Automated Telecommunication Software Testing

An automated model generator for Model-Based Testing

ARMANDO GUTIERREZ LOPEZ
and
IGNACIO MULAS VIELA

KTH Information and Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Automated Telecommunication Software Testing

An automated model generator for Model-Based Testing

Armando Gutierrez Lopez
Ignacio Mulas Viela

Master's Thesis

Industrial Supervisor: Athanasios Karapantelakis
Examiner: Dr. Gerald Q. Maguire Jr.

Stockholm,1 May 2012

Abstract

In Model-Based Testing (MBT) the main goal is to test a system by designing models which describe the functionality of the system to test. Subsequently, test cases are obtained from the model, and these test cases can be executed automatically.

Experience has shown that the learning curve for learning MBT can be steep - especially for people who do not have previous programming experience. This is because the language used to design models uses programming language concepts. In this thesis we describe a tool which automatically generates models, given an initial set of requirements. The advantage of this tool is that users do not need to learn a model-based testing language to design models, but instead they must learn to use a high-level of abstraction and a Graphical User Interface to specify their test cases.

We demonstrate the value of the tool by using it to design models that generate test cases for telecommunications system, but show that this tool can be adapted for use in testing similar systems. The application of this tool can facilitate traditional phase-based software development methods, by saving a considerable amount of time and resources. In addition, when applied to agile software development, the reduced time required for testing because of the use of our tool helps shortening the feedback loops between designing and testing, thus increasing team efficiency within every iteration.

Keywords: Model-Based Testing, Automated Model Generation, Graphical User Interface, State Chart, Agile software development, Scrum

Sammanfattning

I Model-Based Testing (MBT) är det huvudsakliga målet att testa ett system genom modeller som beskriver systemets unktionalitet för att testa. Därefter erhålls testfall från modellen, och dessa testfall kan utföras automatiskt.

Erfarenheten har visat att inlärningskurvan för lärande MBT kan vara branta - särskilt för personer som inte har tidigare erfarenhet av programmering. Detta beror på det språk som används för användning programmeringsspråk begrepp. I denna avhandling beskriver vi ett verktyg som automatiskt genererar modellerna, med tanke på en första uppsättning krav. Fördelen med detta verktyg är att användarna inte behöver lära sig en modellbaserad testning av språket att konstruera modeller, utan de måste lära sig att använda en hög nivåav abstraktion och ett grafiskt användargränssnitt för att ange sina testfall.

Vi demonstrerar värdet av verktyget genom att använda den för att konstruera modeller som genererar testfall för telekommunikationssystem, men visar att detta verktyg kan anpassas för användning vid testning av liknande system. Tillämpningen av detta verktyg kan underlätta traditionella fas-baserade metoder mjukvaruutveckling, genom att spara en avsevärd tid och resurser. Dessutom, när det tillämpas på Agile Software utveckling, minskade tid som krävs för att testa på grund av användningen av vårt verktyg hjälper förkorta återkopplingar mellan design och testning, vilket ökar teamet effektiviteten inom varje iteration.

Keywords: Model-Based Testing, Automated Model Generation, Graphical User Interface, State Chart, Agile software development, Scrum

Resumen

En Model-Based Testing (MBT), el objetivo principal es testear un sistema mediante el diseño de modelos que describan su funcionalidad. En consecuencia, estos modelos generan test cases que pueden ser ejecutados automáticamente en dicho sistema.

La experiencia nos muestra que la curva de aprendizaje en el caso de MBT puede ser pronunciada, especialmente para aquellos sin ninguna experiencia previa en programación. Esto se debe a que los lenguajes usados para diseñar modelos usan conceptos intrínsecos a los lenguajes de programación. En este Proyecto Fin de Carrera, describimos una herramienta que genera automáticamente modelos, dado un conjunto de requisitos inicial. La ventaja que ofrece esta herramienta es que los usuarios no requieren el aprendizaje de ninguno lenguaje de modelado a la hora de diseñar modelos, sino que tan solo deben aprender a utilizar una Interfaz de Usuario Gráfica (GUI), a un alto nivel de abstracción, para especificar sus test cases.

Demostramos el valor de esta herramienta mediante su aplicación en un nuevo sistema de telecomunicaciones en fase de pruebas de Ericsson, mostrando al mismo tiempo que puede ser utilizada en el testeo de sistemas similares. La aplicación de esta herramienta puede facilitar los métodos de desarrollo de software tradicionales mediante el ahorro de una cantidad considerable de tiempo y recursos. Además, aplicada a métodos de desarrollo ágil de software, el tiempo reducido requerido para el testing a causa del uso de esta herramienta ayuda a acortar los plazos entre diseño y testing, y en consecuencia, incrementando la eficiencia del equipo en cada iteración.

Keywords: Model-Based Testing, Automated Model Generation, Graphical User Interface, State Chart, Agile software development, Scrum

Acknowledgements

We would like to especially thank our industrial supervisor and now workmate Athanasios Karapantelakis for making the thesis an outstanding experience due to his support, advice and commitment. Also, a especial mention to our academic supervisor and examiner, Gerald Maguire, for his valuable and precise comments which helped to improve and polish the whole thesis. We would like to take the opportunity to thank Alan Anderson as well for his advice and support during this thesis.

Ignacio Mulas Viela

I want to show my infinite gratitude to my parents, Carmen and José, who have always supported me in every decision I made and have encouraged me to follow my own convictions whichever they were. Also I want to thank my little brother, Luis, for always holding me on and to my grandparents for giving me unconditional support.

I cannot write these lines without thinking of Jesús for always been there when it was needed, Carlos for being my partner on everything, Víctor for being more than a friend in Sweden, Viky for making my new experiences abroad incredible, Chiquero for showing me see the bright side of things, Apa for been with me since long time ago, Jesús Muñoz for having amazing adventures together. I would like to acknowledge a great friendship to good friends from Stockholm, Nadine, Laura (Barna), Sibel, Ceci, Clara, Bea, Sankar, good friends from school Luque, Javi, Marta, good friends from the university, Dani, Javo, Rome, Sergio, Ali G, Pablo, Isma, Polo, Ali Martín, Ana, Laura, Yeyo, Garfio and my thesis and now work mates in Ericsson, Armando, Isaac and Manu. Furthemore, I want to mention a especial person because she has given me support and encouraged me in every challenge either from the closeness or from the distance, Carolina.

Because I would not be the same without all of you, thank you so much!

Armando Gutierrez Lopez

I will never be able to repay all the support that my parents, Jose Antonio y Pilar, and my sister Elena, has given me. In good and bad moments. For me it is enough with just knowing that you have always been there, and you will always be.

I also want to make a special mention to Daniel Jesus Garc/'ia, and Isaac Albarr/'an, for being my partners and my friends since everything started in Sundbyberg almost two years ago. Without you two this would have been completely different. In addition, I want to acknowledge Daniel Camara for his friendship and support. And despite the distance, I could not write this without thinking of Juan Manuel Herrera, for being there and being my support in Granada more than 5 years ago. I also want to acknowledge all the people that have made my life better (and easier) during all this years, like Guillermo, Kiko and Miguel Angel. I also thank my former thesis partners and now work colleagues, Manu, Nacho and Isaac. And last but not least, I want to thank Cristina for holding me in any moment of doubt, and encouraging me to achieve every goal I set out.

Thank you, really.

List of Figures

List of Tables

List of Acronyms

AMG	Automated Model Generator
API	Application Programming Interface
ASM	Abstract State Machine
CLI	Command Line Interface
FSM	Finite State Machine
GDL	GOTCHA Definition Language
GUI	Graphical User Interface
MBT	Model Based Testing
MDL	Murphi Definition Language
OPI	Operative Instruction
OS	Operating System
PLI	Programming Language One
QML	Qtronic Modeling Language
SQL	Structured Query Language
SUT	System Under Test
TAF	Test Automation Framework
TTCN-3	Testing and Test Control Notation Version 3
UML	Unified Modeling Language
URL	Uniform Resource Locator
UTP	UML Testing Profile
XML	Extendible Markup Language

Contents

Chapter 1

Introduction

A software product's life cycle is composed of conception, design, development, testing, and maintenance & migration to a replacement/successor product [71]. During the first period, the product's idea and feasibility are studied and the specifications are decided. Once a company decides to develop a product, the second period begins and the product is designed to meet the given specifications that may detail the product's expected performance and functionality. During the third period, the product's development is carried out and finally, the testing period checks whether all the specifications are met or not and how well they are met [71]. This is followed by product maintenance & migration as shown in Figure 1.1.

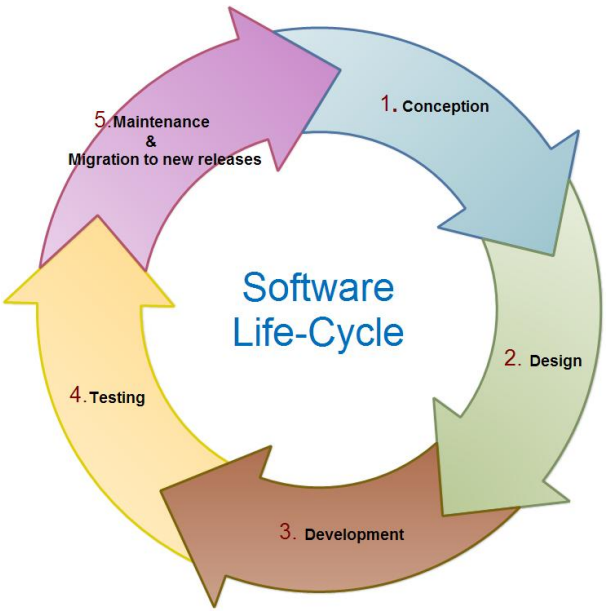


Figure 1.1. A software products' life cycle

The rapid changes that the telecommunications market is subjected to require a great ability to adapt. In addition, customers demand more functionality from the products along with high reliability and quality. Thus companies need faster and more efficient processes. An increasing fraction of the value of these products is based upon the software that is used to realize the product, configure, and operate it. As a result more and more time, money, and effort is being spent by companies on this software.

Testing is one of the most expensive processes within the software development life cycle, accounting for 40% to 70% of all system development costs [62]. Therefore, software verification in practice involves a trade-off between budget, quality, and time [64]. Efficiency in designing and development has been raised considerably reducing the time required to develop software products, but the time and effort required to test products has barely been reduced [30]. As a result, testers either need to work longer hours, or supplementary resources need to be added to the test team in order to meet aggressive test deadlines [37]. Thus, the target of recent research is to automate the testing in order to reduce the overall resources spent during the life cycle of the product, and hence increase the efficiency.

Designing automated tests for telecommunication systems is not a simple matter because it is not possible to create a unique test-suite for all systems. The complexity lies in the fact that these systems have little in common, e.g., at first glance a radio system seems quite different from a wired network system. Therefore, despite some test-cases that can be common to several systems under test (SUT), there test-suites need to be adapted to each SUT. Research has focused on developing a general-purpose method that allow an easy implementation of test-suites.

Different approaches have been developed in order to improve testing. One of the most popular testing methods currently is MBT. Its main goal is to test a system by designing models which describe the functionality of the SUT. Subsequently, test cases are derived from the model and executed. These models describe the expected behavior of the SUT and are used together with test generator adaptors in order to automatically generate a number of test cases [56]. The result of using MBT is a reduced human effort by avoiding manual test generation and execution[68].

Although the concept of "Model-based Testing" has existed for many years, during the last several years it has become popular as a testing approach not only because it decreases the overall time to test, but because it also provides better test coverage [43] versus traditional testing methods [49].

1.1. Problem Statement

MBT encapsulates the complexity of creating test cases into a simpler and more abstract representation of the functionality to be tested (i.e. a model). MBT has

been proven to be effective in the area of test maintenance, since every change can be made in the model, instead of individually modifying each test cases. As test cases are extracted from the model, small changes in the model can translate into large changes in the test cases. Thus, the model's maintenance time is usually smaller than modifying each test case individually (usually contained in test-scripts).

Even though MBT shows some benefits over traditional testing techniques (see Section 2.5), MBT has several problems to address:

- Experience has shown that the learning curve for learning MBT can be steep, especially for people who do not have previous programming experience [75].
- In MBT, many of the details found at the "execution layer" are abstracted by using models. This differs from the traditional testing view which directly transforms the specification of a test case into an executable script [28].
- "Pesticide paradox"[33], one of the biggest drawbacks in traditional automated testing is that fixing a bug leads to decreased efficiency since the bugs that were revealed lead to an escalation of features and complexity, resulting in subtler bugs that have to be found and fixed in order to preserve the product's earlier reliability.
- Currently, the MBT models need to be manually generated from the requirements specifications of the SUT, using some type of modeling tool [57]. Although the extraction of test-cases is automated, the time required to design and implement the models is still high.

In addition to the above, other major obstacles in the testing process are the problems related to requirements specification, which are associated with software systems failures [41]. A correct requirements specification can improve the effectiveness of generating models and test inputs. Unfortunately, there usually is no uniformity in these specifications [78], so in practice generating tests based upon these specification is much less efficient than directly writing test cases.

1.2. Goals

The main goals of this thesis project are designing, implementing and evaluating a tool which is able to automatically generate models, given an initial set of requirements. This implies accomplishing the following objectives:

- Designing a Graphical User Interface (GUI) to enter the test specifications requirements in order to increase the degree of abstraction in testing as much as possible. The benefits of this are:
  - Reduce the required programming and modeling knowledge when a SUT is tested.

  - Reduce the learning time for aspiring MBT users.

- Generate models in an intermediate Extensible Markup Language (XML)-based format, which is an accessible (i.e. machine-readable), and portable format. Thus, a model can be subsequently used by other applications after its creation. However, we have to keep in mind the specific Ericsson test domain where this tool is applied and the time constraint.
- Implement a transformation process which uses the created XML files to generate models in the format of Conformiq Modeler™and Designer™and then use these tools to automatically derive test cases from the model.
- Design and implement a parser to automatically read test specification documents and extract an initial set of machine-readable specifications. Thus, ideally the only thing that a user has to do is to enter an URL to the specifications and the program will complete all the requirements.
- Design and implement a "document pretty-printer" to automatically generate documentation from the implemented model.
- Write a user's guide which describes how to use the Automated Model Generator.

1.3. Scope

In this thesis project we try to fill the gap in the automation between requirements specification and the implementation of models. A model-based application was developed to create models for a set of command line interfaces (CLI) commands. The tool is designed to support any existing or any new CLI commands which could be introduced in the future, as well as new MBT test generation tools. Moreover, the XML-format models can be edited and reused from other SUTs, thus saving time and improving the efficiency of the overall testing processes.

In addition, we propose a solution to the problem of uniformity in requirements specification. Our tool is capable of creating documentation and specifications in a unified format, enabling future researches which could fully automate the verification process using a constrained language.

1.4. Target Audience

Test automation engineers will find useful information about automation solutions throughout this master's thesis. Test organizations interested in automating testing processes will find references to a implementation of an intuitive GUI for doing software testing.

Moreover, this master's thesis may also be interesting for those who need to store information using programming languages through an easy-to-use GUI or to translate information into different languages, i.e., XML, Qtronic Modeling Language (QML), modeling programming languages, and wide-spectrum languages.

1.5. Methodology

This thesis project began with a deep and thorough literature study about traditional testing techniques, and existing automated testing techniques - i.e. script-based testing and MBT. Moreover, model programming languages - i.e. XML and QML, parsing and transcoding techniques, and model-based design have been studied in order to provide a solid background about the context and solution adopted in this master's thesis.

Additionally, a review of related work and state-of-the-art methods is presented in order to extract new challenges that have not been addressed yet. Furthermore, we have taken into account the real needs of a large organization, Ericsson, and have implemented a solution that can accelerate their testing processes.

This thesis project describes an application created in order to simplify the process of designing and generating models, which are subsequently used as input to an MBT tool in order to generate test cases. The Automated Model Generator (AMG) tool creates a preliminary model in XML based on the input given by the user, the "pre-model" (this concept is explained in detail in later sections). A transcoder then generates, based on the previous XML file, the final model using a MBT tool, Conformiq Modeler and Designer. We deploy AMG to test a telecommunications system, but show that this tool can have a bigger impact, i.e. it can easily be adapted for the use in designing similar systems.

We show that by utilizing AMG in combination with a model-based testing tool, we can generate test cases from the models and using appropriate test adaptors generate executable test cases. Thus, users do not need to learn a new model programming language to design models, but instead are presented with an application which allows them to create models using a GUI.

Furthermore, by taking advantage of having the behavior of the system described in a model the documentation can be extracted from this model. Using such machine generated specifications a common format for specifications can be achieved. Additionally, because changes are made on the "pre-model" rather than directly in the specifications, old or incomplete information will automatically be eliminated from the specification when the model is next pretty-printed as a specification.

The tool is implemented in Java based on the results of our literature study of state-of-the-art approaches in MBT and Ericsson's needs for most suitable design.

Finally, using results from a survey we conducted within the company, we quantify the benefits of our tool when it comes to testing, in terms of time to test when using our proposed AMG/MBT approach versus when using manual or scripted testing. The results of this comparison shows the benefit of using MBT together with AMG against other methods.

1.6. Structure

Chapter 1 gives a general overview and a shortly introduction to this thesis.

Chapter 2 describes the necessary background for a good understanding of this thesis' contents. It starts with some short definitions which will help the reader to go through the following sections. It continues with the analysis of current testing techniques and an explanation of the test domain which the tool is adapted to. Thirdly, a deep explanation of MBT is presented and finally, the XML format is explained in detail.

Chapter 3 related work on this topic is mentioned and compared to what our targets are.

Chapter 4 the adopted solution is shown. Firstly, the proposed method and the traditional MBT are compared. Secondly, the architecture of the application is presented and the different modules are deeply explained.

Chapter 5 analyses the results obtained with the usage of our new tool and they are compared with the traditional MBT process in different scenarios.

Chapter 6 presents the reached conclusions and proposes some improvements to the current tool's version.

Furthermore,some appendixes which will help the reader to understand smoothly the content of this thesis appear in the end.

Chapter 2

Background

The intention of this chapter is to give an overview and explanation of all the technical background needed for a good understanding of this thesis. The chapter starts with a brief explanation of essential testing concepts in section 2.1. Actual testing methods are presented and compared, explaining their advantages and disadvantages in section 2.3. The AMG's test domain is presented in section 2.4 and this is followed by a deep study on MBT in section 2.5 as this method is the adopted solution. Furthermore, section 2.5.4 gives an overview of existing MBT tools and finally, a brief explanation of the XML language is given in section 2.6.

2.1. Basic Concepts

In this section, important concepts that appear during this thesis are defined:

- *Test-case*

  A test case is the set of instructions together with conditions and variables under which the tester determines whether an application or a process of the SUT works correctly or not.

- *Test specification*

  The test specifications contains the information about the design and/or the implementation of the test cases.

- *Test scenario*

  The test scenario sets the circumstances under which the SUT is tested, e.g., high/low traffic level on the network. The test scenarios contain test cases, and the sequence in which these test cases must be executed. They may be independent test cases, or a sequence of test cases, each dependent on the output of the previous one.

- *Test-suite*

  A test suite is a collection of test scenarios and/or test cases that are related or that cooperate with each other.

- *Test script*

  A test script in software testing is a set of instructions that will be performed on the SUT to test the system's functionality and/or quality. A text-script contains a sequence of actions to follow and the expected outputs from the SUT. One test script usually corresponds to one test case.

- *Model*

  In the testing context, a model is an abstract implementation of the SUT. It describes the expected behavior when specific inputs or conditions occurs.

- *Test oracle*

  A test oracle is software providing fail/pass verdicts. In automated testing, the tester specifies the expected output from the SUT, which is checked in order to determine if the test has passed or not.

- *Test-coverage*

  Test coverage describes the fraction of the SUT which is tested by the test suite. This includes all the features within the test suite's scope.

- *Regression Testing*

  Regression testing is any type of software testing that seeks to uncover new errors in existing functionality after changes have been made to the SUT. Some tests are common in most of the products, so in order to prevent repetitive test-cases and hence a massive growth of our test suite in consecutive releases, these test cases are either edited or eliminated [59].

- *White/Black box testing*

  White-box testing is a testing approach which tests internal structures and implementations of the SUT. Conversely, the black-box testing approach tests the functionality of an application, measuring the generated output given a specific input.

- *Test environment*

  A test environment is the required context in which to test a product. This environment simulates test scenarios, generates the inputs, and analyzes the outputs of the SUT.

2.2. Testing Framework

The goal of testing is to minimize the defects of products by detecting as many errors as possible before the product reaches the market. Testing on a system consists of a series of <input, output> interactions between this system and the tester. Successful verification of a test step can be perceived as a match of the system's output and the expected output from the tester, given a specific input.

Even though a product has successfully passed through a thorough testing process, it would be unrealistic to think that no errors will ever occur during the product's life-cycle. In fact, the need for extensive tests implies a large budget in order to ensure a better quality. Therefore, companies usually take into account post-release errors that will be corrected in future releases. Thus, the first challenge when testing is to design the appropriate testing process so that the desired functionality and quality are well-tested for a particular release.

Trivial post-release errors are known before releasing a product. The need to correct them sometimes depends on users' demands. Releasing a not fully complete product is currently common for most companies. These releases only purpose is to obtain feedback from the users' in order to improve the next release to meet their demands and avoid or delay unaddressed post-correction errors.

2.3. Software Testing Methods

Software testing is an essential period during the life cycle of any product, especially during software development. Software systems are modified many times during their life cycle due to various reasons, i.e., adding functionality, fixing bugs, or increasing quality. Therefore, tests must be carried out to check the robustness of each new release [42].

Designing, developing, and testing processes must be performed sufficiently to satisfy the market's demands. While the system functionality is set by the user, the company generally wants to reduce costs. This cost reduction is usually achieved by reducing the cost of the hardware. Unfortunately, this often leads to more complex software [55], and hence a larger number of tests need to be carried out in the last stages of the product's life-cycle in the shortest time possible.

The reader must not forget that research and development on testing aims to perform effective testing, i.e., to find more errors in terms of the requirements, design, and implementation in order to increase the product's robustness and quality. Automation potentially performs efficient tests, hence accomplishing the desired goals. Here efficiency refers to executing more test cases per unit time and/or executing a given set of tests in less time.

These investigations lead to increasing automation during the test period. Ideally, testing processes should be completely automated, which means that the SUT is tested without requiring any human interaction. A high degree of automation results in significant time and cost savings [51]. This high degree of test automation is achieved through an increase in the level of abstraction in test design [63].

In the current framework, a fully automated general-purpose method has not been created due to the variance in the nature of software testing. Therefore, a variety of test processes are found in the market that offer solutions with different degrees of automation. Although most of these processes require some form of human interaction, completely automated testing has been only achieved in specialized cases [77].

Specialized solutions are expensive and not flexible, therefore complete test automation is not the target in many cases. However, the main goal of test automation research remains to create a flexible and fully-automated solution [40].

Test automation requires a combination of programming and testing skills which are rarely found in software development teams. Furthermore, the lack of a standardized language for test automation tool interoperability causes problems when these tools are integrated [77]. These are some of the reasons why automated testing processes are still under-developed in comparison to other processes.

In software testing, test cases are executed (1) during the life-time of a release, and (2) in subsequent releases [34]. This subsequent regression testing is one of the major reasons why manual testing has proven to be inadequate in software testing because it consumes a lot of resources during the last releases when both previous and new test cases are executed. In manual testing, testers spend exactly the same amount of time when executing a specific test-case in each consecutive release [55]. This becomes a problem when the number of test cases increases considerably or when there are many different releases. The number of test cases tends to increase because products acquire more and more features, while testing teams face aggressive deadlines. Today, manual testing is not a feasible solution when a a large software product is developed because testing periods would become too long [37].

Therefore, semi-automated general-purpose testing techniques are currently widely used in testing software. Traditional test automation, i.e. script-based testing, coexists (see section 2.3.2) with modern processes such MBT [73], see section 2.5. These methods achieve an increasingly automated execution of test cases, once they are implemented.

2.3.1. Manual Testing

In manual testing a set of tests is directly executed by a tester on the SUT. The tester interacts with the SUT directly by entering a CLI command along with the command parameters (see subsection 2.4.1), then waits for the system's output and checks if this output is correct or not (see figure 2.1).

This method of testing greatly depends on the knowledge and experience of the testers, both to design the test suite and to create all the relevant test-cases. In manual testing, the tester must have well-focused goals in order to achieve a well-structured testing period [55] which can lead to failing to test one or more of the general goals of the SUT and hence, lead to a badly design test suite.

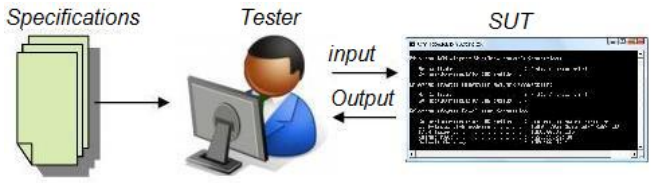


Figure 2.1. Manual testing schema

2.3.2. Automated Testing

Automated testing solutions are wide spread in software companies. General-purpose automated testing methods have been deployed to design test suites, where test execution and MBT test-case extraction are automated. Full-automation is usually costly and has a narrow scope, hence it is not a common practice.

However, automated testing has gained in importance with the increasing number of test-cases in consecutive stages [34]. The main advantage of automated testing is that already implemented tests can be updated and used in the maintenance phase, as well as easily adapted to new releases. a few modification to the overall set of tests allows faster test execution in later stages and hence, reduce the time needed for a particular test-case. This is referred as test "re-usage". Thus, the testing team is able to the increased the test coverage in comparison to manual testing, as each tester can automatically execute previous-designed tests and spend their time creating new tests for the new release.

In this thesis, two different methods have been considered, i.e. script-based testing and MBT.

Script-Based Testing

In script-based testing, testers implement testing scripts that will be executed automatically and compared against the test oracle of each test (see figure 2.2). The value of this method resides in the possibility to re-use or easily adapt of the same test-scripts to consecutive releases in order to perform the same test-cases (or a small variant of them).

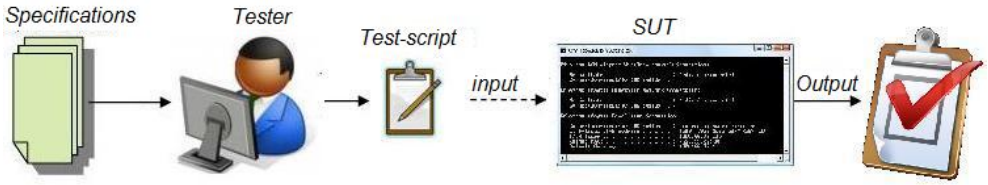


Figure 2.2. Script-based testing schema

The time spent initially implementing test scripts is considerably greater than the time spent for a single manual test cycle, but as the product development stages are completed, the time spent in the execution of the same number of tests via test scripts is significantly lower than the sum of the time that would be required for the equivalent manual testing. Thus, as stated before, the real savings occur in the later releases.

Model-Based Testing

MBT aims to test a system based upon models which describe the intended functionality of the SUT. MBT is an improved method in comparison to the traditional automated testing explained above. This method is illustrated in Figure 2.3 and explained in detail further in section 2.5.

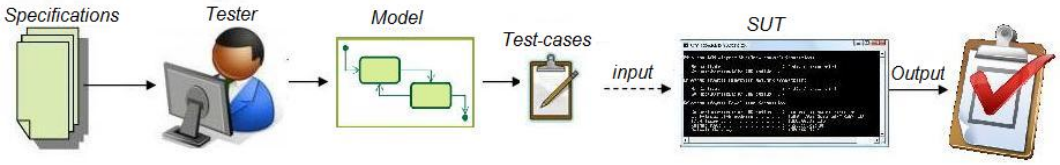


Figure 2.3. MBT schema

2.3.3. Manual Testing versus Automated Testing

Table 2.1 shows a comparison of manual testing against two different types of automated testing with regard to a number of different features. The compared features are:

- Execution time: the execution of test cases is automated in Script-Based Testing and MBT, hence the execution time is much lower than manual execution.
- Implementation time: refers to the amount of time required to execute a test case for the first time.
- Adaptation time: is the time spent when updating a former test case to the current version. MBT editing makes changes in the model so more than one test cases can be modified at the same time, thus It usually reduces the adaptation time per test case.
- Test coverage: is generally proportional to the number of test cases executed in late releases.
- Test cases: the test cases are implemented one by one in the case of script-based testing, but more than one test case can be automatically extracted using MBT.
- Future releases: it shows the method of adaptation followed in testing future releases.
- Investment: it is the amount of money that a company needs to spend in order to use one method. The investment in automated methods is high because it requires a learning process and the benefits are not immediate.

The cost-effectiveness of an automated test depends primarily on the number of times the test is to be executed. The more times a test case is executed the more profitable acquire the the use of an automated solution. Based upon the results shown in [34], test cases are executed at least 5 times and 25% of test cases are executed more than 20 times. Thus, the effort in manual testing rapidly becomes significantly higher than the effort required for automated testing solutions.

Table 2.1. Comparison between Manual Testing, Script-Based Testing, MBT

	Manual Testing	**Script-Based Testing**	**MBT**
Execution time	Long	Short	Short
Implementation time	Short	Long	Medium
Adaptation time	None	Long	Medium
Test coverage	Small	Medium	Large
Test cases	Progressively	Progressively	Partial test cases
Future releases	Repeated	Re-usage	Re-usage
Investment	Small	Large	Large

2.4. Test Domain

As stated before the nature of testing depends on the SUT. In this thesis we focus on the test of CLI to a telecommunications system. A CLI enables humans to communicate to software through commands which invoke a specific task. These commands are text-based and they are input directly to a command shell or via a computer emulator, e.g. PuTTY[1].

CLIs are typically used when there is a low-level of interaction between a human and a computer. The commands invoke specific tasks, e.g. reset the processor, load a new program, save the contents of memory, initiate a connection to a remote device, etc. These types of commands are wide used in many operating systems and software. Perhaps but probably the best known CLI commands are the CLI used in UNIX and MS-DOS.

These commands are used when an Operating System (OS) or a new software product is developed. In the testing framework, these commands are executed against the SUT in order to compare the output of the machine with the expected output for a particular input.

2.4.1. Nature of CLI commands

When an automated method is implemented, such as script-based testing or MBT, the CLI is defined as <command input, system output>pairs. The command input is composed of a command name and this command's parameters. While the system output is the expected system response.

Test-cases are sequences of CLI input-output pairs. These CLI input-output pairs are divided into two groups:

- Command Operation Description (COD): these commands are simple operations, e.g. connect, release, switch processor, reset processor, etc.
- Operational Instruction (OPI): these commands contains sequences of CODs, and they support operations such as switching network terminals, initiating/ending user session, initiating system recovery, transfer data from one device to another, etc.

Note that not all the sequences of CODs are defined as OPIs, there are some CODs that need to be executed after another COD - but they do not compose an OPI. OPI are usually sequences composed by a considerable number of CODs.

Figure 2.4 represents an example of a test-case showing the communication between the tester and the machine. Each of the lines from the tester to the

[1]See http://www.putty.org/

SUT represent an input and all the lines from the SUT to the tester represents the system's output. In the beginning, a connection with the server is requested through the 'ssh' command. Once the SUT's output completes the connection some configuration parameters are sent and the OPI 's sequence starts.

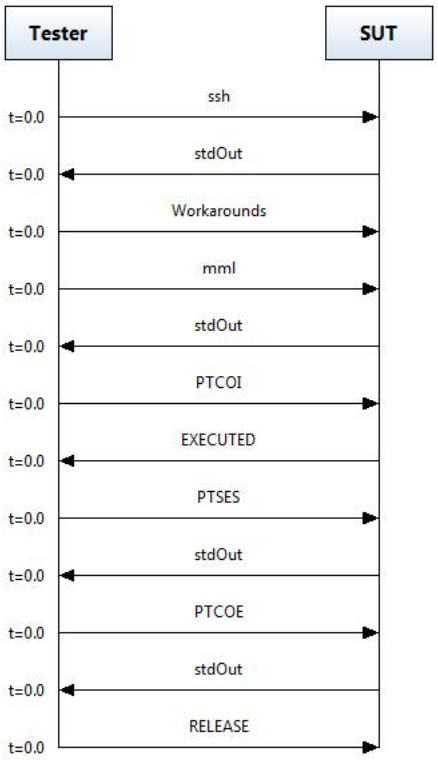


Figure 2.4. Example of a test case: initial load.

During the execution the tester (in the case of manual testing) or the system (in the case of automated testing) enters a given input and checks the output. If the SUT's output matches the expected output the sequence continues to the next command; but if it does not match, then the command flow stops as the test-case has failed. If the sequence is completed, then the test-case is passed.

2.5. Model-Based Testing

Model Based Testing (MBT) has been explored in the literature for many years [28] [39], but in recent years the interest in this topic has grown rapidly. The goal of MBT is to generate models from which tools can automatically derive test cases, thus avoiding hand-crafting of test cases or test scripts [39].

Due to increased competition in the telecom market, customers demand a

reduction in the time for product development and improved maintenance. This situation has encouraged product test organizations to search for techniques to improve the traditional hand-crafted test cases [39]. This effort focus on minimizing costs which has in turn led to the rise of MBT, due to its multiple advantages and benefits.

The purpose of MBT is to avoid the main obstacles in application deployment. Among these ones, time is a critical issue in each stage of the product development life cycle. Once the product development cycle is well underway, developers find that the time spent in integration and testing is a major obstacle in application deployment (i.e., getting the application running in customer's networks - so that the company can be paid). One of the main causes is that the same amount of effort spent on development environments is not applied to testing environments [28]. Test environments are often created in-house and are highly-specialized for a certain scenario. In addition, much of the testing requires human intervention and manual hand-crafting of test cases, so new tests are very difficult to produce when platforms and/or software are modified [28].

Another of the goals of MBT is to capture test requirements, while providing the flexibility needed to respond effectively to product changes. Effective test coverage requires not only a well-organized testing method, but also requires testable requirements [47].

2.5.1. Principles of MBT

MBT is a testing technique used for automated generation of test cases using models of the SUT, created to describe the system functionality and processes at an abstract level[50]. Using MBT software developers and testers can more clearly analyze the requirements specifications of the SUT, automate tasks, easily maintain consistency across the product line development, or generate code for direct deployment on the SUT. In MBT these models are used to automatically generate test cases. In addition, the models can be used in different ways:

- Represent the behavior of SUT,
- Represent the different testing strategies, and
- Represent the different testing environments.

As shown in figure 2.5, the first step is to describe the system in terms of a model. This initial model is abstracted to a level independent of implementation details or equipment required. Here the model should describe how the system should work and how it should not respond. The second step is to describe how the functions of the system interact, eliminating some of the abstraction of the model. The main advantage of this hierarchical development approach is that the model is independent of platforms and application issues, so it can be rapidly adapted to

new environments.

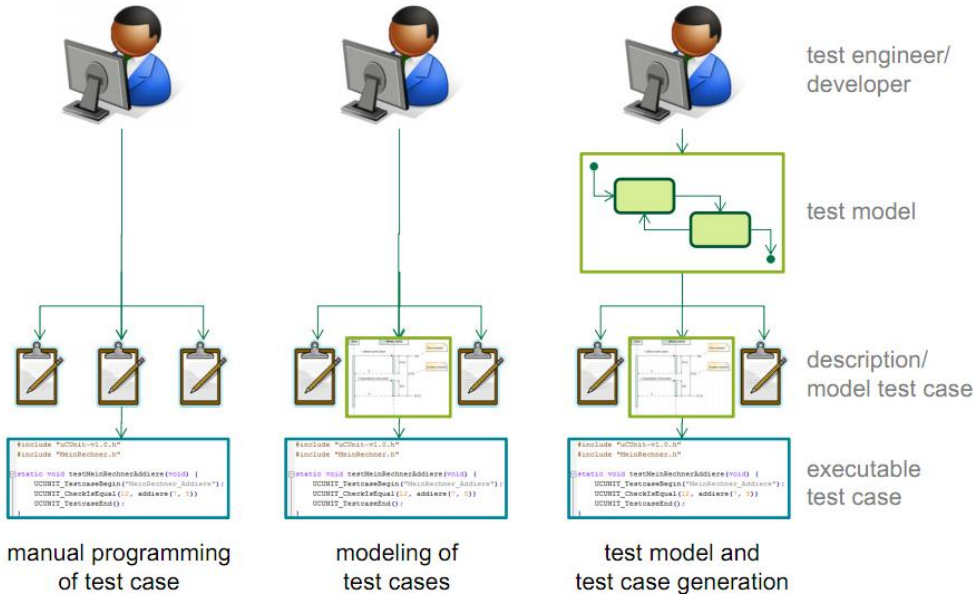


Figure 2.5. MBT approach

Cost savings and resource optimization have been two of the most significant weaknesses in traditional testing [36]. When bugs are detected, the cost of rework and additional regression testing in the traditional manual testing approach is costly and the date of the product release can be negatively affected. In this context, MBT is presented as an efficient solution to this problem in terms of cost, reliability, and software quality. The results of using MBT techniques provide a significant saving of time, lower costs, and higher quality [50].

