering the right flow (the merging based on CSP), we have a set of abstract test cases specified as CSP processes. Recall from Chapter 5 that the first step to perform the unification using CSP is to set up the environment. Basically, the environment is set initializing all necessary unification components (UCs). As previously discussed, the merging based on CSP has several unification components, such as, list of initial models, firing rules, auxiliary functions, renaming and hinging on models, unification interface, unification component, and others. The UCs are responsible for merging the models, but some of them are static components, so they do not suffer changes, once specified it is not necessary to change again. The components such as, firing rules, unification interface and unification component are static components, for instance. Even, other components, such as, list of initial models, alphabets of initial models and renaming operations are considered dynamic components.

The purpose of the structuring step is to configure all dynamic UCs. To configure some UCs it is necessary to interact with the FDR *Explorer* (Chapter 3), Figure 6.3 shows this interaction. Finally, after configuring all unification components (UCs) we have a CSP specification, which contains all necessary structures and a CSP process that captures the unified model. With the CSP unified model it is also possible translate this process into an LTS diagram. As shown in Figure 6.3, this translation step is performed using an FDR *Explorer* script.

6.1.1.3 Task 3 - Generate automatic test cases

Finally, the sequential flow finishes taking all abstract test cases and generating concrete ones (automatic test cases). The last task is the task for generating automatic test cases, and its aim is to translate from a test case format to another. Figure 6.4 shows how it was implemented in the TCRev tool.

The result of the previous task (unified model in LTS or CSP) is the input of this task. However, before generating concrete test cases, the initial step is to extract the abstract test cases from the formal model. To extract abstract test cases we need a specific approach of model-based testing. As previously discussed in Chapter 4, we have related to our work two MBT approaches: one takes a CSP model and generates CSP test cases [Nog06], the another uses the LTS model to generate test cases in LTS format [Car06]. Such MBT approaches are based on guided generations, where *test purposes* drive the generation algorithms. Basically, the test purpose contains information that defines the desirable paths the user wants to test. Thus, based on information passed by test purpose abstract test cases are generated. Currently, the interaction with those MBT approaches in the TCRev tool is being manually performed.

This task finishes by taking each abstract action contained in abstract test cases and translating them into concrete actions. This translation step uses a table that contains all mapping between abstract and concrete actions previously stored. Recall from Chapter 4 that one of the purposes considered by our modelling approach is to create a table with mappings between abstract and concrete actions. The mappings table was implemented using a relational database

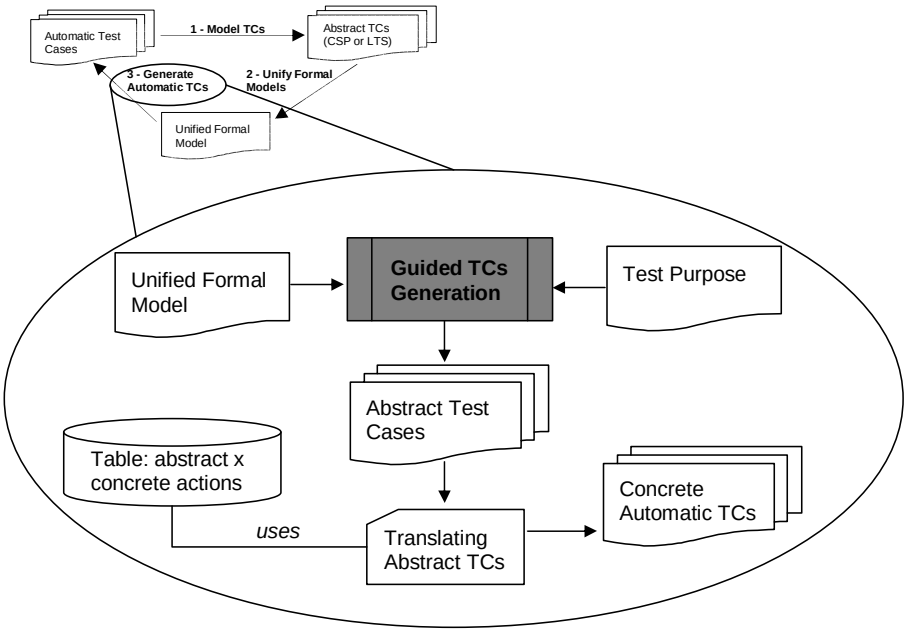


Figure 6.4 Task 3 - Generate Automatic Test Cases

PostgreSQL [Mom01].

6.1.2 Architecture

The TCRev tool was implemented using the Java language [Gra97] and the Eclipse RPC platform [J. 05a]. The purpose of using Java is to guarantee executions in multiple platforms. The Eclipse RPC platform gives an extensibility mechanism that allows add additional *plug-ins* to a main application. Each plug-in represents a bundling of some well-defined functionalities that may or may not extend another plug-in's functionalities. Thus, the TCRev tool was intended to be a plug-in in a major application, the TaRGet. In Section 6.1.3.1 we give an overview about the TaRGeT.

The RCP platform architecture serves as an open tools platform, it is designed so that its components could be used to build just about any client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform. Figure 6.5 shows the architecture of an application implemented following RCP platform.

The Eclipse development platform has pros and cons [J. 05a]. As cons we can highlight:

- The learning curve: a-priori it is not so easy to develop using Eclipse RCP platform. First, it is necessary to know the RCP platform and afterwards how to develop using such platform;
- The workbench does not always fit your needs: The windows management done by Eclipse workbench, sometimes does not adjust all graphic element properly;
- The UI controls have some constraints: The GUI elements provided by platform have some restrictions related to positions and user interactions.

