UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica
XXVIII Ciclo

A Model-driven Approach for the Automatic Generation of System-Level Test Cases

Author:
Ugo GENTILE

Supervisor:
Prof. Valeria VITTORINI
Prof. Nicola MAZZOCCA
Prof. Stefano MARRONE
Coordinator:
Prof.Francesco GAROFALO

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Electrical Engineering and Information Technology

March, 2016

This page is intentionally left blank

*"A 'passing' test doesn't mean 'no problem.' It means no problem *observed*. This time. With these inputs. So far. On my machine. "*

M. Bolton

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Abstract

Department of Electrical Engineering and Information Technology

Doctor of Philosophy

A Model-driven Approach for the Automatic Generation of System-Level Test Cases

by Ugo GENTILE

Systems at the basis of the modern society, as the as the homeland security, the environment protection, the public and private transportations, the healthcare or the energy supply depend on the correct functioning of one or more embedded systems. In several cases, such systems shall be considered critical, since the consequences of their failures may result in economic losses, damages to the environment or even injuries to human life. Possible disastrous consequences of embedded critical systems, suggest that discover flaws during systems development and avoid their propagation to the system execution, is a crucial task. In fact, most of the failures found during the usage of embedded critical systems, is due to errors introduced during early stages of the system development. Thus, it is desiderable to start Verification and Validation (V&V) activities during early stages of a system life cycle. However such V&V activities can account over the 50% of times and costs of a system life cycle and there is therefore the need to introduce techniques able to reduce the accounted resources without losses in term efficiency. Among the methodologies found in scientific and industrial literature there is a large interest in the V&V automation.

In particular, automatic verification can be performed during different stages of a system development life cycle and can assume different meanings. In this thesis, the focus is on the automation of the test cases generation phase performed at the System level starting from System Under Test (SUT) and test specifications.

A recent research trend, related to this, is to support such process providing a flexible tool chain allowing for effective Model Driven Engineering (MDE) approaches [82]. The adoption of a model-driven techniques requires the modelling of the SUT to drive the generation process, by using suitable domain-specific modelling languages and model transformations. Thus, a successful application of the MDE principles is related to the choice of the high-level language for SUT specification and the tools and techniques provided to support the V&V processes.

According to this, the model-driven approach define in this thesis relies on three key factors: (1) the definition of new domain-specific modelling languages (DSMLs) for the SUT and the test specifications, (2) the adoption of model checking techniques to realize the generation of the test cases and (3) the implementation of a concrete framework providing a complete tool chain supporting the automation process.

This work is partially involved in an ARTEMIS European project CRYSTAL (CRitical sYSTem engineering AcceLeration) [23]. CRYSTAL is strongly industry-oriented and aims at achieving technical innovation by a user-driven approach based on the idea to apply engineering methods to industrially relevant Use Cases from the automotive, aerospace, rail and health-care sectors. The DSML that will be presented in this thesis, emerged as an attempt to address the modelling requirements and the design practices of the industrial partners of the project, within a rigorous and well-founded formal specification and verification approach.

In fact, the main requirement that a modelling language suitable for the industry should have is to be small and as simple as possible [43]. Thus, the modelling language should provide an adequate set of primitive constructs to allow for a natural modelling of the system of interest. Furthermore, the larger the gap between the design specification and the actual implementation is, the less useful the results of the design analysis would be. The test case generation is supported by model checking techniques; the SUT and test models are in fact translated in specifications expressed by the language adopted by a model checker. The thesis discusses all the issues addressed in the mapping process and provides their implementations by means of model transformations. A class of test specifications is addressed to exemplify the generation process over a common class of reachability requirements. The model-driven approach discussed in the thesis is applied in the contest of the railway control systems, and in particular on some of the key functionalities of the Radio Block Center, the main component of the

ERTMS/ETCS standards for the interoperability of the railway control systems in the European Community.

The thesis is organized as follows. The first chapter introduces embedded critical systems and outlines the main research trends related to their V&V process. The Chapter 2 outlines the state of the art in testing automation with a particular focus on model-driven approaches for automatic test generation. The same Chapter 2 provides also the necessary technical background supporting to understand the development process of the supporting framework. The Chapter 3 describes the context of the CRYSTAL project and the proposed model-driven approach partially involved in its activities. The Chapter 4 describes the domain-specific modelling languages defined for the modelling of the SUT specifications and of the test generation outcomes. Moreover the guidelines defined for modelling test specifications are discussed. The Chapter 5 focuses on the mapping process that enable the translation of the high-level language for the modelling of the SUT specification to the language adopted by the chosen model checker. The implementation of the overall framework is addressed in Chapter 6. Here model transformations realizing the defined mappings and the architecture of the Test Case Generator (TCG) framework are described and discussed. The Chapter 7 shows the results of the application of the approach in the context of the railway control systems and in particular to the Radio Block Centre system, a key component in the ERTMS/ETCS standard. Chapter 8 end the thesis, giving some conclusive remarks.

This thesis includes materials from some research papers, reported in the bibliography, which are already published in peer-reviewed conferences and journals.

Acknowledgements

The research activities leading to these thesis results have partially supported by the following projects:

- The research project CRYSTAL (Critical System Engineering Acceleration), funded from the ARTEMIS Joint Undertaking under grant agreement n. 332830.
- The research project KNOWLEDGE - *Nuovi paradigmi e tecnologie per la collective knowledge*, POR CAMPANIA FSE 2007-2013, 06 DD 414-13/11/09, CUP B25B09000020007 Asse IV Asse V.

Keywords: Model-driven, Automatic generation of test cases, Model Checking, Domain-specific modelling language, Model transformations,Railway Systems

Contents

List of Figures

List of Tables

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
AST	**A**bstract **S**yntax **T**ree
ATL	**A**tlas **T**ransformation **L**anguage
BNF	**B**ackus-**N**aur **F**orm
CAD	**C**omputer-**A**ided **D**esign
CFG	**C**ontrol **F**low **G**raph
CHM	**C**ummunicating **H**ierarchical state **M**achine
CP	**C**ommunication **P**rocedure
CRYSTAL	**CR**itical **SYST**em Engineering **A**cce**L**eration
CST	**C**oncrete **S**yntax **T**ree
CTC	**C**entralized **T**raffic **C**ontrol
CTL	**C**ontrol **T**ree **L**ogic
DSML	**D**omain **S**pecific **M**odelling **L**anguage
DSTM	**D**ynamic **ST**ate **M**achine
EBNF	**E**xtended **B**ackus-**N**aur **F**orm
EMF	**E**clipse **M**odelling **F**ramework
ERTMS	**E**uropean **R**ail **T**raffic **M**anagement **S**ystem
ETCS	**E**uropean **T**rain **C**ontrol **S**ystem
ETL	**E**psilon **T**ransformation **L**anguage
EU	**E**uropean **U**nion
FSM	**F**inite **S**tate **M**achine
GMF	**G**raphical **M**odelling **F**ramework
GSM-R	**G**lobal **S**ystem for **M**obile Communications – **R**ailway
GUI	**G**raphical **U**ser **I**nterface
ICT	**I**nformation and **C**ommunication **T**echnology
IEC	**I**nternational **E**lectrotechnical **C**ommission
IOS	**I**nter**O**perable **S**pecification
ISO	**I**nternational **O**rganization for **S**tandardization
LTL	**L**inear **T**emporal **L**ogic
M2M	**M**odel-to-**M**odel
M2T	**M**odel-to-**T**ext
MA	**M**ovement **A**uthority
MARTE	**M**odeling and **A**nalysis of **R**eal-**T**ime and **E**mbedded systems)
MB	**M**odel **B**ased
MBSE	**M**odel **B**ased **S**ystem **E**ngineering
MBT	**M**odel **B**ased **T**esting
MD	**M**odel **D**riven
MDD	**M**odel **D**riven **D**evelopment
MDE	**M**odel **D**riven **E**ngineering
MDT	**M**odel **D**riven **T**esting

MOFM2T	**MOF M**odel to **T**ext **T**ransformation Language
OCL	**O**bject **C**onstraint **L**anguage
OSLC	**O**pen **S**ervices for **L**ifecycle **C**ollaboration
PIM	**P**latform **I**ndependent **M**odel
PIT	**P**latform **I**ndependent **T**est
PSM	**P**latform **S**pecific **M**odel
PST	**P**latform **S**pecific **T**est
PROMELA	**Pro**cess/Protocol **Me**ta **La**nguage
QVT	**Q**uery/**V**iew/**T**ransformation
RAMS	**R**eliability, **A**vailability, **M**aintainability, and **S**afety
RBC	**R**adio **B**lock **S**ystem
RCP	**R**ich **C**lient **P**latform
REST	**Re**presentational **S**tate **T**ransfer
RIS	**R**ailway **I**nfrastructure **S**ystem
RSM	**R**ecursive **S**tate **M**achine
RTP	**R**eference **T**echnology **P**latform
SIL	**S**afety **I**ntegrity **L**evel
TCG	**T**est **C**ase **G**enerator
TESQEL	**TES**t **S**e**Q**uence **L**anguage
TSI	**T**echnical **S**pecification for **I**nteroperability
TSP	**T**est **S**pecification **P**attern
TTCN-3	**T**esting and **T**est **C**ontrol **N**otation version **3**
UES	**U**nconditional **E**mergency **S**top
UML	**U**nified **M**odelling **L**anguage
UNISIG	**UN**ion **I**ndustry of **Sig**nalling
UTP	**U**ML **T**esting **P**rofile
V&V	**V**erification and **V**alidation
WS	**W**eb **S**ervice
XMI	**X**ML **M**etadata **I**nterchange
XML	e**X**tensible **M**arkup **L**anguage

To my mother and my father,
to my sister,
to my love Eleonora

Chapter 1

V&V processes in Embedded Critical Systems

The technical and technological growth of Information and Communication Technology (ICT) has been resulted in a capillary diffusion of computer-based systems. A computer-based systems is a system in which some (or even all) parts depend on the correct functioning of computers.