2.5.2. Modeling in context of MBT

An MBT model describes parts of system's behavior without describing the system's implementation. MBT focus on black-box test modeling, i.e. on the functionality of the SUT (a representation of the structural implementation of the system leads to white-box test modeling) [33]. In this black-box testing specific knowledge of the internal structure or code of the SUT is not required. For the purpose of this thesis, we mainly focus on two modeling techniques: *Finite-state machines* and *Unified Modeling Languange*.

Finite-States Machines

Finite-state machines (FSM) represent the system's responses to incoming (both internal and external) events [71].A FSM is a state machine that can be in one of a finite number of states. This machine can only be in one state at a time, and to change from one state to another state it is necessary that a defined trigger event or condition occurs.

One of the main benefits of using FSMs is the ease of dealing with stimulus from the system's environment when describing the behavior of systems, including real-time systems. This behavior is analyzed and described in terms of one or more events which could occur in one or more states. Each model usually represents an object of a single class, and the trace of the states through the system represents all of the possible execution paths [44].

This type of diagram can be combined with UML notation. The advantage of using this notation is that it is a general-purpose notation and is able to describe any kind of state-machine model (see figure 2.6).

On the other hand, one of the main disadvantages of models based on state machines is that the number of states can grow rapidly, so for large systems a good structuring is needed [29]. One way to solve this is problem is to establish some "super states" which encapsulates other states. These super states correspond to a functional decomposition of the system.

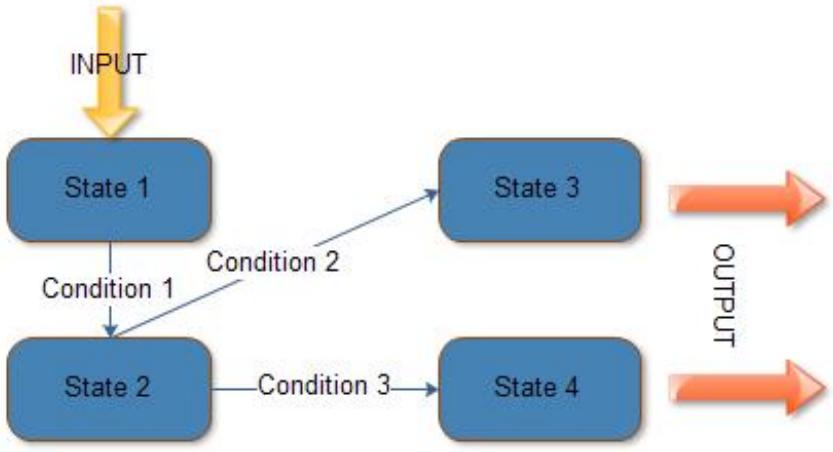


Figure 2.6. State machines diagram

Unified Modeling Language

Unified Modeling Language (UML) is a standardize language used to analyze and design object-oriented software systems [61]. UML is a notational

language that is able to capture numerous aspects of the software development lifecycle including design, implementation, and even testing issues. In short, it is a language used to define models.

Applying UML to both software testing and development enables the testing and design phases to use a consistent specification based approach to generate functional and reliable components with better effectiveness and efficiency [80]. UML's graphical notation allows a wide range of modeling approaches, not only for software systems, but also for workflows, organizational charts, or hardware design.

A UML model is composed of 3 parts:

- *Elements*: Abstractions of objects, actions, etc.
- *Relations*: How elements are related to each other.
- *Diagrams*: A set of elements with its relations.

In order to be able to represent any kind of system, UML has a wide range of types of diagrams to visualize the system from different perspectives. These different types of diagrams include the following:

- Use-Cases Diagram
- Classes Diagram
- Object-based Diagram
- Sequence Diagram
- Collaboration Diagram
- State Diagram
- Activities Diagram
- Component-based diagram
- Deployment Diagram

For the purposes of this thesis the most interesting diagram is the finite-state machine's diagram (see section 2.2.2). Sometimes, state machines can be confused with flowcharts. Figure 2.7 illustrates the differences. The main difference is that a state diagram performs actions in response to explicit events, whereas in the case of flowcharts, events could or could not be explicit [67].

2.5.3. Automation of MBT

The MBT automation method is sometimes referred to in the literature as Test Automation Framework (TAF) [36]. The TAF approach describes how the modelers

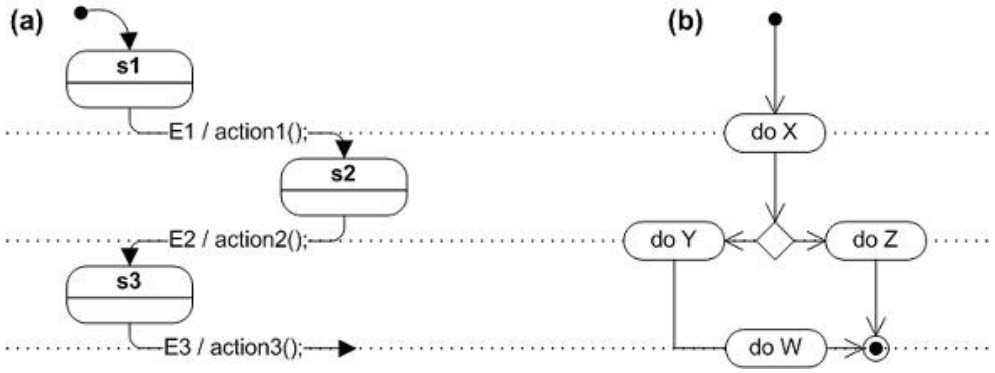


Figure 2.7. Example of UML diagram

develop the logic according to requirements for data and control processing of the SUT using models. This process involves 3 different roles:

- Requirements engineer
- Designer/Developer
- Tester (modeler)

Requirements can include system specification, user documentation, interface control documents, API documents, previous designs, and old test scripts. A requirements engineer usually documents the product's specifications in textual documents. A designer/developer develops a software architecture, design components and the rest of the elements involved in the system. The tester uses any available information to clarify the specification of the model and to properly describe the behavior of the system. All these models are translated by some software component to produce an expected range of test cases. The test generator adapter then creates test scripts, which are executed against the implemented system. Finally a comparison is made between the expected output and the actual output, if they match then the test case has passed, otherwise it has not passed [36]. Figure 2.8 illustrates this process.

2.5.4. Tools and Automation

According to Shafique et al., there are multiple state-based MBT tools available both in research and commercial domains [70]. The criteria for the selection of the tools which are in the scope of this thesis are:

1. *Model-flow criteria*: This criteria refer to states, transitions, transition-pairs, sneak paths (undesired paths), paths, parallel transitions (concurrent paths), and scenario criteria, which MBT tools can support in order to build test cases.

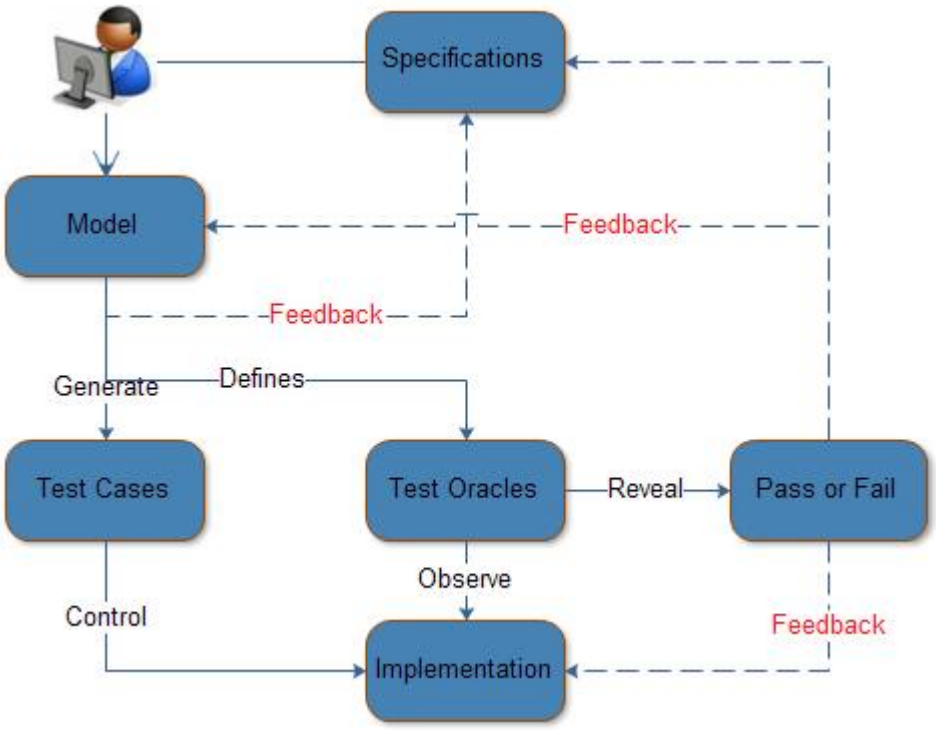


Figure 2.8. Automated MBT process

2. *Related activities criteria*: Existing MBT tools can support a number of activities that facilitate the integration of MBT activities in the process of software development [70]. The related activities include model creation, sub-modeling (i.e. decomposing the model into parts to reduce complexity), model verification (i.e. check properties of the model before test cases are generated), and requirements traceability, i.e., link requirements to parts of the test model (e.g., a transition, a path) in order to see which test cases exercise with requirements [65].

3. *Test scaffolding criteria*: This criteria refers to code developed to facilitate testing, including test drivers (i.e., software to run tests), test stubs (e.g., software to simulate execution environment), and test oracles (i.e., software that provides pass/fail verdicts) [70].

Accoding to the above criteria, we will consider the following tools: GOTCHA-TCBeans, MBT„ MOTES, Test Optimal, AGEDIS, ParTeG, Conformiq Qtronic, Test Designer„ and Spec Explorer. Each of them will be described below and a summary comparison will be given in section 2.5.4.10

GOTCHA-TCBEANS

In *GOTCHA-TCBeans* [9], GOTCHA generates test cases from an finite-state machine (FSM) model while TCBeans provides Java classes to derive and execute test cases on the SUT. The FSM test model is written in GOTCHA Definition Language (GDL), which is an extension of Murphi Definition Language (MDL) [15]. The user can specify test scenarios. GOTCHA generates test cases in XML format for online and offline testing. Test adapters are written using TCBeans classes and a translation table in XML to map calls to methods of the SUT. Test execution traces can be viewed in a TCBeans browser.

MBT

MBT is an open-source tool and does not have a GUI, only command prompt. "MBT" imports an FSM model, possibly composed of sub-models in GraphML format [12], created by a third-party tool, such as yED [27]. MBT provides requirements traceability and different algorithms to generate test cases. It generates skeleton code for test adapters using a default or user provided code template. The user can specify the adapter's behavior, including the oracle. MBT supports both online and offline testing. It also allows the user to generate test sequences to cover specific states, transitions, and requirements. However, *MBT* does not support test case execution.

MOTES

MOTES is an Eclipse plug-in which uses FSMs to generate test cases [14]. It uses third party tools such as Poseidon [18] or Artisan Studio [2] for model creation. It imports the test model in XMI format. MOTES requires three input files to generate test cases in Testing and Test Control Notation Version 3 (TTCN-3) the test model in XMI format: test data description in TTCN-3, configuration data for FSM states, and input/output ports to describe the types of data and the calls that can be sent to the SUT. The generated TTCN-3 test cases can then be executed by a third party (TTCN-3 compliant) tool.

TestOptimal

TestOptimal is a web-based client server tool that tests desktop and enterprise applications [21]. It uses FSM models, created interactively while analyzing the web site being tested. Models can also be imported in GraphML [7], XMI [16], and GraphXML format [48]. TestOptimal provides model validation, simulation, and debugging support. It provides multiple algorithms to generate test cases and supports both online and offline testing. It can be used for stress, load, and

regression testing. TestOptimal automatically generates test adapter class skeletons to wich a tester can add functional logic to run the generated test cases.

AGEDIS

AGEDIS is a test suite for model based testing of component based distributed systems [46]. It includes a modeling tool (Objecteeing UML Modeler [23]), a test suite editor (Spy Editor [26]) and browser, a test case simulation and debugging tool, a test coverage analysis tool, a defect analysis tool, and a test execution report generator, which are all integrated behind a single GUI. AGEDIS also provides a a test model compiler, a test generator engine, and a test execution engine (called Spider). The test models consists of classes, state machines and UML object diagrams. Classes' behavior are described with state machines. Object diagrams describe states of objects and hence the expected state of the SUT. Methods are specified in a intermediate format action language. A mapping between model operations and the SUT interface is provided in an XML file. Test cases are written in an XML file. The AGEDIS coverage analyzer provides information about possible input values which are not covered and methods which are not invoked by the generated test suite. The defect analyzer provides information about failed test cases and groups them with respect to similar failure reasons.

ParTeG

ParTeG (Partition Test Generator) [17] is an open source Eclipse plug-in that generates test cases from a test model, created using the TopCased Eclipse plug-in [22]. The model is composed of a UML 2.0 class diagram and associated state machine diagrams. ParTeG creates a transition tree from the test model (traversing the graph representing the state machine), each path in the tree being a test case. The expressions involved in a path are converted to conditions on input values. These conditions are used to define partitions for input values. The values near the boundaries of these partitions are selected as input values for concrete test cases. ParTeG generates test cases in Java which are executed on the SUT using JUnit. Test cases can be debugged using the Eclipse integrated debugger.

Conformiq

Conformiq [52] has three main parts: the Conformiq Computation Server generates test cases, executes them on model, and analyses results; the Conformiq Modeler is used to create the test model as UML state machines complemented with an action language, Qtronic Modeling Language (QML), similar to Java. Conformiq Client (Eclipse plug-in or standalone desktop application) provides support to create test models using Conformiq Modeler, select test coverage criteria, and

analyze model and test suite execution results. The model can also be created with Enterprise Architect [72]. The model is a combination of a state machine and QML. Conformiq also provides support for requirement traceability. Test inputs and expected output (oracle) are generated from the test model: Abstract test cases can be generated using different algorithms and in multiple languages (such as Java, C, C++, Perl, Python, XML, TTCN-3, etc.). The Conformiq Client environment provides online testing by directly connecting to the SUT using a dynamic link library (DLL) plug-in interface.

Test Designer

Test Designer [19] is part of Smartesting Center Solution suite. Smartesting uses third party tools for a number of features: IBM Rational DOORS [9] for requirements definition and traceability, HP Quality Center [8] to create test adapters, and IBM Rational Software Modeler (RSM) for model creation [9]. The Smartesting Model Checker and Simulator [19] are integrated with RSM.

Spec Explorer

In *Spec Explorer*[20], the test model is an Abstract State Machine (ASM. An ASM is a generalization of FSM, written in C# like pseudo-code, to operate over arbitrary data structures. Transitions between states are specified with actions (functions of the SUT), and rules, which specify the transitions allowed. The abstract model defined by the user is transformed into an internal model where abstract states become concrete states and actions are given actual input values which can be specified by the user. Test cases (with oracle checks) can be executed directly on an SUT implemented in .NET.

OSMO

Open-Source Modelling Objects (OSMO) consists of a number of tools and libraries to help in modeling software behaviour and to help in applying these models. The given models are represented in terms of a finite state machine (FSM). The main tool in this set is currently OSMO Tester, which is a simple model-based testing tool [1]. Both online and offline testing is allowed.

Comparative table

Table 2.2 summaries the tools described in the paragraphs above. As we can see the tools have a variety of target platforms, but all using either a FSM or UML model.

Table 2.2. MBT Tools comparison

Tool Name	Model Type	Category	Target Platform
GOTCHA-TCBeans	FSM	IBM Internal	Java,C/C++
MBT	FSM	OpenSource	General
MOTES	FSM	Research	General
Test Optimal	FSM	Commercial	General
AGEDIS	UML	Research	Java,C/C++
ParTeG	UML	Research	Java
Qtronic	UML	Commercial	General
Test Designer	UML	Commercial	General
Spec Explorer	FSM	Commercial	General
OSMO	FSM	OpenSource	Java

2.5.5. Benefits of MBT

The benefits of using MBT are:

- Significant saving of costs and time due to the automation of the generation of test cases [36].
- Development of an independent model, which can be used by different MBT tools written in Java, Perl, C/C++, SQL, PLI, XML, etc. [36]
- When system functionality changes or evolves, all the test cases can be regenerated with only a few changes in the logic of the model, facilitating test case maintenance [36].
- Models helps clarify unclear and poorly defined requirements. In order to be testable, a requirement must be clear and unequivocal. Some defects are difficult to identify manually when the requirements are documented in textual specifications and identification of such defects depends on human recognition and the experience of the people involved.Additionally, in MBT the model can be iteratively refined, thus progressively identifying a greater number of defects. MBT allows requirement testability analysis [36].

2.6. Extensible Markup Language

The Extensible Markup Language (XML) has established itself as the standard format for describing and exchanging data and documents between different applications [35]. Due to its intuitiveness and its hierarchical structure, XML is used in several existing test tools which offer different degrees of automation. The advantage of XML in the context of MBT is its capacity to describe input data in an

open and standard form through a structure known as an XML schema [35]. From a tester's point of view, the XML schema establishes the basic rules and constraints on parameters and information that different classes of systems and applications can exchange, thus it provides an accurate and formalized representation of the input domain in a format suitable for automated processing, i.e. with XML we are able to describe the SUT in a machine-readable format. Describing the information exchanges between the tester and the expected behavior of the SUT in a clear way is a step forward in automating the testing process.

2.6.1. Benefits of XML

The benefits of using XML are:

1. It is possible to extend XML with new tags which can be created as they are needed, providing a high degree of flexibility.
2. Information encoded in XML is easy to read and understand.
3. XML provides machine-readable context information.
4. XML is a standard that has been endorsed by the leading software vendors.

2.6.2. Estructure of an XML Document

The purpose of XML is to encode well-formed and structured information.This encoding allows software designers to move XML documents between different platforms and applications. An example of an XML document is shown in Figure 2.9. The structure of these documents must follow a series of rules [45]. Some of the key aspects of these rules are [6]:

- There is a root element which contains all other elements.
- Only UNICODE characters are permitted.
- Some special characters, e.g. '&', only appear with a markup delineation role.
- Element tags are case-sensitive.

2.6.3. XML schema

An XML schema is a specification of a type of XML document. It is usually expressed in terms of constraints on the structure and content of documents of a certain type. The schema consists of a combination of grammatical rules for the order of elements, Boolean predicates that the content of the document must satisfy, or other specialized rules governing the elements and their attributes [45].

```
<?xml version="1.0" encoding="UTF-8"?>
- <model>
  - <command>
      <name>Sample</name>
    - <parameter>
        <name>a</name>
        <value>1</value>
      </parameter>
    </command>
</model>
```

Figure 2.9. Example of an XML document

A specification of a XML schema consists of a set of markup declarations which are associated with a XML file. A declaration at the beginning of the XML document declares that the file is an instance of the type defined by the indicated specification [6]. The most important specifications of different XML schema are: Document Type Definition (DTD), XML Schema, and Standard Generalize Markup Language (SGML).

From a tester's point of view, the XML schema establishes the basic rules and constraints on parameters and information that different classes of systems and applications can exchange, thus it provides an accurate and formalized representation of the input domain in a format suitable for automated processing. This formalization is clearly a step forward in test automation.

Chapter 3

Related Work

Extensive work has been carried out in the area of automated testing and MBT. The benefits of automated testing are shown in [40]. In this paper Dustin et al. introduce the concept of "automated test life cycle methodology", i.e. the use of automated tools to support the test process. Their paper shows the benefits of automated processes compared to traditional manual testing processes and presents several different tolls for test automation.

Prestschner [66] has demonstrated the advantages of MBT by comparing automatically generated test suites with manually designed ones. His results show that automatically derived model-based test cases detect significantly more specification errors than hand-crafted test suites.

Aydal et al. present an approach for the automation of MBT using model transformations in [32]. In this paper they present a multi-platform MBT technique where the operations to be tested are modeled in the Alloy language [54], while the implementation of the complete system to be tested is specified in Z formal language. Instead of taking the model of the system or the code of the system as an artifact to be transformed, the test cases are transformed from one format to another. The test cases generated in Alloy are stored in XML format, and transformed into command files, executable in the Z model of the system. This work focused on the test adaptor (the tool that manages the model transformation) and not on the full automation of the process as models have to be manually created in order to generate the expected test cases.

A number of researchers have proposed tools with varying degree of automation. In [30] Aho et al. present a tool for automated test modeling of Java applications. A GUI driver is combined with an open-source MBT tool to form a tool chain to support automated testing. The GUI driver generates models which are used to generate test sequences with the MBT tool. The test sequences are then executed with the GUI driver to generate a test report. The difference with AMG is that

while AMG focuses on CLI testing, the field of application of this tool is restricted to GUI applications.

Iyenghar et al. [53] propose an approach to demonstrate the adoption and applicability of UML for deploying MBT in real-time embedded systems. In this case MBT is performed by using UML Testing Profile (UTP), an extension of UML for testing purposes. In this approach, models are not automatically generated, but manually created.

An integrated tool chain which enables a model-based development of testing scenarios is discussed in [74]. This tool chain is based on a canonical metamodel for testing concerns and a service-oriented model storage and exchange infrastructure, that allows a certain degree of adaptation to new testing requirements. The disadvantage of this approach is that it is more focused on the automation of test cases generation rather than the automation of model generation, as the models are still manually created.

Most of the approaches reviewed above are targeted at generic test domains. However, AMG is targeted towards a specific test domain (CLI testing). Thanks to this narrow testing area it is possible to create a tool containing an intuitive GUI (see also figure 3.1) and capable of automating the model implementation. In this sense, the tool fills a gap in the test solution market, by being one of the first to offer a level of abstraction in model design, on top of another abstract model. The tool will also provide a mechanism to extract requirements from specification documents, thus reducing the human interaction necessary in the process. In addition, in an intermediate step system or process requirements are transformed into XML files (see section 4.6), allowing the input of the generated models into different MBT tools and platforms.

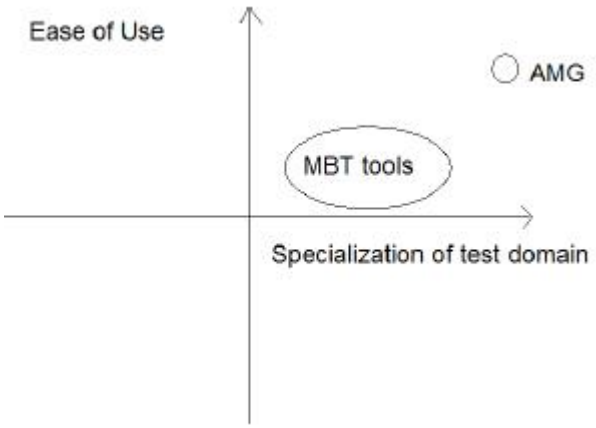


Figure 3.1. AMG position

Chapter 4

Automated Model Generator

In this chapter, the adopted solution is explained in detail and the . In section 4.1 the problems to solve are addressed and the adopted solution is explained in detail. Section 4.2 gives an overview of the whole process using the AMG tool, followed by a short explanation of each the module of the tool. Moreover, the different modules are explained in detail afterwards. These are (i) Document parser (section 4.3), (ii) AMG GUI (section 4.4), (iii) the XML transcoder (section 4.6), (iv) the XML translator (section 4.7), and (v) the pretty printer (section 4.8). There is also a little section (4.5) where it is explained how the program stores the information internally in order to understand how it is possible to convert all the introduced information into the models.

4.1. General overview

As stated before, this tool is implemented in order to facilitate the design and implementation of models in MBT. The target is to reach a point of abstraction where the user does not have to worry about modifying the implemented code and every single detail can be edited from the interface.

Following this idea, an easy-use graphical interface has been deployed where all the characteristic features can be edited and MBT implementation can be used without any knowledge on model programming.

In order to provide the reader with an accurate idea about this new approach to MBT we are introducing in this thesis, traditional MBT and MBT together with AMG tool frameworks are explained.

4.1.1. Traditional MBT

In Figure 2.3 the traditional procedure followed in MBT is shown and in section 2.5, MBT is presented in detail. The advantages and disadvantages are explained in comparison to other methods. However, is has not been discussed the weak points of MBT.

Traditional MBT has still much human interaction and only the two last steps are completely automated, test cases extraction and automated execution. Despite the last two automated process can save a considerably amount of time, the whole process is still slow compared to the expected results that would be obtained with full-automation. There are two main reasons:

- Specifications are not always clear enough.
- The MBT learning curve can be steep - specially for people who do not have previous programming language.

The first one usually causes confusion to the tester. If test specifications are unclear, modeling the expected behavior is not a trivial task and it can take more time than desired. Bad understanding of the system functionality can cause a bad test design and hence, incomplete or longer testing processes.

On the other hand, a new MBT user needs to learn a model programming language and model design conception in order to implement testing models. Therefore a good knowledge in a specific model programming language and a good understanding of model design are needed because the experience has shown that the learning process can be longer if the user does not have any previous experience in programming. Furthermore, as explained before testers usually were assigned to specific task in order to develop specific test cases but designing a model implies to have a global overview of the SUT. This mind change is needed to correctly design a useful testing model.

In practice, these two reasons cause some reluctance in new users and it is why MBT is still not established completely. The investment that a company has to do in order to change former ways of testing (see Section 2.3) is still high and the results of improvement in MBT cannot be seen after some time using it. Hence, companies usually continue with former and inefficient ways of testing or introduce MBT very slowly. The latter option results on an even larger learning process and hence, a delay on getting results from this testing method.

4.1.2. MBT with AMG tool

A new way of performing is presented in order to avoid the difficulties that a MBT user has to face. On the one hand, the process of learning a model language is totally removed thanks to the high degree of abstraction achieved in AMG tool. On

the other hand, AMG tool generates documentation from the implemented "pre-model" which results on common formatted documentation.

Abstracting the user from model programming languages

Automation usually implies developing higher levels of abstraction in order to represent all the information in the best way possible. Through a GUI, the user only interacts with the requirements and he does not need to worry about the code implementation in a specific language to represent them in a specific tool.

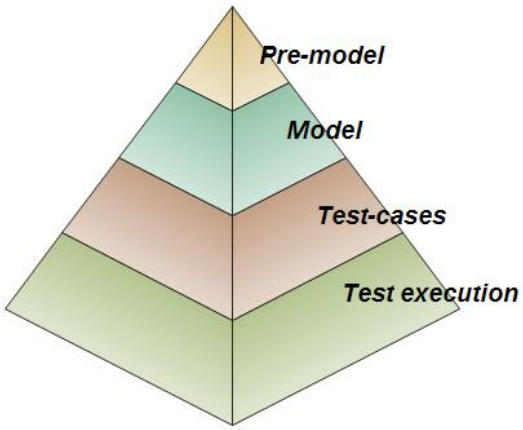


Figure 4.1. Levels of abstraction

As the sequences of commands follow some similar patterns (see section 2.4), it is possible to increment the degree of abstraction here. Thus, an initial representation has been implemented. It is called "pre-model" and it contains all the CLI commands sequence and their parameters together with the conditions necessary to design a correct sequence of commands.

By adding one more level of abstraction, we create a less complex test design environment for users, who simply design their model using an easy GUI, instead of learning a model programming language. Figure 4.1 represents the four levels of test abstraction in the context of our thesis. As we can see, the highest level is always limited by the levels below, i.e., the modeling language used in the model level (QML in the case of this master thesis project) limits the capacity of the pre-model.

An evolution from manual testing to the usage of AMG tool is given in Figure 4.2. The reader can see how the automation of the process has evolved through abstraction because the user is able to change specific details from the highest levels. Even though everything can be modified from the pre-model, the user is able to see and change any detail at any level.

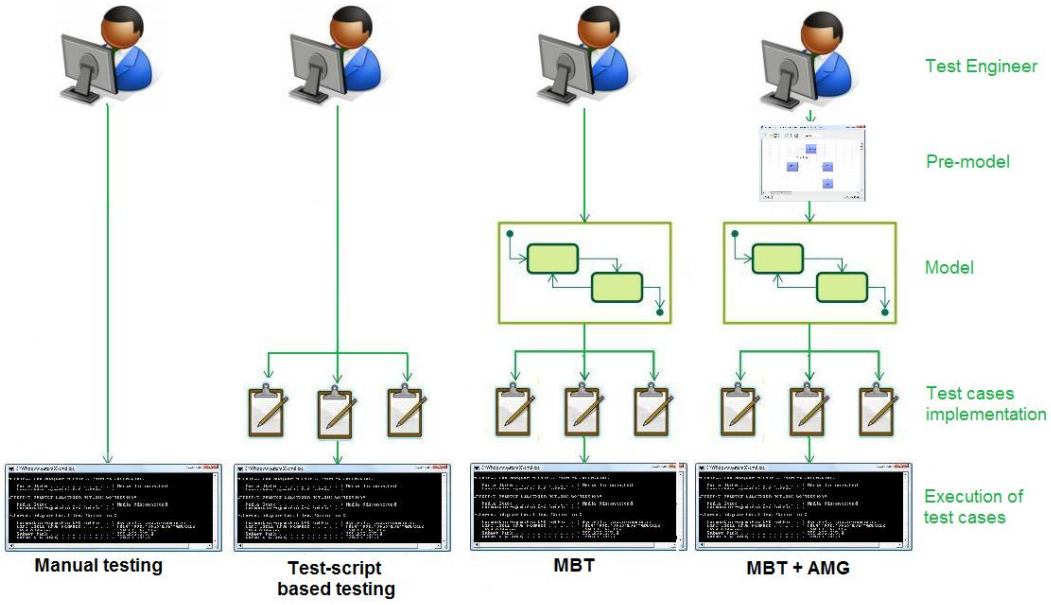


Figure 4.2. Context of AMG tool

Documentation generation

The diversity of formats in requirements documents is a problem when parsing the information to fully automate the testing process. The AMG tool address this issue with two different software modules:

- Document Parser
- Document pretty printer

The document parser is responsible of analyzing Ericsson's requirements documents and extracting the information required for the CLI commands to be modeled. However, the solution to this issue is complex due to the lack of uniformity in the specifications format. Therefore, our aim is to implement a JAVA module which automatically generates documentation from models in a unique format in order to improve the process of parsing. This feedback loop improves the automation of the process of modeling of our tool.

4.1.3. Comparison of Models and Pre-Models

After a careful study on the different approaches of MBT tools, we realized that the used MBT tools are so general and they do not facilitate any help to design or implement the models. The user must know the specific model programming

language required for each tool (usually different tools with slightly different approaches have different programming languages).

Our target is to abstract the user from using model programming language so they do not have to follow any learning process to start using our tool and subsequently MBT processes. The implemented solution is the pre-model which is slightly explained in the previous section.

We know the testers have to implement models composed by <command input, system output> pairs and conditions to join them. Therefore, we are able to create a higher level of abstraction with the common features with the pre-models (see Section 4.1) and translate them into the model described in the desired model programming language with a translator, see Section 4.7.

Models

Models usually represents state charts (see Section 2.5.2 for further details). Therefore, there are two kinds of elements:

- "Boxes" which represent states
- "Transitions" which contain the pre-conditions, action and checking of results.

Pre-Models

Pre-models use a different approach. It is important to stress that is not a state charts because the user is not able to represent states on it. The user is able to define two different elements:

- "Boxes" which represent <command input, system output> pairs.
- "Transitions" which contains the necessary conditions in order to continue the sequence of <command input, system output> pairs.

Example of a Pre-Model and a Model

In this subsection, a simple example is given in order to show the differences in both approaches. Figure 4.3 shows the designed pre-model and the generated model[1].

On the left, the pre-model is illustrated. There is a sequence of two <command input, system output> pairs joined by a transition with a condition. The name

[1]The reader is not asked to understand everything represented in the figure yet. All the details of this transformation are explained in the following sections.

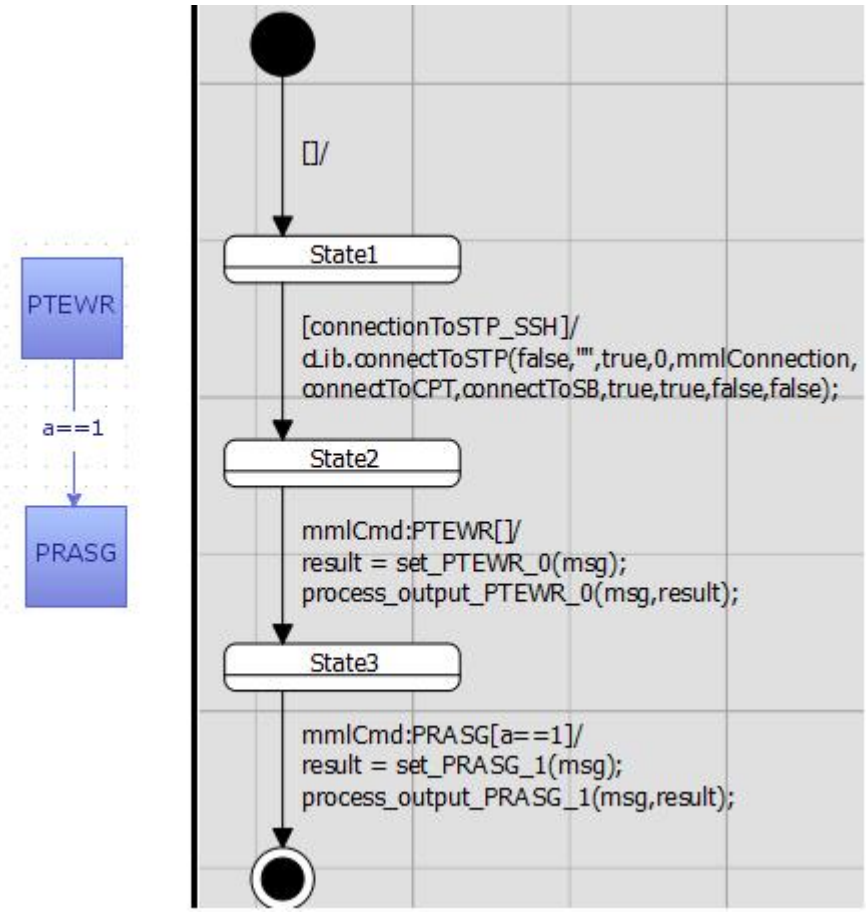


Figure 4.3. a) Pre-model representation (left); b) Model representation on Conformiq (right)

in the box is the same name as the implemented command. The condition can be different verifications from a parameter to the command's output.

On the right, the same information is illustrated but as a state chart. Focusing on the last two states (the first one is the defined pre-conditions in the first command of the pre-model), the reader can see the two commands of the sequence in two transitions. Moreover, this model has an additional file where the value of the parameters are defined.

Note that this is a simple example and all the parts and the translation from the pre-model to the model are explained in the following sections. The purpose of the former chart is just to give a general idea of the different approaches followed to see their differences graphically and in non-case it is not intended to explain them in detail.

4.2. Modularization of the application

The development of the AMG tool is divided into different software modules in order to provide a certain degree of modularity while using an Object-oriented architecture. Figure 4.4 illustrates the different modules, which are the following:

- Document parser: This module is responsible of parsing the requirements documents and extract the required information about the CLI commands to parser, i.e. parameters, range of values, system responses, etc.
- GUI: The AMG GUI is the module where the user can introduce input data to create a pre-model of the process in question. The GUI gives allows the user to introduce the information about each state manually, i.e. without using the document parser.
- XML Transcoder: The XML transcoder creates an XML file to storage the information introduced in the GUI while creating a file that can be used by multiple MBT tools, aside from the used in this thesis (see section 4.7).
- Model Translator: This software module extract the information in the XML files and creating the MBT tool models which will be responsible of deriving the test cases.
- Document pretty printer: This module generates documentation using a certain format which in turn facilitate the document parser's activity.

If the reader wants to go for further details about how this is implemented, Appendix A.1.

4.3. Document Parser

The parser is software which attempts to parse any text specification documents to extract information about the test model. The goal of the document parser is to extract knowledge from product requirements documents, i.e. a specific notation of parameters and system responses for given CLI commands.

The product requirements document is a data set in a human-understandable structure, written by the company in order to specify a set requirements for the features for an existing product or process [58]. In the case of this thesis, Ericsson's requirements documentation is expressed in natural language. In addition, there is not an uniform format for every document, but there is a wide set of different formats in which the documents can be expressed. This represents an additional difficulty to the engineer's task of turning the information expressed in natural language into information expressed in a formal language with a machine-understandable structure.