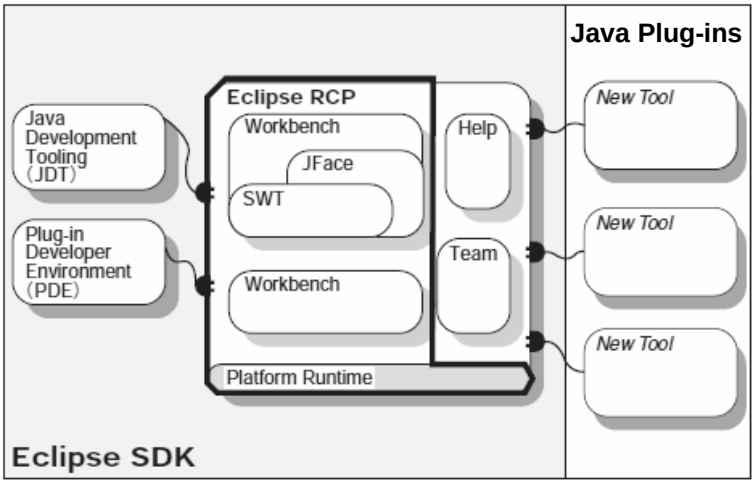


Figure 6.5 Eclipse RCP architecture

Besides the previous difficulties working with RCP platform we can point out as satisfactory factors:

- Reuse of the Eclipse platform and community: all graphic and control management elements are previously supported by the platform. Moreover, all extensions done by the Eclipse development community are available;
- Your application becomes extensible: as previously said, using the RCP platform we can extend our main application adding additional plugins;
- Cross platform: as the Eclipse platform underlies Java platform we gain an application able to run in multiple platform.

6.1.3 Context

In our context, we have the TaRGeT tool, a RCP application that connects with others plug-in applications, such as the TCRev tool, the MBT system, the natural language processing system, and others. Figure 6.6 shows an instance of some plug-ins added on a RCP application (TaRGeT).

6.1.3.1 Overview of the TaRGeT tool

TaRGeT or Test and Requirements Generation Tool is also related to Motorola's context, which is being developed by the Research Team. The TaRGeT tool as suggested, gives support to automatic generation of test cases and requirements. TaRGeT automates a systematic approach for dealing with requirements, design and test artefacts in an integrated way. Test cases can be automatically generated from both use cases written in natural language and UML sequence diagrams. Moreover, existing test cases can be used to automatically generate or indicate necessary updated to requirements document.

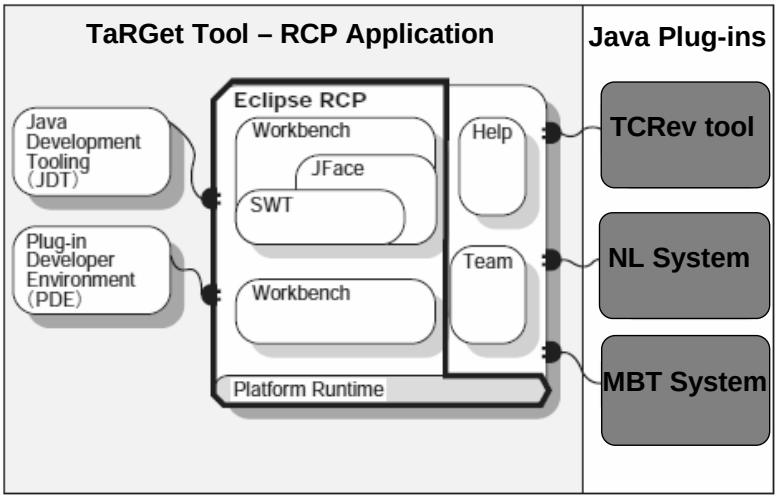


Figure 6.6 Plug-ins on RCP application (TaRGeT)

The TaRGeT's architecture is the same as shown in Figure 6.6. TaRGeT is intended to be like a shell of several other tools. Each tool was developed or is being developed by members of the Research Team. Beyond the TCRev tool, TaRGeT is intended to have several other tools (plug-ins), such as a tool for generating test cases [Nog06, Car06], one for processing natural language [Lei06], one for translating requirements into formal models [CS06] or test cases into requirements [Tor06], and, finally, one for generating UML diagrams [MS06]. The TaRGeT's main functionalities are:

- Test cases generation based on requirements documents. The generated test cases can be either manual or test scripts, and the requirements documents are specified following use cases. The currently approach for generating test scripts follows our modelling strategy defined on Chapter 4.
- Update or complement requirements documents. This updating follows a reverse engineering defined by our modelling approach, where existing test cases are used to build models, and afterwards update former requirements documents;
- Check consistency between test and requirements and traceability among test and requirements artefacts;
- Automatic generation of UML diagrams.

Currently, the TaRGeT is in alpha test phase. This prototype allows the user to generate manual test cases from system requirements documents (use cases). Moreover, traceability data among test and requirements artefacts are collected and can be viewed by the user. In later versions of TaRGeT all above functionalities are expected to be implemented. Figure 6.7 shows us a screenshot taken from TaRGeT's main screen.