Computer-based systems play an important role within the modern society. There are in fact some essential services as the homeland security, the environment protection, public and private transportations, communications, public health and energy supply depending on the appropriate functioning of such systems.

Nowadays, there is the trend to combine hardware and computer-based systems and allow them to interact with the surrounding environment through sensors and actuators, in order to perform a specific task. Such approach is widely adopted within some critical contexts as the transportation (e.g. automotive, aerospace and railway), the healthcare, the energetic (e.g. smart grids, nuclear plants) and the economic (e.g. transactional systems of banks). Such kind of systems, are identified as *embedded critical system* and are at the basis of a large amount of research interest, both from the academician and the industrial contexts. According to [63] embedded systems can be defined as information processing systems embedded into a larger product; when an embedded systems is involved in critical contexts as the ones above mentioned, the become critical embedded systems.

Critical embedded systems are therefore systems in which a failure may result in economic losses, damages to the environment of even injuries to human life [86].

Dealing with such critical systems, means that the most important property to guarantee is the *dependability*. The definition of dependability was proposed

by [60] as a general concept able to encompass other relevant non-functional attributes as availability, reliability, safety and security. However dependability is not only an umbrella concept that contains several non-functional properties but it has long held other means, that include also concepts related to the functional requirements of systems. In fact dependability is often perceived as the degree of trust that users have with the respect to the considered system. There are several reasons supporting this meaning of dependability:

- Systems that are not dependable, are perceived by users as unreliable, unsafe and insecure and are often rejected with an obvious loss of time and money;
- Non-dependable systems may cause a loss of information that are often more important that the systems itself;
- Failures to system dependability, in general, affect a large amount of users and may cause damages to other business application.

These reasons allow to understand that, guaranteeing dependability is not a merely process of assure non-functional properties. In fact, and in particular for critical systems, the assurance of the dependability, can be achieved by taking in to account both functional and non-functional requirements. Next section, will exploit characteristics of embedded critical systems, by classify them, from the point of view of their mission and from the perspective of the failure consequences.

1.1 Classifying embedded critical systems

The expression *Critical system* is a general concept that gives an idea of the problems that may rise during system life cycle. However is worth to have a classification able to better collocate the system under development in a precise category, in order to drive the effort to be spent in design and development and understand the aim of the development itself. Academic and industrial literature is rich of taxonomies related to this and each of them focus on a specific aspect of critical systems. Following taxonomies, allow to understand why is important to consider functional requirements when addressing critical computer-based systems development.

In order to introduce the above mentioned taxonomies, the concept of failure should be clarified. A Failure is the result of system errors that derives from faults in the system. More precisely, it has to be distinguished between:

- *failure*: an event that occurs at some point in time when the system does not deliver a service as expected by its users;
- *error*: an erroneous system state that can lead to system behaviour that is unexpected by system users.;
- *fault*: a characteristic of a software system that can lead to a system error. For example, failure to initialise a variable could lead to that variable having the wrong value when it is used.

Strictly related to the failure consequences, critical systems can be classified in:

- *Business-critical systems*, in which a failure implies economic losses. Examples of business-critical systems are the transactional ones adopted by a Bank or by an airline reservation system;
- *Mission-critical systems* in which a failure may damage some goal-directed activities. An example of a mission-critical system is a power grid.
- *Safety-critical systems* in which a failure may result in injury, loss of life or serious environmental damages. An example of a safety-critical system is a railway or aircraft control system.

The taxonomy above helps in understand requirements that a critical system should be able to satisfy. However, several systems may fall in more then one category. For example, a railway control system is a *mission-critical* system, due to the goal of the provided service, is a *safety-critical* system, because a failure, can expose the system to intentional attacks, that can injury human life, and finally is *business-critical* since each disruption of the service, can result in a loss of money for the provider of the service. Nevertheless, is clear that the development will be guided by the adopted perspective. In fact, when dealing with a mission-critical systems, might be more relevant to take into account non-functional requirements (e.g the availability for a transactional bank system). At the contrary, in a safety-critical systems, functional properties are more important to address. In fact, in case a failure occurs on a railway control system, is worth that the train stops its run until recovery procedures restore the same control system. According to such perspective, embedded critical systems can be classified in:

- *Fail-Safe Systems(FSS)*: systems that lead them self to a safe state, when under failure. Several medical system fall in this category. They in fact avoid that a failure to the system may harm the patient.

- *Fail-Operational Systems(FOS)*: systems that are able to perform their mission, even in case of failure of their control system. An example is the auto-pilot of an aircraft, that, in case of failure, does not compromise the steering of the vehicle.

Disastrous consequences of failures in embedded safety-critical systems, suggest that, discover flaws during system development and avoid their propagation to the system execution phase, is a crucial task. Thus, V&V activities are relevant phases of the a system life cycle and should be executed as soon as possible since most of the failures discovered in embedded critical systems, are introduced during early stage of the development. Furthermore, it has been estimated that the V&V process accounts over the 50% of the total development time and costs.

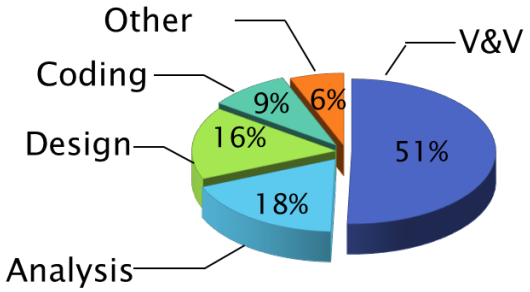


FIGURE 1.1: V&V process accounted costs

Thus, the critical context suggests to adopt well-tried methods and techniques for the V&V process even if it has been proven that several approaches for non-critical systems, as formal methods, have been successful adopted in the development of such systems, since they are able to reduce and optimize the resources involved in the systems development. Definitively, despite the well-know sentence of Dijkstra in 1969 that states that *Testing can show the presence, not the absence of bugs*, V&V activities are the best way to address the safety of a critical embedded system. Next sections provide a deep overview of the V&V process, specially with the respect to the practices commonly adopted within industrial contexts.

1.2 The meaning of V&V

The concepts of *verification* and *validation* are usually considered together, even if relevant differences occur between them. In particular, the main difference

is related to the aim which drives these two phases. Boehm [13] proposes two questions that also nowadays clarify perfectly the difference between *verification* and *validation*:

- *verification* answers to the question "Are we building the product *right*?", that means that verification deals with the proof that a system conforms to its requirements;
- *validation*, instead, deals with the question "Are we building the *right* product?", that means that is finalized to discover if the system works correctly with the respect to its specifications (or in other words with users expectations).

With the respect to the existing and consolidated approaches found in literature, is possible to divide V&V activities in two macro-category: the static-based and the dynamic-based. Static-based approaches focus on inspections and reviews of artefacts produced during the system development and their premise is that the system under test is not put into use. Static-based approaches allow to discover flaws in system design and in general defects even when the considered system is not completed. Furthermore costs accounted for static-approaches are very low.

The main limitation of static-based approaches is that they are not able to discover defects that may arise during runtime interactions of systems components, as timing issues, performances problems or even bad management of memory space. To do this, there is the need to use dynamic-based approaches and in particular testing. Dynamic-based approaches, in fact, are able to proof that a system behaves as expected, or that it respects its functional requirements that is very important for embedded safety-critical system, since the discussed consequences of unexpected behaviours.

Testing is in fact one of the most relevant tasks of V&V activities and can be performed during different stages of system life cycle. According to this, it is possible to distinguish between:

1. *Development testing*, whose objective is to discover defects, during the system development;
2. *Release testing*, in which a separated team performs system testing before the release to the users;
3. *User testing*, in which the system is put into use, often for a selected and limited set of user-type.

Development testing is very important, according to the recommendations provided by international standards as the ISO/IEC 61508 described in Section 1.3.1. In fact, with development testing is possible to discover system defects during early stages of development, or in other words, before the system starts to interact with human life and environment. Development testing, can be conducted with different levels of granularity. In fact, it can be performed as:

- Unit testing, where single module, object or class is tested. Unit testing, normally, is performed in order to test single functionalities provided by the system;
- Component testing, during which, units are composed in order to test their integration and components interfaces;
- System testing, where some or all of the components in a system are integrated. Thus, the system is tested as a whole. Goals of system testing are usually the testing of component interactions and compliance with specifications.