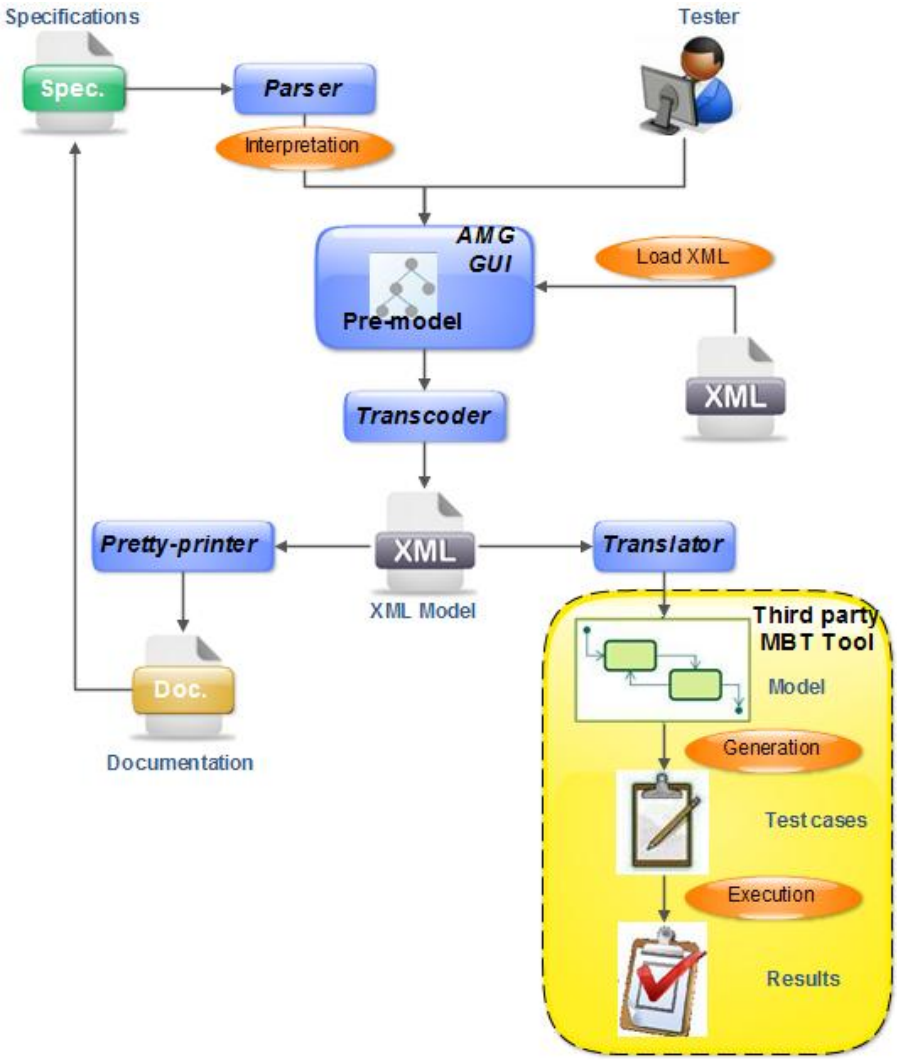


Figure 4.4. Block components of the AMG application.

In the following subsections we will describe the different solutions considered for our tool, and finally, the adopted solution to the problem.

4.3.1. Natural Language Processing

Natural Language Processing is subfield of AI concerned with interactions between computers and natural languages, in terms of exchange of information. This trend leads to the developing of techniques for natural language understanding, which are used to transform human - understandable information into machine-understandable information.

Modern NLP algorithms are based on machine learning and specially, statistical machine learning [79]. This method consists of analyzing a large set of documents and creating a learning model from it, with automatically created rules for determining the part of speech where a certain word belongs, and typically based on the nature of the word in question, the nature of surrounding words, and the most likely part of speech for those surrounding words [79].

On the other hand, other NLP algorithms are based on created rules. The procedure used during machine learning focus on the most common cases, but in the case of manually created rules the effectivness of the algorithm depends on engineer's expertise. In addition, models generated by statistical algorithms are resilient against incorrect inputs, whereas models of manually-created rules can omit this kind of errors, leading to false results [38]. In such cases, in automatic learning models the rules can be more accurate by training the algorithm (i.e., by providing more input data). However, the accuracy of hand-written rules can only be increased creating more and potentially more complex rules, which is a difficult task.

4.3.2. Controlled Natural Language

Natural language is easy to use but imprecise and ambiguous and hence cannot be used as a basis for automatic test case generation without using complex techniques of machine learning to obtain the information required (see section 4.2.1) [69]. However, these methods are hard to use for non-trained personnel.

A solution to this problem is the use of controlled natural languages (CNL) for the formulation of requirements. A CNL is a subset of a natural language where vocabulary and grammar have been restricted in order to reduce the ambiguity and complexity of the text. A CNL is usually restricted to a certain domain. Therefore, before defining a CNL, the domain has to be analyzed carefully in order to specify the set of words allowed in the vocabulary together with the grammar which defines how the vocabulary must combine in order to describe the system [69].

Requirements documents written in a CNL facilitate the pattern recognition and thus, the extraction and analysis of the information required.

4.3.3. Pattern Recognition Algorithms

The pattern recognition algorithms consist of analyzing the product requirements document and establishing patterns in order to acquire the required information for the system. This solution is much less efficient than NLP or CNL because this "wide spectrum" languages more easily capture the specifications at varying levels of abstraction. By contrast, and despite of the efficiency, pattern recognition methods are more easy to implement, because it does not require a

previous training as NLP or CNL techniques nor implementing complex statistical machine learning methodologies. On the other hand, the larger the difference is between the formats used in requirements documents, the larger the number of patterns to be implemented.

4.3.4. Adopted Solution

In the case of this thesis, our tool must work with existing documents expressed in natural language. Therefore, a CNL cannot be used over documents already written in natural language, despite of being the most efficient solution due to its capacity of capturing requirements at varying levels of abstraction. In addition, another important factor in the implementation of the tool is the available time. Given the time restrictions for the implementation of this tool and specifically, for the implementation of the document parser, the adopted solution was a pattern recognition algorithm. Although the efficiency of NLP techniques is bigger, the short period of implementation of these techniques is the key in this module.

Requirements Formats

Ericsson's product requirements documents specify the information about the CLI commands in the following formats:

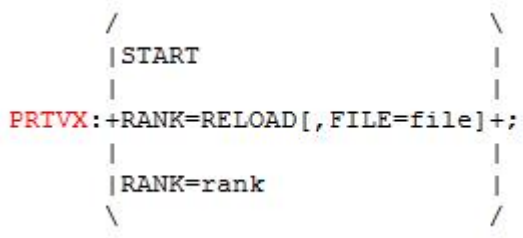


Figure 4.5. In this format, the extracted information must provide the following formats: PRTVX:START (format 1) PRTVX:RANK=RELOAD (format 2) PRTVX:RANK=RELOAD, FILE=file (format 3) PRTVX:RANK=rank (format 4)

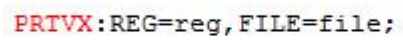


Figure 4.6. In this format, the extracted information must provide the following format: PRTVX:REG=reg, FILE=file(format 1)

The pattern recognition algorithm implemented identifies the different symbols seen in figures 4.5, 4.6, and 4.7, extracting the parameters together with

```
PRTVX [ -a] [ -c] [ -s -Q] [ -d device ] [ -i device -I device]
```

Figure 4.7. In this format, the extracted information must provide the following formats: PRTVX -a; PRTVX -a -c; PRTVX -c; PRTVX - a -c -s -Q; PRTVX -c -s-Q; PRTVX -s -Q ; (...)

the parameters dependencies. In figure 4.5, the symbols "+", "\", "/", "|", "[", and "]" specify the delimitation of the different formats and parameters. On the other hand, in figure 4.7 the only symbols used are "[", and "]'s'. Note that the commands illustrated in the pictures above reflect but are not part of the real system specifications.

4.4. AMG GUI

In sections 4.1.2 and 4.1.3, we explained the reasons to develop a higher degree of abstraction for MBT and detailed the degrees of abstraction present in MBT (see Figure 4.1).

In this section, a detailed description of the GUI implementing this higher abstraction concept is presented so that the reader is able to understand its purpose and function. We will start with an explanation of the GUI followed by the pre-model creation.

The user must define in our tool the sequence as the one defined in the specifications. Using AMG tool the design becomes more intuitive, as the tool introduces a process of specifying subsequent pairs of <input command, system output>. Subsequently, the tool automatically creates a XML model out of this command sequence.

The process from reading the specifications to getting the results is explained in detail in the following subsections.

4.4.1. GUI Description

Figure 4.9 illustrates the different sections of the main window of the application. This screen is displayed when the user executes the program and its from here where he is able to manage all its features.

The following sections describe the user interface of the main application window in more detail.

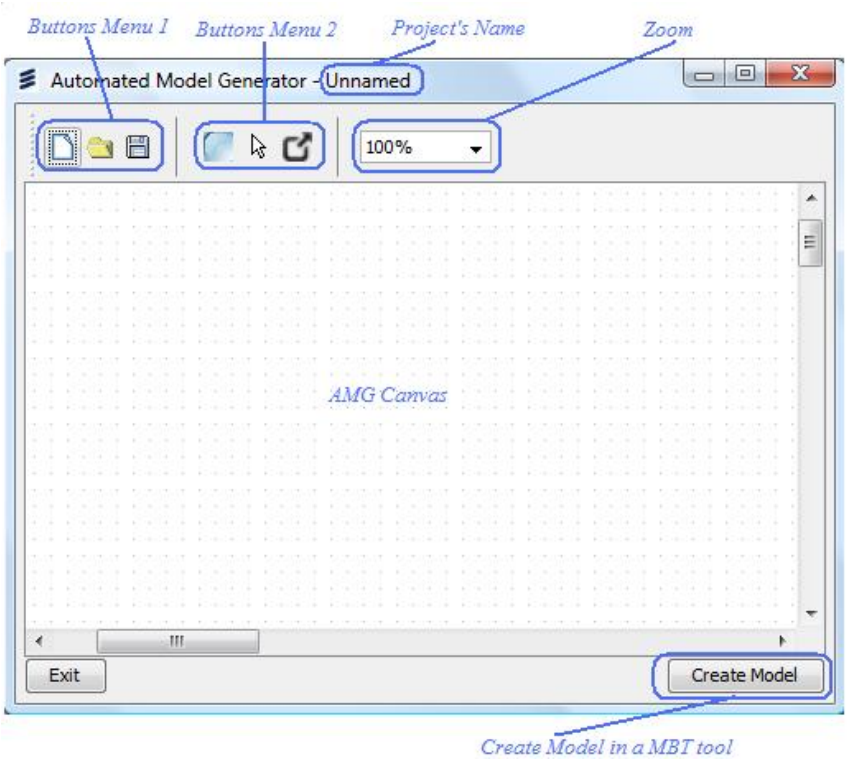


Figure 4.8. AMG GUI sections

Figure 4.9. The main application's window of AMG. The toolbar section consists of a series of buttons to create, open and save a model (Buttons Menu 1), a series of buttons to design the model (Buttons Menu 2) as well as a zoom feature to zoom in and out (Zoom). The name of the project is displayed on the application titlebar (Project's Name). The model is designed in the white area called AMG Canvas. This area provides a grid to help a user better place the model components. Users can use the "Create Model" button in the lower part of the application to create a new model compatible with a third-party MBT tool.

Buttons Menu 1

There are three buttons implemented here, they are respectively from left to right:

- "New pre-model" which removes everything and creates a new pre-model.
- "Load XML file" which removes everything and load a pre-model already implemented in XML format.
- "Save XML file" which saves the implemented pre-model on a XML file.

To check more details about load/saving models in XML format see Section 4.6.

Buttons Menu 2

In this container, there are three buttons implemented too, they are respectively from left to right:

- "COD" Enables the AMG canvas to add more boxes in the chart.
- "Select" Disables the creation of boxes in the AMG canvas.
- "Expand" Open the COD screen (see Section 4.4.3) in order to define the required data on each command.

Project's Name

The name of the current project appears together with the name of the tool.

Zoom

On this box the actual zoom is displayed. Here the user can specify the desired zoom opening the box and choosing one of the options or he can change it pushing CTRL + mouse wheel. In both cases, the value of the current zoom is displayed on the box.

Create Model

Pushing this button the translator (see Section 4.7) starts functioning and the tool translate from the XML format to the model programming language used in the MBT tool.

4.4.2. AMG Canvas

This is the front-end in the AMG tool and its main screen. As stated before in section 4.1.3, the possible elements on this canvas are command blocks (visualized as boxes) each of them containing a command operation description (COD), transitions.Transitions can have conditions to be fulfilled before going from one COD to the next. Mixing both, the user is able to model any sequence of commands.

Each command block contains an instance of the *COD.class* (see Section 4.4.3) with each command's information defined by the user. The user only sees the name of the command in the front of the box in order to maintain the model simple and not overload it with a lot information. However, the user is able to check the information through a tooltip which will appear if the user allocates the mouse on

a box and wait a couple of seconds. The information displayed here is explained in Section 4.4.2.

Moreover, command blocks are joined by transitions which are instances of the *conditions.class* (this class is presented in section 4.4.4). These transitions information but a short description of the condition is hidden as well in order to keep the simplicity.

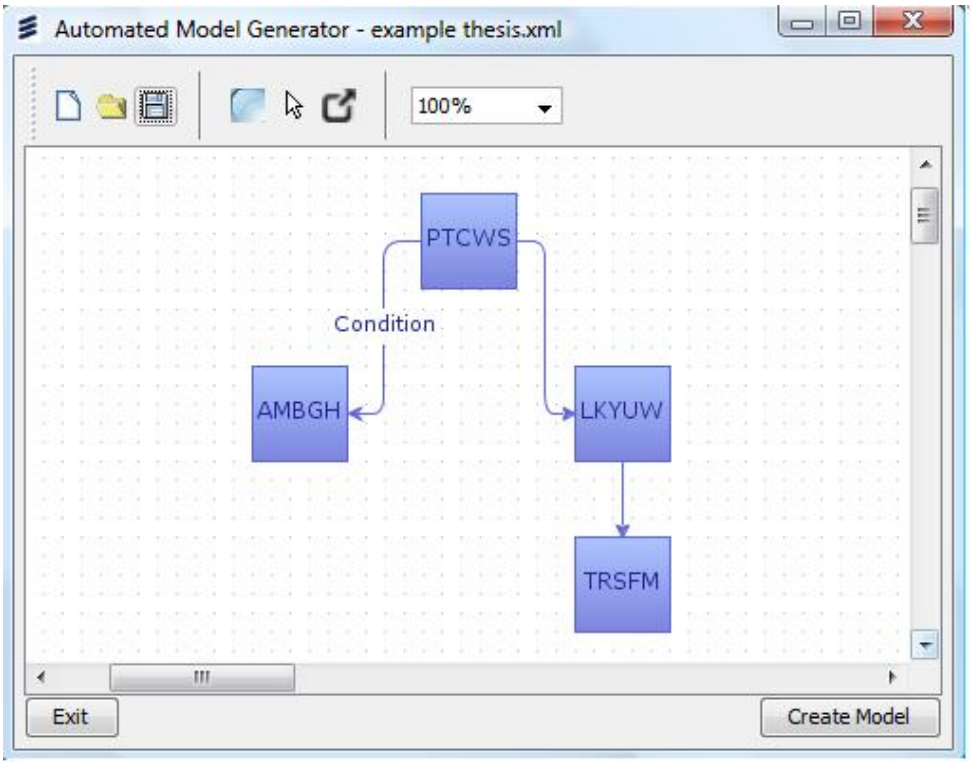


Figure 4.10. AMG canvas with an example of a pre-model

Figure 4.10 illustrates an example model drawn on the canvas. The model has two different paths containing different commands and therefore different sequences. AMG tool give support to this kind of sequence allowing the user to have a more compact model. Note that the design of pre-models are intended to have one origin but can have multiple endings.

Tooltips

As the information displayed on the boxes and the transitions is insufficient to see all the contained information, tool-tips have been implemented to see the contained information from the canvas. When the user places the mouse pointer on

any box or transition a tooltip appears with the information contained inside the box (see Figure 4.11) or the transition (see Figure 4.12)

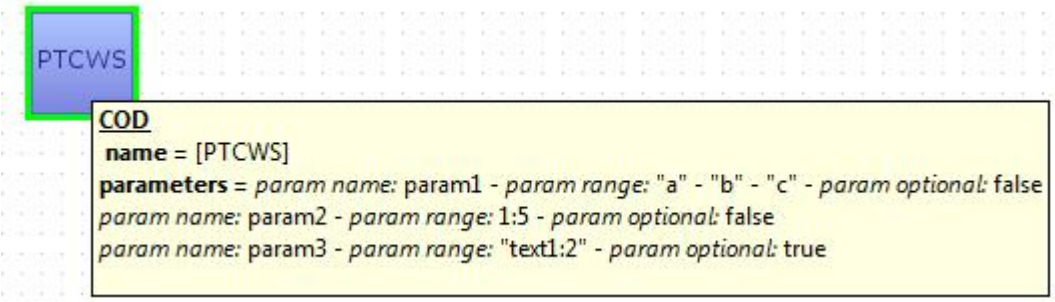


Figure 4.11. Example of a box tool-tip: In the command PTCWS, there are three parameters with the possible values defined

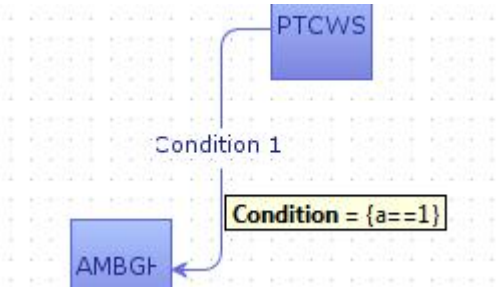


Figure 4.12. Example of a tool-tip in a transition

4.4.3. COD

As the canvas needs to be as simple as possible, the value of each command is stored individually in each box. When the user wants to edit this information, he only needs to select the desired box and push the button "Expand" (see section 4.4.2), and a new screen will appear with the data of the COD.

When the user opens this screen, the command name text-field must be written to continue and update the information in the AMG canvas. The other parameters are optional and they can be filled-in only if it is required for the command that is being implemented. Figure 4.13 illustrates the former example of the command PTCWS (see figure 4.11) with the information filled in.

The parts of the application are described in more detail below.

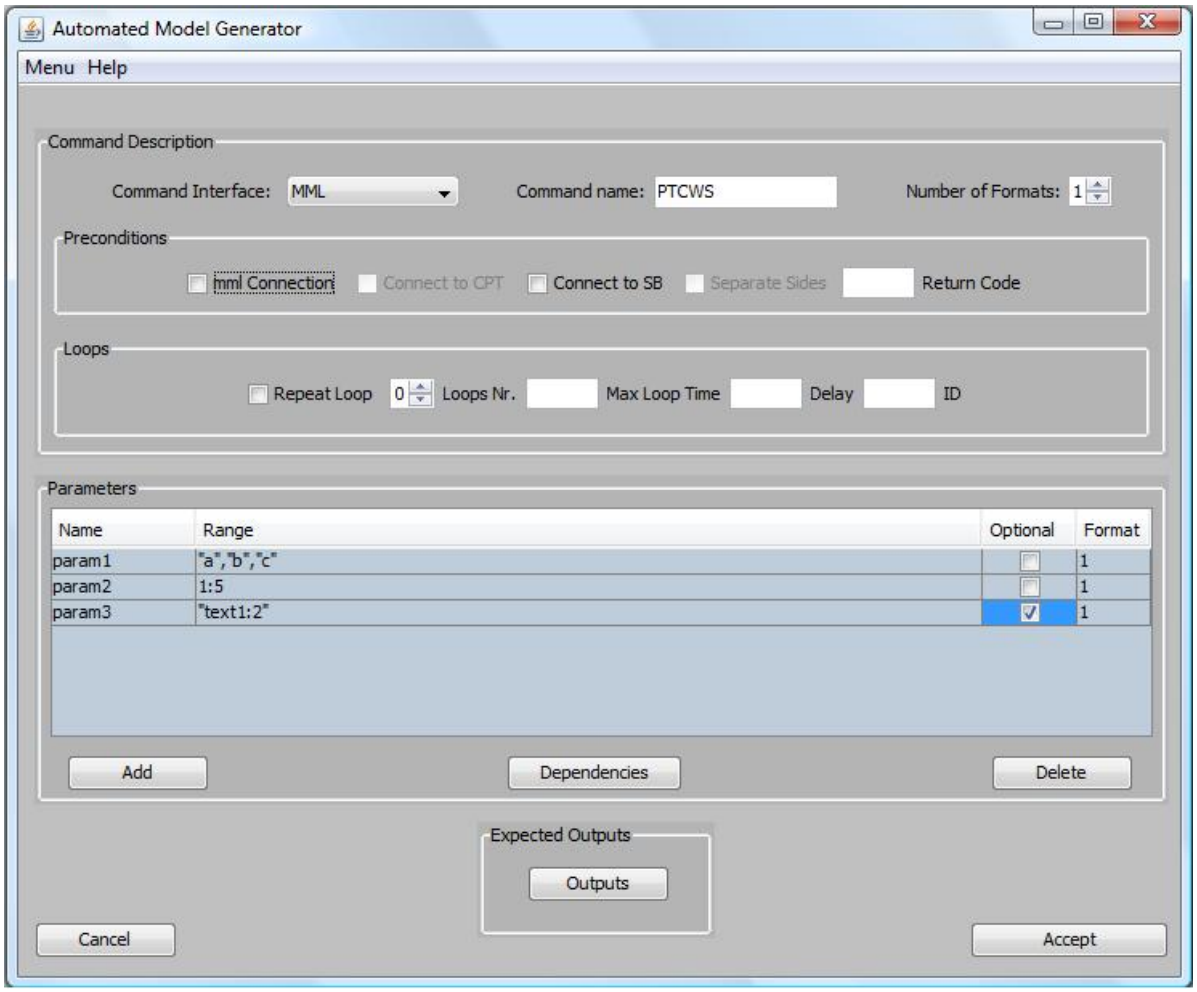


Figure 4.13. COD specification window with an implemented example

Command description

In this section, there are three buttons implemented too, they are respectively from left to right:

- Command Interface: This drop-down box allows the user to choose the kind of interface that the command belongs to. There are two implemented interfaces, MML and APG.
- Command name: it is the only mandatory text field and it contains the name of the command to be modeled.
- Number of formats: the number of formats can vary from one command to another hence, the amount of formats must be specified here.

The implemented command line interfaces are two but the application can be extended to support any other CLI.

Preconditions

Some preconditions are pre-implemented in order to reduce even more the spent time when implementing the pre-model. These pre-conditions are only checked on the first COD of the sequence in the AMG canvas and they are a set of connection options to establish a session towards one of the SUT's shells.

- MML connection: initiates a Man-Machine Language (MML) session.
- Connect to CPT: it connects to the central processor terminal (CPT).
- Connect to SB: Connects to the standby side of the Central Processor (CP).
- Separate Sides: this command separates the CP sides after connection.
- Return Code: The return code of the command for verification of results. By default it is set to "0" (denoting successful command execution), however it may be another one.

Moreover, in order to avoid mistakes, the check-boxes are enabled or disabled depending on the selection of other preconditions, e.g. "separate sides" needs a CPT connection so it is disabled if CPT is not selected.

Loops

Loops can be defined in order to repeat a particular instruction. This allows us to specify a number of repetitions instead of programming a counter in the model. The user can also define the delay between loops or define an identifier. There are five sections:

- "Repeat Loop" Select this checkbox if the COD needs to be repeated.
- "Loops Nr" Select the number of repetitions.
- "Max loop Time" Specify the max time in seconds to keep repeating from current command.
- "Delay" Specify the number of seconds to wait before subsequent repetitions.
- "ID" Identifier binding one inbound and one outbound message to each other.

Parameters

Each command has its own set of parameters with their respective values. Commands do not have to agree on the number of parameters therefore, our tool allows the user to define any number of commands in a dynamic table.

Looking at the Command Specification window, there are two buttons on the bottom of this section where the user is able to add or delete parameters to the command. "Add" button creates a new row in the table for a new parameter. The "Add" button, a new empty line will appear in the table. Clicking on "Delete" button, the selected rows will be deleted. If no rows are selected, the program will remove the last one.

Moreover, each parameter has four characteristics:

- "Name of the parameter" where the user defines the name of the command.
- "Range of values" where the user defines the possible value of the parameters.
- "Mandatory / optional" is a checkbox, if the checkbox is selected the parameter is optional, otherwise it is mandatory. In case of an optional parameter, the model generated will in turn generate test cases with and without the presence of this parameter.
- "Format" is the format which the parameter belongs to.

We will pay special attention to the declaration of expressing parameter values in the GUI. Looking at the current command specifications from the CODs, we have noticed that there are different sets parameter values in terms of their nature (i.e. integers or strings) as well as their value range. AMG supports both textual parameters (alphanumericals) as well as numbers and mixed-type parameters (i.e. alphanumericals mixed with numbers). In order to be able to express all different types of values for parameters, we created the following notation scheme:

- *Text-type values*

  This type of value must be enclosed within quotation marks. If more than one values are possible, they can be separated by commas, e.g. "Text1", "Text2", "Text3". This will create three test cases, the first one with value Text1, the second with value Text2 and the third one with value Text3.

- *Numeric values*

  In this case, the user can define parameters in three different ways. To specify a set of single values, the user must type them separated with commas. To specify a range of possible values the user has two options. Firstly, to define a range from which the model will generate test cases for every single value in the range . The colon character separates the lower bound of the range from the higher bound. Secondly, the user can define a range from which a random

value will be chosen and one test case will be generated with this value. This is expressed using the slash character "/" which separates the lower bound from the higher bound of the numerical set from which the random value is chosen. The following cases are examples of aforementioned notations:

- User Input: 1, 2, 3
  Description: The tool will create 3 test cases, the first one with value 1, the second with value 3 and the third one with value 6.
- User Input: 1:5
  Description: the tool will create 5 test cases, the first one with value 1, the second with value 2.
- User Input: 1/5
  Description: one test case is created with a random value between 1 and 5.

- *Mixed values*

  Numerical and text-type values are sometimes mixed and the parameter's value range is composed by both types. In this case the user has to define the range between quotation marks as if it was a text. Moreover, is is possible to use the different options to handle the numerical values, as was explained in the former section. Examples of this notation:

  - "AD-1" , "AD-4", "AD-65" Three test cases are created, first one with value AD-1, second with value AD-4, and the third one with value AD-65.
  - "AD-1:10", the tool will create 10 test cases, first one with value AD-1, second with value AD-2, third with value AD-3, and so on until AD-10.
  - "AD-1/5", One test case is created prefixed by string "AD-" and suffixed by a randomly chosen integer between 1 and 5.

Section 4.5 explains how our solution translates from this parameter value notation to the model information that will be written in the XML model.

Dependencies

In some CODs, certain command parameters have to coexist with other parameters, but they cannot be present individually. For example, parameter if parameter a depends on parameter b, then test cases with both a and b present and with both a and b not present are valid, but cases with only a or b present are not. These parameter dependencies can be defined in AMG using the dependencies window. When users click on the button "dependencies" (see Figure 4.13), the dependencies frame pops up and the user can define the dependencies.

This frame contains the optional parameters on each format defined in the

command specification window (see figure 4.13). This solution is implemented using java vectors[2] which stores instances of the class JCheckbox. A vector is an open-ended data structure from one side (hence the word vector) - therefore it cannot really get "full". Each optional parameters within a command has its panel containing the check-boxes of the other optional parameters in the same format. Using these checkboxes and tabs, the user can define the dependencies of the owner of the panel and the other optional parameters.

Figure 4.7 shows an example of dependencies between parameters. This format of the PRTVX command has some dependent parameters (defined between square brackets), e.g. the first two dependent parameters are $-s$ and $-Q$. The dependency declaration is illustrated in figure 4.14.

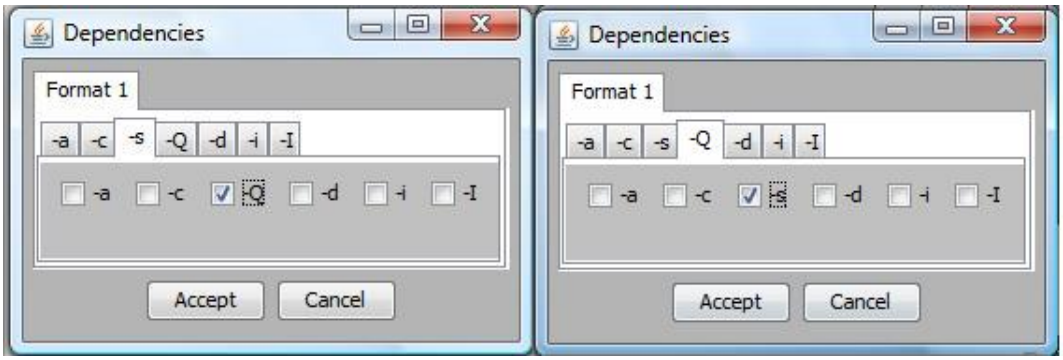


Figure 4.14. Dependencies table containing the names of the optional variables.

What is shown in Figure 4.14 is one of the most basic cases, although we could find dependencies more complex like in the following examples:

<command> param1 [param2 [param3 param4]]

In this expression, we have the following dependencies:

- Param4 depends on param3 and param2.
- Param3 depends on param4 and param2.
- Param2 depends on param1.

Further explanation on why the user must define these parameters' dependencies like this is presented in section 4.5.

Outputs

The Outputs window is accessible from the command specification window (see figure 4.15). Using this window, users can specify the criteria which determine

[2]See http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Vector.html

whether a test case passes or fails (also known as test oracles). Since our solution focuses on CLIs, these test oracles are regular expressions of expected system outputs (i.e. response to a command), which have to match the received outputs in order to pass the test case. The user is able to define one expected output per command format. Figure 4.15 shows the frame where the user defines these expected outputs. The design is made simple in order to maintain this idea of simplicity in the whole GUI. The number of boxes depends on the number of formats which were defined by the user the "command description" section. The GUI displays as many text boxes as number of command formats.

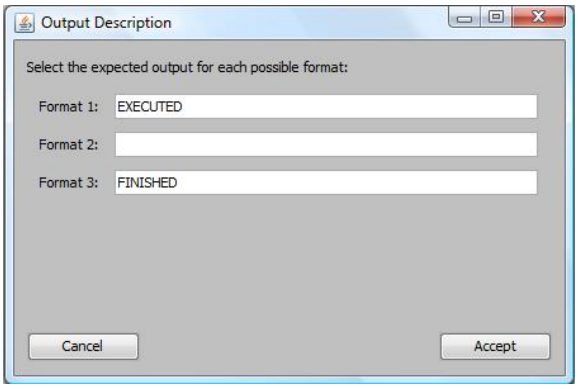


Figure 4.15. Output screen in a command composed of two formats.

We defined a vector where we add or delete text fields, the number of text fields is the same as the number of formats defined in the main screen (see figure 4.13). Therefore, if the user changes the number of formats in the main screen, it is detected and a new frame containing the same number of text fields as number of formats is shown.

Menu Bar

Figure 4.16 illustrates the menu bar in the command specification window (also see figure 4.13). It is composed of "Menu" and "Help". From the "Menu" option, the user is able to start designing a new model, and load and save models from/to XML files and exit the application. The help menu contains a link to the user guide (which is also included in this thesis - see Appendix C).

Note that these XML documents only contain the information of a simple command and hence, it is not the XML model that was described in section 4.4.1 because it does not contain the sequence information and other commands information. The storage and retrieval functionality of individual commands was implemented in order to reuse commands created in previous models, or store commands for future

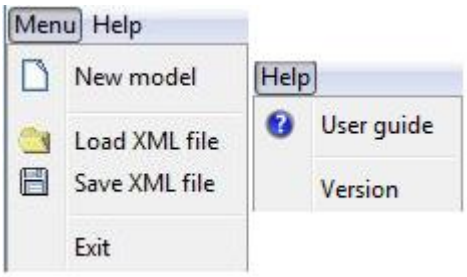


Figure 4.16. On the left, the "menu" is open and on the right, the "Help" is open.

use. This helps shorten the time required to design models in AMG, especially if a sufficient set of CODs were defined and stored in the past.

The reason why we decided to put a menu instead of allocating these options on the main application window is that there was already enough information in the frame so we decided to hide some options. These options are "new model", "load model", "save model", the user guide or the current version of the program.

The functionality of these buttons are presented below:

- "New model" which deletes all the information defined in the GUI.
- "Load model" which loads a XML model.
- "Save model" which saves a XML model.
- "User guide" which links the user to the user guide (it is uploaded in an Ericsson's internal website)
- "Version" which shows a panel with the information of the version, authors and place.

4.4.4. Transitions

The transitions are needed in the representation of a sequence in order to define a condition between two commands or when multi-paths are represented in a model. The transitions are defined as instances of the class "conditions" and these instances have two fields. On the top of the screen, a short description field appears in order to write a short description of the condition (this field can be in blank). Below this, a text-field is used to define the condition. These conditions are boolean expressions where the value of a variable is checked.

Moreover, there are two attributes that will be specified when this instance is created, the target and the source which refer to the boxes that are linking. If the user decides to write something here, this text will appear in the transition. This objects are represented as frames and their design is shown in Figure 4.17.

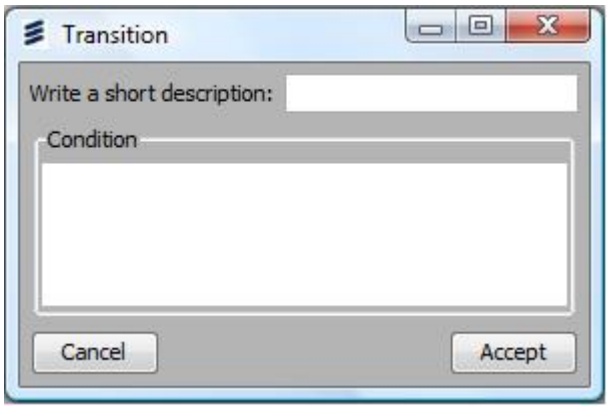


Figure 4.17. Transition frame

4.5. Storing the information

This section explains in detail how we collect and process all the information taken from the GUI. There are two kinds of informations, on the one hand there is information related to the commands' sequence, and on the other hand there is information related to the commands themselves.

4.5.1. Command block and transition storage

Storing command blocks

As stated before, each command block contains the information of one command. These commands are modeled from CODs (see section 4.3.4.1) and therefore share some common properties. An instance of the Java class Hashmap contains all the CODs. We chose a hashmap because this class allows you to get any of the stored COD instances given a key, because all the objects stored in the hashmap are linked to a key defined when this object is created.

Upon creating a new command block, a new ID-number is generated and assigned. Using this key, we can retrieve the command block from the hashmap. Once the user defines and stores the information of a command, a new COD instance is created and stored in the hashmap with the box's ID as the key. Therefore, we achieve a relation between the stored information in the hashmap (useful information to create the XML model afterwards) and the graphical information (useful for visualizing of the sequences). Figure 4.18 illustrates the storage of command blocks defined in AMG's user interface (left) to a hashmap storage structure (right). A unique identifier known as "key" is assigned to every command block and is used to retrieve the stored information for future use.

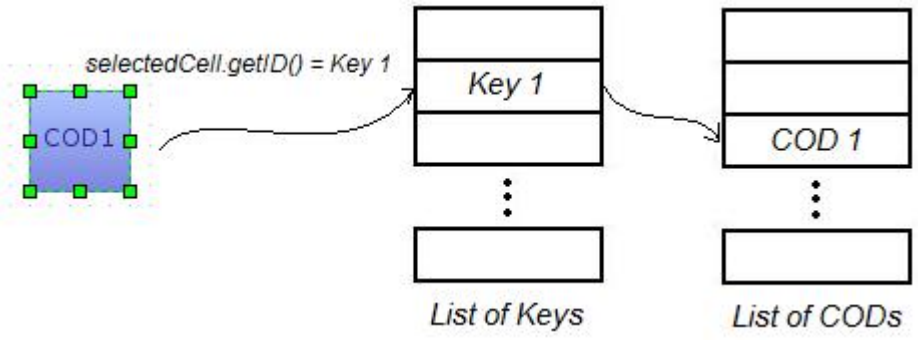


Figure 4.18. Creating/Editing a COD

Storing transitions

The Jgraphx library used by AMG, was created for handling state-machine diagrams and it does not offer the possibility to create an ID number for transitions. Given the lack of an ID number as a unique identifier, storage of transitions cannot be implemented using a hashmap. The lack of an ID number makes hard to implement a hashmap with the transitions' information. Therefore, a list was created as storage structure.

This list contains instances of the class "conditions", as it was described before in section 4.4.4. If the user decides to create a new transition, a new instance of the class conditions is created in the list and the respective transition is drawn in the canvas between the two commands. If the user decides to edit an old transition, the list's stored condition objects are checked one by one. The selected transition target and the source are compared with the transition targets and sources stored in the list through an iterator. When they match the user is able to modify this transition and it is restored again with the new changes.

4.5.2. Conditions and dependencies storage

Transititions

As said in previous sections, the test scenario is a series of CLI commands. In the model, this scenario consists of two parts: States (i.e., command blocks), and transitions (conditions and the triggered actions between command blocks). Once the sequence of commands is properly modeled in the GUI, an easy and intuitive way of describing the model is required in order to storage that information in a XML document. The models are described as sequential maps using a boolean matrix. The rows represent the outputs and the colums represents the inputs.

The cell (*i, j*) has a“true” value if there is a transition from the state *i* to the state *j*.