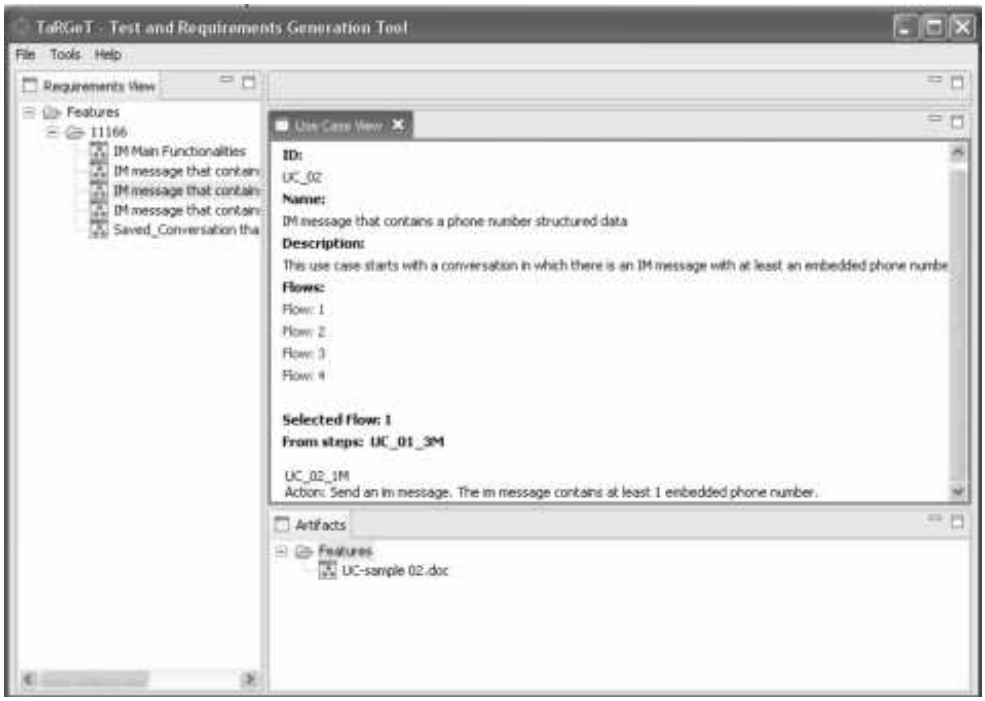


Figure 6.7 TaRGeT's main screen

CHAPTER 7

Case Study

In this chapter we show a case study performed in the Motorola's context. This case study aims to perform and evaluate the whole systematic procedure presented in the previous chapters.

In this case study we deal with test cases and requirements documents. The original test cases were generated inside the BTC project by the automation team and they describe test scenarios of the messaging application. Moreover, these requirements documents are based on the messaging feature and were specified following the work presented in [CS06].

More details about the feature used in this case study are described in Section 7.1. The case study was performed using the TCRev tool presented in Chapter 6, and the step of modelling test cases using TCRev tool is presented in Section 7.2. Section 7.3 describes the step of unifying formal models using models specified through test cases and requirement documents. Finally, Section 7.6 presents the analysis and final considerations concerned with this case study.

7.1 Feature Description

The first step to perform the case study was to select the feature to evaluate. We decided to select the messaging feature because this feature describes several interesting scenarios and many of them are able to be automatized using automatical scripts.

In the Motorola's context, artefacts responsible for specifying a set of functionality of cell-phone are divided in different set of features. Thus, artefacts related to similar functionalities are grouped in the same feature. For instance, test cases and requirements which describe scenarios of handling message operations, such as sending, receiving, and saving messages, are grouped in the messaging feature. Other artefacts related to multimedia operations, such as playing audio and video files, or visualizing pictures, are in the multimedia feature.

7.1.1 The messaging feature

The messaging feature involves a large set of functionalities where, these functionalities are related to each other. Actions related to send and receive messages (SMS, MMS, EMS), send and receive emails, and services of instantaneous messaging are examples of functionalities of the messaging feature. Due to extensibility of the messaging feature, this feature is also considered as a whole component.

In the Motorola's context, a component differs from a feature by having other integrated sub-features. So, a component describes one or more related features. For example, inside the

messaging component we have other sub-features, such as the *Hot Messages* feature, which is responsible for uniquely describing actions related to the *Hot Messages Folder*. In our case study we are concerned with the sub-feature of sending and receiving messages, either using messages (SMS, EMS, MMS) or emails.

7.1.1.1 Sending and saving usual messages (SMS, EMS, and MMS)

These scenarios involve operations of sending and receiving messages. There are three kinds of usual messages, which are basically based on the contents of the messages. The first and most popular is the *short messaging service* or SMS. An SMS message involves only text characters. We also have the *enhanced messaging services* or EMS. In EMS messages we can handle more elaborated messages, involving images and melodies pre-installed in the phone. Thus, while SMS transmits pure text, EMS is able of transmitting text, pictures and melody and presenting them in a nice way. EMS is a technology designed to work with existing networks, nevertheless it became obsolete after emerging the MMS or *Multimedia Messaging Service*.

The MMS message is a new standard in mobile messaging service and represents one of the most amazing technologies in cell-phone applications. Like SMS and EMS, MMS is a way to send a message from one mobile to another. However, the difference is that MMS can include not just text, but also sound, images and video not pre-defined in the phone. Furthermore, it is also possible to send MMS messages from a mobile phone to an email address. Formats that can be embedded within MMS include text (formatted with fonts, colours, etc), images (JPEG, GIF format), audio (MP3, MIDI) and even video files using the MPEG standard.

7.1.1.2 Sending and saving email messages

These scenarios define all actions related to send and save email messages. For these scenarios the phone must support communication protocols, such as SMTP, POP3 or IMAP4. Moreover, the phone must be able to send/receive messages to/from server.

It is necessary that there are no roadblocks between the server and the phone. A roadblock is caused by a difficulty in the communication or even a lost of signal. Messages sent in broadcast (multiple receivers) are also concerned with this kind of scenario.

7.2 Modelling Test Cases

After selecting the feature, the next step is to select the set of test cases that will be used for modelling as abstract test cases.

In the selection step it is necessary to assure that all proposed test cases are sound, that means test cases which describe a correct behaviour of the SUT. To verify whether a test case is sound, we must check its actions (steps) with the requirement documents. If actually the test case specifies a correct behaviour it becomes a candidate for the modelling approach. Otherwise, it should not be considered. That verification is necessary because some test cases can be specified or implemented incorrectly, and such tests should be discharged by the modelling approach, because they would add wrong behaviour to the test model.

In our work, we focus in automatic test cases or test scripts. Moreover, the selected test scripts are only feature basis. One test case is feature basis if it has only sequential flows, no interaction flows with other features. A sequential flow in test cases are commonly named as *feature testing* or *unit testing*. They have such a definition because are concerned with the test of a specific feature of the SUT per time. Integration and interaction scenarios are not considered in feature testing.