1.3 V&V of embedded safety-critical system in industrial practice

Identify system hazards as early as possible and try to reduce them to an acceptable level, is the best way to follow in order to address safety issues. However, well-tried techniques, adopted during a system life cycle, are not able to cope with the complexity of nowadays systems and technologies. The inadequacy reflects on the fact that the adoption of these methods do not guarantee the proper level of safety of system under development that is instead necessary even for embedded safety-critical system. The safety issue has been addressed and discussed for a long and created the branch of system safety engineering. System safety engineering deals with the identification of systems hazards, the determination of their causes, the development of proper countermeasures and, once applied, the verification of the strategies effectiveness. With the respect to industrial development, a common practice is to adequate the system life cycle to a accepted reference standard, in order to achieve an high level of quality, that increase the system trustiness. Standards can be defined as common languages that not only engineers but also industries, companies, and countries can use to try to ensure a good, high-quality product [8]. The adoption of standards in embedded safety-critical system is necessary in order to control the behaviour of the

system during its whole life cycle and to estimate in time, and with minor costs and times possible, architectural choices, functional and non-functional properties and consequences of possible failures.

1.3.1 Reference standards for embedded safety-critical systems

According to the above mentioned scenarios, there are several national governments, public and private agencies, industries and international standardization organizations which collected and promulgated directives to follow in order to improve the safety of embedded critical-systems. The International Electrotechnical Commission (IEC) and the International Organization for Standardization (ISO) are two of the most productive international entities in the production of such standards. The IEC, in particular, promulgates the IEC/EN 61508 that is one of the most important and diffused standards, at the basis of several directives for embedded critical systems as the *ISO/FDIS 26262* for the automotive domain, the IEC/EN CENELEC 50126 focused on railway control systems, the *IEC 61513* for computer-based system in nuclear plants.

In order to better understand the relevance of these standards, the meaning of *safety* to be addressed, has to be clarified. In fact the term *safety* may assume different meanings: for example, addressing safety in electrical contexts means that is introduced a proper mechanism able to reduce the risk of electrocutions during the usage of electrical devices. In the case of this dissertation, the safety addressed is the *Functional Safety*. Functional safety is a concept that can be applied to a wide amount of industrial domains. It in fact provides the assurance that the safety-related systems will offer the necessary risk reduction required to achieve safety for the considered equipments. This concept was introduced by IEC/EN 61508 as a consequence of the increasing diffusion of the electrical, electronic or programmable electronic systems also within critical contexts.

A more formal definition, is the one presented in the ISO/IEC 61508 [53]. Given the following definitions:

- Equipment under control (EUC): equipment, machinery, apparatus or plant used for manufacturing, process, transportation, medical or other activities.

- EUC control system: system which responds to input signals from the process and/or from an operator and generates output signals causing the EUC to operate in the desired manner.

- EUC risk: risk arising from the EUC or its interaction with the EUC control system, i.e. the risk associated with functional safety.
- Electrical/electronic/programmable electronic system (E/E/PE): as for PES.
- Programmable electronic system (PES): system for control, protection or monitoring based on one or more programmable electronic devices, including all elements of the system such as power supplies, sensors and other input devices, data highways and other communication paths, and actuators and other output devices.

The functional safety is the part of the overall safety relating to the EUC and the EUC control system which depends on the correct functioning of the E/E/PE safety-related systems, to other technology safety-related systems and to external risk reduction facilities.

Furthermore the ISO/IEC 61508 defines another relevant concept that is the SIL (Safety Integrity Level) which represents the integrity level of safety functions of a system as measurement of safety required/obtained by itself (a definition applicable to all kind of industries for both hardware and software components). It is important to underline that the SIL is related to a single safety function and not to the entire system or to its individual components: within a given system a lot of safety features will exist, each of them related to a particular hazard. To each of them an appropriate SIL should be associated. The whole set of components of each security system must respect the related SIL class. The IEC/EN 61508 does not define the SIL to achieve for the specific application domain but it lists all the operations that must be carried out, i.e. a risk analysis of the specific system and an assessment of acceptable risk as a combination of the probability and the hazard level. This standard covers the complete safety life cycle decomposed into 16 phases addressing analysis, realisation and operation of a safety critical system. The main concepts defined by this standard rely on the fact that safety must be considered from the beginning of the life cycle, that not-tolerable risks must be reduced and that zero risk can never be reached.

With the respect to this, one the crucial task of the development cycle demanded by the IEC/EN 61508 for safety critical systems is the elicitation of requirements whose purpose is the reduction of the risk. Such requirements are called *Safety Requirements*. Safety requirements, as the other requirements, can be specified at a high level and should be validated during early stages of system development. Safety related concepts are interpreted in specific application domains and can be covered by different standards. Next sections provide a brief overview of these standards, in particular related to transportation domains.

1.3.2 Aerospace domain

The avionics domain is rich of standards that regulated not only the system development itself but in particular embedded systems involved in the control and management of the aircraft. A first relevant standard is for sure the *DO-178B* - "Software Considerations in Airborne Systems and Equipment Certification" [35]. This standard, in particular, addresses safety issues related to safety-critical software that interacts or even is at the basis of some airborne systems. With the respect to the IEC 61508, it defines five levels of software criticality (from A that is the catastrophic to E that is the no effect level) in order to define the accepted failure rate for the related components. At the contrary of the DO-178B that addresses software systems, the *DO-254* - "Design assurance Guidance for Airborne Electronic" [35], deals with complex electronic hardware, involved in the functioning of airborne system, such as Field Programmable Gate Arrays (FPGA) and other special purpose devices.

1.3.3 Automotive domain

Within the automotive domain, a relevant standard is the *ISO 26262* [54] that inherits recommendations from the IEC 61508 and apply its principles to automotive processes. This standard provides some relevant characteristics: first a safety life cycle, that takes into account the system under development from its conception to the decommissioning. Second, according to the SIL concept, it defines an *Automotive Safety Integrity Level (ASIL)* that permits to identify the risk class, of each item driving the development, in order to reduce the overall risk. Furthermore, it provides the requirements for verification and validation of system under development, to ensure an acceptable level of safety.

1.3.4 Railway domain

In the railway context, one of the major standards that must be considered, is the *CENELEC 50126* (and the subsequent *CENELEC 50128 and 50129*)- "Railway Applications - The Specification and Demonstration of Dependability, Reliability, Maintainability and Safety (RAMS)" [16]. These standards apply the principles of the IEC 61508, by encouraging the interoperability among the European rail industry. The concept of interoperability means that, rail equipments developed according to the CENELEC standards, should be applied to any member state, within the European railway context. CENELEC standards recommend that possible failures and hazards have to be identified during the overall life

cycle, properly corrected or mitigated, by considering their occurrence rate and effort to spend, and finally the risk has to be evaluated. More in detail, the CENELEC 50126 describes the processes and methods that are used to specify the most essential and important aspects for operability and safety in the railway domain; the EN50128 and the EN 50129 give a set of requirements which have to be satisfied during the safety-critical software (the former) / hardware (the latter) development, deployment and maintenance phases.

The above discussed standards, show three essential elements: (i) a reference safety life cycle, (ii) a way to identify the proper risk level, in order to drive the development and (iii) the need to start verification and validation activities, with a certified process, at early stages of the system development With the respect to this, a life cycle explicitly suggested is the 'V' lifecycle (or 'V'-model) where the design is implemented during the descendent activities and V&V are performed during the ascending ones. The same standards also define the list and the minimal contents of the documents and deliverables that have to be produced during the life cycle of an embedded safety-critical system: in particular it is clear that the entire set of requirements, test cases and reports have to be traced in appropriate documents. It also imposes the clarification of the traceability between requirements and test reports in order to demonstrate how each requirement has been satisfied by outcomes of test cases.

1.3.5 The V-model lifecycle

Most of the failure during the usage of a safety-critical system start from errors introduces during early stages of system development. According to this, is worth to start the V&V process of system at early stage of its development. A model widely-adopted in literature is the one named *V-model*. The V-model life cycle was proposed as an extension of the well-know and used water fall model, for the development of software-based systems. Stages of this development model, are depicted in Figure 1.2 and detailed below:

The typical structure based on a "V" organizes development phases into levels of complexity with the most complex item on top and least complex item on bottom. This places the design next to verification in the sense that the acceptance activities are intrinsically linked to those of development.