Dependencies

A parameter dependency refers to the situation where a command parameter only appears in the CLI if another parameter does. Figure 4.5 illustrates this concept (the variable “FILE” can only appear if the variable“RANK” does). To solve this question we considered two techniques: Backus–Naur Form (BNF) grammar, and B-tree data structures. BNF notation allows a better coverage of the dependencies but requires more time of implementation due to the wide range of commands and formats. On the other hand, B-tree data structures, i.e., a tree data structure that keeps data sorted and allows sequential access and insertion, require less time of implementation and the user can easily specify the dependencies. Therefore, and taking into account the time constraint, the approach used to solve this issue was the B-tree data structure (see Chapter 6, Future Perspectives).

We will use the following example to illustrate this concept: *PRTMT A [B [C [D E]]]*. In this example, the command PRTMT has 5 parameters, where B, C, D and E are optional at different levels (see Figure 4.19). Each parameter dependency only has to be assigned to the parameters in the same level or the nearest top level. This means that a series of assignments at different levels will cover the dependencies between all the parameters of the command.

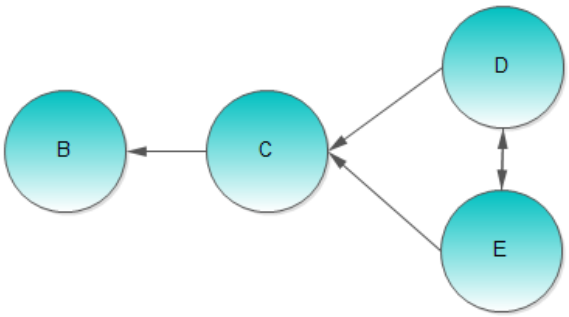


Figure 4.19. Example of parameter dependency

4.6. XML Transcoder

The XML transcoder module analyzes the information from the pre-models created in the GUI (see section 4.4) and generates an XML file which helps 2 main purposes:

- Storage of Information: This software module contains the classes *XML-Reader*, and an *XMLWriter*, which use the Document Object Model (DOM) interface [4] to read and write XML documents.
- Integration with different MBT tools: The XML format allows the stored models to be used by different MBT tools in order to provide flexibility to the tester.

4.6.1. XML schema

An XML schema defines and describes the type of XML document generated. For this thesis project, the xml schema language used is XML Schema [25]. XML Schema is the W3C-recommended schema language for XML. It provides a means for defining the structure, content and semantics of XML documents.

One of the greatest advantages of using XML Schema is the support for data types, which facilitates the definition of data patterns and formats, and to convert data between different data types. In addition, it is easier to validate the correctness of the information [60]. XML Schemas also ensure consistency of the data format, i.e., the sender and the receiver will expect the same document format, thus avoiding missunderstandings in the content of the document.

The open-source Java library, JGraph [11], used in the GUI to depict the pre-model (see section 4.1.3), automatically generates an XML document with customized tags that Jgraph uses to identify the different nodes and transitions of the model. This preliminary XML document is parsed by AMG in order to load the model. This feature of the library facilitate the task of generating the XML files from the models. The information users enter in the GUI (see section 4.4) is then added to the JGraph-like XML documents following the rules established by the XML Schema. Figure 4.20 illustrates this process.

4.6.2. Parsing Methodology

The most widely used parsing methodologies are *Simple API for XML parsing* (SAX) and *Document Object Model* (DOM). DOM is a interface that allows to dynamically access and update the content, structure and style of XML documents [76]. DOM is a tree-based parser, which allows traversing in any direction. DOM can both read and write in the XML documents. In addition, the user can easily update the document, adding or deleting nodes because DOM uses a tree-based data structure for storing the XML document and facilitates the random access and manipulation of the XML data [24].

On the other hand, SAX is a open interface based on JAVA language which uses a streaming method to analyze and get the information from a XML document

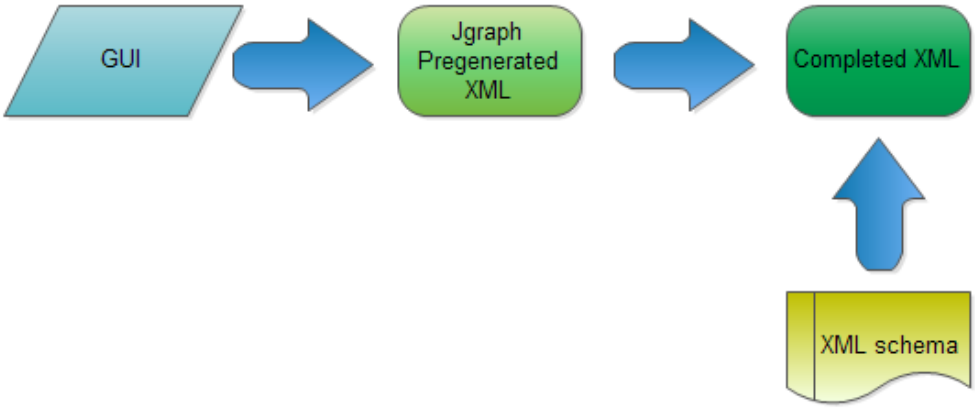


Figure 4.20. Transcoding of user's data

[76]. SAX presents the document as a serialized event stream (i.e., a sequence of calls to a handler function as each chunk of XML syntax is recognized)" [24]. It occupies less memory than DOM, and generally runs faster. However, SAX can be only used for sequential processing of the XML documents, and these can not be manipulated.

Table 4.1. SAX vs DOM

	SAX	**DOM**
Parsing	Parses node by node	Store the entire XML into memory before processing
Storage	Does not store the XML document in memory	Ocuppies more memory
Node Manipulation	Can not add or delete nodes	Nodes can be added or deleted
Node Traversal	Top to bottom traversing	Traversing in any direction
Structure	Event-based parser	Tree-based parser

For the purposes of this master thesis's project, which requires the creation of XML documents and the continuous update of these, the chosen methodology is DOM. Despite of the less efficient usage of memory, the features described above (see table 4.1) fits better with the design requirements of our tool. The class *XMLTranslator* parses the XML document and extract the information as detailed above (see section 4.5). The information is then transfered to the "model translator" module (see section 4.7).

4.7. Model Translator

Model Translator is a JAVA-based software module, whose function is to generate a final model which will be used to generate the test cases to be executed by the SUT. The input to this module is provided by the class *XMLReader* (see section 4.6, which parses the information stored in the XML file and transfer the data in the format described in section 4.6.2. This information is analyzed by the *Model Translator* in order to create the model. Subsequently, a third party MBT tool is used to generate test cases: Conformiq Modeler™and Designer™Tool [3]. This approach is shown in Figure 4.21.

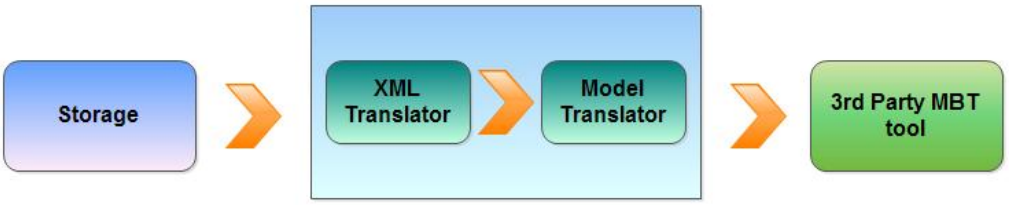


Figure 4.21. Test Adaptor

4.7.1. Model Adaptor Templates

Conformiq Designer™is an MBT test generation tool which automatically generates test cases out of models. This tool tries to achieve maximum coverage of requirements while at the same time keeping the number of generated test cases as small as possible. The advantage of using this tool is that the models it uses for input are created in XML, and therefore, AMG can easily create new models automatically. The format of the models consists basically on two parts: A nominal part, and a graphical part. The nominal section stores the number of states (operations) and their content, as well as the transitions and the different conditions to trigger each transition. On the other hand, the graphical section stores all the characteristics related to the visualization of the model, i.e. location (coordinates) and size of each state, length of transitions, etc.

Model Translator uses the XML model generated previously (see section 4.6), to create a new Conformiq model using an approach based on templates. These templates consider the number of states and transitions of a certain model and determine their position on the Conformiq canvas. This information is inserted into the template together with the rest of the data required for Comformiq to depict the models.

4.7.2. Qtronic Modeling Language

The final step is the model generation is the generation of the code in Qtronic Modeling Language (QML). QML is an object-oriented language that is used in Conformiq Designer to specify models using textual notation. A JAVA-derived language is combined with UML state-machines in order to design models. As seen in Figure 4.22, notation of transitions has basically three parts specified as *guard[trigger]/Action*, where:

- Trigger: It specifies the pattern of data to match and switch to the next state. Here, we typically describe a command incoming to the system from a tester, e.g. an incoming TRSFM command on the mmlCmd interface.
- Guard: It specifies a boolean condition for the transition to fire.
- Action: It specifies the action to perform if the transition fires. An action contains a block of QML code which can be in the XML (XMI in Conformiq notation) file itself or in a source code textual file (CQA type). In this code we typically describe the values of parameters of the incoming command declared in the trigger, as well as the expected system response.

CQA Library

QML is used to specify the logic of the model. A CQA file is created to write the methods for treating each command. Each method consider the number of formats of parameters of each command and combines them to create the different test cases. In order to achieve this goal, every interface and command must be previously specified in another CQA file using the QML notation. The function of the CQA library is to define the possible inputs and outputs of the test domain.

The inbound and outbound interfaces are first specified, together with the allowed commands in each interface. These commands are then specified along with their paramaters according to the requirements document. QML allows *Integer*, *String* or *Boolean* parameters. if optional, the tag *Optional<TypeOfParameter>* is placed before the parameter in question. The following example will illustrate these concepts:

```
1 system {
2     /* Back-end interfaces */
3    Inbound mmlCmd : PXRLT;
4    Outbound mmlPrt: EXECUTED;
5
6     /* User connection interfaces */
7     Inbound userIn : telnet, ssh, ftp, sftp, comCli,
         comNetconf;
8     Outbound userOut : stdOut;
```

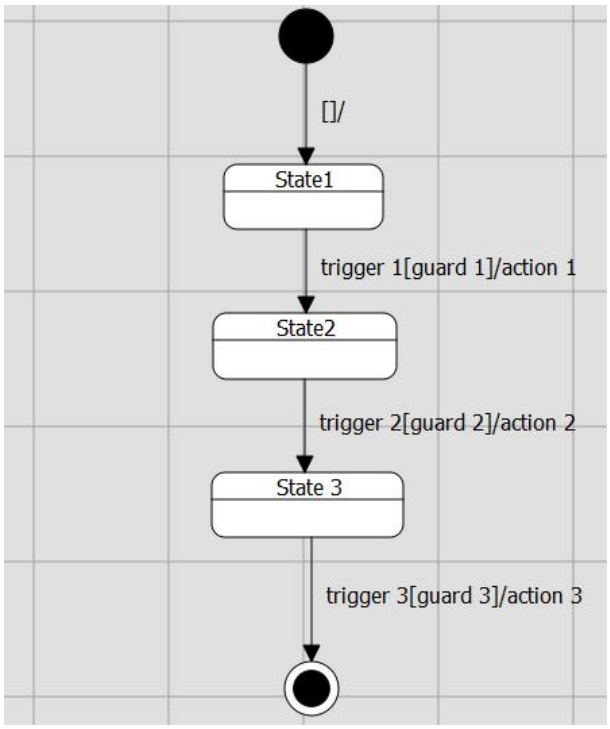


Figure 4.22. Conformiq model example generated a pre-model. Note the complexity of this model, which uses UML-specific notation, as opposed to the simplicity of the pre-model illustrated in figure 4.3, which does not require any special knowledge to create. In addition to this graphical part, AMG also generates a complement of code in a JAVA-derived language used by Conformiq.

```
9  }
10 record EXECUTED {
11     /* Procedure printout for MML commands */
12     int sessionId = 0;
13     String printout = ":\nEXECUTED\n:";
14     String matchMethod;
15 }
16 record PXRLT {
17     int sessionId prefer 0;
18     int timeout prefer 60;
19     String reg;
20     String data;
21     Optional<String> hidata;
22 }
```

As seen in the example above, two inbound interfaces are declared, *mmlCmd* and *userIn*; and two outbound interfaces, *mmlPrt* and *userOut*. For simplicity reasons, in the example only two of the commands are shown (*PXRLT* and *EXECUTED*).

The *RECORD* field defines the messages allowed in an inbound or outbound interface. Each record field has a series of parameters which can be string or integer and have a predetermined value by using the special command *prefer*. The parameter *hidata* in the record field *PXRLT* is declared as optional, i.e., the parameter can appear or can not appear in an incoming message *PXRLT*.

The information about the CLI commands extracted from the XML document must match with the existing CQA library of the system.

Command Functions

As said above, every inbound command has a number of fields, integers or strings. The combination of these fields results in a number of different test cases which specify all the possible combinations for a particular command.

The data combinations for each command is processed in a function which is specified together with the rest of the commands' functions in a separate CQA file.If the *combine-all* construct is used, Conformiq Designer calculates a set of all possible combinatios in the region and generate a test for each combination. The class *XMLTranslator* parses the information about every parameter and extract the data about the number of different formats, parameters, outputs required for each format,etc., which are then used to create the functions.

For Example

```
public int set_PTPBP(PTPBP incomingMessage){
   combine_all{
         require(
         ((incomingMessage.bn =="0")
        &&  ((incomingMessage.addr =="1")
                 || (incomingMessage.addr =="2")
                 || (incomingMessage.addr =="3")
                 || (incomingMessage.addr =="4")
            || (incomingMessage.addr =="5"))
        &&  ((ispresent(incomingMessage.nrw))
                && (incomingMessage.nrw =="3")
                       || (incomingMessage.nrw =="4") )
                 || (!ispresent(incomingMessage.nrw)) );
   }
        return 0;
}
```

In the example above, we can see the case of one single command with 3 parameters. *require* checks that the boolean argument supplied is true. *incomingMessage* is the instance of the current message. The three arguments are: *bn*, *addr*, and *nrw*. The *OR* operator specifies the range of alues that a particular parameter can

take. The *OR* operator is also used to difference between two different formats of the same command. In addition, for those parameters marked as optional, *ispresent* checks that the parameter in question is present in the message.

4.8. Pretty-Printer

The documentation generator known as pretty printer allows us to generate a readable textual document using the information encoded by the XML model. The pretty printer has some predefined rules that can be configured in order to achieve the desired format for the generated documentation.

Maintaining and/or creating new documentation according to the software developed is solved in our tool which automatically generates the documentation taking the XML data. Furthermore, this generated documentation can be read by the parser without parsing errors hence the program can re-read the generated output to re-generate the XML model. This facilitates making transformations in the documentation which lead to changes in the model.

If the documentation comes from another source or it was not generated with this pretty printer, then the parser will make its best-effort attempt to generate a pre-model, but the resulting pre-model must be reviewed by the tester. Although the first representations of the documentation are sometimes inaccurate, they will be improved with time. The final result will be a common documentation format that is fully machine readable.

Chapter 5

Analysis of Results

AMG has been tested on an Ericsson's telecommunications system. To collect the data required for the analysis of the efficiency and the added value of the AMG tool, we have established two parallel tracks of work. The first track of work consists of testing our tool with models of different complexity to measure the efficiency of testing when using AMG. The second track consists of a survey completed by a team of 15 testers in order to conduct a research study about the value of MBT versus traditions testing techniques based on the experience of the testers.

5.1. Testing Scenario

In order to perform correct measurements on the benefits of using the AMG tool, a careful study has been carried out. The purpose of this study is to find out the complexity of the models that are going to be implemented.

We have assumed that the complexity of a model depends directly on the number of parameters and their values. A model with a high number of parameters and values is translated into larger implementation time in comparison to a model which has fewer parameters and values. Thus, some charts includes the nature of the commands to model.

As we mention before in section 2.4 the test domain, where our tool is used, is CLI commands. We distinguish between CODs and OPIs, therefore an analysis of each one is carried out.

5.1.1. Analysis of CODs

Figure 5.1 illustrates the number of commands with a specific number of parameters. The reader can see that most of them have less than 5 parameters and it can

be said that it follows an exponential decay where the most common are commands with zero parameters. This statistic has been calculated with about 100 commands chosen randomly from the specifications sheets.

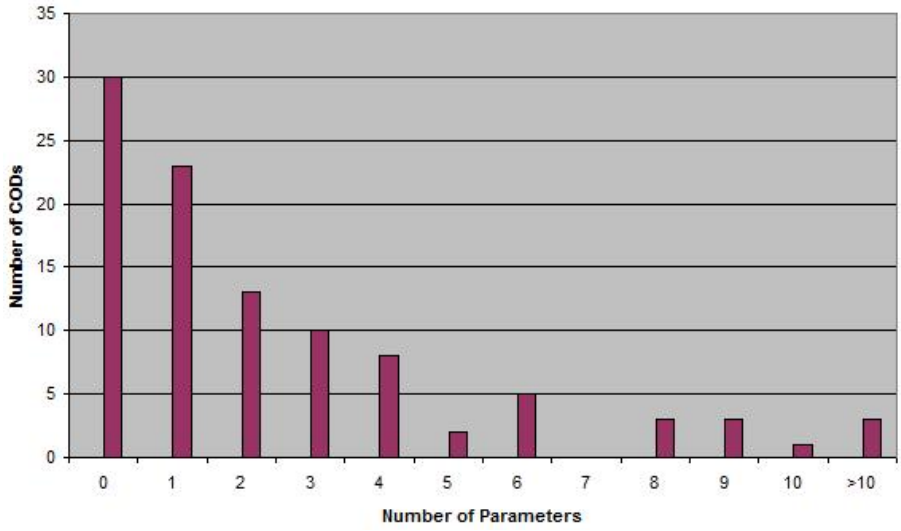


Figure 5.1. Analysis of the number of parameters of CODs

In order to be able to make a later comparison,we have consider to define three different scenarios depending on the number of parameters of each COD. The different commands used in every scenario are not real and serve illustrative purposes only. We will consider three different ranges:

- A command with two or less parameters
- A command with three or more parameters but fewer or equal to five
- A command with more than five commands

Figure 5.2 shows the percentage of commands within each range.

Scenario 1

Scenario 1 includes the commands with two or less parameters. As seen in Figure 5.2, most of the CLI commands belong to this range. We will consider the command *PTCFP* (see the PTCFP specification in the Appendix D), which has two parameters: *ADDR*, which defines an address with an hexadecimal number from 0 to H'3F, and *NRW*, which is an integer with values between 1 and H'40. In addition, the parameter *NRW* is optional.

In this scenario we will consider a range of values from 0 to 2 with different ranges in the parameters. Table 5.1 shows the amount of time needed to create the

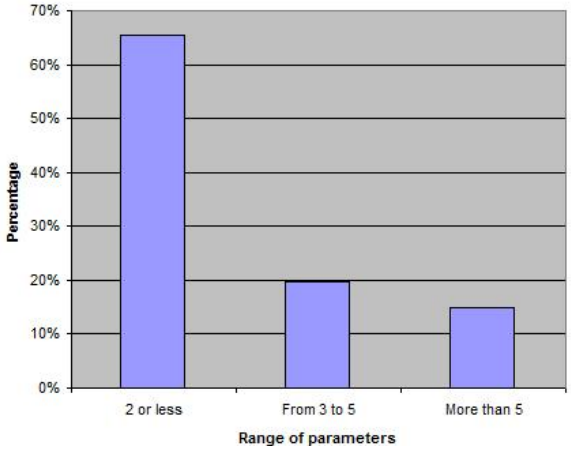


Figure 5.2. Percentage of the different CODs' ranges

model and generate the test cases. The time is measured on the implementation of both ADDR and NRW. Consequently, the expected number of derived test cases is 12.

In the last column, an average of the three precious columns is shown. Note that this time is the average of the efforts of three people who had experience with both AMG and Conformiq Designer and designed models.

Table 5.1. CODs - Scenario 1

Technique	Time 1	Time 2	Time 3	Average
MBT	10'	13'	12'	11' 39"
MBT + AMG	3' 20"	6' 45"	8'	6'

From our measurements, we calculated the improvement from using AMB versus using Conformiq in this scenario to be 195%.

Scenario 2

Scenario 2 includes the commands with a number of parameters between 3 and 5. The complexity of this scenario is larger than in the previous scenario, so a higher number of combinations and, consequently, test cases can be generated. We will consider the command *PTWRP* (see the command specification in Appendix E), which has 4 parameters: *DUMP*, a string-type identifier which can take values from 4 to 10 characters; *REG*, another string-type identifier which can take values from 3 to 15 characters; *RPBH*, a integer with a value from 0 to 9; *RPBIS*, a string-type identifier. The command specification contains two different formats. The parameter RPBH is optional.

For this scenario, we will consider a range of three values for the RPBH parameter (the only parameter which can take values). The number of test cases derived from the model is 462. Table 5.2 shows the amount of time needed to create the model and generate the test cases.

Table 5.2. CODs - Scenario 2

Technique	Time 1	Time 2	Time 3	Average
MBT	27'	38'	30'	31' 39"
MBT + AMG	9'	10' 37"	13'	10' 52"

From our measurements, we calculated the improvement from using AMB versus using Conformiq in this scenario to be 298%.

Scenario 3

In this scenario are included the commands with more than 5 parameters. These commands represents the 15% of the total of CODs. In this case, we will model the command *APFPL* (see the command specification in Appendix F). *APFPL* has 6 parameters: *CP*, a string with 1-7 characters; and a series of optional parameters which requires no values, such as *k*, *l*, *p*, *q*, *s*. The command admits 5 different formats.

For this case, we will consider, as in the previous scenarios, a range of three values for each parameter. The number of test cases derived from the model is 50. Table 5.3 shows the amount of time needed to create the model and generate the test cases.

Table 5.3. CODs - Scenario 3

Technique	Time 1	Time 2	Time 3	Average
MBT	40'	34'	30'	34' 39"
MBT + AMG	8' 30"	9'	11'	9' 30"

From our measurements, we calculated the improvement from using AMB versus using Conformiq in this scenario to be 364%.

5.1.2. Analysis of OPIs

In order to analyze the complexity of the OPIs, we assume that this complexity is directly proportional to the number of actions an OPI has. In order to understand this measurement the reader must assume an ideal case where all the actions have the same complexity. To give a valid measurement, the complexity of the actions of each OPI are assumed to be on average the same.

Keeping this in mind, a direct correlation between complexity and number of actions can be established. Figure 5.3 illustrates the percentage of the number of actions per OPI. This statistic has been calculated with data retrieved from 100 OPIs chosen randomly. It can be seen that most of the OPIs are composed of sequences between 10 and 50 actions. Note that the nature can be either check a condition to continue or to execute a COD, so 50 actions do not mean necessary a sequence of 50 CODs.

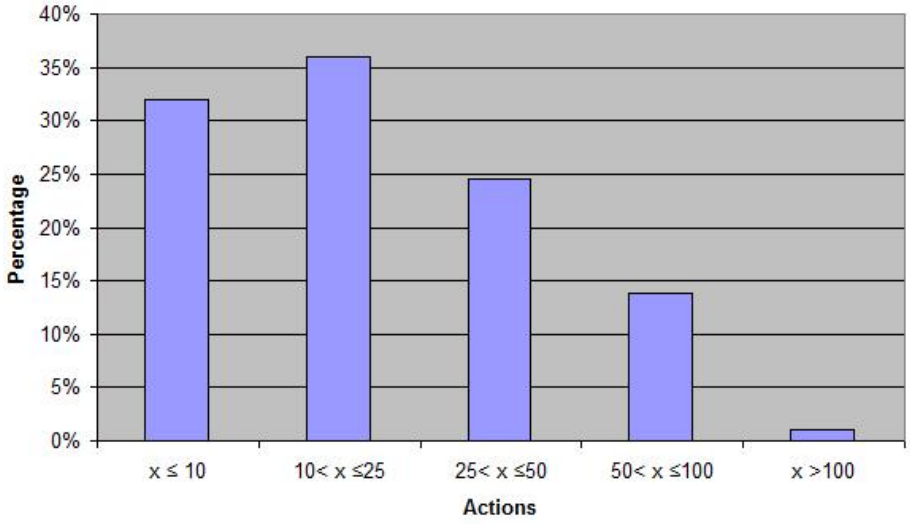


Figure 5.3. Percentage of the different OPIs' ranges

In the case of OPIs, we have performed a research to measure the improvoment of the AMG tool in the MBT context. We have divided the OPIs in 4 categories, based on their complexity. We will conside r the averages expressed in Figure 5.2. Table 5.4 shows the times measured in the different categories.

Table 5.4. Comparison of times in OPIs

Number of CODs	Technique	Time
5	MBT	60'
5	MBT + AMG	12'20"
10	MBT	2h45'
10	MBT + AMG	25'
50	MBT	16h
50	MBT + AMG	1h45'

Table 5.5 shows the percentage of improvement given by our tool. As seen in the obtained results, the efficient of the AMG tool increases with the length of the OPI, due to the simplicity provided by the automation of the model generation.

Table 5.5. Improvement in OPIs

Number of CODs	Improvement
5	489%
10	660%
50	914%

Different techniques have been developed in order to improve the efficiency of V&V processes. In this paper, some of these techniques have been presented and compared. The adoption of MBT is not a trivial decision as it requires a change in the way of thinking across the organization as both engineers and managers have to change the way they view testing. However, we have found that the ease of creating test cases with AMG together with the use of a capable MBT test case generator can be beneficial in the long term.

To prove our claims, we conducted a survey on different testing techniques currently used. This survey was distributed among professional testers, some empirical results were stored and processed and they have been used to obtain conclusions. The purpose of the survey was to acquire solid data about the different techniques that testers of the organization were using. The answers give us an average time estimation for creating a test case using each technique, as well as the average number of test cases created per day per test technique. In addition, we measured the adoption rate of MBT within Ericsson (see section 5.1.5). We received 15 answers from the testing team, with a relatively low variance in their responses (as explained below).

5.1.3. Comparison of the different testing approaches

We have observed that the average number of test cases performed per person and day, within a Ericsson SCRUM (an iterative software development method) team, using traditional manual testing is 18. In the case of test-case scripting that number is reduced to 2, while using MBT (without our tool) 23. We consider a SCRUM Agile sprint (the basic unit in SCRUM development) to be 3 weeks long.

Traditional manual testing is suitable for a small number of test cases, however, the larger the number of test cases, the less effective is the manual testing approach. As observed in Figure 5.4, the time needed to run a test case manually is shorter than writing a whole script, but it has to be perform again in every iteration.

On the other hand, the test-case scripting approach is adaptive, but with the disadvantage of high cost in terms of time. Test-case scripting is a costly approach as it requires a great deal of time to create the necessary test scripts. Additionally, test case coverage could seem insufficient for the great cost in time and effort, but it is in long-term planning where we can find its real value, which is the possibility of being used for regression-test environments. As seen in the results, this testing approach

becomes less effective in time, where larger amounts of test scripts are available and the scripts have to be modified individually to fit new deliveries of the SUT. In comparison, when using MBT, a single model can be used to automatically generate test scripts, reducing the need to maintain each test script individually. Upon a new delivery of the SUT, the only change needed is to edit the model, and not every test script, as scripts are generated anew from the model as soon as the editing process is complete. This makes MBT an approach suitable for regression testing, saving time and thus increasing the amount of delivered software per iteration in an Agile development environment.

Given a standard test suite comprising of test cases of 10 CLI commands in sequence, our tool can create the model in 10 to 40 minutes, depending on the complexity of the process to model. Each model is able to generate a large combination of test cases at a time. In addition, the complete model, consisting of the required UML logic and JAVA-like source code, is automatically created, so the time needed for modeling and generation of test cases is drastically reduced. According to our research, the average time to write a test-script is 4 hours, and the average time for manually create a model, 2.7 hours. This is a significant saving in time for both manual MBT and test-case scripting, because the great disadvantage in both of them, as expressed by the consulted testers, is the time spent by the tester in writting the code of the script or the model.

5.1.4. Comparative Graph

We collected data from professional testers in our test department using a survey. The results of this survey were collected and compare the efficiency in test approaches in Figure 5.4. This figure shows average values, rather than individual responses.

To check the validity of our results, the variances of collected data are as follows. In manual testing the number of tests created by each tester per workday can vary between 10 and 35. In script-based testing each tester can write from 2 to 4 test-scripts per workday which are available in subsequent sprints. MBT users are able to write between 2 and 3 models per workday, each model composed of 10 test cases on average. The duration of each sprint is set to 3 weeks.

The workload of one single tester is measured and on average a prediction of his efficiency is calculated. This efficiency is measured as follows:

$$Ef = Tc * (1 - t_{updates}/t_{sprint}) \tag{5.1}$$

where Tc is the total number of executed test cases per sprint, t_{sprint} is the duration of each sprint and $t_{updates}$ is the time spent in updating the test cases. In addition, the efficiency is normalized to make a relative comparison among the four techniques.

$$Ef_N = Ef/Ef_{max} \tag{5.2}$$

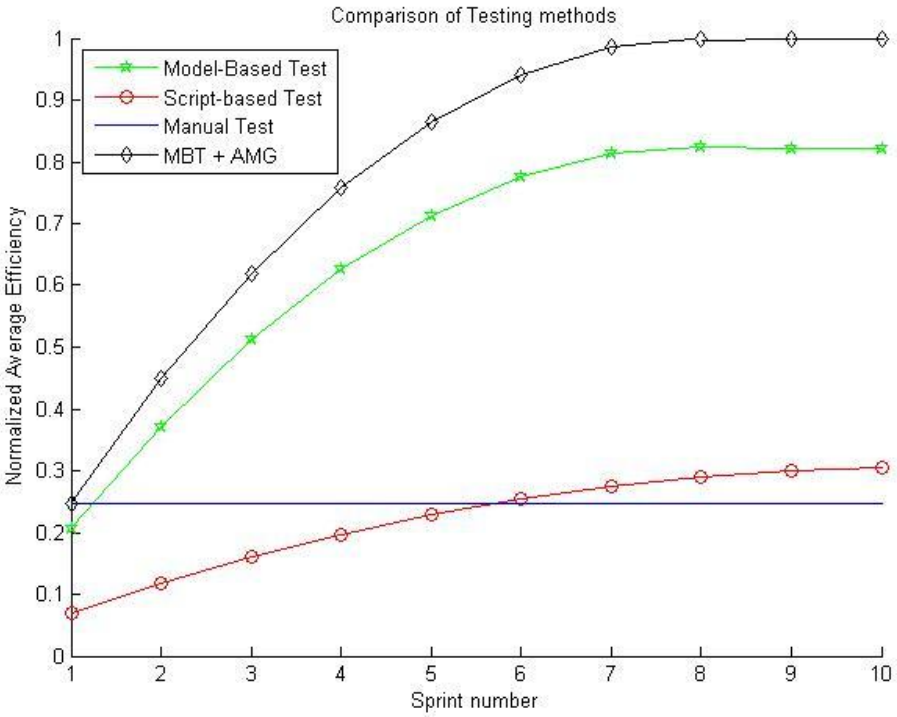


Figure 5.4. Comparison of testing methods efficiency

where Ef_N is the normalized efficiency represented in figure 5.4. It is calculated by normalizing the efficiency with the maximum value of it represented by Ef_{max}.

Manual testing

This method has a high efficiency-rate in the beginning of a project but a low one in the end. Due to the lack of automation, a single tester has to manually repeat test cases from previous sprints. Thus, the number of executed test cases on average is the same in each sprint.

Script-Based testing

Written test scripts can be re-used in consecutive sprints, therefore the tester only needs to update the test suite manually, then all the test cases can be executed automatically. This semi-automated method increases the number of executed test cases in each sprint, leading to increased efficiency.

Traditional Model-Based testing

Model-based testing automates generation and execution of test scripts. Therefore, a single tester is able to evaluate more test cases than the previous two methods (see Figure 5.4). Furthermore, the spent time in updating the models is less than updating the scripts, thus the tester has more time to implement new models leading to new test cases.

Model-Based testing with AMG tool

The AMG tool abstracts the implementation of the desired model. After extracting the model from the specification test cases are created automatically. Therefore, the tester is able to execute models in less time, hence the efficiency is even higher than the "traditional" MBT method.

5.1.5. MBT within the company

Interest in MBT has been promoted by companies, rather than universities or government organizations, due to its improvement of business processes. This has particularly attracted the interest of large companies [10] [13]. According to our research, the importance of MBT within the structure of product development is becoming widely accepted as 82% of consulted testers think that it is more efficient than other testing techniques. In addition, 100% of the surveyed testers were willing to learn and improve their knowledge about MBT.

Among the main difficulties addressed when using MBT, most users point to problems related to the creation of models, e.g. writing the code, implementing the structure or even debugging the model. Our tool helps to avoid these problems, automatically implementing the structure of the model, without debugging, or any other effort needed. The efficiency of the current MBT approach (with manual model creation) is less efficient than including our tool in the MBT process, which largely reduced the human effort. Given a standard process, the average time spent by a tester in creating the model is 2.7 hours, while using our tool is on average 35 minutes.

5.2. Amdahl's law

The presented results are focused on the performance on average of a single tester. Supposing that the time he spent updating models, he cannot implement new ones. As we are trying to measure the tool's performance we will analyze the improvement from the process perspective. Amdahl's law measures the improve-

ment of an entire process when only the performance of one part is improved [31]. This usually means that the process is parallelized or sped up in some point.

Firstly, the speedup factor is calculated, $F_{speedup}$:

$$F_{speedup} \leq \frac{p}{1 + f * (p - 1)} \tag{5.3}$$

where p is the number of times faster (after the improvement), f is the fraction of time (before improvement) spent in the part that was not improved.

Secondly, using this factor we are able to calculate the percentage of improvement in the whole process:

$$Improvement = \left(1 - \frac{1}{F_{speedup}}\right) * 100 \tag{5.4}$$

Due to time constraints, it was impossible to measure the average duration of the parts which integrates the whole MBT process (see figure 2.8). Taking the results showed in [5], the duration is divided into:

- Specifications analysis & model implementation 65%
- Documentation & project management: 6%
- Extracting test & creating test harness: 12%
- Execute test cases & closure: 17%

Table 5.6 summarizes the results obtained and the improvement achieved:

Table 5.6. Process improvement

Partial improvement	**Speedup factor** ($F_{speedup}$)	**Improvement (%)**
1.95	1,205	17,05
2.98	1,303	23,25
3.64	1,340	25,38
4.89	1,385	27,84
6.60	1,422	29,69
9.14	1,452	31,17

Note that the *partial improvement* is written in as times faster and the improvement is written in percentages. In order to calculate the $F_{speedup}$ following the Amdahl's law, we supposed that the percentage of understanding and implementing the model is equal to 65% as stated before.

An improvement up to 31% can be achieved in the whole process with our tool compared to a simple MBT process.

Chapter 6

Conclusions and Future Improvements

In this chapter the results of this master's thesis project are discussed and some improvements are proposed. Section 6.1 presents a discussion of the results achieved in the previous chapter. Section 6.2 shows possible features to improve in the future.

6.1. Discussion

The main thesis' goal was to develop a method to reduce the time spent on the implementation of models when using MBT. The best way to reduce time in repetitive processes, i.e. testing processes in a product life cycle, is to automate the process and the results obtained are really promising. Thanks to the abstraction of a particular type of models which were composed of <command input, system output>pairs, the automation was carried out and a significant improvement of the model implementation times was made which leads to a significant improvement on the whole process.

Note that the results shows an improvement up to 31% over the MBT process which means that the implementation and designing was made around 9 times faster. One conclusion can be extracted here the more complex the system is, the larger is the amount of saved time.

The presented results shown in section 5, are based on people with knowledge of both AMG and Conformiq. Although we still have not conducted any measurements on the amount of time it takes for people to start using the AMG tool, we know from previous measurements according to the experience of our testers that the time it takes to learn and start using MBT ranges from 7 to 19 days, depending on the availability of resources to conduct the MBT training and the previous experience of the trainees with programming languages. From our estimations, the time to learn AMG is less then that, because the people do not need to have any background

in programming languages, but only know how to use standard Windows or Linux applications.

Moreover, the modularity of the designed tool and the XML as the intermediate format give it a lot of flexibility and enables it have a big impact in the company. Thanks to these facts, the company thinks about integrating some of the tool's features into a bigger project and therefore, develop a commercial tool.

On the other hand, there are two sections that were not completed due to the lack of time. The first one is the automatic generation of documentation which can enable the company to have updated information of modifications or new features. The second one is the implementation of an HTML parser which could enable the tool to fill-in all the requirements directly reading the specifications. They were planned but not accomplished, therefore they are explained in the section.

Anyhow, the big challenge is introducing the tool to the company's testing engineers. Traditional methods are still used even though there are much more efficient methods developed, so hopefully the reduction of complexity to get started through this new tool can remove the difficulties of learning new testing methods.

6.2. Future improvements

The results as shown in Section 5, indicated good performance and an increment in efficiency when using AMG. The time to generate and implement models is shorter and the required experience and knowledge have been reduced. However, AMG can still be improved.

The natural language parser was not possible to implement due to time constraints. Although it is the optimal solution because ideally a natural language parser can read any document, and although some ambiguity would still be present, the benefits could still be of value.