After verifying the test cases, we selected a set of 117 test scripts for applying the modelling approach. All test cases are related to functionalities described in Section 7.1.

As described in Chapter 4, the task of modelling tests cases aims to take original test cases and translate them into abstract ones. An abstract test case can be represented either in LTS or CSP formalism. Abstract test cases are used to build a formal model of the SUT, or test model.

With all selected test cases, the TCRev tool performs the modelling approach. Initially it is necessary to inform the directory where the test cases are and the type of translation format (LTS or CSP). For instance, if the user selects the LTS format all outputs are also generated in LTS. After the modelling task, we got 117 abstract test cases specified in LTS or CSP (based on user's option). In order to validate both kind of generation (LTS and CSP) we performed the modelling approach with both formalisms, thus a total of 234 abstract test cases were generated. Chapter 4 shows an example of a real test case (test script) given as input for the modelling approach, and how it is translated into abstract formats (LTS and CSP).

By using our modelling approach we also got a mappings table with about 230 records. Each record contains a mapping between formal events (either LTS or CSP events) and high level sentences written in English. Using such a mapping it is possible to translate test cases from abstract formats (CSP or LTS) into real formats (manual or automatic). Moreover, such mappings are used by other initiatives inside the research team.

7.3 Unifying abstract test cases

After modelling test cases as abstract ones, the next step is to build a unique representation of the feature under test. The unique representation is a formal model (test model) which describes behaviours of the SUT captured from real test cases.

To build the test model we used both unifying approaches, based in LTS and CSP. The following sections give the result of unifying in LTS and CSP approaches. Please refers to Chapter 5 for further details about the unifying approaches.

7.3.1 Unifying approach based on LTS

Using our unifying approach based on LTS, we just need to run TCRev Tool informing the directory where the abstract test cases written in LTS are. After that, TCRev Tool returns a new directory containing the unified model (test model) also specified in LTS.

In this case study we got a unified formal model with 114 different behaviours. Each behaviour represents a different sequential flow on the test scenarios. Notice that, the unified model contains less behaviours (flows) than total of 117 initial test cases. That occurred because among the total set of test cases there were redundant test cases. Redundant tests means

repetitive test scenarios. The detection of redundant tests is one of the advantages of our unifying approach. With our unifying approach we can easily eliminate repetitive test scenarios, which could be contained on the generated test model.

7.3.2 Unifying approach based on CSP

To perform the unifying approach based on CSP it is necessary to configure all unification components. Many configuration steps are configured automatically, by using the TCRev tool. However, to get other unification elements, such as the communication alphabets, we need to interact with other external tools, for instance the FDR and FDR *Explorer* tools. The configuration step of unifying approach based on CSP is well described in Chapter 5. Here, we are concerned with describing the results and found challenges.

After performing all configuration steps, we have the CSP unified model stored in specification files. The generated CSP use model represents the behaviour of the system observed by the test cases. However, in this case study, we could not explore the generated use model in CSP. One can explore a use model by visualising the graphical representation in LTS or checking properties, such as deadlock, livelock, determinism, and others. As the CSP use model is totaly based on parallel communications between the unification components (*UC*) and the test processes (test cases communicates their own events), this brings to us an overhead issue.

That was one drawback of the unifying approach based on CSP we have faced. The amount of 117 test cases is considered low in terms of Motorola's context. The limitation is related to performance issues. Performance issues were affected in the unifying approach by using many processes in parallel. Notice that, for each two processes we need an internal unification component. So, for instance, in a unification with 4 models the unifying approach generates a parallel composition with 6 processes (as shown in Figure 7.1). The amount of generated parallelism directly affects the performance. The bigger parallel composition we have got using the unifying approach based on CSP was a parallelism with 19 processes. In that case, 19 processes involve 10 initial models and 9 internal unification components.

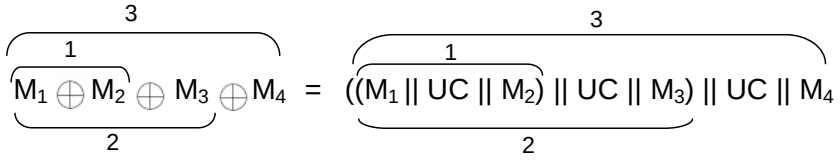


Figure 7.1 Unification with 4 CSP models

7.4 Generating Formal Models from Requirements

The approach for generating formal models from requirements is proposed by [CS06]. This work defines a process for generating a formal model from usual requirement documents. The requirement documents are written in English and follow the use case scheme. Figure 7.2 shows the process for generating formal models from requirement documents.

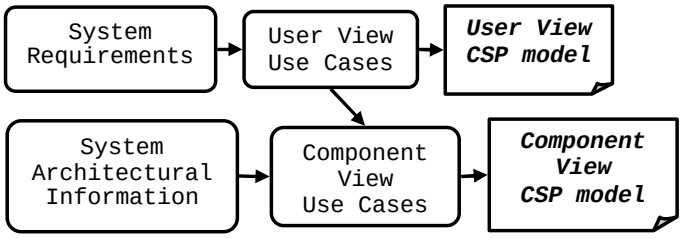


Figure 7.2 Process for generating formal models from requirement documents

The formal models of Figure 7.2 are two CSP models: **User View CSP Model** and **Component View CSP Model**. The user view model is generated from requirements documents whereas the component view model is generated from architecture documents. Initially, both resources (requirements and architecture documents) must be written using templates of use cases documents, *User View Use Cases* and *Component View Use Cases*, respectively. After they had been written in standard use cases, it is possible to generate CSP models.