There are several phases in the life cycle, starting from the conception to the decommissioning of a system, specifying inputs, requirements, deliverables and verification techniques of each phase. For sake of brevity, these phases can be grouped into four macro-activities, which are:

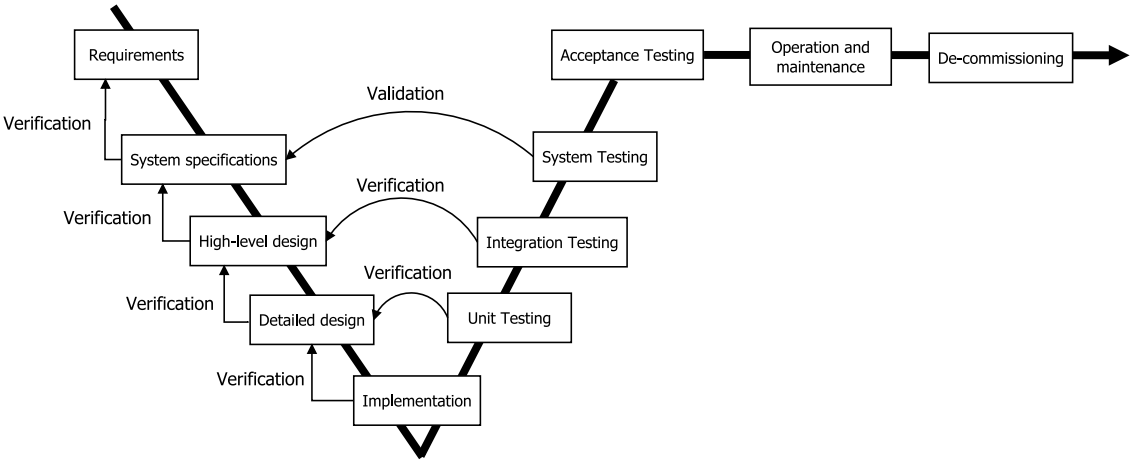


FIGURE 1.2: The V-lifecycle

- *Definition*: the objective of this phase shall be to develop a level of understanding of the system sufficient to enable all subsequent life cycle tasks to be satisfactorily performed. In this phase are defined: the mission profile, the boundaries, the application conditions influencing the characteristics of the system. At last, also in this phase, the overall requirements for the system are defined as well as the overall demonstration and acceptance criteria for the system.

- *Design*: during this phase, requirements are apportioned to all subsystems and specific acceptance criteria are defined at subsystem level; subsystems conforming requirements are hence created and their compliance with target is demonstrated through the usage of proper formal models (combinatorial as Reliability Block Diagrams, Fault Trees and also state-based models).

- *Testing*: after the realization of the subsystems, the objectives of this phase are to validate that the total combination of sub-systems, components and external risk reduction measures comply with the requirements for the system; in this phase specific characteristic are collected in order to assess compliance with provided requirements of the complete system, and hence accept the system for entry into service.

- *Operation*: the objective of this phase shall be to operate (within specified limits), maintain and support the total combination of subsystems, components and external risk reduction measures such that compliance with system requirements is maintained; the performance of the system shall be monitored in order to maintain confidence in the system.

1.4 Research trends

In this section some relevant research trends, within the context of the V&V of embedded safety-critical systems, are discussed, in order to better clarify the contribution of the thesis.

1.4.1 Testing automation

According to the industrial needs, described in Section 1.3 and with the aim to reduce time and costs, there is a diffuse trend that moves towards *Testing Automation*. The term *Testing Automation* in another general term which can be applied to different phases of testing activities, as the generation or the execution of test cases. According to the objectives of the present dissertation, test automation refers to the automatic generation of test cases. A deep discussion about testing automation and automatic generation of test case, will be provided in Chapter 2. A widely adopted approach for automatic test cases generation the Model-based testing (MBT). MBT is a testing paradigm that relies on models of a system under test and/or its environment to derive test cases for the system model [93]. A complete discussion on model-based testing process, will be provided in the Section 2.3. MBT, is often identified as a black-box testing technique, due to the fact that a model, is an abstract representation of the system behaviour. In fact, one of the main issue when dealing with MBT is the formalism to adopt for the representation of the SUT. Dias Neto et al. [27] state that in order to cope with a rigorous process and increase the quality of the software under development, there is the need to use a formal model. In fact, according to the concept that, within critical system is worth to identify issues at early stage of development, there is the need to have specification as much possible consistent with the concrete system expectations [70]. In this sense, formal methods are able to provide the necessary rigorous process for the realization of a formal and reliable system specification.

1.4.2 Formal methods

Formal methods are mathematical-based languages, techniques and tools for the specification and the verification of complex systems [**clarke**]. They can be involved in several stages of a system development. However, with the respect to the V&V activities, a strong research trend is to adopt formal methods, for the modelling of system specification in order to start as soon as possible verification activities.

Formal methods have the capability of combine low level informations with system behaviour, in order to minimize the number of errors and even in order to identify possible failures during early stages of development. Industry are interested in adoption of formal methods for two main reasons:

- first they are recommended by several widely adopted international standards (e.g. the CENELEC 50128);
- they increase the trustiness in the system, that allow for a better marketing of the system itself.

However there are some disadvantages that limit the adoption of such methods. First formal methods are based on rigorous and mathematical-based languages, that require high-specialized skills and that means greater costs for the industry. Second formal specifications are quite distant from user ones and often is difficult to monitor the system development.

Formal methods, are extremely advised because they allow for the description of both the system and of its properties, with a rigorous mathematical language, that allows for their verification. A number of formal methods and techniques have been developed by the scientific community in the past decades and applied to the development of critical systems, including railway applications [11]. In this context model checking techniques stand out. Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

Heimdahl [49] shows the potentialities of Model Checkers, in the testing automation of aircraft guidance systems. Despite the potentialities, there are some limitations in the adoption of model checking techniques:

- Model checking, as stated, is based on a rigorous mathematical formalism, and often it needs a high-qualified staff to deal with such formalism. Such implies new costs to sustain;
- there is a gap between system specification, made with a model, that leads to a diffidence against model-based techniques included model checking.

In order to simplify the adoption of formal methods, Finite State Machines (FSMs) are widely used in modelling systems where control handling aspects are predominant. Statecharts [47] extend FSMs with hierarchy, concurrency and communication among concurrent components. Hierarchy is achieved by injecting FSMs into states of other FSMs. Concurrency is achieved by composing FSMs

in parallel and by letting them run synchronously. The second limitation is usually addressed by migrating from the more general context of the Model-based testing, to another paradigm, strictly related to the previous one, that is Model-driven testing (MDT). In a model-driven testing, the model became the key artefact that drives the whole system life cycle. Applied to the testing process, it means that, the model of the system and test, drive the specification, the generation and the execution of test cases. Characteristics of model-driven will be deepened address in the next chapter.

1.4.3 Recent European projects addressing testing automation

European community has been conducted several cooperative projects aiming at the improvement and the industrialization of embedded safety-critical systems. The main objective of such projects is to integrate large enterprises, suppliers, and vendors coming from different critical domains and enable them to cooperate in order to improve and innovate research in the embedded critical system field. CESAR project [17], for example, aims to boost cost efficiency of embedded systems development, safety and certification processes by bringing significant innovations in systems engineering disciplines like requirements engineering, component-based development and by introducing a reference technology platform. SafeCer [81] aims to increase efficiency and reuse in development and certification of safety-relevant embedded systems by providing process and technologies which enable composable qualification and certification of systems/subsystems. OPENCOSS [76] was a FP7 project aiming to produce the first European-wide open safety certification platform to reduce time and costs for (re)certification of safety-critical embedded systems, in particular for the railway, avionics and automotive markets. DECOS [26] focused on the development of an architecture-based design methodology and the associated COTS hardware and software components, with certified development tools and advanced hybrid control technologies, allowing Europe's leading position in highly developed control systems in the avionics (Airbus) and automotive industries and in the area of dependability of software-intensive systems. Within the context of embedded safety critical system, V&V activities are one of the most addressed issues. MBAT [64], for example focuses on integration of formal techniques within testing activities in order to improve the efficiency of testing and the effectiveness of formal methods analysis. The PRESTO project [78] aims at improving test-based embedded systems development and validation, while considering the constraints of industrial development processes. The above list of projects,

that is not intended to be exhaustive, set up the basis for the CRYSTAL (Critical System Engineering Acceleration) project, a Joint Untertaking ARTEMIS project whose aim is to enable sustainable paths to speed up the maturation, integration, and cross-sectoral re-usability of technological and methodological bricks of the factories for safety-critical embedded systems engineering in the areas of transportations (aerospace, automotive, and rail) and healthcare providing a critical mass of European technology providers. Within the CRYSTAL project, V&V is a relevant process. CRYSTAL in fact, in order to reduce costs related to V&V activities, encourages the integration between the different steps of the same V&V process. The CRYSTAL project, will be deep described in the Chapter 3.

1.5 Thesis contribution

According to the industrial needs and to the recommendations of international standards, there is the need to automate some stages of the life cycle of embedded safety-critical systems. The original contribution of this thesis is in the definition of a model-driven framework, able to enable the automatic generation of proper formal models, starting from an high-level specification of a system behaviour. In particular, formal models described in this work are involved in the context of the V&V activities of a embedded safety-critical system life cycle and in particular in the automatic generation of test cases. To realize such framework, model-driven principles have been exploited. More in detail, in order to capture the needs of the application domain, and enable the automatic generation of test case, proper domain specific modelling languages (DSMLs) have been defined. In particular topics related to the definition of formal syntax and semantics of DSMLs have been address. Another relevant activity described in this thesis is in the definition of a structured methodology to address mapping issues related to the translation from the high-level languages defined and the languages adopted to describe the generated formal models. A concrete framework, based on model transformations realizing the defined mapping, has been realized during the thesis work. The validation of the developed framework has been performed by applying it within the context of the mass-transit transportation and in particular within the railway domain. Even if the focus is on automatic generation of test cases, the proposed Model-Driven methodology is useful also for the complete design (based on centralized information), hence it is possible to extend it for automatic verification of several non-functional properties of embedded critical systems as the survivability and the vulnerability.