Given the ambiguity of NLP, we propose the use of a controlled language in the technical specifications. A wide-spectrum language sacrifices some expressiveness capabilities (e.g. complexity of grammar and richness of vocabulary) in order to have a more formal structure and be machine-readable. The pretty-printer generates documentation with a common format and common expressions, therefore a parser can be implemented based on this. The generated documentation is reviewed by a tester and uploaded as the specifications. This ensures the maintenance of correct specifications. We called this the "documentation loop" and it is illustrated in Figure 6.1.

Instead of creating the parser and then the specifications, we propose to do it in reverse. Firstly, define the controlled language and secondly, implement a parser which can parse what the pretty printer wrote before.

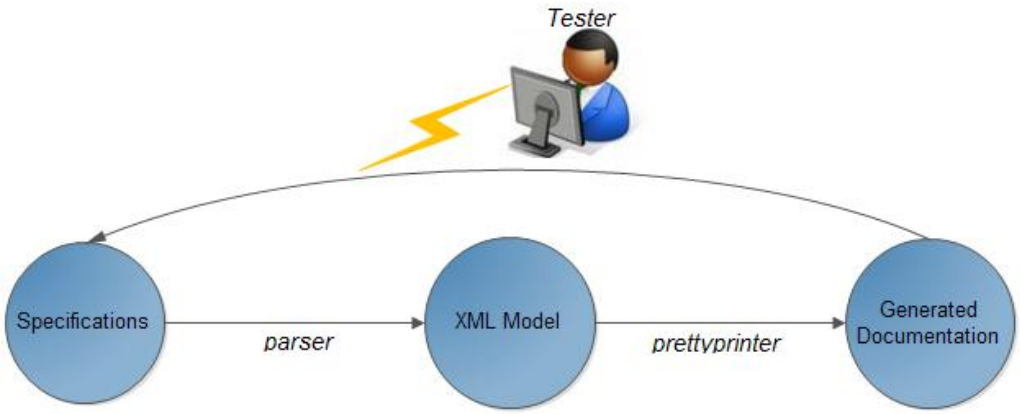


Figure 6.1. Documentation loop

The advantages of implementing this method would be mainly two:

- Full-automation and hence, considerably reduced testing time.
- New documentation

With this implementation the tester could modify directly a model from the specifications, he would not even have to know how our tool works. Editing a model from the specifications will lead to another model when it was read by the the parser. Therefore, the user could have new models and new test cases just redefining the specifications.

This change in our minds can lead to a new approach in the test suites because if they are analyzed, testers can focus on explaining in a correct way every single detail in the specifications. Currently, the specifications are written in order to explain a procedure already done and hence others can reproduce it afterwards. However, we propose to write commonly formatted specifications with every single detail explained on them in order to extract the test suite from there. In this way, the risk for implementation errors would be minimized since the specifications are well-written.

Writing specifications can be a challenging task and if we extend our idea even further, the tester could design a model from the specifications or from the pre-models which give this idea even more flexibility to the user. Therefore, he can choose the way to implement new models, through specifications, pre-models or a mixed implementation.

We also propose another lead of future work about the investigation of the feasibility of handling parameter dependencies using BNF grammar. This approach would substitute the b-tree data structures, which fulfill the requirements of the domain of this tool (i.e., a specific CLI domain and a certain 3rd-party tool,

Conformiq), but would be less effcient if future works try to extend the functionality and the scope of this thesis project. Libraries as JFLEX or JLEX could be useful in this case.

Another feasible future approach would be the implementation of a software module which captures already-created UML diagrams. If a software module could be implemented in order to extract the system behavior from the model used for designing the system instead of creating specifications from it, it will help to extract the requirements directly without any intermediate step (written specifications). The main benefit of this approach is that the system models could be used to capture the requirements specifications directly, and no NLP (and subsequently, automation loop) would be needed.

Bibliography

[1] About - osmo - simple model-based testing tool - google project hosting. http://code.google.com/p/osmo/wiki/About.

[2] Artisan studio - products - atego. http://www.atego.com/products/artisan-studio/.

[3] Conformiq | automated test design. http://www.conformiq.com/.

[4] Document object model (DOM) specifications. http://www.w3.org/DOM/DOMTR.

[5] An evaluation on model-based testing at ericsson.

[6] Extensible markup language (XML) 1.1. http://www.w3.org/TR/2004/REC-xml11-20040204/#NT-doctypedecl.

[7] The GraphML file format. http://graphml.graphdrawing.org/.

[8] HP quality center software | HP enterprise software. http://www8.hp.com/us/en/software/software-product.html?compURI=tcm:245-937045.

[9] IBM rational DOORS software. http://www-01.ibm.com/software/awdtools/doors/.

[10] Integrate model-based testing to find quality problems early in development. http://www.ibm.com/developerworks/rational/library/integrate-model-based-testing-to-find-quality-problems-early-in-development/index.html?ca=drs-.

[11] JavaScript and java diagram library components. http://www.jgraph.com/.

[12] mbt.tigris.org. http://mbt.tigris.org/.

[13] Model-Based testing of infotainment systems on the basis of a graphical Human-Machine interface. http://ieeexplore.ieee.org.focus.lib.kth.se/stamp/stamp.jsp?tp=&arnumber=5617165.

[14] Motes - elvior website,. http://www.elvior.com/mbt-2/features.

[15] Murphi description language and automatic verifier. http://verify.stanford.edu/dill/murphi.html.

[16] OMG formal specifications. http://www.omg.org/spec/.

[17] ParTeG - homepage. http://parteg.sourceforge.net/.

[18] Poseidon tool. http://www.gentleware.com/.

[19] SMARTESTING - optimize your test center. http://www.smartesting.com/index.php/cms/en/home.

[20] Spec explorer. http://msdn.microsoft.com/en-us/library/ee620411.aspx.

[21] TestOptimal - Model-Based test automation. http://testoptimal.com/.

[22] Topcased - home. http://www.topcased.org/.

[23] The UML, SOA, BPMN, EA and MDA convergence for model driven engineering. http://www.objecteering.com/.

[24] W3C document object model. http://www.w3 .org/DOM/.

[25] W3C XML schema. http://www.w3.org/XML/Schema.

[26] XML editor. http://www.altova.com/xml-editor.

[27] yEd - graph editor. http://www.yworks.com/en/products_yed_about.htm.

[28] Majdi Abuelbassal and Mohammad Hossain. Model driven testing. *System*, 75080(972), 1997.

[29] M. A Ahmad and A. Nadeem. Consistency checking of UML models using description logics: A critical review. In *2010 6th International Conference on Emerging Technologies (ICET)*, pages 310–315. IEEE, October 2010.

[30] Pekka Aho, Nadja Menz, Tomi RÃďty, and Ina Schieferdecker. Automated java GUI modeling for Model-Based testing purposes. pages 268–273. IEEE, April 2001.

[31] G.M Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. pages 483–485, 1967.

[32] E. G Aydal and J. Woodcock. Automation of Model-Based testing through model transformations. In *Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009. TAIC PART '09.*, pages 63–71. IEEE, September 2009.

[33] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold, New York, 2nd ed. edition, 1990.

[34] S. Berner, R. Weber, and R.K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579, 2005.

[35] A. Bertolino, Jinghua Gao, E. Marchetti, and A. Polini. Automatic test data generation for XML schema-based partition testing. In *Second International Workshop on Automation of Software Test , 2007. AST '07*, pages 4–4. IEEE, May 2007.

[36] Mark Blackburn, Robert Busser, and Aaron Nauman. Why Model-Based test automation is different and what you should know to get started. *Test*, 2004.

[37] A. Cervantes. Exploring the use of a test automation framework. In *Aerospace conference, 2009 IEEE*, pages 1–9, 2009.

[38] S. W.K Chan. Inferences in natural language understanding. In *, Proceedings of 1995 IEEE International Conference on Fuzzy Systems, 1995. International Joint Conference of the Fourth IEEE International Conference on Fuzzy Systems and The Second International Fuzzy Engineering Symposium*, volume 2, pages 935–940 vol.2. IEEE, March 1995.

[39] 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. *Proceedings of the 21st international conference on Software engineering ICSE 99*, 1999(May):285–294, 1999.

[40] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[41] K. El Emam and N.H. Madhavji. Measuring the success of requirements engineering processes. In *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*, pages 204–211. IEEE Comput. Soc. Press.

[42] S. Eski and F. Buzluca. An empirical study on Object-Oriented metrics and software evolution in order to reduce testing costs by predicting Change-Prone classes. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 566–571, 2011.

[43] R.D.F. Ferreira, J.P. Faria, and A.C.R. Paiva. Test coverage analysis of uml state machines. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 284 –289, april 2010.

[44] Martin Fowler and Kendall Scott. *UML distilled: a brief guide to the standard object modeling language*. Addison Wesley, 2000.

[45] S. Harrusi, A. Averbuch, and A. Yehudai. XML syntax conscious compression. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 10 pp.– 411. IEEE, March 2006.

[46] Hartman and K. Nagin. The AGEDIS tools for model based testing. *ACM SIGSOFT Software Engineering Notes*, 29(4):129–132, 2004.

[47] Bill Hasling, Helmut Goetz, and Klaus Beetz. Model based testing of system requirements using UML use case models. pages 367–376, 2008.

[48] Herman and M S Marshall. GraphXML – an XML-based graph description format. *World*, 1984:33–66, 2000.

[49] Gerard J. Holzmann. Economics of software verification. In *PASTE '01 Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 80–89. ACM Press, 2001.

[50] Chaw Yupar Htoon and Ni Lar Thein. Model-based testing considering cost, reliability and software quality. In *6th Asia-Pacific Symposium on Information and Telecommunication Technologies, 2005. APSITT 2005 Proceedings*, pages 160–164. IEEE, November 2005.

[51] C.Y. Huang, J.H. Lo, S.Y. Kuo, and M.R. Lyu. Software reliability modeling and cost estimation incorporating testing-effort and efficiency. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 62–72, 1999.

[52] Huima. Implementing conformiq qtronic. 4581 LNCS:1–12, 2007.

[53] P. Iyenghar, E. Pulvermueller, and C. Westerkamp. Towards Model-Based test automation for embedded systems using UML and UTP. In *2011 IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–9. IEEE, September 2011.

[54] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, November 2011.

[55] S.J. Jang, H.G. Kim, and Y.K. Chung. Manual specific testing and quality evaluation for embedded software. In *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 502–507, 2008.

[56] Teemu KanstrÃłn. *A Framework for Observation-Based Modelling in Model-Based Testing*. Doctoral dissertation, Faculty of Science, University of Oulu. VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O. Box 1000, FI-02044 VTT, Finland.

[57] M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, and M. Satama. Towards deploying Model-Based testing with a Domain-Specific modeling approach. In *This paper appears in: Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006*, pages 81–89. IEEE, August 2006.

[58] Beum-Seuk Lee. Automated conversion from a requirements document to an executable formal specification. In *16th Annual International Conference on Automated Software Engineering, 2001. (ASE 2001). Proceedings*. IEEE, November 2001.

[59] Y. Liu. *WCDMA test automation workflow analysis and implementation*. PhD thesis, Master's thesis, Royal Institute of Technology, 2009. Available at: http://web. it. kth. se/∼maguire/DEGREE-PROJECT-REPORTS/090423-Yike_Liu-with-cover-a. pdf, 2009.

[60] Ralph LÃďmmel, Stan Kitsis, and Dave Remy. Analysis of XML schema usage. pages 1–38, 2005.

[61] Mohammed Misbhauddin and Mohammad Alshayeb. *Extending the UML Metamodel for Sequence Diagram to Enhance Model Traceability*. IEEE, 2010.

[62] R. Mitsching, C. Weise, A. Kolbe, H. Bohnenkamp, and N. Berzen. Towards an industrial strength process for timed testing. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 29–38, 2009.

[63] T. Pajunen, T. Takala, and M. Katara. Model-Based testing with a general purpose Keyword-Driven test automation framework. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 242–251, 2011.

[64] Jiantao Pan. Software testing, July 1999. "Topics in Dependable Embedded Systems" , Carnegie Mellon University.

[65] Pezze and M. Young. *Software Testing and Analysis*. Wiley, 2008.

[66] A. Pretschner. Model-based testing. In *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*, pages 722– 723. IEEE, May 2005.

[67] Miro Samek. *Practical UML statecharts in C/C++: event-driven programming for embedded systems*. Newnes, October 2008.

[68] Pedro Santos-Neto, Rodolfo Resende, and Clarindo PÃądua. Requirements for information systems model-based testing. In *SAC '07 Proceedings of the 2007 ACM symposium on Applied computing*, pages 1409–1415. ACM Press, 2007.

[69] M. Schnelte. Generating test cases for timed systems from controlled natural language specifications. In *Third IEEE International Conference on Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009*, pages 348–353. IEEE, July 2009.

[70] Muhammad Shafique and Yvan Labiche. A systematic review of model based testing tool support. *Review Literature And Arts Of The Americas*, (May):1–21, 2010.

[71] Ian Sommerville. *Software Engineering*. Addison Wesley, 7 edition, May 2004.

[72] SparxSystems. Enterprise architect. pages 1–4, 2011.

[73] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

[74] M. -F Wendland, J. Grossmann, and A. Hoffmann. Establishing a Service-Oriented tool chain for the development of Domain-Independent MBT scenarios. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pages 329–334. IEEE, March 2010.

[75] S. Wieczorek and A. Stefanescu. Improving testing of enterprise systems by Model-Based testing on graphical user interfaces. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pages 352–357. IEEE, March 2010.

[76] Yazhou Xiang, Bin Zhang, Guohui Li, Yan Li, and Xuesong Gao. The application and analysis of netconf subtree filtering based on SAX and DOM.

In *2010 International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, volume 3, pages 758–761. IEEE, March 2010.

[77] Z. Xiaochun, Z. Bo, L. Juefeng, and G. Qiu. A test automation solution on GUI functional test. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 1413–1418, 2008.

[78] Tao Xie. Improving effectiveness of automated software testing in the absence of specifications. In *22nd IEEE International Conference on Software Maintenance, 2006. ICSM '06.*, pages 355–359. IEEE, September 2006.

[79] Li Zhao and Feng Li. Statistical machine learning in natural language understanding: Object constraint language translator for business process. In *IEEE International Symposium on Knowledge Acquisition and Modeling Workshop, 2008. KAM Workshop 2008*, pages 1056–1059. IEEE, December 2008.

[80] Weiqun Zheng and G. Bundell. Model-Based software component testing: A UML-Based approach. In *6th IEEE/ACIS International Conference on Computer and Information Science, 2007. ICIS 2007*, pages 891–899. IEEE, July 2007.

Appendix A

Application Architecture

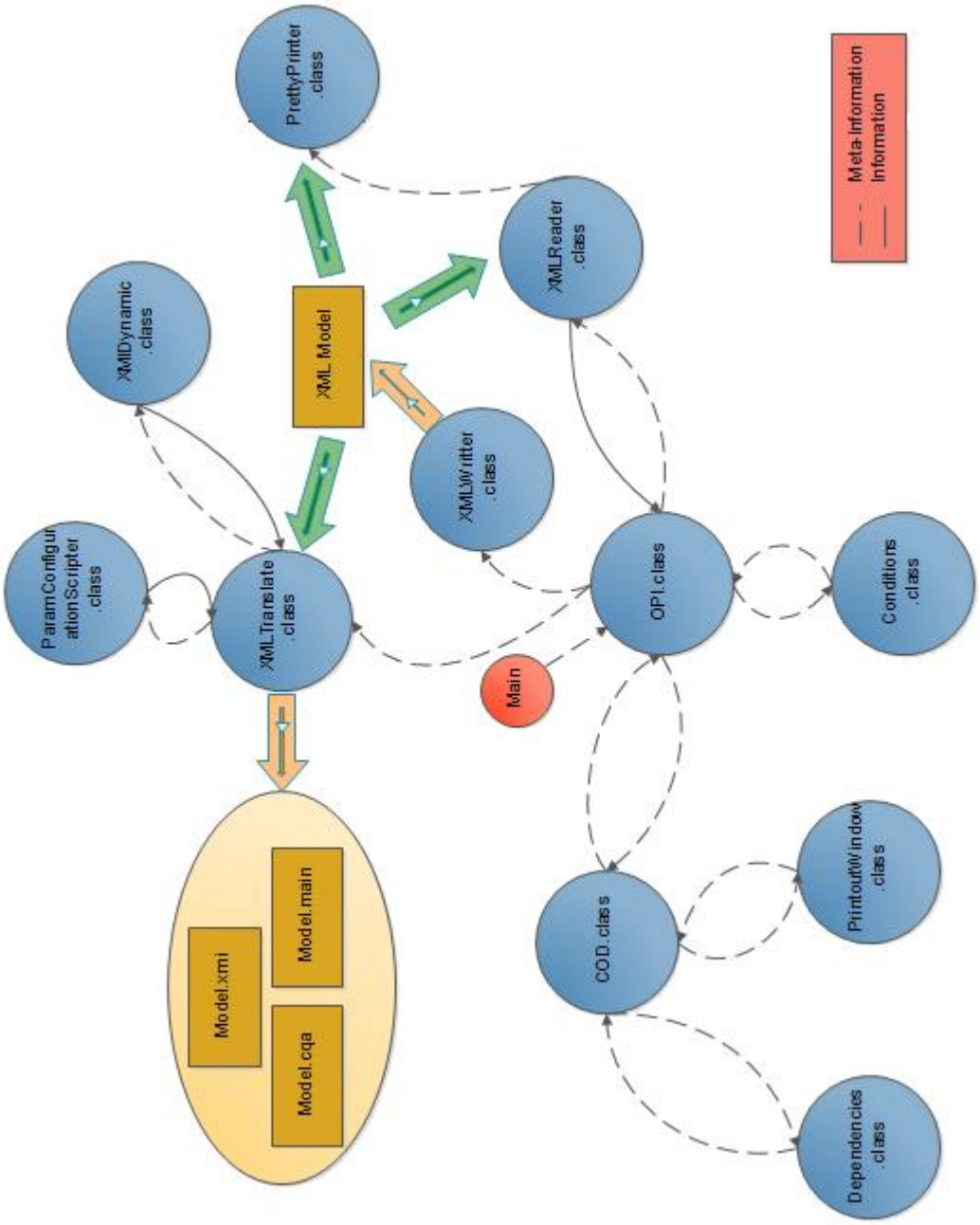


Figure A.1. Class Diagram

Appendix B

Example of XML document generated by the AMG tool

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:jGraph="http://www.jgraph.com/" xmlns:xsi="http://www
    .w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http:
    //graphml.graphdrawing.org/xmlns http://graphml.
    graphdrawing.org/xmlns/1.0/graphml.xsd">
<key attr.name="nodeData" attr.type="string" for="node" id="d0
    "/>
<key attr.name="edgeData" attr.type="string" for="edge" id="d1
    "/>
<graph edgedefault="directed">
<node id="9">
<data key="d0">
<jGraph:ShapeNode>
<jGraph:Geometry height="50.0" width="50.0" x="160.0" y="-10.0
    "/>
<jGraph:label text="COD"/>
</jGraph:ShapeNode>
<command>
<name>PTSAP</name>
<type>MML</type>
<numberOfFormats>1</numberOfFormats>
<preconditions>
<isMMLConnection>true</isMMLConnection>
<isConnectToCPT>true</isConnectToCPT>
<isConnectToSB>false</isConnectToSB>
<isSeparateSides>false</isSeparateSides>
<returnCode>0</returnCode>
</preconditions>
<parameter>
```