Both models (user and component view) describe system behaviours, but they concern to different abstraction levels. The approach indeed suggests the use of refinement notions to prove that the component view model (lower abstract) refines the user view model (higher abstract). The refinement notion is now shown in Figure 7.2. In our unifying strategy we are only concerned with CSP models in the user view.

Figure 7.3 represents a user view use case. Summarizing, a user view use case is divided into main and exception flows. For each flow is specified a start (**From Step:**) and end (**To Step:**) points, which defines the start and end of the execution flow. Each step of use case contains fields for: step identification (it must be unique, for example, *1M* for first step of main flow and *2E* for second step of exception flow), user action, system state, and system response.

The translating consists in taking the field contents **Use Action** and **System Response** of each step and supplies them to a natural language processing system [Lei06]. The result of the natural language processing is a CSP event of the user view model. This step is similar to that we use for modelling real test cases as abstract test cases (see Chapter 4).

The field content **System State** defines whether an exception flow must be taken or not. The exception flows are introduced in the CSP model using external choices (`[]`). Finally, a use case is translated into a CSP process. Figure 7.4 shows the CSP process in user view generated from the use case of Figure 7.3.

Each step of use case was translated into a CSP sub-process (such as, *UC_02_1M*, and *UC_02_2M*). The exception flow is introduced by an external choice between a main and exception subprocess (for instance, *UC_02_3M* and *UC_02_5E*).

After translating 3 user view use cases into CSP processes, the final CSP model of user view is built using the unifying approaches. In this case study, we generated the unified model using the approach based on LTS and CSP. The unified model generated from 3 use cases consists in about 50 different behaviours (flows).

UC_02 - Incoming message moved to the Important Messages folder

Related requirement(s): REQ_1302, REQ_1326

Description: User accepts an incoming message and moves it to the Important Messages folder.

Main Flow
From Step: 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 Flow
From Step: 3M
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 7.3 An instance of a user view use case

7.5 Unifying Formal Models from Test Cases and Requirements

After modelling both test cases and requirements as formal models it is possible to unify them in a unique behavioural unit. Again, in this case study, we tried to unify the initial models (test cases and requirements) using both approaches based on LTS and CSP.

Using the LTS approach for unifying formal models we got a unified model with about 170 different behaviours. That matches the expected result by unifying a test model with 117 flows and another model with more 50 flows generated from the requirement documents.

```
UC_02_1M = (
  steps -> read.DT_ITEM.(INCOMING_MSG,{}) ->
  expectedResults -> display.DT_VALUE.(MSG_VALUE,{}) ->
    UC_02_2M
)

UC_02_2M = (
   steps -> open.DT_MENU.(CSM_MENU,{}) ->
   conditions -> isstate.DT_LIST.(FEATURE,{ON}) ->
   expectedResults -> isstate.DT_VALUE.(HOT_MSG,{}) ->
     UC_02_3M
   []
     UC_02_5E
)

UC_02_3M = (
   steps -> select.DT_ITEM.(IMPORTANT,{}) ->
   conditions -> isstate.DT_ITEM.(MSG_STORAGE,{}) ->
   expectedResults -> isstate.DT_VALUE.(MSG_MOVED,{}) ->
     UC_02_4M
   []
     UC_02_1E
)

UC_02_4M = (
   steps -> wait.DT_ITEM.(SECOND,{AT_MOST.2}) ->
   expectedResults -> isstate.DT_VALUE.(MSG,{HIGHLIGHTED}) ->
     SKIP
)
```

Figure 7.4 CSP process in user view generated from Figure 7.3

7.6 Advantages of the modelling and unifying approaches

After having the unified model containing both information from requirements and test cases, the next steps are to check system properties using model-checking in the unified model and to use the unified model in an MBT approach to generate new test scenarios. The new generated test cases will contain additional information from the requirements, so they are considered more complete test cases than originally.

Our unifying strategy is basically considered a mechanism for underpinning the MBT approaches. Moreover, we can use our modelling approach to eliminate redundant test cases and also update the original requirement documents.

Finally, other purpose of our unifying approach is to work together with an MBT approach able to generate test scripts following abstract test cases. The generated automatic test cases will contain new information provided by requirements documents. By adding new information on generated test scripts it is possible to insert script commands until not mapped on our mappings table. So, if a new entry is supposed to be added on our mappings table it will be simply noted with a special tag. This tag is the basis for implementing a way to supply a code for that new entry. Other works propose new ways to supply a code implementation for those new entries in our mappings table. The test case from Figure 7.5 states a special tag, which indicates a new entry in the mappings table.

```
TestCase TC_SELECT_AND_PLAY_A_MULTIMEDIA_FILE:

  Setup:

    description("Insert a multimedia file.");
      phone.goToIdle();
      phone.startApp(MULTIMEDIA);
      phone.insert(“C:\sound.mp3”);


  Steps:

    description("Open Multimedia application");
      phone.goToIdle();
      phone.startApp(MULTIMEDIA);

    description("Select the multimedia file.");
      phone.scrollToMultimediaFile(Content.SOUND_MP3);
      phone.verifyHighlight(Content.SOUND_MP3);

    prototype("Play the selected multimedia file.");
      /*  (“prototype” tag)
       *  This code is intended to be empty.
       *  A new code implementation must be put here.
       */


  PostCondition:

    description("Delete the inserted multimedia file");
      phone.goToIdle();
      phone.startApp(MULTIMEDIA);
      phone.delete(“C:\sound.mp3”);
```

Figure 7.5 An automatic test case with special tag (new entry)

As shown in Figure 7.5, the special tag (**prototype**) states a new entry in the mappings table. All prototype tags are input to other works responsible for generating a code implementation suitable to script command.

CHAPTER 8

Conclusion

There is no doubt that the use of formal methods is important to software development. Thus, new approaches for validating, checking or testing software based on formal methods are emerging. Concerning the specific combination between formal methods and software testing, the MBT approach is the most notable progress. In MBT, a system is supposed to be modelled using a formal notation and because there is only one interpretation of this representation, it is possible to generate test cases correctly and, in general, automatically. The MBT approach is still more relevant when the context of use is related to industrial case studies.