Chapter 2

State of the art in automatic test case generation

The testing activities affect up to the 50% of the total costs and time of a critical system development cycle. Thus, in order to reduce this effort, academician and industrial interests have been focused on the concept of testing automation. Literature is rich of definitions addressing the this concept:

"The management and performance of test activities, to include the development and execution of test script so as to verify test requirement, using an automated test tool "

(Dustin et al.[29])

"A system that includes technologies that support automatic generation and/or execution of tests for unit, integration, functional, performance, and load testing"

(Huzinga et al.[51])

Testing automation means that tests are run and checked without manual intervention (...) you make use of a test automation framework to write and run

Sommerville [86]

This chapter is focused on well-know techniques and research trends adopted for testing automation and in particular for the automatic generation of test case. The chapter will provide the needed background to understand research activities at the basis of the present thesis. First a brief overview of the a general testing automation process is provided in order to understand which are its the key factors. Then, with the respect of the state of the art, Model-based techniques for automation of test cases generation are discussed.

2.1 The process of automation

As emerging from previous definitions, the process of testing automation, is composed by several stages taking into account different aspects of a testing process. Despite the absence of reference standards, several methodologies have been proposed in the year as an attempt to provide a systematic schema to perform testing automation. Among such proposed methodologies, one of the most famous it the *Automated Test Lifecycle Methdology (ATLM)* proposed by [29]. The ATLM, depicted in Figure 2.1, shows that, the automation process is anything but easy, due to the presence of several stages to address:

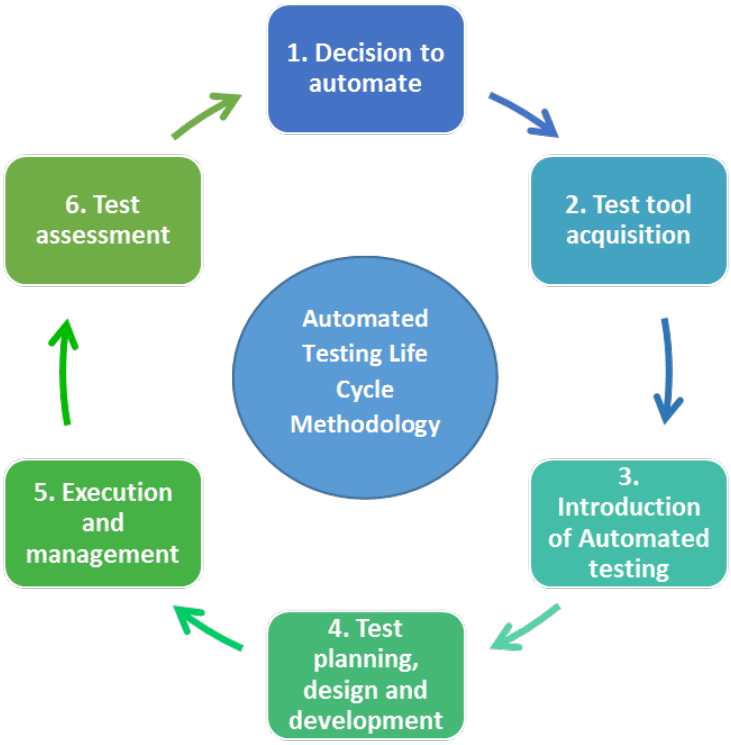


FIGURE 2.1: The ATLM process ([29])

1. *Decision to automate*: the first step might seem obvious but in some cases it may not be necessary to introduce testing automation. This step, in fact, should help the team to outlines the potential benefits of automation and the effort to spent in an automated test development; There are several possible reasons that motivate the introduction of testing automation:

    - Manual testing is a laborious task, that can lead to human errors;
    - Automation allows to better verify that the SUT meets its requirements;

- Automation reduces the presence of defects and failures of the developed system;

2. *Test tool acquisition*: during this step, the software team, can decide to acquire an existing automation tool suite, or develop it from scratch. The outcomes of the step varying on the basis of the choice. In the first case, a set of criteria to drive the choice have to be defined, and the tool suite vendor has to be contacted. In the second case, this step can be assimilated to the design phase of a software development cycle.

3. *Introduction of Automated testing*: this phase is focused on the process that allows to integrate the automation strategy within the considered software system. This phase can be divided in two sub-steps:

   - Test Process Analysis, which ensures that the overall test strategy is in place and adapted to the considered system under development;
   - Test Tool Consideration, that focuses on the benefits that can be achieved from the introduction of the tool suite for automation, during the system development.

4. *Test planning, design and development*: this and the next one are the most important steps as they include the identification of the test procedures, defines the tests cases and allows for their generation. The test plan identifies among others test items, the features to be tested, the testing tasks, the test environment, the test design techniques and a preliminary testing schedule. Then, according to the plan, test cases are designed in terms of the steps that have to be followed in order to verify a given system feature. Finally the test development allows to obtain test script that can be executed on the system.

5. *Execution and management*. In this step, the previously obtained scripts are executed on the system and the results are collected, in order to perform the assessment.

6. *Test assessment*: this step has a twofold objective: first it allows to evaluate the automated testing process, in order to exploit eventual weakness and improve them at an early stage of development. The second, is the result analysis to find out the defects distribution of the considered system.

The first three steps of such general methodology are more comparable to management process that has to carry out the advantages of an automation process and the needed effort to introduce it within an existing development process. Instead the process that lead to the effective automation of testing, is composed by three defined phases:

1. Test Generation, in which high-level description of test cases is generated from test specification. The outcomes of this step are usually not executable;
2. Test Development, that allows to translate the generated test cases in codes or scripts written in machine-readable language and therefore executable;
3. Test Execution, in which, scripts are executed on the SUT;
4. Test Reporting, in which outcomes from tests executions are analysed and compared with predicted outcomes.

With the respect to the context of the testing automation, this work is focused on the *Test Generation* step. Next sections provide the state of the art related to approaches for automatic test case generation from an high-level specification of the SUT.

2.2 Approaches to the automatic test case generation

There exist several approaches able to automatically derive test case from high-level specification of system functional requirements. Such approaches can be classified with the respect on the basis of the adopted perspective. Below the classification of test cases generation approaches, is performed from the point of view of the knowledge about the SUT.

2.2.1 White-box generation approaches

In general, white-box testing, refers to a testing methodology focusing on the internal structure of the SUT. With the respect to the topic of the test case generation, most of the approaches based on white-box techniques relies on the construction of the Control Flow Graph (CFG) of the SUT. The CFG is a directed graph $G=\{V, E, s, e\}$ in which:

- V is the set of Nodes. Each node, represents a single statement.

- E, is the set of Edges. An edge allows to connect nodes and represent an execution flow between two statement.
- s, is the entry point of the considered system.
- e, is the end point of the considered system.

The objective of the automation with a white-box technique is to find, in an automatic way, an admissible path on the CFG that represent a possible execution of a test case. Several white-box test generation techniques exist in literature:

- **Random Testing**: this approach allows for the generation of random test data in order to give as input to the SUT. Random testing is normally used during integration or unit testing due to its wide-range structural coverage. Furthermore it introduces several advantages:
  - is inexpensive, because it requires only a random inputs generator that is well supported by existing software libraries;
  - is useful to perform stress testing on the SUT;
  - it allows to calculate software reliability, from random test results.

  Random testing however does not assure for the full code coverage and requires human resources able to analyse results of testing.
- **Path-Oriented Methods** : these methods allow for the generation of test data, using paths on the CFG. The analysis could be statical or dynamical. In the first case, generation is made without program execution. One of the well-know techniques in this context is the *symbolic execution* that consists in the execution of a program using symbolic values for variable instead of their actual values. Statical methods allows to obtain inequalities that describe the conditions necessary to cross the path. However general formulation of a path-oriented testing is NP-Hard and its necessary to introduce linear constraints and linear programming techniques. In contrast to the statical approach, the dynamical one performs the analysis during the run-time of the program under test. Program execution flow is monitored during the run-time and, if case of any deviations from the expected flow, some heuristic or meta-heuristic techniques (as the simulated annealing) or the backtracking are used to identify the problem [65]. The obvious limitation of these approaches is that the generation of a single test suite might be achieved after many iterations.

- **Goal-Oriented**: in contrast to the path-oriented, goal-oriented approaches are able to generate test suite even in case of non-complete paths. There are some relevant techniques based on a goal-oriented approach: **chaining**, in which a path is obtained by following sequences of events, representing nodes that must be visited in order to reach testing objectives. Another technique is the **assertion-based**. An assertion represents a pre or post-condition that must be satisfied during program execution. The objective of testing, in this case, is to find a path that satisfies the identified assertions. Assertions and nodes that must be visited imply that the test oracle is embedded in the code.