```
<name>store</name>
<value>DS, PS</value>
<format>1</format>
<optional>false</optional>
</parameter>
<parameter>
<name>hiaddr</name>
<value>"1:2"</value>
<format>1</format>
<optional>true</optional>
</parameter>
<parameter>
<name>addr</name>
<value>"0:9"</value>
<format>1</format>
<optional>false</optional>
</parameter>
<parameter>
<name>nrw</name>
<value>"1:5"</value>
<format>1</format>
<optional>true</optional>
</parameter>
<output>NULL</output>
</command>
</data>
</node>
</graph>
</graphml>
```

Appendix C

AMG User Guide

AMG:

Automated Model Generator

- User Guide -

Ignacio Mulas Viela

Armando Gutiérrez

Ericsson AB

Stockholm, March 2012

Table of Contents:

1 Tool Motivation

Automatic Model Generator (AMG) was created in order to automate testing of operation and maintenance (O&M), command line (CLI) interfaces in APG43 or help reducing the required time to design complex tests.

A big amount of test cases can be generated by just using a GUI-based parameter insertion tool, without having to deal with the complexity of test scripting languages. As soon as users enter the parameters in the GUI, the tool generates a model which can be subsequently be used as input to a Model-Based testing (MBT) tool. Subsequently, test cases are generated from the MBT tool and are rendered to executable test scripts using the proper test adaptor (i.e. a program that transcodes the generated test cases to executable scripts) (see also figure 1).

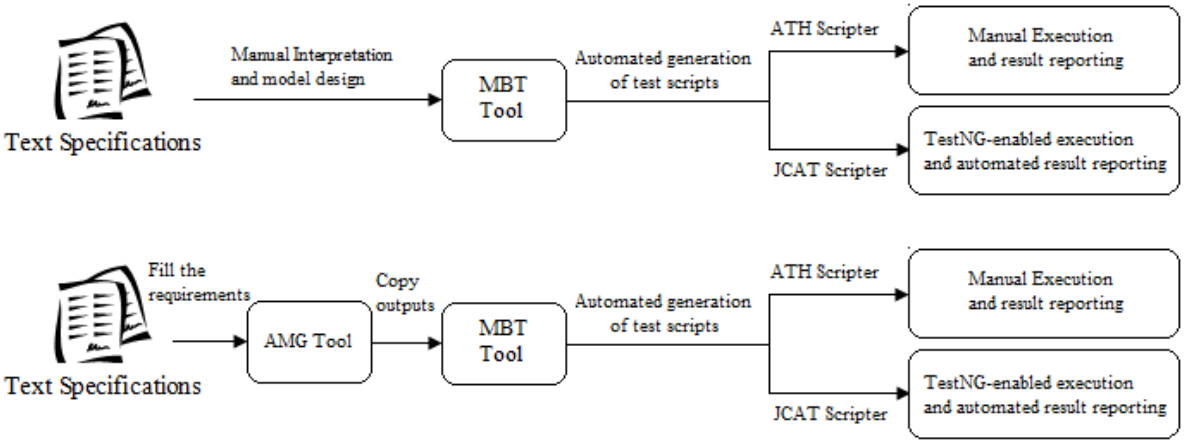


Figure 1. (a) Actual process (b) Process using AMG tool

This figure illustrates the current way of working with MBT (a), wherein models are designed manually. With the usage of AMG, models are designed with minimal user feedback, thus resulting in a reduction of the time required to create a model (b). This also leads to a reduction of total time required for testing.

2 Installation

This program does not require any kind of installation but some additional software is required.

2.1 Additional Required Software

To run an executable .jar file. Thus, please make sure that you have installed Java VM version 6. The program can be obtained from http://www.java.com/en/download/chrome.jsp?locale=en

Moreover, the user must have installed “Conformiq Designer (CD)” software for MBT. It can be a plug-in on Eclipse or a standalone application (although installing the Eclipse plugin is recommended). In order to obtain and install Conformiq Designer, please refer to the guide on ericoll:
https://ericoll.internal.ericsson.com/sites/AXE_I_V_MBT/AXE_UG/Lists/Categories/Category.aspx?Name=1.%20Getting%20started:%20Installing%20the%20MBT%20tool%20suite

AMG software outputs will be processed later by “CD” software with the purpose of creating the proper test cases that will be tested afterwards in the system under test with the AMG output as an input.

3 Required background knowledge:

AMG is made for generating models containing the necessary code for creating the test cases afterwards with a MBT processor like CD. Therefore, it is essential that the user has some knowledge in model based testing (MBT) theory as well as be familiar with basic Object-Oriented Programming (OOP) concepts.

4 Program Overview:

This section explains the functionality and purpose of each feature of the AMG tool and contains careful instructions about how to use it.

4.1 Graphical User Interface

This is the front-end in the AMG tool and its main screen. On this screen, the user defines the sequence of commands and the conditions necessary to continue from one command to the next one. This sequence is usually required when the user wants to model OPIs because they are composed of simpler commands (CODs).

If the user just wants to model a single COD, only one box is needed in the canvas.

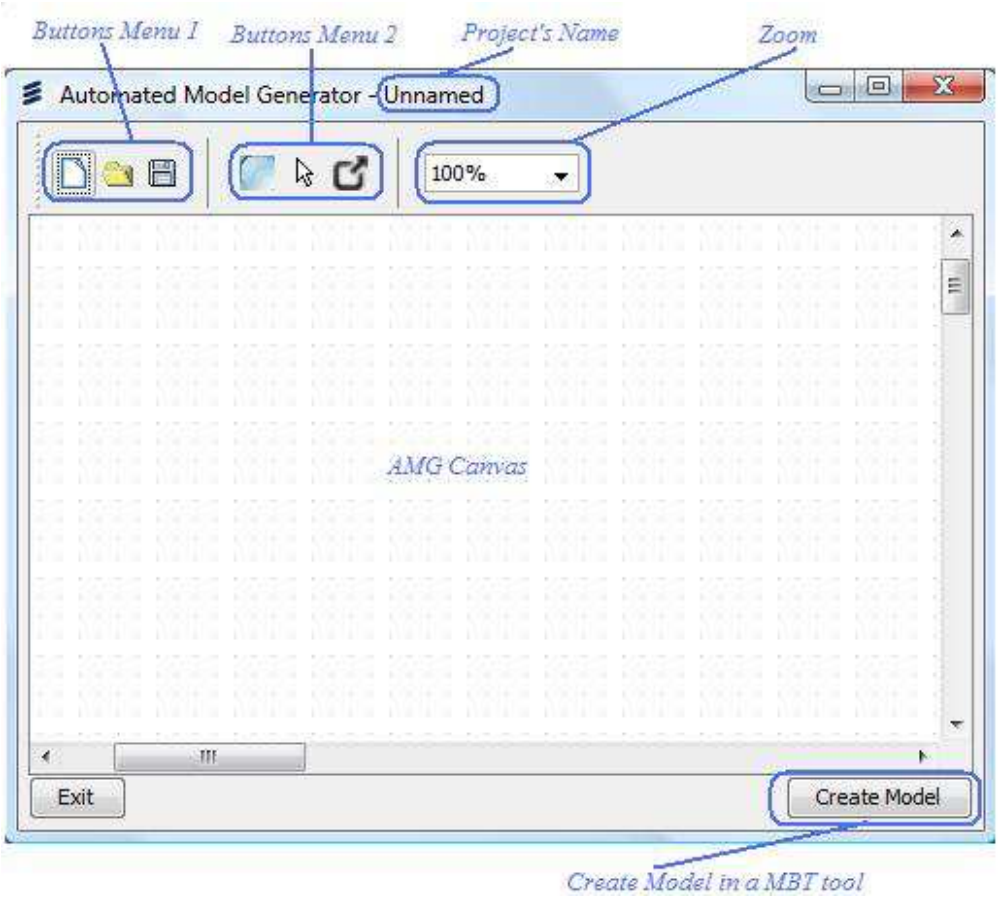


Figure 2. Main Screen

4.1.1 Buttons Menu 1

There are three buttons implemented here, they are respectively from left to right:

- "New pre-model" which removes everything and creates a new pre-model.

- “Load XML file” which removes everything and load a pre-model already implemented in XML format.
- “Save XML file” which saves the implemented pre-model on a XML file.

4.1.2 Buttons Menu 2

In this container, there are three buttons implemented too, they are respectively from left to right:

- “COD” Enables the AMG canvas to add more boxes in the chart.
- “Select” Disables the creation of boxes in the AMG canvas.
- “Expand” Open the COD screen (see Section BB) in order to define the required data on each command.

4.1.3 Project’s Name

The name of the current project appears together with the name of the tool.

4.1.4 Zoom

On this box the actual zoom is displayed. Here the user can specify the desired zoom opening the box and choosing one of the options or he can change it pushing CTRL + mouse wheel. In both cases, the value of the current zoom is displayed on the box.

4.1.5 Create Model

Pushing this button, the tool translates from the XML format to the model programming language used in the MBT tool.

In order to push this button the user must have saved the implemented pre-model as an XML model in the selected directory. Pushing “Create Model” button the user creates the necessary files to translate the XML model into a Conformiq model. The outputs, as it is said before, “XMLname.cqa”, “XMLname.xmi”, “XMLname_main.cqa”, and “template_use_case_ConfigurationData.cqa”. The XML name is the chosen name for the XML file saved before. These outputs will be saved directly in the directory: C:\Users\name_user\Automated Model Generator. This will be changed in the following versions so the user can select the directory that he wants.

4.1.6 AMG canvas

As stated before, the possible elements on this canvas are “boxes” containing a command operation description (COD), and “transitions” with the conditions to be checked. Mixing both, the user is able to model any sequence of commands.

Each “box” contains a COD and the command’s information defined by the user, such as parameters’ values, preconditions, etc. These values are visible either in the tooltip (explained later) or opening

the “box”. On this screen, the user is only able to see the name of the command in the front of the box in order to maintain the model simple and not overload it with a lot information.

Moreover, “boxes” are joined by “transitions” which contains the conditions to be checked out between two CODs. These transitions information but a short description of the condition is hidden as well in order to keep the simplicity.

4.1.7 Tooltips

As the information displayed on the boxes and the transitions is insufficient to see all the contained information, tool-tips have been implemented to see the contained information from the canvas. When the user locate the mouse on any box or transition a tooltip appears with the information contained inside the box (see Figure 3a) or the transition (see Figure 3b).

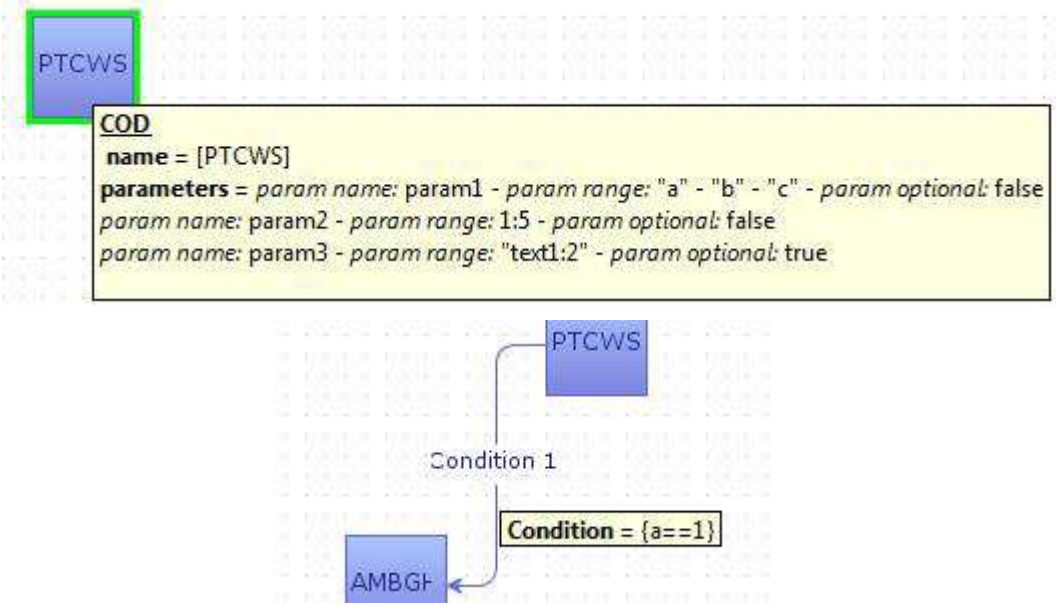


Figure 3. (a) Box' Tooltip (b) Transition Tooltip

4.1.8 Robot

An automatic Conformiq project creation has been created through a sequence of commands. The only requirement to use this robot is that the user must open first “Conformiq” or Eclipse/Conformiq designer and have it in the background of the tool.

If this checkbox is selected, the sequence of actions will be done automatically when the model is created. Thus, the user will have prepared a new project in the workspace of “Comformiq” called “XMLname_project”.

4.2 COD

This is the main screen for modeling COD specifications in AMG Tool. It is composed by the following panels: (i) Menu (ii) Command interface, name and number of formats (iii) Preconditions (iv) Parameter input (v) Command output declaration for verification and (vi) Storage area and automated project creation settings. All these elements are explained below in detail.

4.2.1 Menu

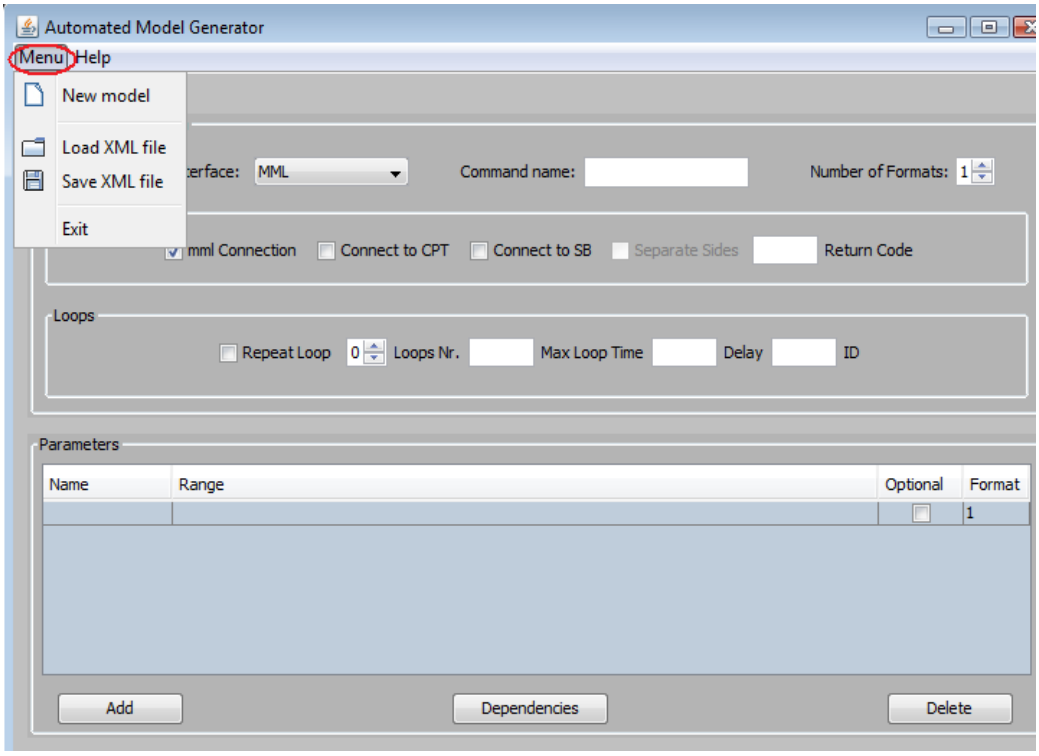


Figure 4. Model Generation Settings

It is possible to use the menu on the top left part of the workspace to save or retrieve previous model generation configuration documents.

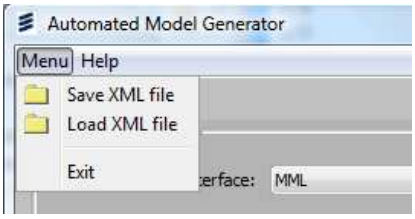


Figure 5. Menu's items

- **Save model**: a XML file is saved with the information of the command that the user has introduced in the GUI.
- **Load model**: a new window pops up where the user can specify the name of the command to load. The information is loaded in the GUI, so the user can change it (or not) depending on the case. Anyway, the "Create Model" button will generate the automated model.

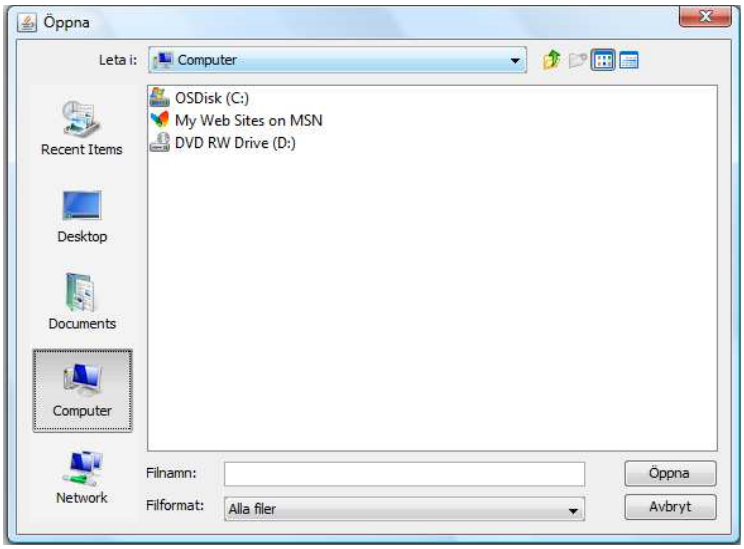


Figure 6. Opening file dialog

4.2.2 Command interface

The command interface specifies the type of interface to which command belongs. The interfaces allowed are APG (for APG shell commands, together with support for O/S shell commands) and MML (for MML and CPT commands) (see figure6).

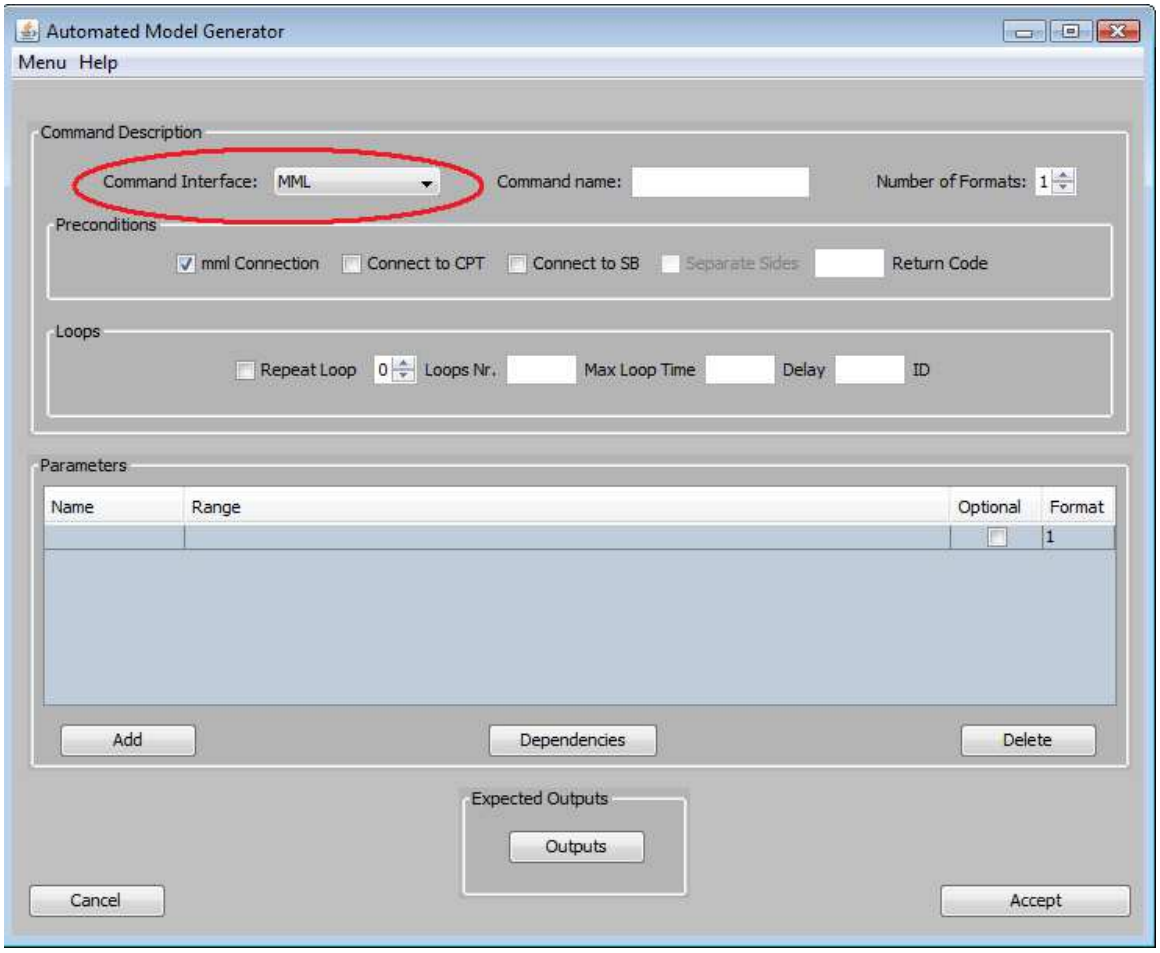


Figure 7. Choosing the interface of the command to model

4.2.3 Command name

Specify the name of the command to model.

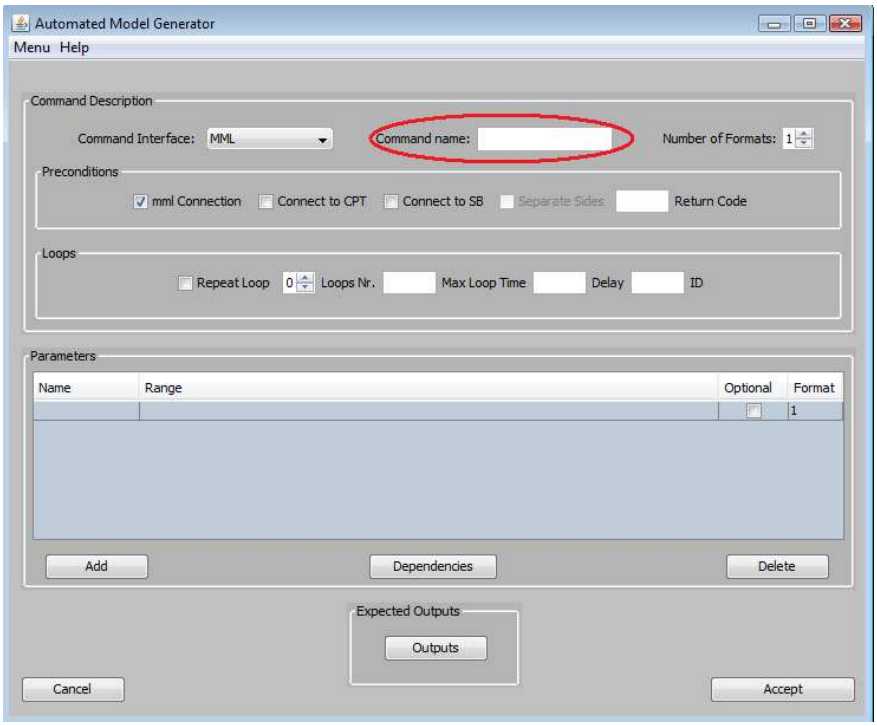


Figure 8. Specifying the name of the command to model.

4.2.4 Number of Formats

Specify the number of the formats of the command to model.

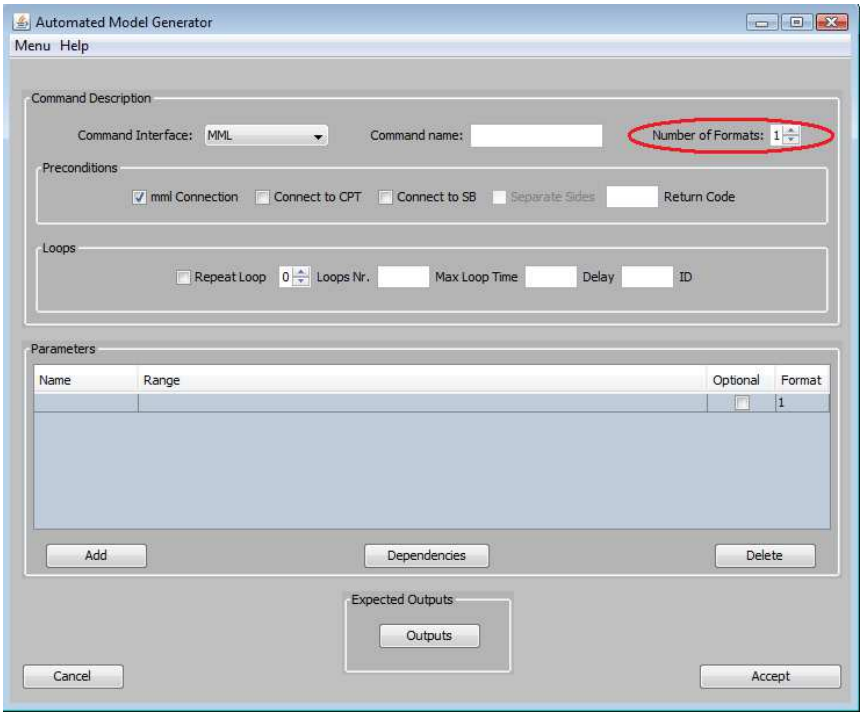


Figure 9. Number of Formats

4.2.5 Connection Preconditions

These preconditions set the prerequisites that certain commands need before its performance. E.g. for a command that needs to be connected to CPT , the user will click on the corresponding option, and in the model a PTCOI command will be executed before in order to connect to the CPT shell. The preconditions allowed are the following:

- **MML connection**: If the command interface is MML this option will be automatically enabled).
- **Connect to CPT**: A PTCOI command is send to establish connection with CPT shell.
- **Connect to SB**: If the command is issued on the Standby (SB) side.

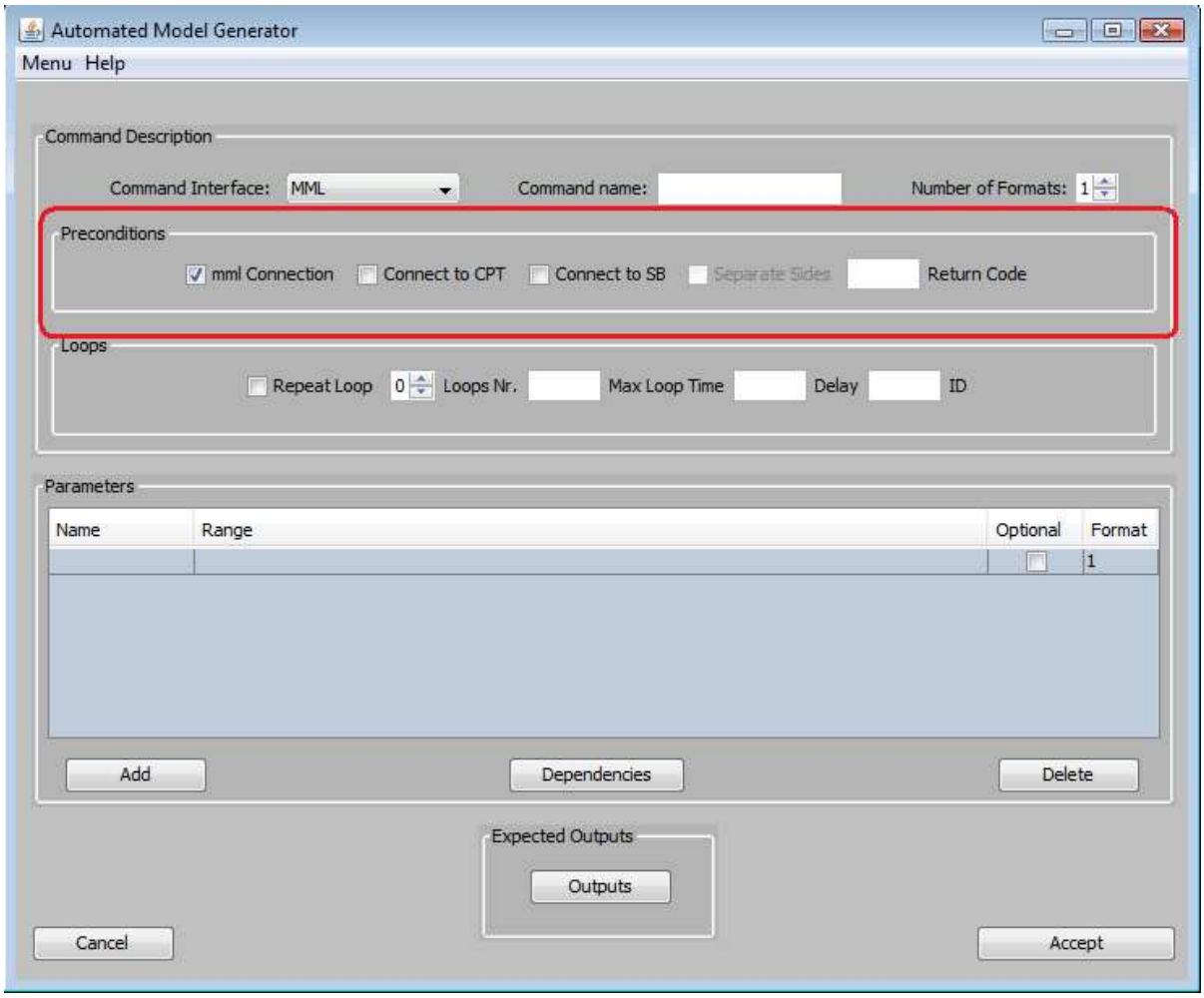


Figure 10. Precondition Configuration

- **Separate Sides**: If this option is marked, a PTSES command is send before the command execution and the SB side is separated. To be enabled, it must be connected to CPT (it requires a PTCOI to connect).

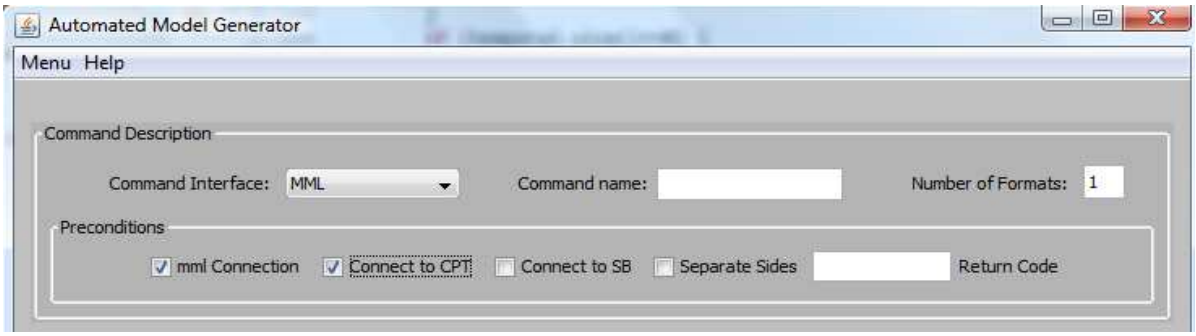
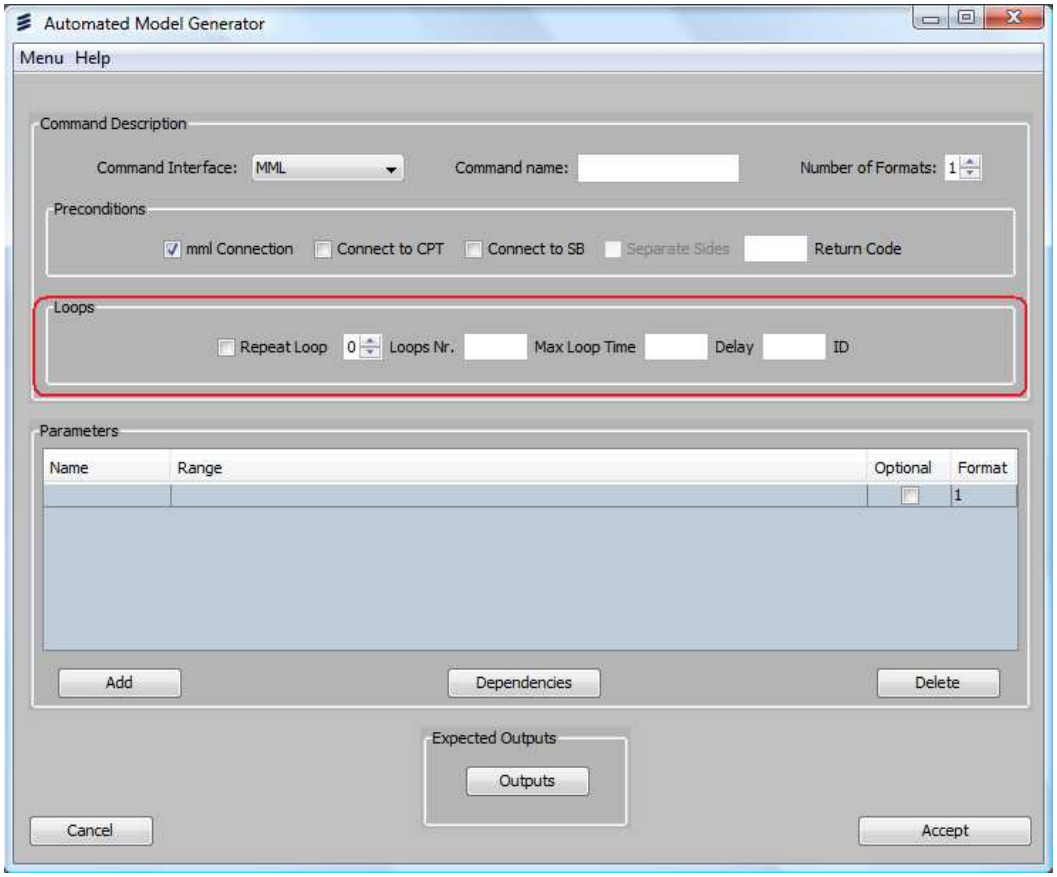


Figure 11. Precondition Configuration: Configuring standby side connection

4.2.6 Loops

Loops can be defined in order to repeat a particular instruction. This allows us to specify a number of repetitions instead of programming a counter in the model. The user can also define the delay between loops or define an identifier. There are five sections:

- "Repeat Loop"
- "Loops Nr"
- "Max loop Time"
- "Delay"
- "ID"



4.2.7 Parameters

In this section the user defines the parameters for the desired command. These parameters are typically retrieved from ALEX CODs, but they can also be manpages in case the command belongs to the APG interface. As a command can have multiple parameters, there are two buttons in the bottom in charge of adding or removing parameters: Add and Delete respectively.

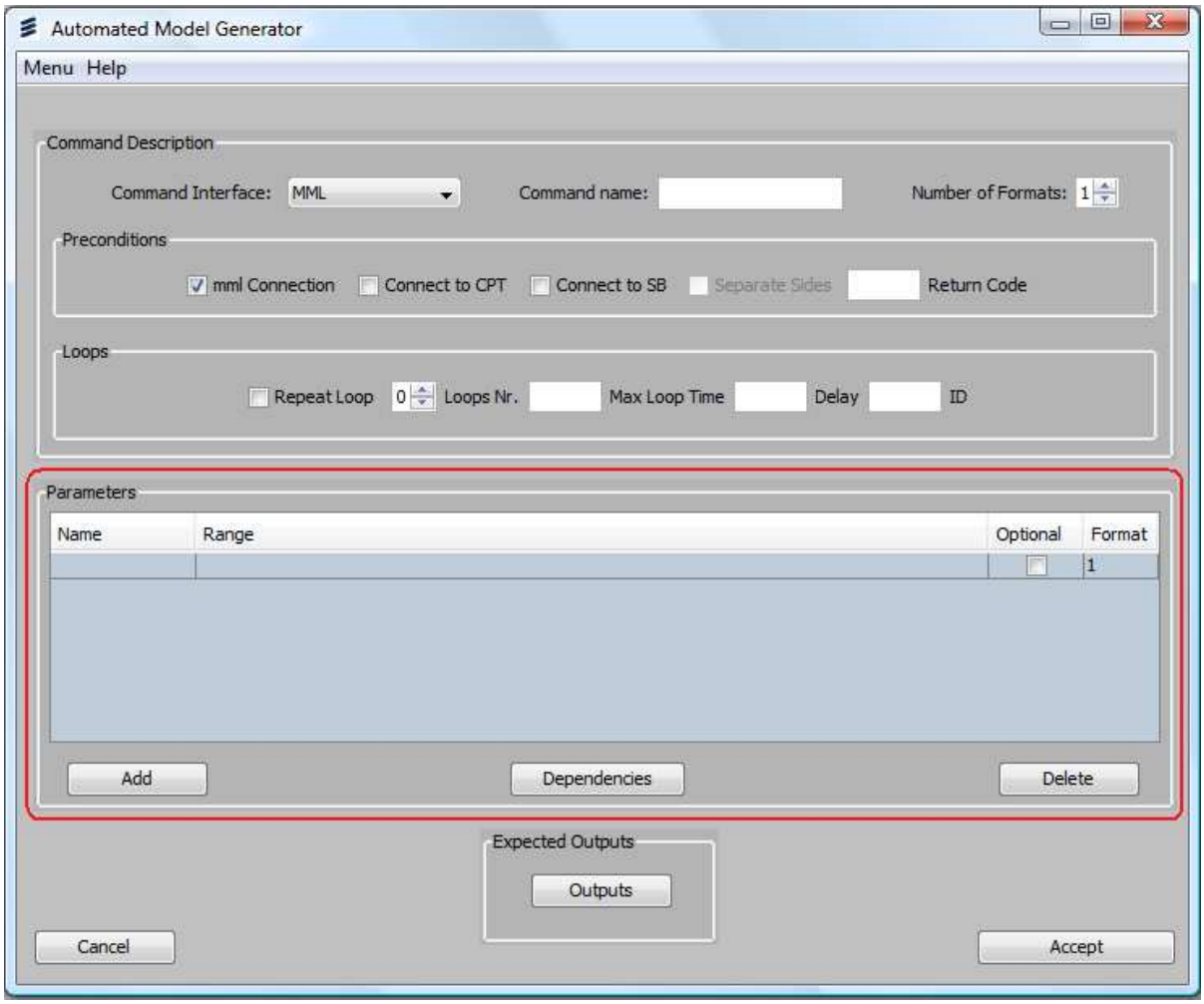


Figure 12. Parameter declaration section

4.2.7.1 "Add"/"Delete" button

"Add" button functionality is to create a new row in the table for a new parameter. Clicking on this button, a new empty line will appear in the table.

Clicking on "Delete" button (on the right), the selected rows will be deleted. If no rows are selected, the program will remove the last one.

4.2.7.2 Parameters' table

A command has a number of characteristics:

1. Name of the parameter

2. Range of values

3. Mandatory / optional

4. Format of the command in which this parameter belongs to (as there can be commands that can be issued with different sets of parameters - i.e. formats)

Parameter Name

Here the user defines the name of the command.

Range

Note: Figure 12 at the end of this section illustrates various parameter declaration examples.

Depending on the parameter, there are different range types from numerical parameters to text-type parameters. AMG supports both textual parameters (alphanumericals) as well as numbers and mixed-type parameters (i.e. alphanumericals together with numbers). The rest of this section describes the notation under which the parameters are expressed in AMG.

Text-type Commands:

This type of parameters must be defined between **quotation marks**. If more than one value is possible, they can be separated by commas.

Example: “Text1”, “Text2”, “Text3” This will create three test cases, the first one with value Text1, the second with value Text2 and the third one with value Text3. **(see Figure 12, param1)**

Numeric Commands:

Here, the user has three options. To specify a list of single values, the user must write them separated with commas. To specify a range of possible values the user has two options. On the one hand, the user can define a range where the model will generate test cases for every single value in the range with the char “:”. On the other hand, the user can define a range where a random value within the specified range will be chosen and one test case will be generated based on this value, with the slash character “/”.

Here are three examples of this notation **(see Figure 12, param2, param3, param4)**:

- 1,3,6 The tool will create 3 test cases, the first one with value 1, the second with value 3 and the third one with value 6.

- 1:5 The tool will create 5 test cases, the first one with value 1, the second with value 2…

- 1/5 The tool will create 1 test case with a random value between 1 and 5.

Mixed Commands:

Sometimes some parameters’ ranges are composed by both former types, text and numbers. For example, in the REG parameter of the PTSRP command, there can be 8 IPNET registers, from IPNET0 to IPNET 7, 8 TIPCNET registers, from TIPCNET0 to TIPCNET7, etc (see http://calstore.internal.ericsson.com/alexserv?AC=LINK&ID=16585&FN=27_19082-CNZ214278Uen.D.html&PA=PTSRP&ST=FULLTEXT#SearchMatch1). In this case the user has to define

the range between **quotation marks.** For the numerical options, the user can use the ones specified in the former section.

Examples of this notation **(see Figure 12, param5, param6, param7)**:

- "AD-1" ,"AD-4","AD-65" The tool will create 3 test cases, first one with value AD-1, second with value AD-4, and the third one with value AD-65…

- "AD-1:5" The tool will create 5 test cases, first one with value AD-1, second with value AD-2…

- "AD-1/5" The tool will create 1 test case with a random value between AD-1 and AD-5. Note that although AD-1:5 could be represented with notation "AD-1", "AD-2", "AD-3", "AD-4", "AD-5", as discussed in the "Text-Type Commands" section above, this is not an optimal solution, since there can be commands with a large finite set of values, for example, consider the 64 values of parameter REG of PTRML and PTRMP commands (see http://calstore.internal.ericsson.com/alexserv?ID=16585&DB=16577-ipw2.alx&FN=19_19082-CNZ214278Uen.B.html)

Example with parameters of different type written in the interface:

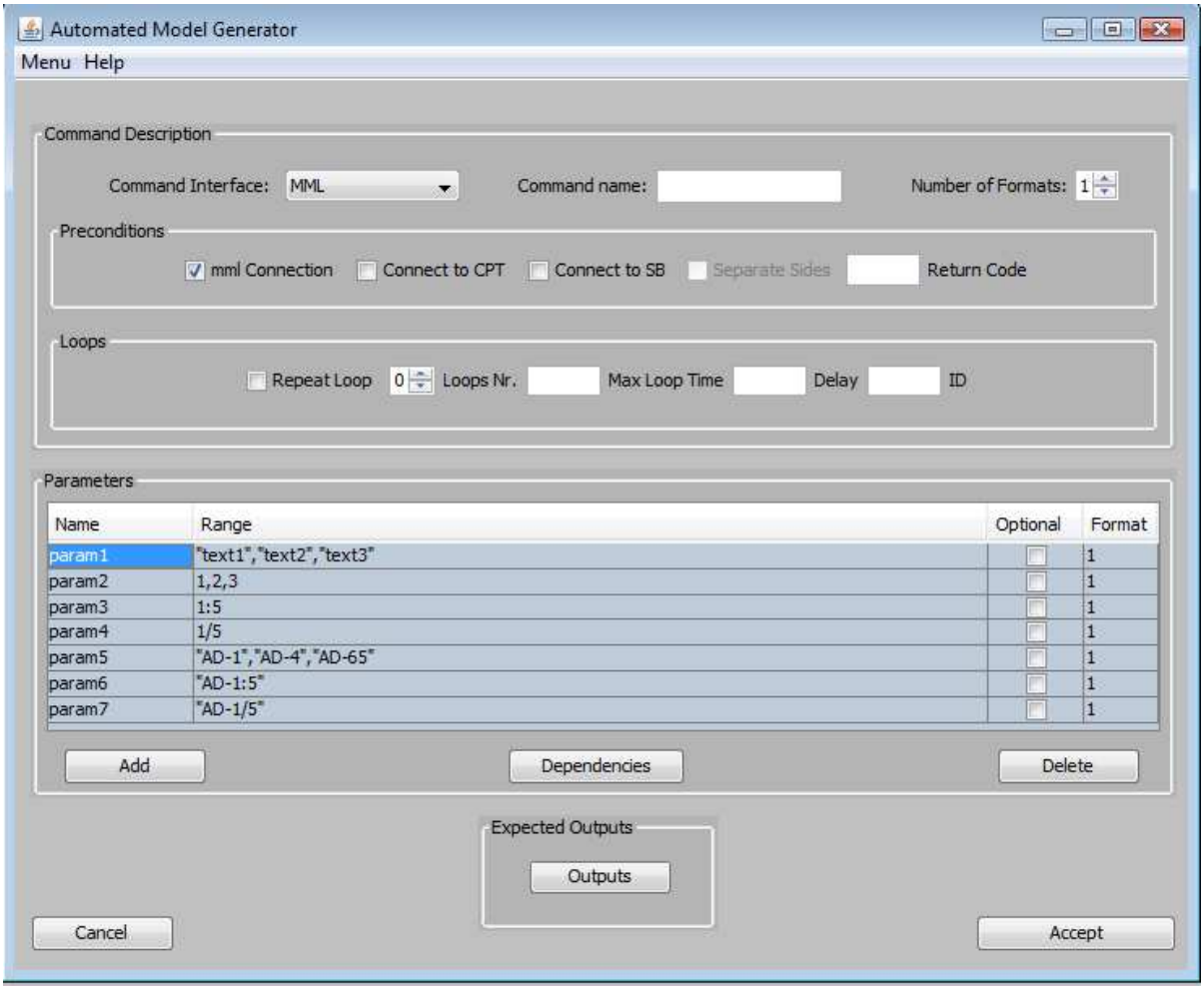


Figure 13. Parameters examples

OBS! It is important to stress that the command to model is written in the AXE common file and the type of value is the same in the tool and in the AXE file.

If the user wants to model a new command (or one that it is not in this file), he must include this command in the AXE file. (See section 6)

Mandatory / optional

There is a checkbox placed on the table to define a parameter as optional which means that one parameter can appear in a specific format/s or not (this leads on different test-cases).

A parameter becomes optional when the checkbox is selected.

Format

The format where the parameter belongs to should be defined here in case that there are multiple formats. If a parameter appears in more than one format, the user must define the number of the format separated by comas. For example:

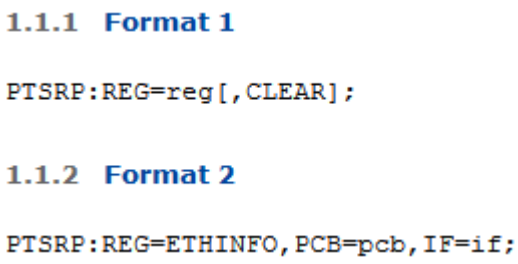
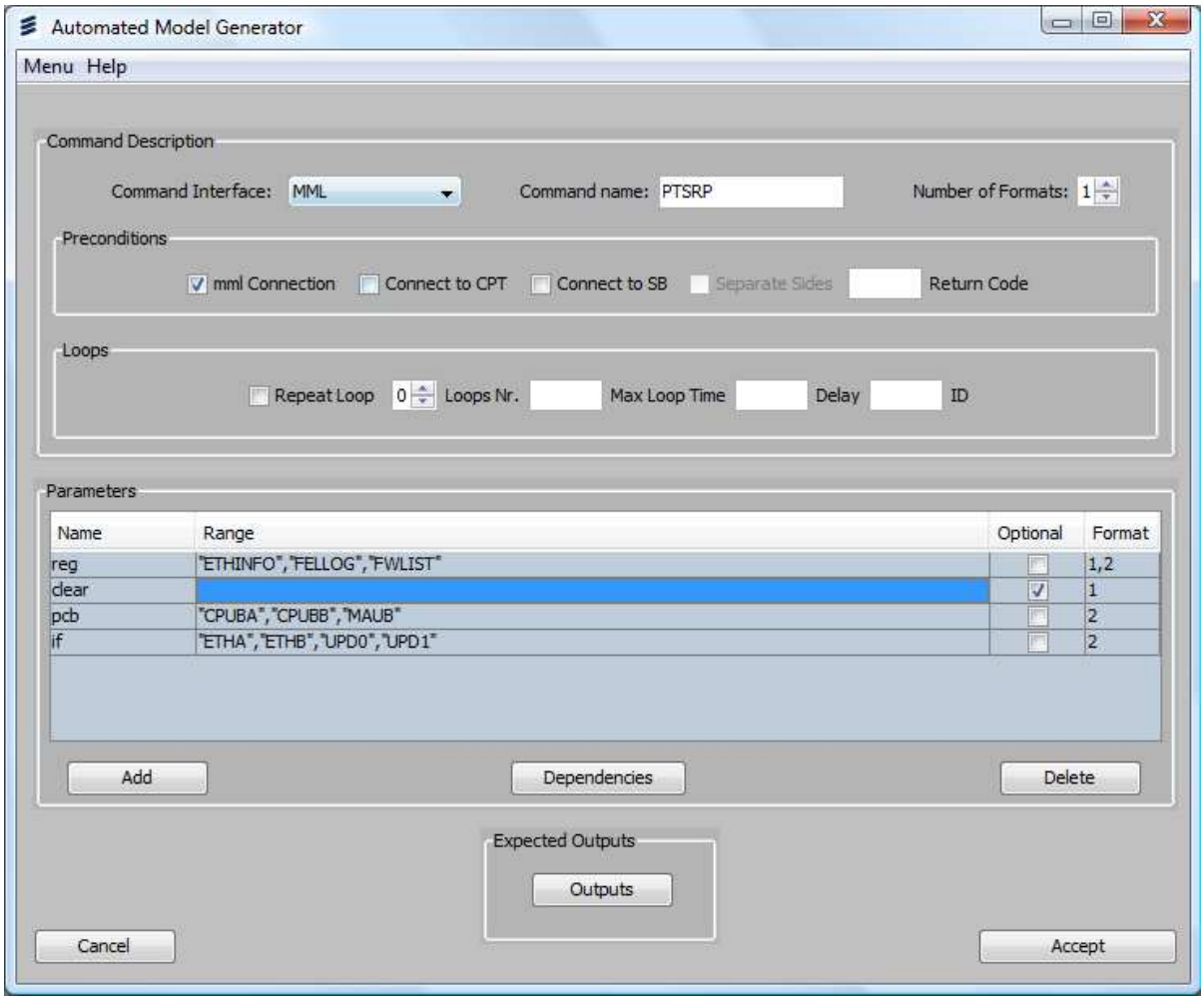


Figure 14. Example of two formats with common parameters

In this case the parameter “reg” is present in two formats, so we define the formats in the last column separated by commas.



The range is taken from the specifications (http://calstore.internal.ericsson.com/alexserv?id=16585).

4.2.7.3 Dependencies

Sometimes, some **optional** parameters require other optional parameters to be present in the same format. In these cases, we can say that these parameters have **dependencies** with other/s. Let's see an example in the figure 13 below:

```
Format 1: MML Session

mml [ -a] [ -c] [ -s -Q] [ -d device ] [ -i device -I device] [ -y lines]
```

Figure 15. MML command format1 description.

As we can see, in the format #1 of the MML command we can find the parameters "-i" and "-I". If we observe properly, both of them have to be present, so they have dependence with each other ("-i" is needed to enter "-I" and vice versa). It is the same case that with the dependency between "-s" and "-Q".

This is the most basic case, although we could find dependencies more complex like in the following examples:

<command> param1 [param2 [param3 param4]]

There we have the following dependencies:

- Param4 depends on param3 and param2 (it cannot appear if they don't)
- Param3 depends on param4 and param2

<command> [param1 [param2 [param3 param4]]]

In this case, although we have the previous dependencies, we have to add one more:

- Param4 depends on param3 and param2 (it cannot appear if they don't)
- Param3 depends on param4 and param2
- Param2 depends on param1.

NOTE: Something important to remember is that a command only has dependencies with other commands in case they are both **optional**. A command cannot be dependent on any other commands which are not optional, because in that case is going to appear always.

"Dependencies" button

In the program, we can specify relationships between parameters clicking on the button "Dependencies", in the bottom of "Parameters" panel.

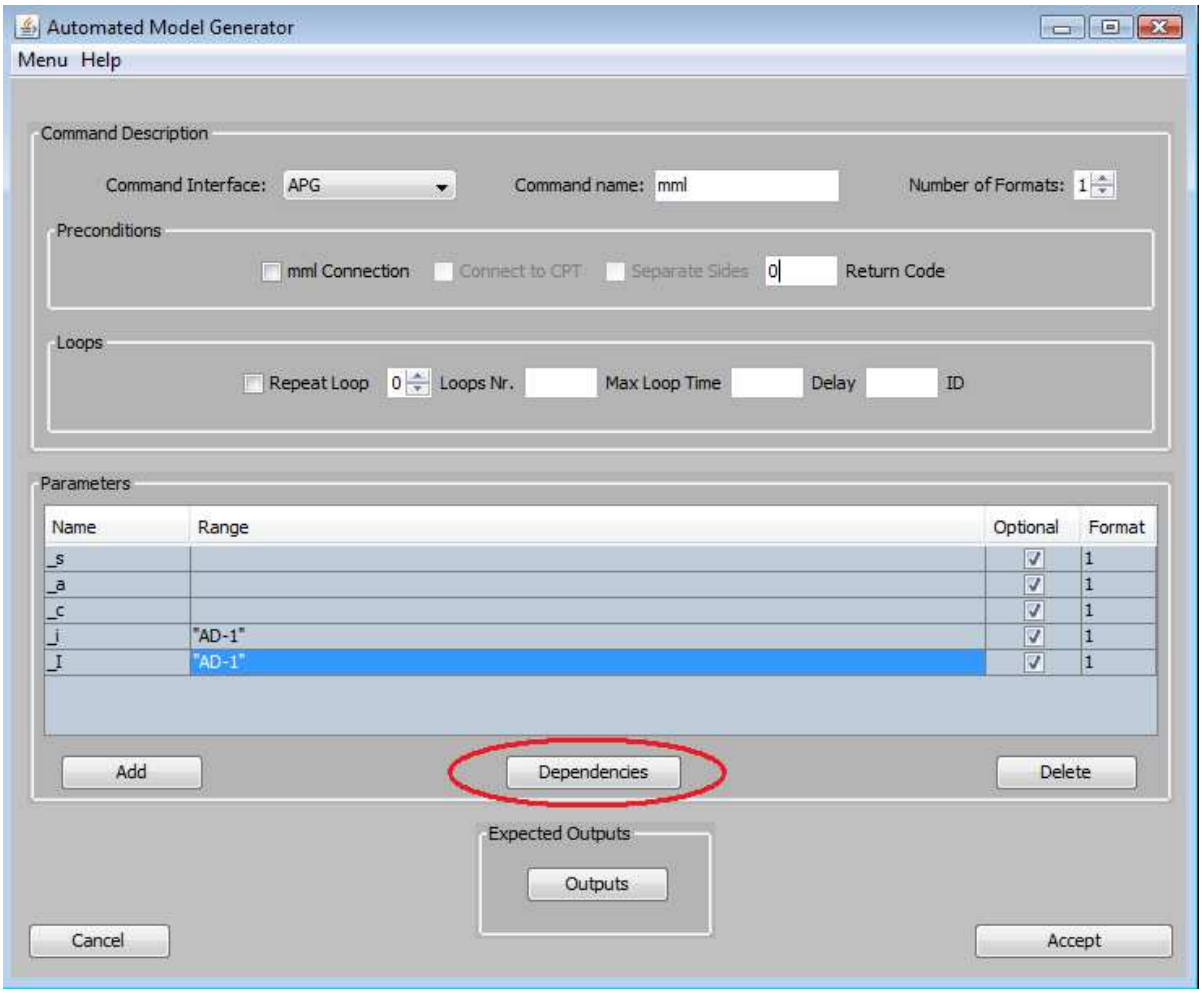


Figure 16. Dependencies button.

Another window pops up, where it is possible to set inter-dependencies. To illustrate this process we will use the example of the MML command. We will follow the next steps to specify the dependency between “i” and “I”.

1. Once all the parameters have been introduced in the table, with the proper format, click on “Dependencies”. A new window pops up (see figure 14).

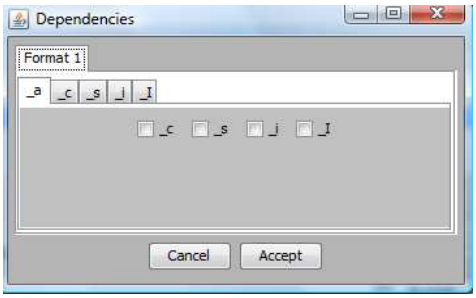


Figure 17. Dependencies editor.

2. In the “Format 1”, click on the “_i” tab, and click on “_I”. Afterwards, click on the “_I” tab and click on “_i”, and click on “accept” (see figure 15).

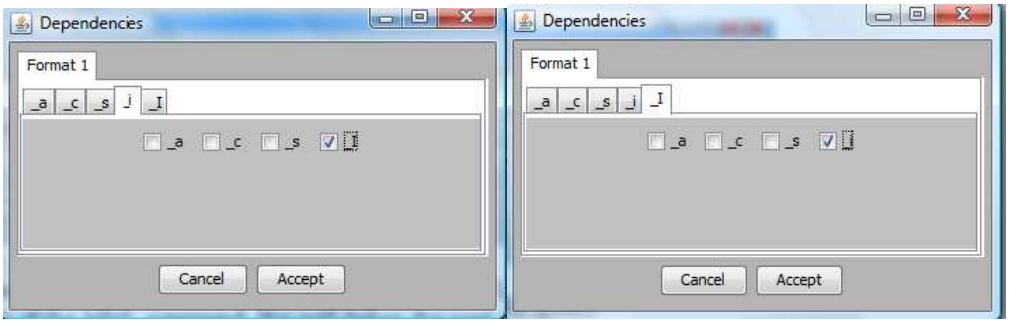


Figure 18. Setting dependencies

3. Create the model

As we stated before, the only requirement is to specify the dependencies of a parameter on the same statement, and in the most immediate level of optionality, e.g. in the previous case *<command> [param1 [param2 [param3 param4]]]* we would click on the following dependencies.

- In the “param4” tab, click on “param3” and “param2”.
- In the “param3” tab, click on “param4” and “param2”.
- In the “param2” tab, click on “param1”.

4.2.8 Outputs

In this section, the expected outputs/printouts from the command must be defined.

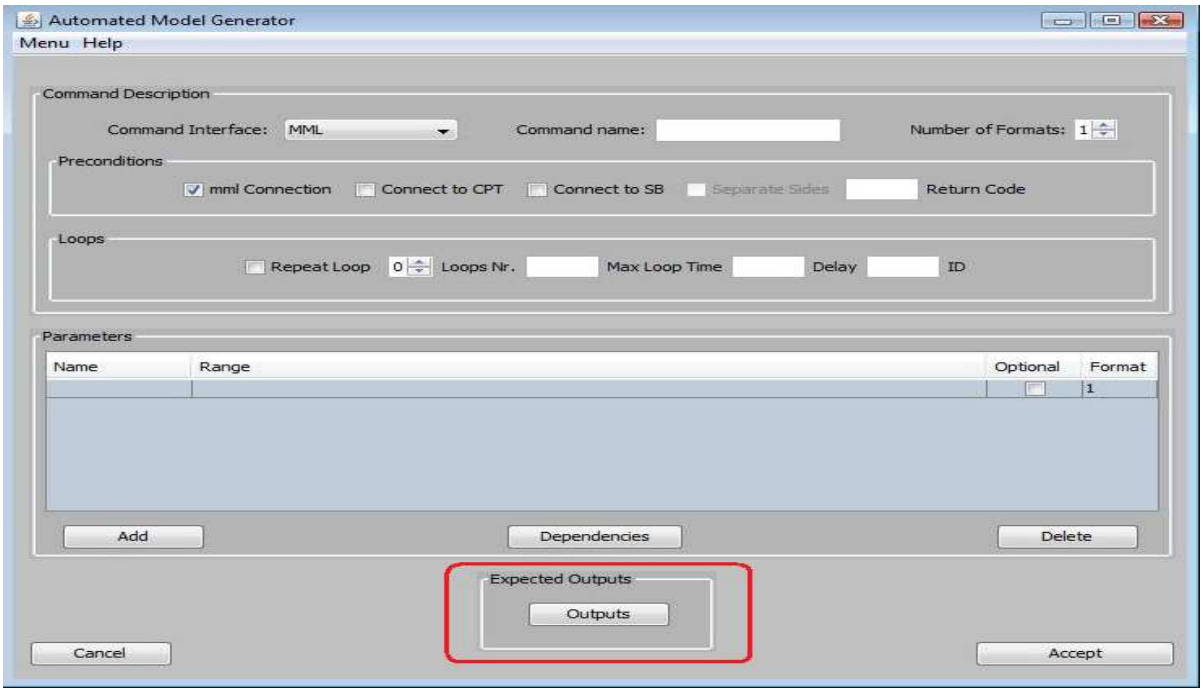


Figure 19. Button for accesing output declaration editor.

The user will be able to define one output per format. Once the user has set the possible number of formats, if he clicks on "outputs" a screen like the one below will appear. The user has to write the expected output for each format on the fields. The outputs are written using regular expressions in ATH-Format. To learn more about regular expressions, please consult the MBT AXE user guide: https://ericoll.internal.ericsson.com/sites/AXE_I_V_MBT/AXE_UG/Lists/Categories/Category.aspx?Name=Appendix%201:%20Using%20regular%20expressions%20in%20printouts

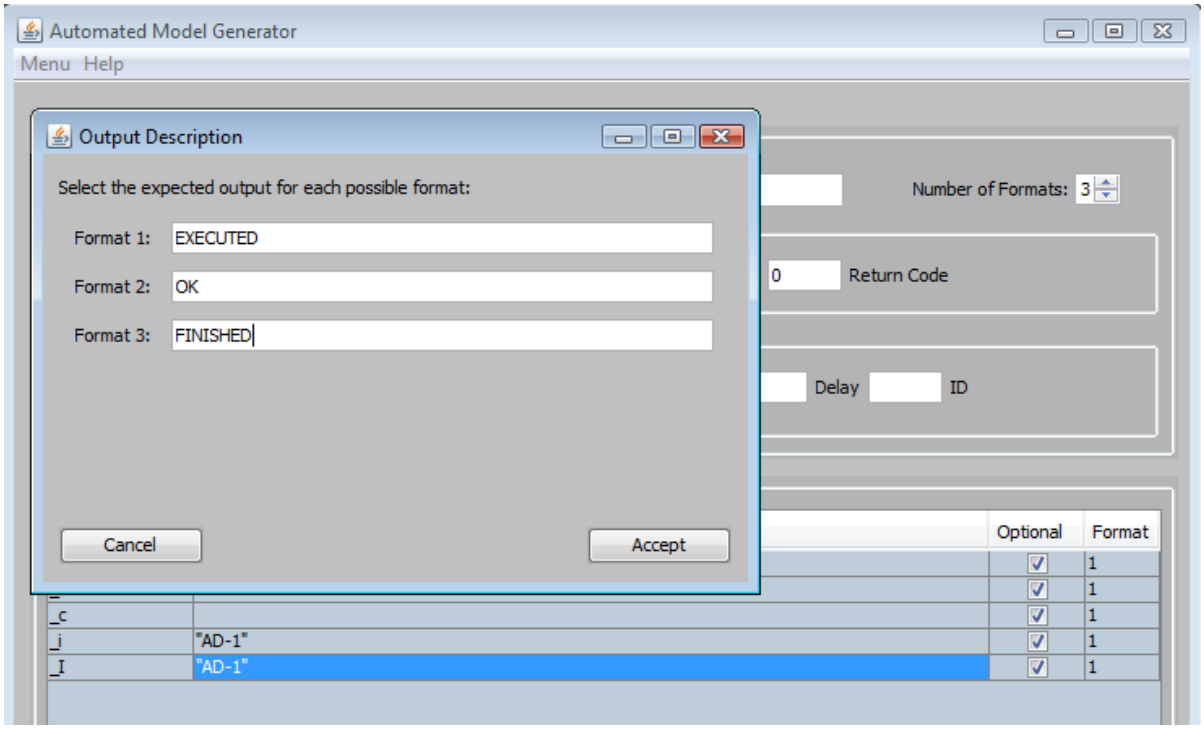


Figure 20. : A screenshot of the command output editor

Clicking on “Accept”, the tool will save the expected outputs and they will be written in the generated test script.

Note: If the user does not click on the “Accept” button or he lets the texts fields in blank, the tool will understand that there is no expected outputs/printouts, so they will not be checked in the auto-generated script afterwards.

4.2.9 Saving a COD

4.2.9.1 XML format

An implemented COD can be saved as a XML format in order to complete the requirements automatically on following occasions. It is explained how it is done in Section 4.2.1.

4.2.9.2 Saving data

All the requirements are saved automatically when the user pushes the button “Accept”.

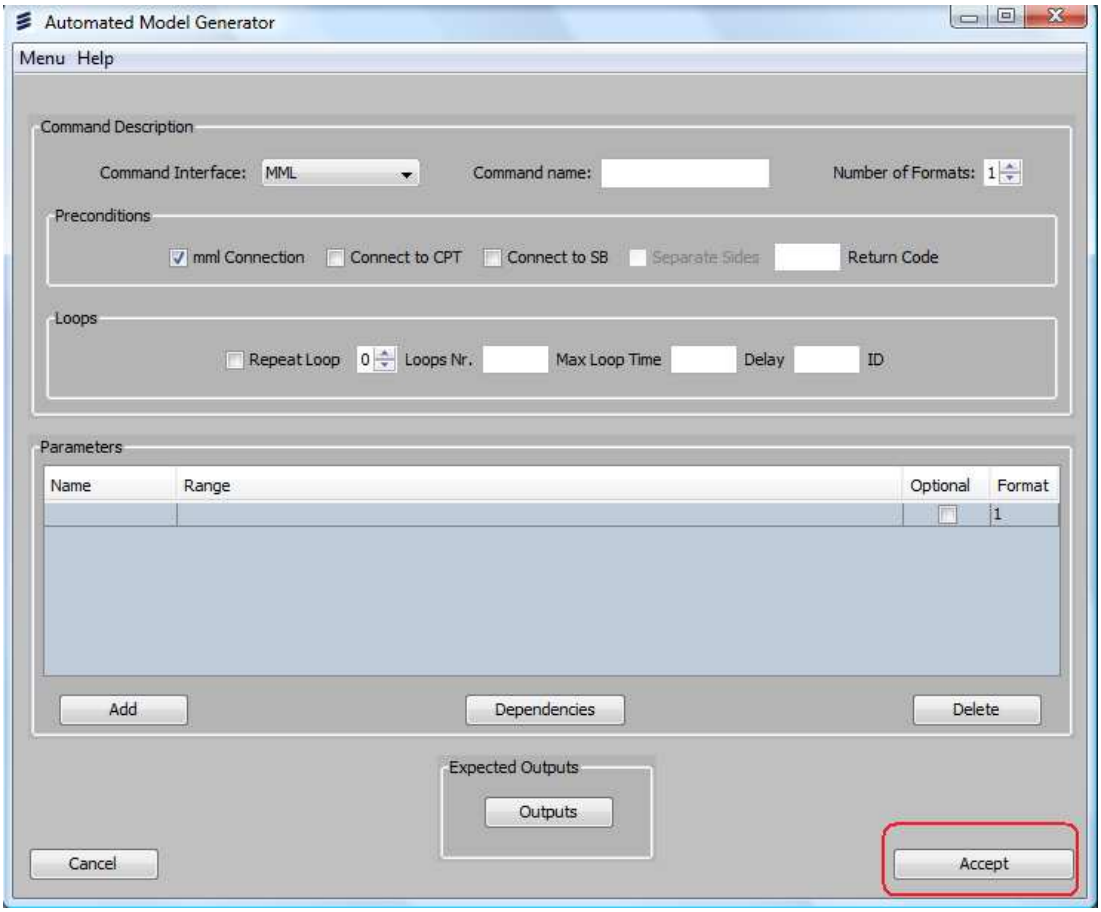


Figure 21. Location of storage and robot settings

All the written requirements are saved and the user can continue modeling the sequence in the AMG canvas.

5 Procedure steps with an existing COD in the library:

5.1 Test cases generation Instructions

These are the steps that need to be followed by the user to generate test cases using AMG:

1. Open AMG tool
2. Select the command's type to implement
3. Fulfill the required information in the interface (Section 4)
4. [OPTIONAL] If "Create project" is on, the user must set Eclipse with Conformiq interface in the background (Section 4)
5. Click on "Create model"
6. Select the created files ("command_name.cqa", "command_name_main.cqa", "template configuration.cqa" and "command_name.xmi") and copy into the project file in Conformiq (drag and drop), in "model" folder.
7. Link the files "AXE Common" and "command_library" to the existing project (drag and drop)
8. Click on the button "generate test cases from model" 

5.2 Executing the test cases

9. In the project file, there is a green icon named "DC", click with the right button on it and select "new" and "Create a new Scriptbackend"
10. User must select which Scripter wants to use (Confomiq, ATHScripter…)
11. Clicking on "Render test cases with enabled scripts"  the user will create the test scripts which contains all the desired test cases
12. Execute the test cases

6 Procedure steps with a non-existing COD in the library:

If the user wants to implement a missing command (in the AXE Common file) or a new command, he will have to edit AXE Common file as follows:

1. Open the file "AXE_Common.cqa" [1] and the "commandLibrary.cqa"[2]
2. If it is a mml command: write the command name in the "Inbound mmlCmd" section.
   If it is an apg command: write the command name in the "Inbound apgCmd" section.

[1] If the user does not have a local copy, he can download from the repository from:https://teamforge.lmera.ericsson.se/svn/repos/ath/DEVELOPMENT/Models/Common/AXE_Common.cqa

[2] If the user does not have a local copy, he can download from the repository https://teamforge.lmera.ericsson.se/svn/repos/ath/DEVELOPMENT/Models/Common/commandLibrary.cqa

3. Write the following lines in the document afterwards:

```
record Command_name {
    int sessionId prefer 0;
    int timeout prefer 60;
    // From Here write the parameters' name
}
```

   NOTE: If the parameter is optional, the user must indicate it on the record as:
   Optional <Type> Parameter_name
4. Save the modified file
5. Follow the "procedure steps with an existing command" (Section 5)

7 Example of a COD

In this section, a new command will be implemented so the user can easily see and understand the whole process. The new command to model is:

APG Command
Format 1:
Myexample param_1 [param_2] [param_3 [param_4 param_5]]
Where the values can be: param_1 = "AD-1", "AD-3", "AD-5"
param_2 = [10,20]
param_3 = "AD-10" – "AD-100"
param_4 = "YES", "NO"
param_5 = [10,15]
Format 2:
Myexample param_1 param_6
Where the values can be: param_1 = "AD-1", "AD-3", "AD-5"
param_6 = "YES","NO"

Outputs: Format 1: "OK"
Format 2: no need to check printouts/outputs

The notation below is the same as the one that appears in the documentation. "[]" means optional parameter, the comma separates the possible values and the ranges are defined with a "-".
Thus, following the steps in Section 6:

1. Open the file "AXE_Common.cqa" (make sure that you have "commandLibrary.cqa" in your computer too)
2. We write in apgCmd: myExample

```
/* APG command interfaces */
Inbound apgCmd : Myexample, hostname, prcstate, alist, mml, trautil, cpfls, cfeted, rm, q,
                 cluster, phaprint, prcboot, Y, N, sfcexec, sfcfb, ptcoi, bupdef, bupls, bu
                 ispprint, CONNECT, RELEASE, ls, dir,cat, startServices_MML, ps, unknownCom
                 shelfmngr, kill, startTOCAP, PRINT, SET, cqrhlls, cqrhils, date, clhls, cr
Outbound apgPrt : stdOut, CPF_FILE_TABLE, date_printout, RELOAD_PARAMETERS;
```

3. We write the parameters in the space reserved for Inbound APG commands:

```
record Myexample {
    int sessionId prefer 0;
    int timeout prefer 60;
    String param_1;
    Optional <int> param_2;
    Optional <String> param_3;
    Optional <String> param_4;
    Optional <int> param_5;
    String param_6
}
```

4. We save the file.
5. Open AMG Tool and fulfill the fields required (See Section 5)
   In the example below, we will specify that for param_2 and param_3 the system will only test one random value in the range and for param_5, it will test all the possible values (6 values).

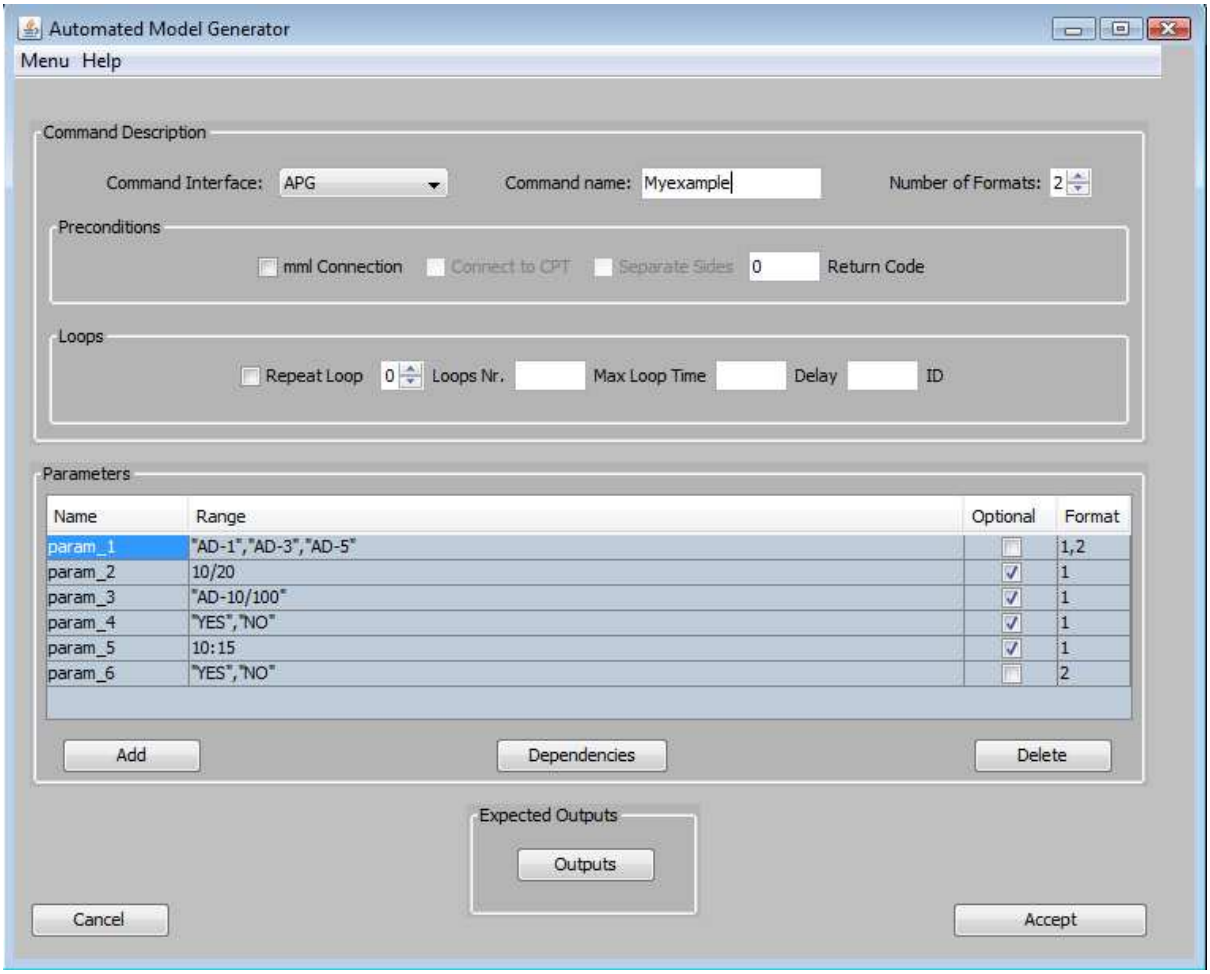


Figure 22. Myexample parameters

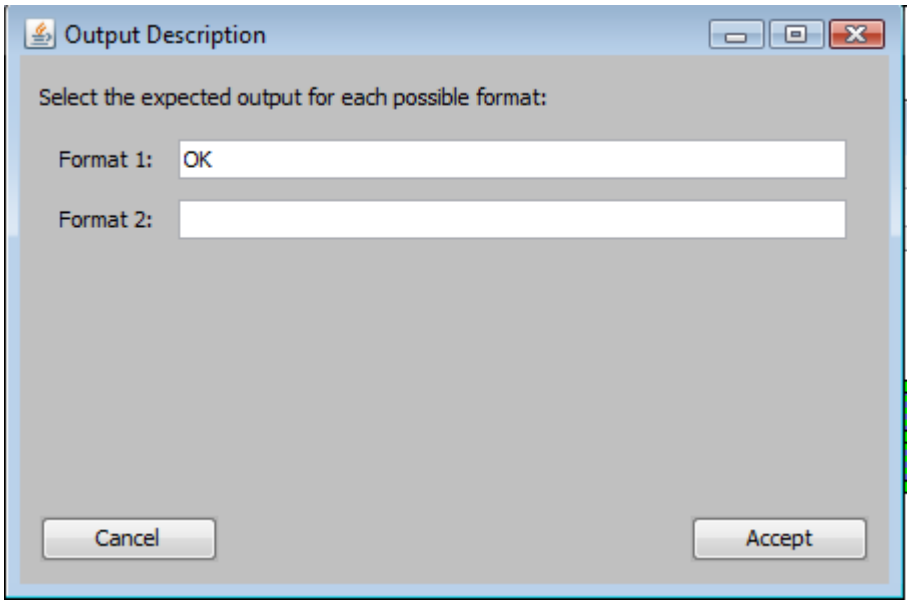


Figure 23. Myexample output

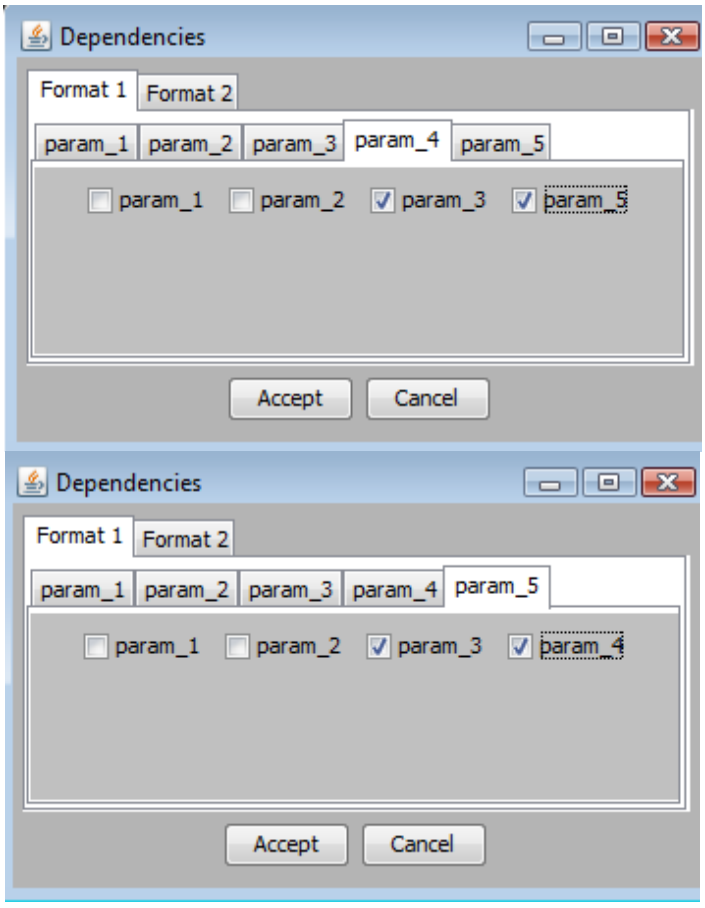


Figure 24. Myexample dependencies

6. Click on “Create model”.
7. Copy the files to the project created in Conformiq (do not forget to have Conformiq opened in the background) in the “model” folder inside “Myexample-test” folder. (drag and drop)
8. Link to “AXE_Common.cqa” and “commandLibrary.cqa”.(drag and drop in “model” folder and click on link files)

9. Click on the button “generate test cases from model” 
10. Execute the test cases following the instruction in Section 5

If the user wants to implement an existing command which is already in the “AXE_Common.cqa” file, he has to omit the steps from 1 to 4.

OBS! The type of the parameters must match with the type defined previously in the “AXE_Common.cqa” file.

8 Example of an OPI

The modeled OPI is the instruction ‘Initial Load’ which is defined in the APG43 Manual, in page 27. It is a simple example so the user can easily get the idea of implementing an OPI and extend it to other instructions.

8.1 Implementation

It is composed of two boxes because the first steps can be introduced as pre-conditions in the first command modeled. Thus, the sequence is as follows:

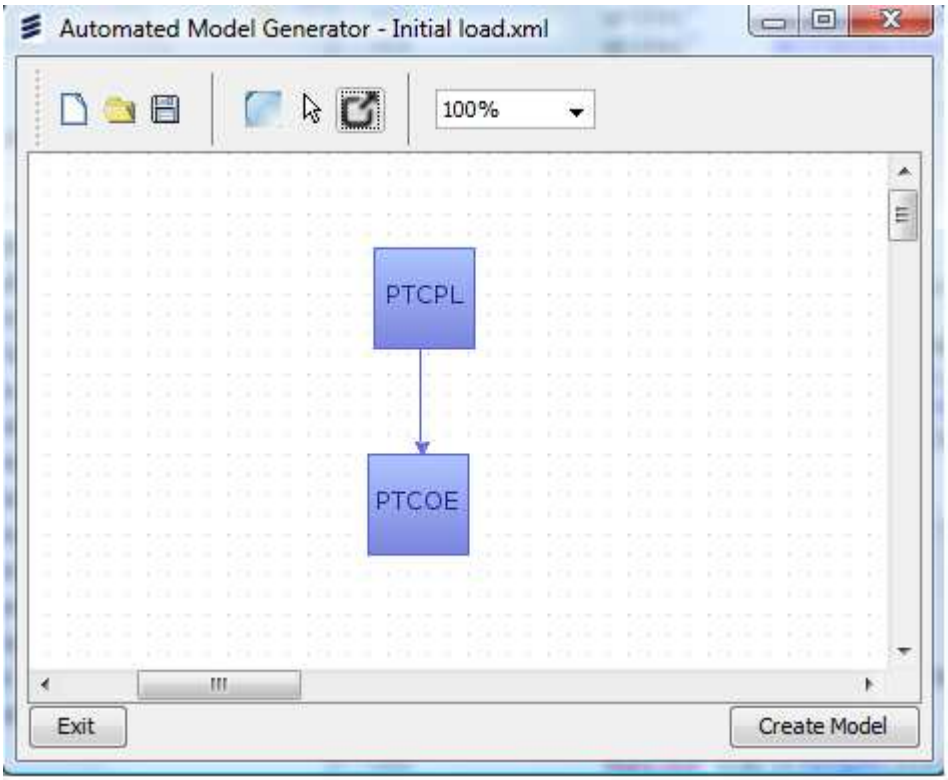


Figure 25. Pre-model

where the PTCPL has some pre-conditions selected and two parameters defined. MML connection, CPT connection and Side B connection are selected (in this example, side B is used). In addition, parameters ‘file’ and ‘cs’ are specified as it is said in the specifications and the expected output. Everything is shown in the following snapshots:

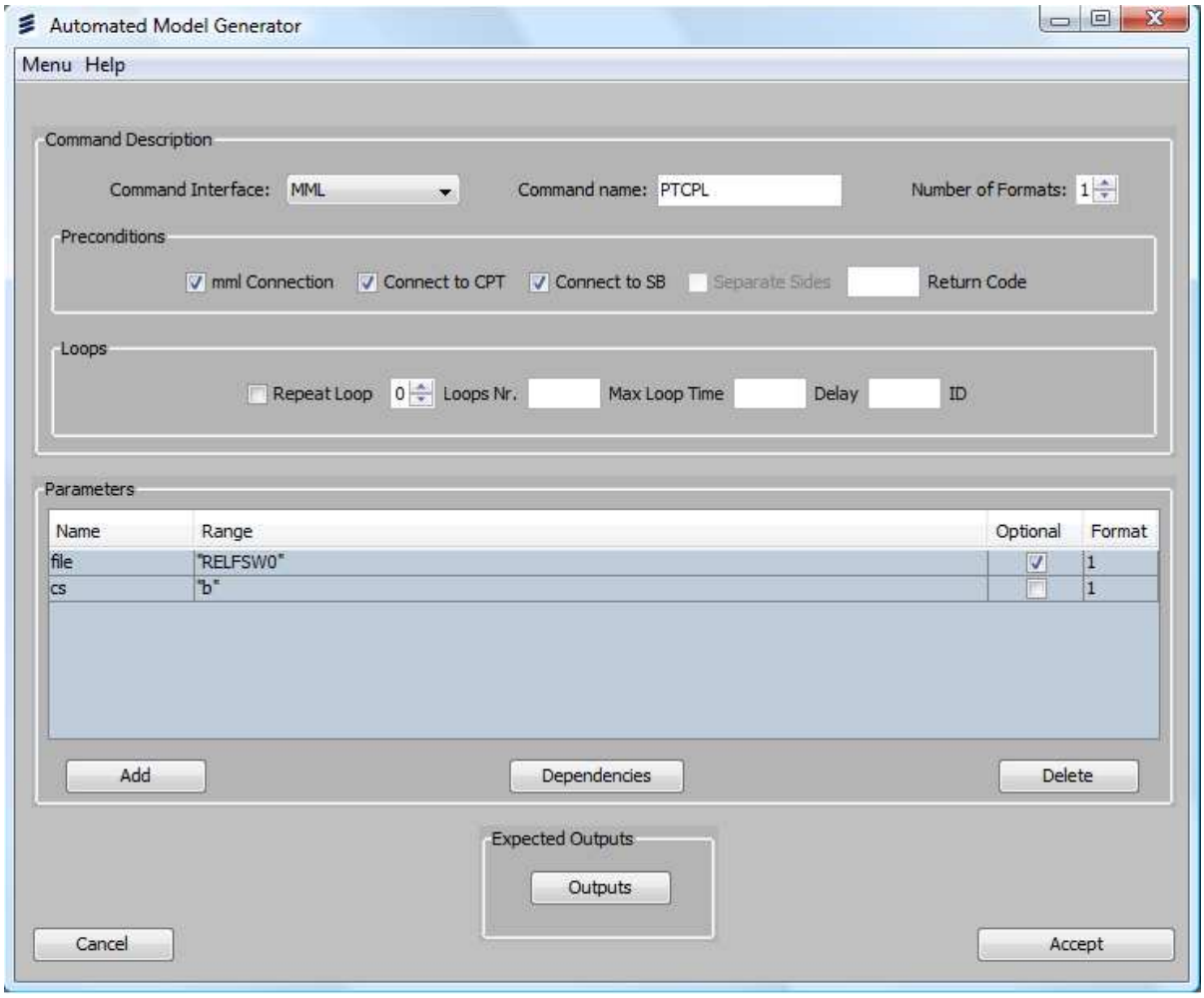


Figure 26. PTCPL requirements

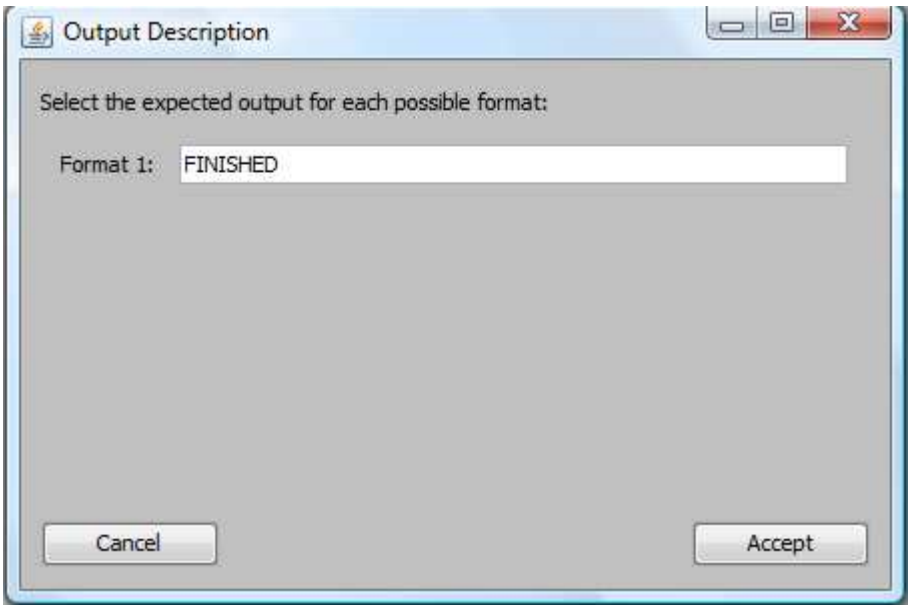


Figure 27. PTCPL's output

And the PTCOE which has not any parameters or outputs:

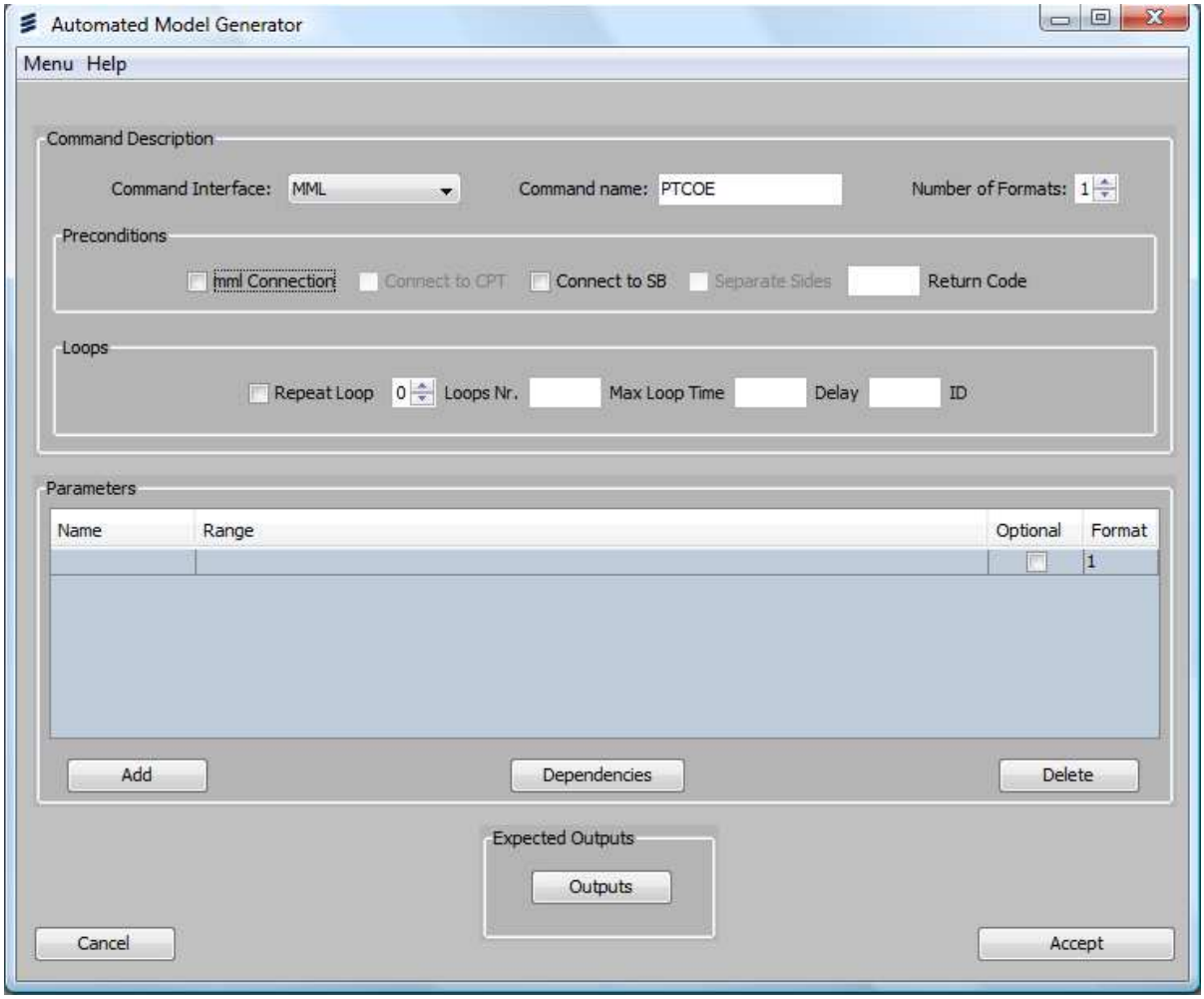


Figure 28. PTCOE requirements

Note that this command checks the output after its execution and therefore, as this value is checked here, the user does not need to specify any condition in the transition between the PTCPL and the PTCOE.

8.2 XML Model

Once the whole model is implemented as explained before, we have to save it as an XML model. For this purpose, the button “Save model” is pushed and a directory is selected in the dialog.

Figure 29. Saving dialog

Then, we can proceed with the next step and generate the model in QML (Conformiq). We choose the name 'Initial Load.xml'.

8.3 Conformiq Model

To generate this model, the button 'Create Model' is pushed and automatically the model is transformed into the following Conformiq model and it is automatically called 'Initial Load.xmi':

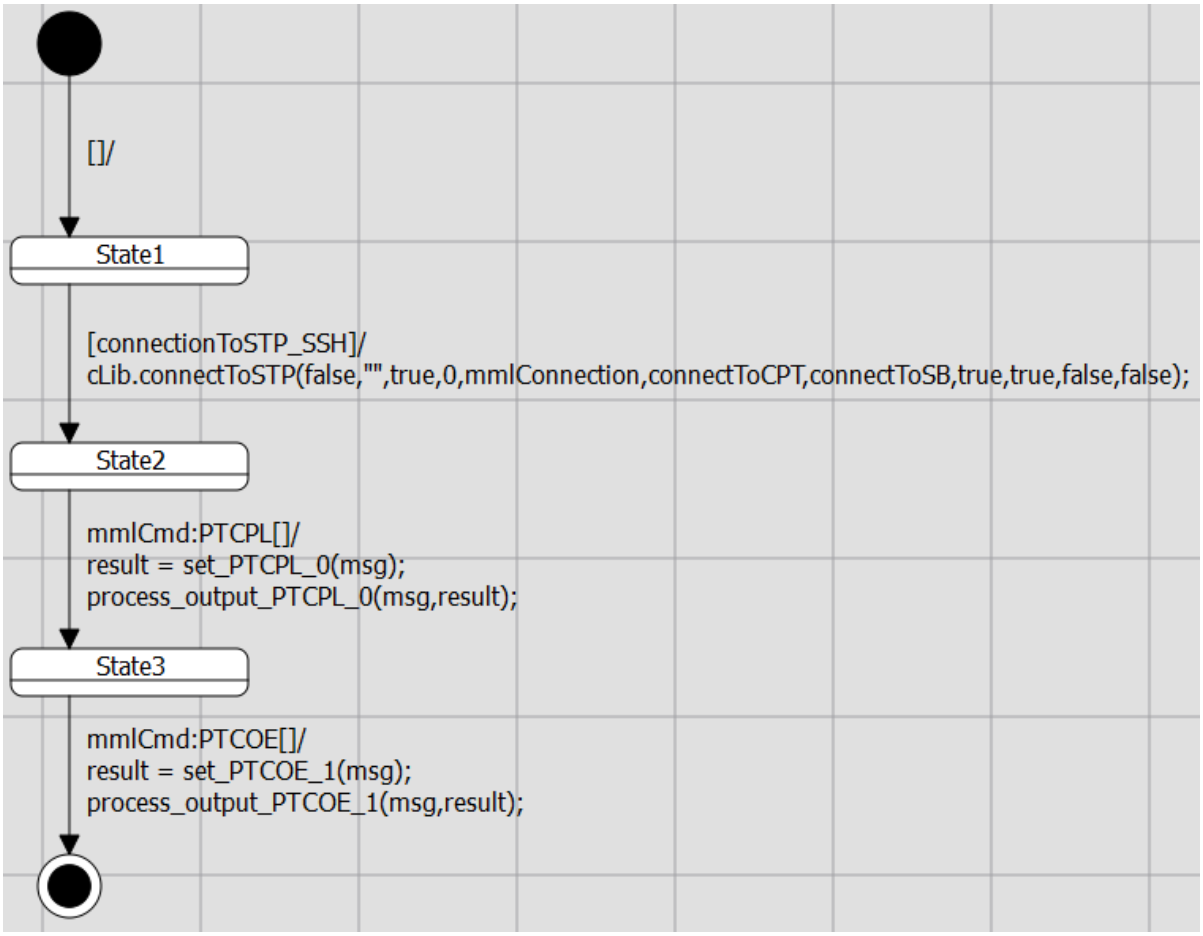


Figure 30. Conformiq model

The reader can appreciate that all the pre-conditions are defined in the function *connectToSTP* which is defined in the library. The parameters and their values together with the expected outputs are defined in the 'Initial Load.cqa'.

8.4 Execution

Once everything is imported to Conformiq, the test-cases can be generated and executed through Conformiq.

9 Contact Details

If you have any suggestions, feedback or relevant information, you can contact to:

Ignacio Mulas Viela: ignacio.mulas.viela@ericsson.com

Armando Gutierrez: armando.gutierrez@ericsson.com

Appendix D

PTCFP specification document

PROCESSOR TEST CENTRAL, FORMAT, PRINT

1 Format

1.1 Command