However, as MBT assumes that a formal model of the system under test must be available and in industry it is not the case in general, the Anti-MBT approach was conceived as a direct extension of the MBT approach. In the Anti-MBT approach, the system model is not defined a priori, and the goal is to synthesize system representations following system descriptions or executions. In usual Anti-MBT approaches the system model can follow either diagrammatical or formal descriptions. A formal description is more precise and consistence once we have a strong mathematical formalism to be followed. Once the system model had been created the MBT approach can be applied as usual.

Another benefit of using the MBT approach is related to translating certain representations into others. In this direction the goal is to translate abstract representations into real ones. An abstract representation, formal or informal, is useful for symbolic representations of the system, but for practical terms it is necessary to use a real representation. For example, we saw in Chapther 4 that a test case can be represented either as an abstract description using a formal language as basis (LTS or CSP) or real descriptions (manual descriptions or test scripts). Thus, in the test cases context using abstract test cases we can check properties or generate system models, but it is not possible to execute them. Just using a real representation it is possible to be executed.

The Motorola's software testing center (or the Brazil Test Center–BTC), is a reference in the area of software testing. In the BTC context, more specifically in the Research group, formal methods and software testing have been used in real situations. In the Research group some MBT approaches have been developed to assist the process of testing in cell-phone's applications. In those approaches, formalisms such as LTS and CSP are strongly used to create system behavioural representations or system models. Using a system model it is possible to check system properties and generate good test cases as well. The generated test cases can be executed (manual or automatic) in real scenarios.

8.1 Contributions

In this work we presented some developed approaches to improve two MBT approaches used in the Motorola's context. The first is the approach of modelling test cases and the second is the approach of unifying formal models. The first approach presented in Chapter 4 is concerned with building system formal models, by which there are not system formal models available or they are out of date. Thus, we used system executions, described by system test cases, to synthesize system formal models. The test cases considered in the evaluation of this first contribution were described using test scripts written by the automation team of Motorola and the scope was the feature of messaging.

Related to the generation of system model we presented in Chapter 5 two unifying (or merging) approaches: one based on the LTS formalism, and another on the CSP formalism. In the LTS based approach we used models described in LTS state-machines. Thus, the initial test cases (test scripts) were first translated into LTS test cases, and then the set of LTS test cases were unified by our unification algorithm. The unification algorithm was implemented in the Java programming language, where inputs were initial LTS models and the output was a unified LTS model.

The unification approach based on CSP consists in taking a set of CSP models as input and generating a unified model in CSP as output. Similarly to the LTS approach, the first step to perform the CSP unification is to translate the initial models into CSP formalism. Thus, the same set of 117 test cases (test scripts) were translated into CSP test cases. With the set of CSP test cases it is possible to generate the unified CSP model. After merging all initial models we have a unified view of the system. With a unified model we verify system properties using a more detailed view of the system, generate new artefacts, and also update older artefacts.

As another contribution of our work we have the translation of abstract representations into real ones. The translation approach was presented in Chapter 4. In this work, we developed a strategy to translate abstract models (LTS or CSP) into real models (manual or test scripts). The translation strategy is based on a mappings table, which contains pairs of events, in the form (abstract event, real event). Using this mapping one can translate an abstract model into a real model, and vice-versa. The strategy was implemented using a Java algorithm able to insert new records in the table and searches for previous records.

Finally, in Chapter 6, we shown the TCRev tool. The tool TCRev (acronym for Test Case Reverse) is responsible for specifying (modelling) automatic test cases into abstract or formal test cases in LTS and CSP formalisms. Beyond generating abstract tests the tool is also responsible for taking all abstract tests and building a complete formal model.

To validate the presented approaches we performed some case studies. In this work we presented a case study concerning the feature of messaging (Chapter 7). The initial artefacts used in this case study were requirement documents and automatic test cases. The requirement documents were written following the controlled natural language (CNL), as specified in the work [CS06]. The test scripts were generated by Motorola's automation team.

8.1.1 Considerations

The main consideration is related to the unification approaches. Using the LTS approach and 117 test cases as input we generated a unified model with 114 different behaviours. The difference between the test cases given as input and the unified model is concerned with redundant test cases as was explained in Chapter 5.

Nevertheless, the CSP approach does not behaviour as expected. Due to performance issues this step was compromised and we could not generate the unified model in CSP specifications. As discussed in Chapter 5, the CSP approach involves several parallel composition, and once we have a considerable number of models as input this approach became impractical in real terms.

Several attempts were done to measure the maximum amount of parallelism supported by the CSP approach. As conclusion we could generate about 18 parallel compositions, which involves 10 models (test cases) and 9 unification components (UCs) in parallel compositions. In the Motorola's context that becomes impractical, as long as we have several test cases to be composed in order to generate a formal model.

8.2 Future Works

Test cases are supposed to be up to date, whereas many requirement specifications are out-of-date. The result of applying the unification approach using test cases and requirements will generate an updated use model. Thus, a unified model from tests and requirements can be useful as input for new approaches which aim to extract and generate new requirement documents following abstract representations. The goal is to develop an approach for extracting all information using formal descriptions and translating back into original requirement documents. The new requirement documents will contain new information which was provided by updated test cases. Recently, new approaches also related to Motorola's context are being developed to achieve such results.

As future work, we intend to improve the unifying approach based on CSP in the sense that making it more applicable in practice. That is, basically avoiding the overhead created by parallel compositions. A possible way to do that is to use data refinement. Probably, to do that it will be necessary to combine CSP with a new formalism, which considers good data definitions, such as those supported by Z [WD96] specifications. Possible formalisms in this direction are CSP-Z [MS01] and Circus [WC02].