- **Genetic Algorithms**: such algorithms consist in search heuristics based on a mechanism similar to the natural process of evolution ([89]). Genetic algorithms, starting from a random population of solutions (called chromosomes), perform a recombination process and gene mutation operations in order to evolve the population of solutions and find sets of optimal solutions. Genetic algorithms are more involved in the automatic optimization of the test cases generated with other approaches.

2.2.2 Black-box generation

Black box-testing (also called input-output testing) is based on a "zero-knowledge" that considers a complex system as an opaque box. Test a SUT with a black-box approach means that the expected and the actual output of the systems given a generic input are equals. According to black-box philosophy, test cases are generated by taking into account system specifications without considering the internal structure of the system. There are several black-box techniques, widely used in software engineering, whose automation is the object of several scientific works. Some of them, are focused on the test case generation exploiting the equivalence partitioning techniques or boundary value analysis [79]. The equivalence partitioning is a well-know black-box technique in which the objective is to reduce the total number of test cases necessary, by partitioning the input conditions into a finite number of equivalence classes. Instead a boundary value analysis, the generation is based on boundary conditions which are the values at, immediately above or below the boundary or "edges" of each equivalence classes [95].

The main advantage of a black-box test generation is that is easy to implement; however its difficult to balance the test effectiveness and the test efficiency. Effectiveness is difficult to predict and depends on a thorough and extensive

(thus costly and time consuming) test specification and execution process. In particular for critical systems, given the high number of variables involved, the required simulations (or test-runs) are prohibitive; thus the process is necessarily either infeasible or incomplete, with possible risks on system safety. Furthermore it can be proven that exhaustive black-box testing is impossible to achieve with no information about system implementation [71]. With the respect to test adequacy, it can only be assessed by means of empirical techniques, e.g. when errors/test curve flattens out [52]. Another important limitation is that black-box testing approaches are based on a system specification, which is usually expressed in natural language and thus needs to be corrected, integrated and refined several times during system life cycle. Therefore, its completeness and coherence are not guaranteed, especially for complex systems.

2.2.3 Gray-box generation

To overcome the limitations of black-box strategies and achieve an high degree of completeness and coherence, gray-box approaches have been introduced, specially for embedded critical systems. Gray-box testing approaches support functional testing in allowing test engineers to fine tune the test-set with the aim of an effective coverage of functionalities with the minimum effort in time. The result is a significant reduction in test-set complexity while maintaining or improving test-effectiveness. Several gray box testing approaches are present in the scientific literature focusing at different testing levels (i.e. the kind of software artefacts under test: unit, integration, system or user/acceptance testing). While the knowledge of white-box testing is clear (for software intensive systems it is the source code), it is not clear what the form/nature of knowledge at the basis of gray-box testing approaches. A lot of heterogeneous approaches exist in scientific literature: [7] starts from a formalized specification of the system or of its components; [24] is able to generate test cases for software, starting from contract-based specification techniques and for example [77] exploits the Finite State Machines representing the SUT.

This thesis is focused on system level testing, and in particular on how gray-box test case generation can improve the quality of generated test cases still remaining a feasible solution. Furthermore, system testing is concerned with testing the behaviour of an entire system because unit and integration testing are often conducted by means of invasive methods. The rationale for system-level testing is the necessity to have a level of testing where all the components interact in the same way in which they would interact during the operational phase

of the software/system.

For these reasons, system level testing has been traditionally accomplished by means of black-box testing. As previously stated, this approach expresses its limitations in case of safety-critical systems. First limiting to the observation of the interfaces can prevent the testing engineer to observe the possible passage through some hazardous states. Second, as stated in Section 1.3.1 international safety and quality standards prescribe methodologies and techniques for system level testing with a limited or full knowledge of the system. Furthermore complex systems, can have internal emergent dynamics that must be known and controlled during the testing phase and system testing is the first moment when these dynamics start to appear. Such problem is worsening in case of critical systems where emergent behaviours can bring the system into unexpected hazardous states: thus black-box testing is considered as not sufficient to eliminate all the possible system hazards and motivate the adoption of gray-box approaches. In order to perform gray box (system-level) testing and having "limited knowledge" about the SUT, a model of the internal system behaviour is needed. Indeed, gray box testing approaches are quite always model based-testing approaches, too. Many approaches that exploit models for gray-box testing are present in the scientific literature, for example in [61] where UML activity diagrams are used to generate test cases, [77] where FMS are used to understand about the internal behaviour of the system or the work in [28] where the starting point of the generation is represented by a set of architectural models of the system specified with the AADL language. In the context of the gray-box approaches, next section exploits model-based ones for test case generation and in particular poses the focus on approaches based on the modelling of the internal structure of the system.

2.3 Model-based test cases generation

The model-based generation of test cases, is often included in the category of the Model-based Testing (MBT). MBT is a testing approach based on the construction of a model both of the SUT and of its external environment, derived from requirements specifications. It is usually considered a form of black-box testing, because tests are generated from an high-representation of the system. However, as discussed in the previous section, when information about the internal structure of the system are provided, it is usually identify as a form of gray-box testing. Academic and industrial literature is rich of surveys focusing on the MBT; for this reason, several taxonomies have been proposed in order to clarify

its meaning. The expression "Model-based" applied to generation of test cases, can be applied in different forms. According to this [93], identifies the following meanings:

- Generation of test input data from informations about the domains of the input values (domain model). This reduces hand-made work but does not provide any information to know whether a test has passed or failed;
- Generation of test cases from an environment model. A model represents the environment of SUT. This approach does not provide information on testing outcome, too;
- Generation of tests cases from a behavioural model. A model describes the expected behaviour of the SUT. This approach needs oracle information to compare outcomes;
- Generation of test scripts from abstract representation of tests. Models describe test cases. From them proper transformations generate machine-readable tests.

The [93] taxonomy, was extended by [94] in order to address the embedded critical systems related to critical sectors as the railway, the healthcare or the avionic. A summary of [94] taxonomy is depicted in Figure 2.2

[94] taxonomy is very helpful, because not only surveys main approaches found in literature, but also underlines the key factors of a MBT process. In fact, there are some relevant factors, identified by [94] with the respect to the model-based generation of test cases:

- the selection criteria, that drives the generation of test cases. Among the criteria collected by [94] the approach described in this work is focused on requirement coverage, that means that we are interested to trace, on the model, SUT requirements and verify that generated test cases cover them;
- the choice of the technology to perform the model based generation. There are several manual and automatic approach. Among them, the one exploited in this work, is the one based on Model Checking.
- the result of generation, that allows to classify the approaches on the basis of the nature of the generation results. In fact, generation results, can be, in turn, model, or executable test scripts/code.