```
PTCFP:ADDR=addr[,NRW=nrw];
```

1.2 Parameters

`ADDR=addr`	Address to print. Numeral 0 - H'3F.
`NRW=nrw`	Number of words to be printed out. If parameter NRW is omitted, then NRW=1 is used. Numeral 1 - H'40.

2 Function

The command is received in a support processor.

The command is used when reading one or more consecutive words in REFM in the CP side CPT is connected to.

The command is accepted if CPT is connected.

CPT is connected by the command PTCOI .

3 Examples

3.1 Example 1

```
PTCFP:ADDR=H'1B;
```

The contents of the word with address H'1B in REFM is printed out.

3.2 Example 2

```
PTCFP:ADDR=H'05,NRW=H'3;
```

The contents of REFM at addresses H'05-H'07 are printed out.

4 Printouts

4.1 Check Printout

No.

4.2 Procedure Printouts

NOT ACCEPTED
fault type

Fault type:

FUNCTION BUSY
The function is busy.

FORMAT ERROR
The command is only accepted if CPT is connected.

FAULT CODE 1
CPT NOT CONNECTED
The command is only accepted if CPT is connected.

FAULT CODE 2
DATA LINK ERROR
Error on data link to CP.

FAULT CODE 3
TIMEOUT IN MAU OR CPU
Time out in cooperation with either MAU or CPU.

FAULT CODE 4
COMMAND NOT SUCCESSFULLY COMPLETED
Fault detected during the command execution.

FAULT CODE 7
UNREASONABLE VALUE
Unreasonable value.

FAULT CODE 8
CP SIDE NOT SEPARATED
The CP side is not separated.

FAULT CODE 14
CPT ALREADY IN USE
CPT is already in use.

FAULT CODE 24
ILLEGAL ADDRESS
The address is illegal.

4.3 Answer Printouts

CPT MESSAGE REFM

4.4 Result Printouts

This command has no result printouts.

5 Logging

Not relevant.

6 Command Category Group

The authority depends on the category group of the path-building command. The category group within the support processor is H.

7 Command Receiving Block

CPT2D

8 Glossary

REFM	REgister File Memory

9 References

9.1 Command Descriptions

PTCOI Processor Test, Connection, Initiate

Appendix E

PTWRP specification document

Processor Test, WEJEG Register, Print

1 Format

1.1 Command

```
      /-                                -\
      |REG=reg,RPBIS=rpbis[,RPBH=rpbh]|
PTWRP:+                                 +;
      |DUMP=dump,RPBIS=rpbis            |
      \-                                -/
```

1.2 Parameters

`DUMP=dump` Dump Identifier
Identifier 4 - 10 characters

`ERREVENT`
: Regional Processor Bus Interface, Serial (RPBIS) Error Event

`ERRLOG`
: RPBIS Error Event Log

`ETH`
: Ethernet interface information

`EVENT`
: RPBIS Event Log

`LINKSTATE`
: RPBIS Link Status Log

`MODE`
: RPBIS Regional Processor Handler (RPH) Mode Log

`RPBISSTATE`
: RPBIS State Log

`SIGNAL`
: RPBIS Signal Log

`STAT`
: RPBIS Statistics

`REG=reg` Register Identifier

Every register starting with RPBH will require the RPBH parameter in the command.
Identifier 3 - 15 characters

`CPINFO`
: Central Processor (CP) Information for own RPBIS register

`CPINFOTWIN`
: CP information for twin RPBIS register

`CPUBLNKSTS`
: Status of Transparent Inter-Process Communication (TIPC) links

towards CPUBs register

ECCCNTR

Error Correction Code (ECC) Counter register

INBNDBUFCNGSTN

Congestion in RPBIS Inbound Buffer register

INBNDBUFLMT

Limit in RPBIS Inbound Buffer register

INBNDBUFSIZE

Size of RPBIS Inbound Buffer register

INBNDBUFUSED

Number of signal positions used by RPBIS Inbound Buffer register

MAUBLNKSTS

Status of Dual Link Protocol (DLP) links towards Maintenance Unit Board (MAUB) register

OUTBNDBUFCNGSTN

Congestion in RPBIS Outbound Buffer register

OUTBNDBUFLMT

Limit in RPBIS Outbound Buffer register

OUTBNDBUFSIZE

Size of RPBIS Outbound Buffer register

OUTBNDBUFUSED

Number of signal positions used by RPBIS Outbound Buffer register

RESTARTGEN

CP Restart Generation register

RHLSTS

Regional Processor Bus Handling Logic (RHL) Status register

RPADDRCONTFETCH

Regional Processor (RP) Address for Continuous Fetch Mode register

RPBHIBIL

Regional Processor Bus Handler (RPBH) Inbound Interrupt Level register

RPBHIBNS

RPBH Inbound Number of Signals register

RPBHIBO

RPBH Inbound Offset register

RPBHIBPTR

RPBH Inbound Pointer register

RPBHIBSA

RPBH Inbound Start Address register

RPBHIBT

RPBH Inbound Timer register

RPBHLOCK

Lock bits for RP Bus Handler register

RPBHMODE

RPBH Mode register

RPBHNISF

RPBH Number of Inbound Signals Location register

RPBHNOSV

RPBH Number of Outbound Signals Valid register

RPBHOBDP

RPBH Outbound Descriptor Pointer register

RPBHOBIT

RPBH Outbound Interrupt Threshold register

RPBHOBND

RPBH Outbound Number of Descriptors register

RPBHOBO

RPBH Outbound Offset register

`RPBHOBSA`
RPBH Outbound Start Address register

`RPBHRIC`
RPBH Regional Processor Bus-Serial (RPB-S) Interface (RI) Control register

`RPBHRIP`
RPBH RI Parameter register

`RPBHRPBLKTBL`
RPBH RP Blocking Table register

`RPBHSTATE`
RPBH State register

`RPBHSTPRSN`
RPBH Temporary Stop Reason register

`RPBISINFO`
RPBIS state Information for own RPBIS register

`RPBISINFOTWIN`
RPBIS state Information for twin RPBIS register

`RPBISSTOPRSN`
Reason for Temporary Stop of RPBIS register

`RPHMODE`
RPH Mode for own RPBIS register

`RPHMODETWIN`
RPH Mode for twin RPBIS register

`TWINLNKSTS`
Status of Twin Communication Links towards twin RPBIS register

`RPBH=rpbh` RPBH Number value
Numeral 0 - 9

`RPBIS=rpbis` RPBIS Address
Identifier 3

`A-0`
RPBIS 0 on A Side

`A-1`
RPBIS 1 on A Side

`A-2`
RPBIS 2 on A Side

`B-0`
RPBIS 0 on B Side

`B-1`
RPBIS 1 on B Side

`B-2`
RPBIS 2 on B Side

`ALL`
All RPBIS on A side and B side

2 Function

This command prints the register or dump contents from one or all RPBIS boards present in the system.

This command is received in a Support Processor (SP).

The command is only accepted if Central Processor Test (CPT) is connected.

CPT is connected by the command PTCOI .

Each RPBIS has 10 RPBHs (RPBH-0 to RPBH-9).

The order does not remain after system restart.

3 Examples

3.1 Example 1

```
PTWRP:REG=RHLSTS,RPBIS=ALL;
```

Print the contents of RHL status register of all RPBIS boards present in the system.

3.2 Example 2

```
PTWRP:DUMP=SIGNAL,RPBIS=A-0;
```

Print the contents of signal log from RPBIS-A-0.

3.3 Example 3

```
PTWRP:REG=RPBHSTATE,RPBIS=A-0,RPBH=0;
```

Print the contents of state register from RPBH-0 of RPBIS-A-0.

4 Printouts

4.1 Check Printout

No.

4.2 Procedure Printouts

```
NOT ACCEPTED
```
fault type

Fault type:

FUNCTION BUSY
The function is busy.

FORMAT ERROR
The command or a parameter was incorrectly specified.

FAULT CODE 1
CPT NOT CONNECTED
The command is only accepted if CPT is connected.

FAULT CODE 2
DATA LINK ERROR
An error was detected on the data link to the CP.

FAULT CODE 3
TIMEOUT IN MAU OR CPU
Time-out occurred in cooperation with either Maintenance Unit (MAU) or Central Processor Unit (CPU).

FAULT CODE 4
COMMAND NOT SUCCESSFULLY COMPLETED
A fault was detected during execution of the command.

FAULT CODE 7
UNREASONABLE VALUE
The parameter was specified with an unreasonable value.

FAULT CODE 52
RPBIS NOT PRESENT

An RPBIS board is physically not present in the system.

FAULT CODE 54
TIMEOUT IN MAU OR RPBIS

Time-out occurred in cooperation with either MAU or RPBIS.

4.3 Answer Printouts

CPT MESSAGE RPBIS REGISTER

4.4 Result Printouts

This command has no result printouts.

5 Logging

Not relevant.

6 Command Category Group

The authority depends on the category group of the path-building command. The category group within the Support Processor is H.

7 Command Receiving Block

CPT2D

8 Glossary

CP	Central Processor
CPT	Central Processor Test
CPU	Central Processor Unit
DLP	Dual Link Protocol
ECC	Error Correction Code
MAU	Maintenance Unit
MAUB	Maintenance Unit Board
RHL	Regional Processor Bus Handling Logic
RI	RPB-S Interface
RP	Regional Processor
RPBH	Regional Processor Bus Handler
RPBIS	Regional Processor Bus Interface, Serial
RPB-S	Regional Processor Bus-Serial
RPH	Regional Processor Handler
SP	Support Processor
TIPC	Transparent Inter-Process Communication

9 References

9.1 Command Descriptions

`PTCOI` Processor Test, Connection, Initiate

Appendix F

APFPL specification document

Adjunct Processor File System, Printout List

Contents

1 Format

1.1 Single-CP System

```
apfpl -p file [-subfile]

apfpl [ -l] [ -q] [ -s] [file]

apfpl -l [ -k] [ -s] [ -q] [file]

apfpl [ -l] [ -q] file-subfile[-generation]

apfpl -l [ -k] [ -q] file-subfile[-generation]
```

1.2 Multi-CP System

```
apfpl -cp cpname | CLUSTER -p file [-subfile]

apfpl -cp cpname | CLUSTER [ -l] [ -q] [ -s] [file]

apfpl -cp cpname | CLUSTER -l [ -k] [ -s] [ -q] [file]

apfpl -cp cpname | CLUSTER [ -l] [ -q] file-subfile[-generation]

apfpl -cp cpname | CLUSTER -l [ -k] [ -q] file-subfile[-generation]
```

2 Parameters

2.1 Options

`-cp cpname | CLUSTER`
: CP Name
This option specifies the Central Processor (CP) name or CLUSTER in a Multi-CP System.

The command is executed against the file system at the CP level related to the specified CP that is a folder named the same as the CP name. If CLUSTER is specified, the command is executed against the file system at Cluster level that is a folder named CLUSTER.

Text string 1 - 7 characters

`-k`
: Compression status
This option is used to list compression status.

`-l`
: Long listing
This option is used to list additional file information.

`-p`
: Path listing
This option is used to list the physical path to a specific file.

`-q`
: Quiet listing
This option is used to suppress all headings.

`-s`
: Subfiles listing
This option is used to list all subfiles belonging to a composite file. The option is ignored for simple files.

2.1.1 Operands

`file`
: CP file name
This operand specifies the CP file. This operand is not case sensitive.
Identifier 1 - 12 characters

`file-subfile[-generation]`
: Subfile name
This operand specifies the subfile. The subfile name consists of three parts, separated by hyphens. The first part indicates a composite main file, the second part and the optional third part indicates the subfile extension. This operand is not case sensitive.

file

CP File name

Identifier 1 - 12 characters

subfile

Subfile name

Symbolic name 1 - 12 characters

generation

Generation name

Symbolic name 1 - 8 characters

3 Function

This command is used to print the names and attributes of one or all of the files in the CP file system. In a Multi-CP System the file system at CP or Cluster level is addressed with the option -cp. It is also used to print the physical path to a specific file.

The result of the command execution does not remain after system restart.

4 Examples

4.1 Example: 1

In the following example, all main files are listed.

```
apfpl
CPF FILE TABLE

FILE                                    TYPE  CMP  VOLUME
RELFSW0                                 reg   yes  RELVOLUMSW
RELFSW1                                 reg   yes  RELVOLUMSW
RELFSW2                                 reg   yes  RELVOLUMSW
LOGFILE                                 reg   no   LOGVOLUME
LO                                      reg   yes  TEMPVOLUME
TRA01                                   inf   yes  TRANSFER
TRA02                                   inf   yes  TRANSFER
TRA03                                   inf   yes  TRANSFER
TRA04                                   inf   yes  TRANSFER
TRA05                                   inf   yes  TRANSFER
TRA06                                   inf   yes  TRANSFER
```

4.2 Example: 2

In the following example, all main files located in the Cluster file system are listed. Note this example is valid only for a Multi-CP System.

```
apfpl -cp CLUSTER
CPF FILE TABLE

FILE                                    TYPE  CMP  VOLUME
RELFSW0                                 reg   yes  RELVOLUMSW
RELFSW1                                 reg   yes  RELVOLUMSW
RELFSW2                                 reg   yes  RELVOLUMSW
LOGFILE                                 reg   no   LOGVOLUME
LO                                      reg   yes  TEMPVOLUME
TRA01                                   inf   yes  TRANSFER
TRA02                                   inf   yes  TRANSFER
TRA03                                   inf   yes  TRANSFER
TRA04                                   inf   yes  TRANSFER
TRA05                                   inf   yes  TRANSFER
TRA06                                   inf   yes  TRANSFER
```

4.3 Example: 3

In the following example, all simple files and composite main files, including additional file information, are listed.

```
apfpl -l
CPF FILE TABLE

FILE                                TYPE  CMP  VOLUME
RELFSW0                             reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
2048                                               0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
RELFSW1                             reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
2048                                               0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
RELFSW2                             reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
2048                                               0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
LOGFILE                             reg   no   LOGVOLUME

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
1024                                               0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
LO                                  reg   yes  TEMPVOLUME

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512                                                0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA01                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
REMOTE01                            FILE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11         0   0 [ 0R  0W]
```

```
FILE                                TYPE  CMP  VOLUME
TRA02                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
TRA02                               BLOCK

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA03                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
REMOTE03                            NONE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA04                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
TRA04                               BLOCK

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA05                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
REMOTE05                            FILE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA06                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
REMOTE06                            NONE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]
```

4.4 Example: 4

In the following example, all simple files and composite main files, including additional file information, of the CP2 file system are listed. Note this example is valid only for a Multi-CP System.

```
apfpl -cp CP2 -l
CPF FILE TABLE

FILE                                TYPE  CMP  VOLUME
RELFSW0                             reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
2048                                               0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
RELFSW1                             reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
2048                                               0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
RELFSW2                             reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
2048                                               0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
LOGFILE                             reg   no   LOGVOLUME

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
1024                                               0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
LO                                  reg   yes  TEMPVOLUME

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512                                                0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA01                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
REMOTE01                            FILE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                           11          0   0 [ 0R  0W]
```

```
FILE                                TYPE  CMP  VOLUME
TRA02                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
TRA02                               BLOCK

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA03                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
REMOTE03                            NONE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA04                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
TRA04                               BLOCK

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA05                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
REMOTE05                            FILE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]

FILE                                TYPE  CMP  VOLUME
TRA06                               inf   yes  TRANSFER

TRANSFER QUEUE                      MODE
REMOTE06                            NONE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
512      100                            11           0   0 [ 0R  0W]
```

4.5 Example: 5

In the following example, all subfiles of composite file RELFSW0 are listed. Option -l is used for a long listing and option -k for compression status.

```
apfpl -s -l -k RELFSW0
CPF FILE TABLE

FILE                                TYPE  CMP  VOLUME
RELFSW0                             reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
COMPRESSED
   1024                                             0   0 [ 0R  0W]
yes

SUBFILES                                    SIZE  USERS
COMPRESSED
RELFSW0-R0                                  1024   0 [ 0R  0W]   yes
RELFSW0-R1                                  1024   0 [ 0R  0W]   yes
RELFSW0-R2                                  1024   0 [ 0R  0W]   yes
RELFSW0-R3                                  1024   0 [ 0R  0W]   yes
RELFSW0-R4                                  1024   0 [ 0R  0W]   yes
RELFSW0-R5                                  1024   0 [ 0R  0W]   yes
```

4.6 Example: 6

In the following example, all subfiles of composite file RELFSW0 , that is located in the Cluster file system are listed. Option -l is used for a long listing and option -k for compression status. Note this example is valid only for a Multi-CP System.

```
apfpl -cp CLUSTER -s -l -k RELFSW0
CPF FILE TABLE

FILE                                TYPE  CMP  VOLUME
RELFSW0                             reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
COMPRESSED
   1024                                             0   0 [ 0R  0W]
yes

SUBFILES                                    SIZE  USERS
COMPRESSED
RELFSW0-R0                                  1024   0 [ 0R  0W]   yes
RELFSW0-R1                                  1024   0 [ 0R  0W]   yes
RELFSW0-R2                                  1024   0 [ 0R  0W]   yes
RELFSW0-R3                                  1024   0 [ 0R  0W]   yes
RELFSW0-R4                                  1024   0 [ 0R  0W]   yes
RELFSW0-R5                                  1024   0 [ 0R  0W]   yes
```

4.7 Example: 7

In the following example, all subfiles of composite file RELFSW0 are listed. The headings are suppressed.

```
apfpl -s -q RELFSW0
RELFSW0                                reg    yes   RELVOLUMSW
RELFSW0-R0
RELFSW0-R1
RELFSW0-R2
RELFSW0-R3
RELFSW0-R4
RELFSW0-R5
```

4.8 Example: 8

In the following example, all subfiles of composite file RELFSW0 , that is located in the CP4 file system, are listed. The headings are suppressed. Note this example is valid only for a Multi-CP System.

```
apfpl -cp CP4 -s -q RELFSW0
RELFSW0                                reg    yes   RELVOLUMSW
RELFSW0-R0
RELFSW0-R1
RELFSW0-R2
RELFSW0-R3
RELFSW0-R4
RELFSW0-R5
```

4.9 Example: 9

In the following example, the attributes of subfile RELFSW0-R0 with additional information are listed. Option -k is used to list compression status.

```
apfpl -l -k RELFSW0-R0
CPF FILE TABLE

FILE                                   TYPE   CMP   VOLUME
RELFSW0-R0                             reg    yes   RELVOLUMSW

TRANSFER QUEUE                         MODE

RLENGTH   MAXSIZE   MAXTIME   REL      ACTIVE         SIZE   USERS
COMPRESSED
2048                                                     0    0 [ 0R   0W]
yes
```

4.10 Example: 10

In the following example, the attributes of subfile RELFSW0-R0 with additional information are listed. Option -k is used to list compression status. CP1 is the CP searched for the subfile RELFSW0-R0. Note this example is valid only for a Multi-CP System.

```
apfpl -cp CP1 -l -k RELFSW0-R0
CPF FILE TABLE

FILE                                TYPE  CMP  VOLUME
RELFSW0-R0                          reg   yes  RELVOLUMSW

TRANSFER QUEUE                      MODE

RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
COMPRESSED
2048                                                 0   0 [ 0R  0W]
yes
```

4.11 Example: 11

In the following example, the physical path to the main file of the infinite file YATSI00 is listed. Note this example is valid only for APG40 environment.

```
apfpl -p YATSI00
CPF FILE TABLE

FILE                                PATH
YATSI00                             L:\FMS\data\CPF\TEMPVOLUME\YATSI00
```

4.12 Example: 12

In the following example, the physical path to the main file of the infinite file YATSI00 is listed. Note this example is valid only for APG43 in a Single-CP System.

```
apfpl -p YATSI00
CPF FILE TABLE

FILE                                PATH
YATSI00                             K:\FMS\data\CPF\TEMPVOLUME\YATSI00
```

4.13 Example: 13

In the following example, the physical path to the main file of the infinite file YATSI00 is listed. CP1 is the CP searched for the infinite file YATSI00. Note this example is valid only for a Multi-CP System.

```
apfpl -cp CP1 -p YATSI00
CPF FILE TABLE

FILE                                    PATH
YATSI00
K:\FMS\data\CP1\CPF\TEMPVOLUME\YATSI00
```

4.14 Example: 14

In the following example, the physical path to the simple file SIMPLE is listed. Note that this example applies to APG40 only.

```
apfpl -p SIMPLE
CPF FILE TABLE

FILE                                    PATH
SIMPLE                                  L:\FMS\data\CPF\TEMPVOLUME\SIMPLE
```

4.15 Example: 15

In the following example, the physical path to the simple file SIMPLE is listed. Note this example is valid only for APG43 in a Single-CP System only.

```
apfpl -p SIMPLE
CPF FILE TABLE

FILE                                    PATH
SIMPLE                                  K:\FMS\data\CPF\TEMPVOLUME\SIMPLE
```

4.16 Example: 16

In the following example, the physical path to the simple file SIMPLE is listed. CP1 is the CP searched for the simple file SIMPLE. Note this example is valid only for a Multi-CP System only.

```
apfpl -cp CP1 -p SIMPLE
CPF FILE TABLE

FILE                                    PATH
SIMPLE
K:\FMS\data\CP1\CPF\TEMPVOLUME\SIMPLE
```

4.17 Example: 17

In the following example, the physical path to the subfile COMPO-FILE1 is listed. Note this example is valid only for APG40.

```
apfpl -p COMPO–FILE1
CPF FILE TABLE

FILE                                    PATH
COMPO–FILE1
L:\FMS\data\CPF\COMPVOL\COMPO\FILE1
```

4.18 Example: 18

In the following example, the physical path to the subfile COMPO-FILE1 is listed. Note this example is valid only for APG43 in a Single-CP System only.

```
apfpl -p COMPO–FILE1
CPF FILE TABLE

FILE                                    PATH
COMPO–FILE1
K:\FMS\data\CPF\COMPVOL\COMPO\FILE1
```

4.19 Example: 19

In the following example, the physical path to the subfile COMPO-FILE1 is listed. CP1 is the CP searched for the subfile COMPO-FILE1. This example is valid only for a Multi-CP System only.

```
apfpl -cp CP1 -p COMPO–FILE1
CPF FILE TABLE

FILE                                    PATH
COMPO–FILE1
K:\FMS\data\CP1\CPF\COMPVOL\COMPO\FILE1
```

5 Printouts

5.1 Diagnostics

```
Exit code: 0
```

Indicates a successfully executed command, and should return the user prompt.

```
Configuration Service cannot fulfill the request
Exit code: 56
```

The block Configuration Service (CS) cannot fulfill the request because an error occurs.

```
CP is not defined
Exit code: 118
```

CP does not exist in current environment.

```
File was not found: <CP file name>
Exit code: 23
```

The file was not found in the CP file system.

```
General fault
Exit code: 1
```

Error when executing.

```
Illegal option in this system configuration
Exit code: 116
```

Option is not supported in the current environment.

```
Incorrect usage
Exit code: 2
```

The command was incorrectly specified.

```
Internal program fault: <detailed information>
Exit code: 43
```

A program error has occurred.

```
Invalid file name
Exit code: 18
```

The file name is not correct.

```
Not a composite main file: <CP file name>
Exit code: 32
```

The requested operation is only allowed for a composite main file.

```
Physical file error: <operating system message>
Exit code: 41
```

An error has occurred in the physical file system.

```
Unable to connect to Configuration Service
Exit code: 55
```

The requested CS server is not responding.

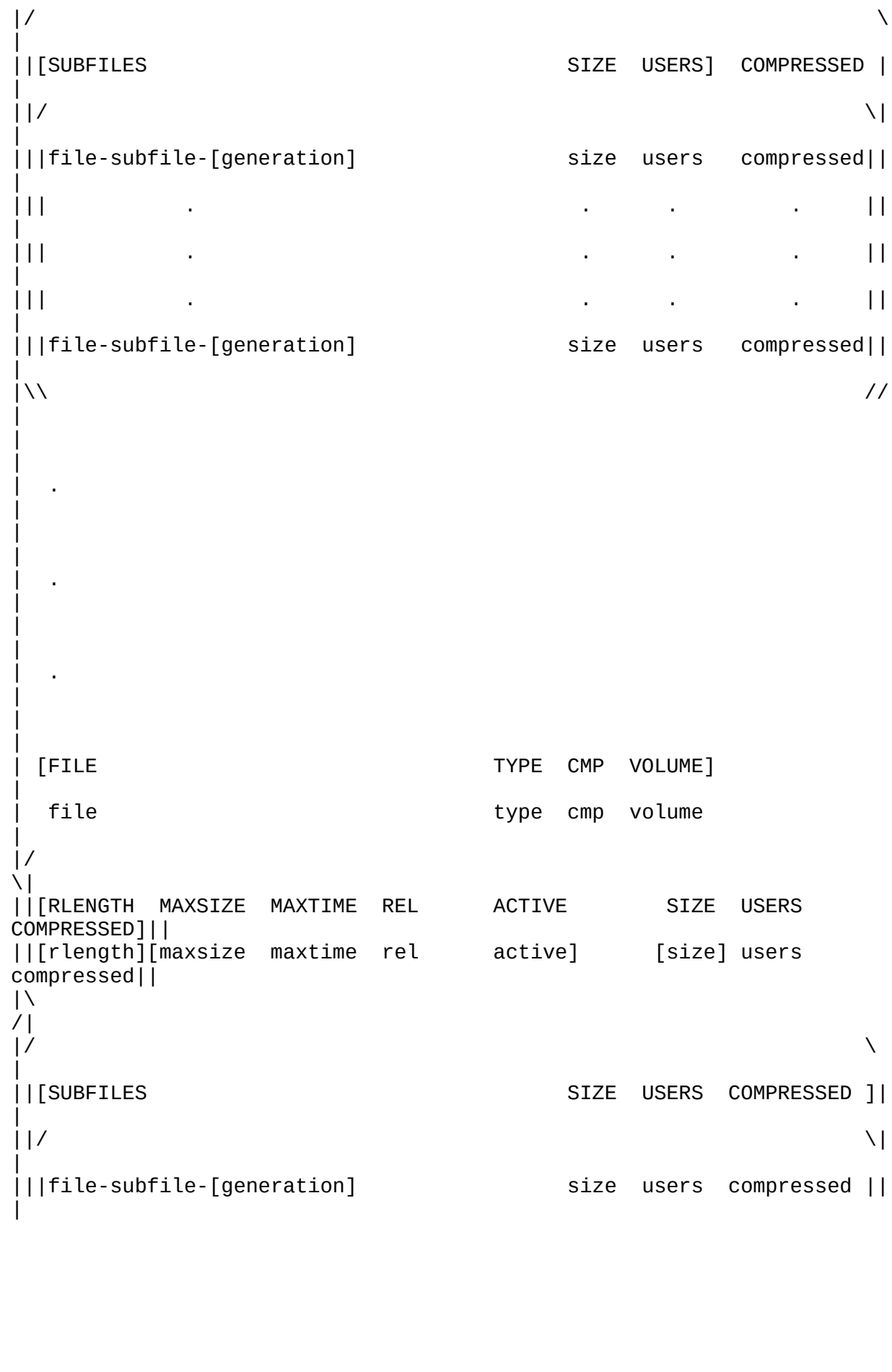
```
Unable to connect to server
Exit code: 117
```

Request is unable to connect to server.

5.2 Answer Printouts

5.2.1 Printout Format

```
Format I: -l and -s options are not used.
[CPF FILE TABLE]
/                                                    \
|[FILE                                TYPE  CMP  VOLUME]|
|/                                                    \|
||file                                type  cmp  volume||
|| .                                   .     .     .   ||
|| .                                   .     .     .   ||
|| .                                   .     .     .   ||
||file                                type  cmp  volume||
\\                                                    //
Format II: At least one of the -l and -k or -s options are used.
[CPF FILE TABLE]
/
\
| [FILE                                 TYPE  CMP  VOLUME]
|
|  file                                 type  cmp  volume
|
|
|
| [TRANSFER QUEUE                       MODE]
|
|  transferqueue                        mode
|
|/
\|
|||[RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS]
COMPRESSED||
|||[rlength][maxsize  maxtime  rel      active]      [size] users
compressed||
|\
/|
```


[SUBFILES SIZE USERS] COMPRESSED
file-subfile-[generation] size users compressed
file-subfile-[generation] size users compressed
[FILE TYPE CMP VOLUME]
file type cmp volume
[RLENGTH MAXSIZE MAXTIME REL ACTIVE SIZE USERS COMPRESSED]
[rlength] [maxsize maxtime rel active] [size] users compressed
[SUBFILES SIZE USERS COMPRESSED]
file-subfile-[generation] size users compressed

```
|||           .                                .      .        .      ||
|
|||           .                                .      .        .      ||
|
|||           .                                .      .        .      ||
|
|||file-subfile-[generation]                  size  users             ||
|
|\\                                                                   //
|
\
/
Format III: Listing of a subfile.
[CPF FILE TABLE]
[FILE                                TYPE  CMP  VOLUME
]
 file-subfile[-generation]           type  cmp  volume
/
\
|[RLENGTH  MAXSIZE  MAXTIME  REL      ACTIVE        SIZE  USERS
COMPRESSED]|
|[rlength]                                        [size] users
compressed |
\
/
Format IV: Listing of a path to a specific file.
CPF FILE TABLE
FILE                                 PATH
file[-subfile]                       path
```

5.2.2 Printout Parameters

active
: Active subfile
This parameter indicates an active subfile. The parameter is only printed for infinite files.

cmp
: File class
This parameter indicates whether the file is simple or composite.

The value is one of the following:

yes
: The file is composite.

no
: The file is simple.

compressed
: Compression status
This parameter indicates if compression is set for a file.

The value is one of following:

yes
Compression is set on the file.
no
Compression is not set on the file.

file
CP file name

maxsize
Maximum size of a subfile
This parameter indicates the maximum size of a subfile belonging to an infinite file, expressed in records. If no is printed, no maximum size is defined. The parameter is only printed for infinite files.

maxtime
Maximum active time of a subfile
This parameter indicates the maximum active time of a subfile belonging to an infinite file, in minutes. If no is printed, no maximum active time is defined. The parameter is only printed for infinitefiles.

mode
Transfer mode
The value is one of the following:
BLOCK
Block mode
FILE
File mode
NONE
Inhibit mode

path
Physical file path
This parameter indicates the path to the file in the physical file system.

rel
Release condition
This parameter indicates whether the active subfile will be changed when the file closed. The parameter is only printed for infinite files.

The value is one of the following:

yes
The subfile belonging to an infinite file will be changed when the user closes the file.
no
The subfile belonging to an infinite file will not be changed when the user closes the file.

rlength
Record length
This parameter indicates the record length, in octets.

size
File size
This parameter indicates the size of the file, expressed in records. The parameter is only printed for simple files and subfiles.

transferqueue
Transfer queue name
Name of the defined transfer queue for the file.

type
File type

The value is one of the following:
reg
Regular file
inf
Infinite file

users
Number of file users
This parameter has the format u [rR wW] where u is the total number of users, r is the number of users with read access R andw is the number of users with write access W. If X is displayed instead of the number of users, the file is opened for exclusive read or write access. Note that the same user may open the file for both read and write access.

volume
Volume name
This parameter indicates the volume where the file is stored.

6 Files

No file information is applicable to this command.

7 Command Owner

CPF

8 Glossary

APG
Adjunct Processor Group

CP
Central Processor

CS
Configuration Service

9 Reference List

This document has no references.

TRITA-ICT-EX-2012:55

www.kth.se