Although we mentioned that our work is strictly related to other works also inserted in the Motorola's context, we did not combine them into a unique approach. Thus, as extension of this work can be to explore a way to combine all related approaches. For instance, using a unified approach one can generate abstract models from test cases and requirements, unify them into a unique model, and apply MBT approaches to generate new test cases. All this using a unique tool able to perform all the functionalities. Recall from Chapter 6 that the TaRGeT tool is the current attempt of the Research team to this integration.

Finally, as future work we propose to prove mathematically the properties of the merge operator. We detailed that the merge operator must satisfy some properties, such as identity,

associativity, and commutativity. In this work, we explored that using the properties of the CSP basic operators used by the merge operator, which are the parallelism and external choice operators. However, a formal proof is necessary to really validate those properties. The idea is to use some rules of the parallelism and external choice to prove the properties of the merge operator.

Bibliography

[A. 99] A. Belinfante and J. Feenstra and R. de Vries and J. Tretmans and N. Goga and L. Feijs and S. Mauw and L. Heerink. Formal test automation: A simple experiment, 1999.

[AAM06] A.Figueiredo, W. Andrade, and P. Machado. Generating Interaction Test Cases for Mobile Phone Systems from Use Case Specifications. In *A-MOST'02: In-proceedings of the 2nd Workshop on Advances in Model-based Software Testing. Co-located with the 17th IEEE International Symposium on Software Reliability Engineering*, 2006.

[And86] S. J. Andriole. *Software Validation, Verification, Testing and Documentation: A Source Book*. Petrocelli Books, Inc., Princeton, NJ, USA, 1986.

[Ash98] P. Ashenden. *The Student's Guide to VHDL*. Morgan Kaufmann Publishers Inc, 1st edition, 1998.

[Bar88] A. Barr. Natural Language Understanding. pages 441–446, 1988.

[BCE+06] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, New York, NY, USA, 2006. ACM Press.

[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.

[BIMP04] A. Bertolino, P. Inverardi, H. Muccini, and A. Polini. Towards Anti-Model Based Testing. In *In Proceedings of Fast Abstract Session at IEEE International Conference on Dependable Systems and Networks DSN 2004*, pages 124–125, 2004.

[Bin99] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BMJ] L. Bousquet, H. Martin, and J. Jzquel. Conformance testing from UML specifications.

[BSS95] E. Brinksma, G. Scollo, and C. Steenbergen. *LOTOS Specifications, Their Implementations and Their Tests.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[Car06] E. G. Cartaxo. Test Case Generation by means of UML Sequence Diagrams and Label Transition System for Mobile Phone Applications. Master's thesis, Federal University of Campina Grande - UFCG, 2006.

[CDEG03] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. Multi-valued Symbolic Model-Checking. *ACM Trans. Softw. Eng. Methodol.*, 12(4):371–408, 2003.

[Coh04] M. B. Cohen. *Designing test suites for software interaction testing. Ph.D. Thesis.* PhD thesis, The University of Auckland, Department of Computer Science, 2004.

[CS06] G. Cabral and A. Sampaio. Formal Specification Generation from Requirement Documents. In *SBMF'06: Proceedings of the 6th Brazilian Symposium of Formal Methods*, pages 217–232. SBC/Instituto de Informática da UFRGS, 2006.

[D. 01] D. Clarke and T. Jeron and V. Rusu and E. Zinovieva. Automated Test and Oracle Generation for Smart-Card Applications. In *E-SMART'01: Proceedings of the International Conference on Research in Smart Cards*, pages 58–70, London, UK, 2001. Springer-Verlag.

[Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model-Checking*. PhD thesis, Eindhoven University of Technology, 1996.

[DBG01] J. Dushina, M. Benjamin, and D. Geist. Semi-formal test generation with Genevieve. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 617–622, New York, NY, USA, 2001. ACM Press.

[DJK+99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *International Conference on Software Engineering*, pages 285–294, 1999.

[DRPAFV02] R. Diaz-Redondo, J. Pazos-Arias, and A. Fernandez-Vilas. Reusing Verification Information of Incomplete Specifications. In *In Proceedings of the Workshop on Component-Based SE*, 2002.

[EC01] S. Easterbrook and M. Chechik. A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints. In *International Conference on Software Engineering*, pages 411–420, 2001.

[ED89] P. Van Eijk and Michel Diaz. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA, 1989.

[FGH+93] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. In *European Software Engineering Conference*, pages 84–99, 1993.

[FHP02] E. Farchi, A. Hartman, and S.S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, pages 41(1):89–110. [439, 444, 446, 447, 448, 449, 450, 454, 456, 457, 458, 459], 2002.

[For97] Formal Methods (Europe) Ltd. *FDR User's Manual*, version 2.28 edition, 1997.

[For98] Formal Methods (Europe) Ltd. *PROBE Users Manual*, version 1.25 edition, 1998.

[Gra97] M. Grand. *Java Language Reference*. O'Reilly, 2nd edition, 1997.

[Het88] B. Hetzel. *The Complete Guide to Software Testing*. QED Information Sciences, Inc., Wellesley, MA, USA, 2nd edition, 1988.

[HMBH94] D. Holtje, N. H. Madhavji, T. Bruckhaus, and W. Hong. Eliciting formal models of software engineering processes. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 28. IBM Press, 1994.

[HN99] A. Hartman and K. Nagin. TCBeans, software test toolkit. In *In Proceedings of the 21th International Software Quality Week (Wq 1999)*, pages 445–450, 1999.

[Hoa83] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 26(1):100–106, 1983.

[Hol03] G. J. Holzmann. *The Spin Model Checker*. Pearson Education, 2003.

[How86] W. E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[Hut07] G. Hutton. *Programming in Haskell*. Cambridge University Press, 1st edition, 2007.