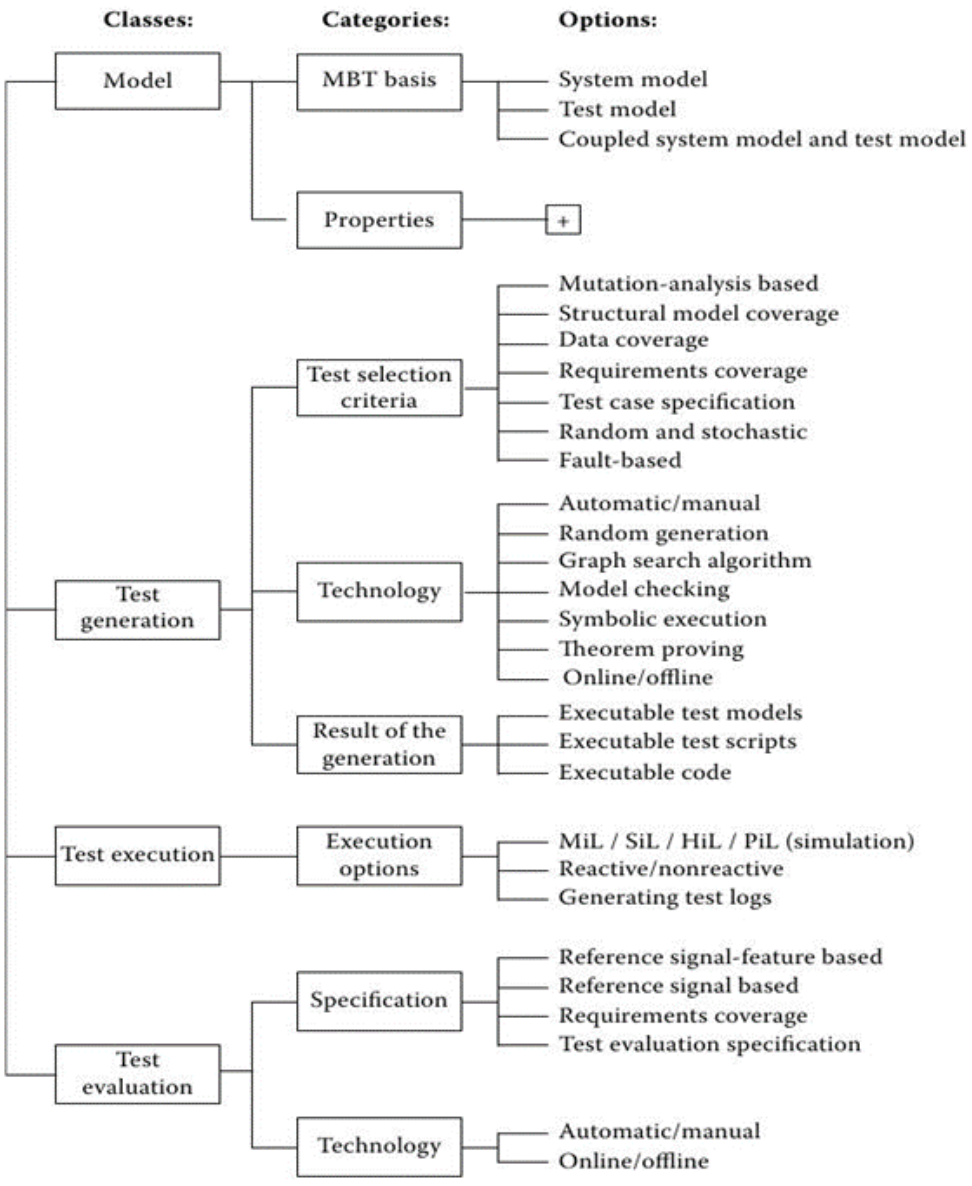


FIGURE 2.2: MBT taxonomy for Embedded Systems [94]

2.3.1 Automatic generation through Model Checking

A well-know technique to automatically generate test cases is the adoption of model checking techniques.

Model checking is a formal technique, used to analyse a finite-state representation of a system for property violations. The core of the technique is the model checker which is able to analyse all reachable states of a system model, in order to verify a property. There are two main reasons that lead to the adoption of model checking for testing generation automation ([40]):

1. The model checker can be used as a oracle for test outcomes evaluation;
2. If a violation is detected, the model checker produces a *counterexample*, that is a sequence of analysed states whose crossing lead to the violation.

Several approaches that apply model checking techniques to testing automation have been surveyed in the years ([27, 36]). Black et al [12] combine mutation analysis with model checking in order to automatically generate complete test suites from formal specifications by defining new specification mutation operators Cimatti et al.[**Cimatti:2013**] describe a formal approach for the validation of functional requirements of hybrid systems. In their work, for instance, counterexamples generation is adopted to verify, in a formal way, the consistency of requirements, during early stages of system development.

The idea emerging from the surveyed works is that a test can be modelled as a property that the system must satisfy. In fact, to verify a given property P, is necessary to to translate it in a Linear Temporary Logic (LTL) or in a Computational Tree Logic (CTL) and to give it to a model checker. However, with the respect to the generation of test cases, the negation of P is given in input to the model checker that, in order to demonstrate the violation, produces a counterexample which is a trace of steps that represents a single test case of a test suite (Figure 2.3).

As the same surveys state, there are some open issues with the respect to the automatic generation with a model checker. In particular the problem of the state explosion when dealing with complex and large models, that may produce a state-space which makes intractable the specification by a model checkers software. The second relevant problem is related to how to force the model checker to generate a complete test suite, since it usually return only one counterexample for each found violation.

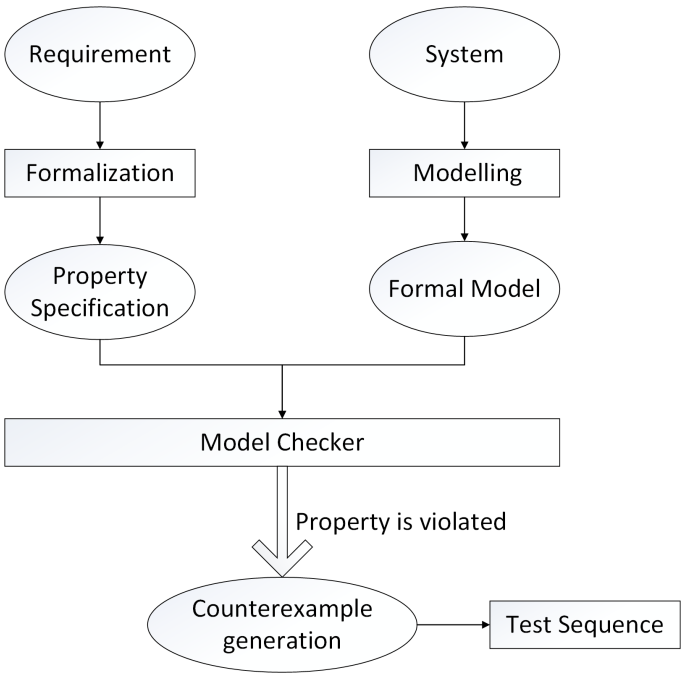


FIGURE 2.3: Testing with Model Checking process

2.3.2 Model-based vs Model-driven generation

Despite Model based (MB) and model driven (MD) use models as primary artefact in system development and encourage both the adoption of a domain-specific modelling language, sometimes there is a confusion in understanding the differences between these two concepts. Several differences exist that definitely separate MB and MD. First the concept of MB was introduced in the 1970 as an attempt to overcome the concept of computer-aided design (CAD) and to support the system development process during its whole life cycle. Model driven, instead, was introduced later, to use abstract representation of a system to concretely generate system code from it. From this point of view, MD can be considered as a subset of MB. Moreover, there is an important difference also in the objectives of these two paradigms. Models produced with a MB approach not necessarily are used to produce concrete system artefacts. In fact, models in MB approach are used to discover, at early stage of development and before the implementation, lacks in requirements specification or in system design. Models in a MD approaches instead, thanks to the support of model transformations, are usually used as inputs to produce new models, that allows to analyse the system from different point views or to obtain concrete realizations of the same system (as for example system code). A relevant example is the the research trend that encourages the adoption of formal methods in industrial settings . In fact, with the introduction of MD, is possible to model the SUT specification with a simple

and user-friendly formalism and deriving a complex formal model, in automatic way without any effort spent by the modeller.

2.4 Model-driven approaches enabling test case generation

According to the previous section, Model-Driven Engineering (MDE) is a software development methodology in which the central concept which drives the entire system development is that *everything is a model*. In fact, model-driven, was introduced to address the increasing complexity of systems and platforms with a abstraction process able to express domain concepts in a light-weight way and to develop, from such abstract representation, the system. A first introduction of the model driven concepts can be traced back to the white paper of [85] in which is presented the Model-Driven Architecture (MDA). The MDA is an OMG initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations. Not all of these technologies will directly concern the transformations involved in MDA. MDA does not necessarily rely on the UML, but, as a specialized kind of MDD (Model Driven Development), MDA necessarily involves the use of model(s) in development, which entails that at least one modelling language must be used. Any modelling language used in MDA must be described in terms of the MOF language, to enable the metadata to be understood in a standard manner, which is a precondition for any ability to perform automated transformations.

Model-driven engineering go over the classical general purpose programming languages by introducing the possibility to adopt a specific language for the considered domain. The presence of a Domain-specific modelling language, allows knowledge and complexity of the considered domain with a specific formalization of the application structure, of the behaviour and of the system requirements. Model-driven allows to migrate from a development process of kind "construct-by-correction" that is based on verification of the late-time properties of a system, to "correct-by-construction", that means the effort of verification is concentrated in the initial stages of the system life cycle.

Moreover, the ability to define high-level languages closer to the user (for abstraction and ease of use) as well as the ability to generate automatically models from the first, allows the usage of formal methods in a "transparent" way and can facilitate their adoption also within sceptical contexts as the industrial ones.

2.4.1 Model driven generation overview

Model-driven testing (MDT) is the application of model-driven principles to testing activities. The result of this combination is well-described by [25], that identify three different levels of software artefacts: the first is the one which specifies the system from the functional point of view, without any reference to the concrete platform, on which test will be executed. Such level is the one of the Platform Independent Tests (PIT). Using model transformations, the PIT is refined in a more concrete model called Platform Dependent Test (PDT) from which is possible to generate concrete artefacts, representing the executable test scripts.