[J. 03a] J. Philipps and A. Pretschner and O. Slotosch and E. Aiglstorfer and S. Kriebel and K. Scholl. Model-based test case generation for smart cards. In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03). (Electronic Notes in Theoretical Computer Science, vol. 80)*, pages 168–182, Trondheim, Norway, 2003. Elsevier.

[J. 03b] J. Philipps and A. Pretschner and O. Slotosch and E. Aiglstorfer and S. Kriebel and K. Scholl. Model-based test case generation for smart cards, 2003.

[J. 05a] J. McAffer and J.Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional; Bk&CD-Rom edition, 2005.

[J. 05b] J. Sun and J. S. Dong. Synthesis of Distributed Processes from Scenario-Based Specifications. In *FM*, pages 415–431, 2005.

[KB94] F. Khendek and G. Bochmann. Incremental Construction Approach for Distributed System Specifications. In *FORTE'93: Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques, VI*, pages 87–102. North-Holland, 1994.

[KB95] F. Khendek and G. Bochmann. Merging Behavior Specifications. *Formal Methods in System Design: An International Journal*, 6(3):259–293, June 1995.

[KRPM99] I. Kr, U. Radu, G. Peter, and S. Manfred. From MSCs to statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, pages 61–71, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

[KVZ98] H. Kahlouche, C. Viho, and M. Zendri. An Industrial Experiment in Automatic Generation of Executable Test Suites for a Cache Coherency Protocol. In *IWTCS: Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems*, pages 211–226, Deventer, The Netherlands, The Netherlands, 1998. Kluwer, B.V.

[L. 06] L. Freitas and J. Woodcock. FDR Explorer. In *In REFINE 2006 MACAU, to appear as LNCS*, 2006.

[Lei06] D. Leitão. NLForSpec: Translating Natural Language Descriptions into Formal Test Case Specifications (in Portuguese). Master's thesis, Informatics Center of Federal University of Pernambuco - CIn/UFPE, 2006.

[Lew91] D. Lewine. *Posix Programmers Guide*. O'Reilly Media, 1st edition, 1991.

[LT88] K. Larsen and B. Thomsen. A Modal Process Logic. In *In Proceedings of LICS'88*, pages 203–210, 1988.

[M. 91] M. Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing*, 3(1):21–57, 1991.

[Mom01] B. Momjian. *PostgreSQL: introduction and concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[MS01] Alexandre Mota and Augusto Sampaio. Model-checking csp-z: strategy, tool support and industrial application. *Sci. Comput. Program.*, 40(1):59–96, 2001.

[MS06] P. Machado and A. Sampaio. Viewing CSP specifications with UML-RT diagrams. In *SBMF'06: Proceedings of the 6th Brazilian Symposium of Formal Methods*, pages xxx–yyy. SBC/Instituto de Informática da UFRGS, 2006.

[Mye04] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 2nd edition, 2004.

[Nog06] S. Nogueira. Guided CSP Test Case Generation Guided by Purposes. Master's thesis, Informatics Center of Federal University of Pernambuco - CIn/UFPE, 2006.

[PERH05] W. Prenninger, M. El-Ramly, and M. Horstmann. Case Studies. In M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-based Testing of Reactive Systems: Advanced Lectures*, number 3472 in LNCS. Springer-Verlag, 2005.

[Ran02] C. Rankin. The Software Testing Automation Framework. *IBM Systems Journal*, pages 41(1):126–139, 2002.

[RG00] J. Ryser and M. Glinz. Using Dependency Charts to ImproveScenario-Based Testing, 2000.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall (Pearson), 1997.

[SA99] J. Shen and J. Abraham. An RTL abstraction technique for processor microarchitecture validation and test generation, 1999.

[Sca98] B. Scattergood. *The Semantics and Implementation of Machine Readable CSP*. PhD thesis, University of Oxford, 1998.

[Sch00] S. Schneider. *Concurrency and Real-Time System the CSP Approach*. Jogn Wiley & Sons, 2000.

[SG03] B. Spitznagel and D. Garlan. A Compositional Formalization of Connector Wrappers. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 374–384, Washington, DC, USA, 2003. IEEE Computer Society.

[Sha85] S. Shatz. Post-failure Reconfiguration of CSP Programs. *IEEE Trans. Softw. Eng.*, 11(10):1193–1202, 1985.

[TB99] J. Tretmans and A. Belinfante. Automatic Testing with Formal Methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12 1999. EuroStar Conferences, Galway, Ireland.

[Tor06] D. Torres. SpecNL - Generating Natural Language Descriptions from Test Case Specifications (in Portuguese). Master's thesis, Informatics Center of Federal University of Pernambuco - CIn/UFPE, 2006.

[UC04] S. Uchitel and M. Chechik. Merging Partial Behavioural Models. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 43–52, New York, NY, USA, 2004. ACM Press.

[UKM03a] S. Uchitel, J. Kramer, and J. Magee. Behaviour Model Elaboration Using Partial Labelled Transition Systems. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 19–27, New York, NY, USA, 2003. ACM Press.

[UKM03b] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115, 2003.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of circus. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 184–203, London, UK, 2002. Springer-Verlag.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[WJH03] B. Welch, K. Jones, and J. Hobbs. *Practical Programming in Tcl/Tk*. Prentice Hall PTR, 4th edition, 2003.

[yWo] yWorks, http://www.yworks.com/products/yfiles/doc/developers-guide/. *yFiles Developer's Guide*.

Dissertação de Mestrado apresentada por **Clélio Feitosa de Souza** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Modelling and Integrating Formal Models: from Test Cases and Requirements Models**", orientada pelo **Prof. Alexandre Cabral Mota** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Alexandre Marcos Lins de Vasconcelos
Centro de Informática / UFPE

Prof. Sergio Castelo Branco Soares
Departamento de Sistemas Computacionais / UPE

Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 26 de abril de 2007.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.