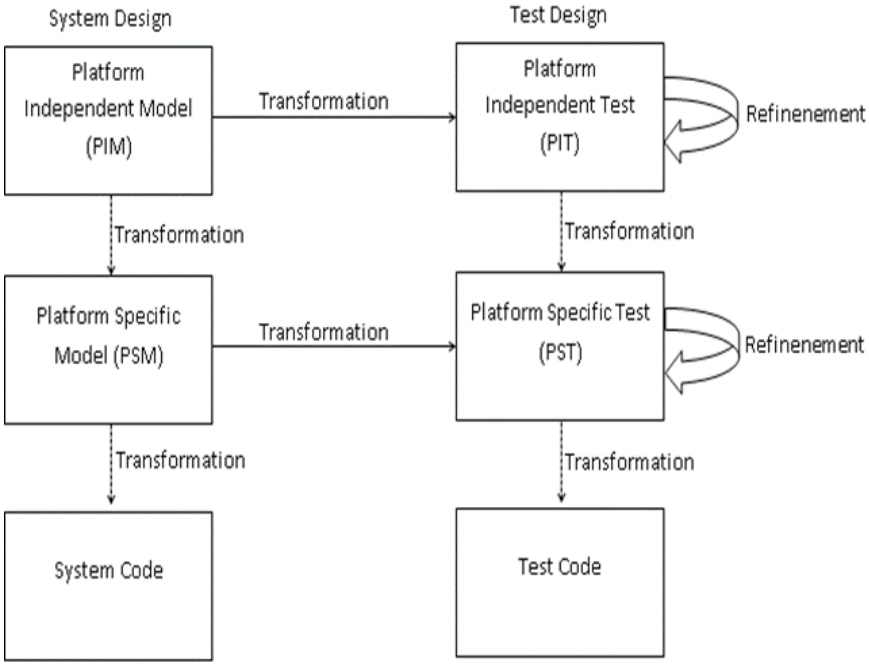


FIGURE 2.4: Model Driven Testing approach [25]

According to MDT process described in Figure 2.4, also from the point of view of the model transformations is possible to distinguish between two levels:

- Vertical transformations that are defined inside the test design process, from the PIT. This kind of transformation allows to obtain the PST and to generate test code for a specific technology platform.
- Horizontal transformations that are defined across the system and the test design steps. This is a very innovative point because horizontal transformations allow performing testing activities early in software development cycle. The first transformation step builds an abstract description of the test suite from an abstract description of the system, without its platform specific model. The transformation between PSM and PST has the same

rational; it allows producing a concrete test suite before the system code is available.

To perform automatic generation of test cases with a model-driven approach, introduce two relevant advantages:

1. first, testing activities may start early in the development process, because a platform independent model can be easily linked with the platform independent test;

2. testing activities are better supported by tools and languages during the whole development cycle.

Several model based approaches are based on techniques borrowed from MDT. Even if they are often considered as "model driven testing", they just use one or more MDT features, such as DSMLs, UML profiling or model transformations. In this work we distinguish between the MDT process above described and such approaches to which we refer as "model driven testing techniques" since they are a sub-set of MBT. Some relevant works show the advantages of model-driven testing techniques within software development. One of the most common ways to establish MDE principles in software development is the use of UML diagrams and its extensions (UML Profiles). UML (Unified Modelling Language) is the most widely adopted general purpose modelling language in software engineering, that provides facilities for the visual design of a system. Javed et al. [55] propose an approach that, starting from UML sequence diagrams, allows to obtain test suite by using a model transformation. Crichton [22] uses UML and applies several transformations in order to generate test case. Mingsong et al.[68] presents an approach starting from UML activity diagrams : in this approach they randomly generate abundant test cases from a SUT written in Java code. Then, they run the program with the generated test cases and obtain the corresponding program execution traces. By comparing these traces with the given activity diagram and according to the specific coverage criteria, they obtain a reduced test case set which meets the coverage criteria. Whilst on the UML language, also UML Profiles, are widely used within the context of the model-driven. An UML Profile is an UML extension mechanism that allows to define custom stereotypes, tagged values, and constraints A relevant example of this mechanism is the UML Testing Profile (UTP) [9]. UTP introduces concepts like test components or test control which are used to realize the test behaviour of the system. Other approaches are able to generate an high level representation of a test suite (and then executable test code) starting from an UTP model of a

SUT. UML is often associated to OCL constraints in order to generate automatically executable test cases. An example of this approach can be found in [14] in which the authors use a subset of UML diagrams constrained with OCL to generate test cases. The use of OCL constraints is necessary to prevent ambiguous behaviours. This point is very important in model driven approaches because it is necessary to have a clear understanding of syntax and semantics of source and target models in order to be able to perform model transformations [84]. Another language often used to introduce model driven techniques is the Testing and Test Control Notation version 3 (TTCN-3) which is a strongly typed test scripting language used to automatically determine whether a system fulfils its requirements. An example of TTCN-3 applications in testing can be found in [92] in which MDE principles are used for testing in ICT domains. The application of model driven to the test case generation process is not limited the UML context. There are in fact other relevant application both coming from the academic and industrial literature. [39] present a test case generation approach based on symbolic execution to obtain data inputs and enumerate event sequences that can lead to maximize code coverage of a GUI application In [48] an example of the application of MDT to web-based distributed services architecture is described and [45] shows an application of model driven testing within the automotive domain. This work describes an application related to the safety control of air bag systems. Despite these and other examples, the application of MDT is far to being considered an assessed practice in industrial settings.

2.5 Technical background

A test generation process based on model driven techniques often means that are involved DSMLs and model transformations. This section provides a background on such techniques, surveying the principal technologies enabling the definition of a DSML and the realization of a model transformation.

2.5.1 Domain-specific modelling languages

According to model-driven principles above discusses, a DSML is useful to describe the internal behaviour of the SUT and the tests specifications. In order to define a modelling language, two aspects should be addressed: the syntax, that represents the way with which is possible to express modelling concepts and the semantics, that is the meaning of such modelling concepts. With the respect to the syntax, a further differentiation can be made; in fact is possible to distinguish

between the abstract syntax, that only specifies domain concepts and the concrete syntax that provides the notation to express abstract concepts. There exist several approaches to follow, in order to define a DSML. Selic in his work [83], summarizes such approaches in:

- Definition by refining an existing general modelling language (e.g. the UML Profile mechanism);
- Definition by specializing some general constructs of another modelling language;
- Definition by the development of a new modelling language. Such approach is often named *from scratch*.

The design choice between above approaches, should be made with the respect to the domain that has to be modelled. In fact, the introduction of a DSML is usually made to simplify the modelling process. The problem with general-purpose languages is that often some specific concepts can be expressed in very inefficient and inexpressive way. Thus, having a language that primitively provides such concepts allow to simplify the overall modelling process. This means also that, all unneeded concepts should be avoided in the definition of a new DSML. This concept is very important because, with the respect to first two approaches (extension and specialization) often there is the need to restrict the expressiveness of the extended/specialized language. Furthermore, the concrete syntax should be as much as possible near to the background of the domain users, that will adopt the considered DSML. With the respect to this, in some contexts, there is the need to define a DSML from scratch. However, also in this case there are some limitations: first the effort in the development, due to the fact that both the syntax and semantics should be defined from scratch. Moreover is has to be considered that all the tooling infrastructure should be provided, and usually developed from scratch. On the other hand, the definition from scratch allows for compact and easy specific modelling language, able to provides primitives to express directly specific modelling issues and that is able to avoid ambiguous behaviours. Except for the first two approaches, to define a language from scratch, there are some steps to be accomplished. In particular a meta-modelling process able to describe domain specific concepts and their relationships is needed. Also meta models need languages describing them; usually, within the model-driven context, metamodels allow to define the abstract syntax of a DSML, generally in form of class diagram. Then the abstract syntax is mapped to a concrete syntax i.e. the DSML constructs provided to the modeller (syntactic mapping). Hence,

the choice of the meta-modeling language is of great importance, as this choice has consequences on the entire process of test generation. Below a discussion of the main meta-modelling languages and on how they can determine different ways to implement and support the generation process is provided.

2.5.2 Meta-modelling process

The choice of the meta-modelling language should be conducted on the basis of the objectives for which a DSML is introduced. In fact, meta-modelling languages can be classified in:

- Editing-oriented, when the aim of the meta-modelling language is finalized to the realization of user-friendly environment for the editing of the models;
- Analysis-oriented, when the objective is the static analysis of the model during its construction;
- Simulation-oriented, when the objective of developed DSML is to execute the models and/or to perform simulations.

On the above classification, and to the best of the knowledge derived for the surveyed literature, the found meta-modelling languages have been grouped in four categories:

- Meta Object Facility (MOF) based: MOF is a OMG standard and is the basis of MDA (Model Driven Architecture);
- Eclipse Modelling Framework (EMF) based: this solution exploits the potentialities of Eclipse. It can also be conjugated with the usage of UML;
- Grammar-based: the meta-model is defined by using grammars like BNF or EBNF;
- All other approaches: several approaches belong to this category, both relying on open source and commercial solutions. As example, MetaEdit+ [67] is a commercial suite that enables the generation of full code directly from models and allows for defining a new domain specific language from scratch.

Table 2.1 provides summarizes the different possibilities and their consequences on Editing, Analysis and Simulation. Here below, they are described in more details.

TABLE 2.1: Meta modelling solutions

	MOF-based	EMF-based	Grammar Based	Other approaches
	Meta-modelling languages			
	- UML Profiles	- Ecore - GMF	- BNF - EBNF	- MetaEdit - Atom3 - MetaGME
Editing	- Reuse of UML Concepts - Large set of supporting tools and IDEs	- Very user-friendly - Graphical IDEs provide palette with the meta modelling concepts	- High readability due to the informal nature of the BNF - Wide support of tools - Can be customized for a large set of applications	- Often proprietary technologies and IDEs are not very user-friendly - Most of them are not open source
Analysis	- Analysis realized through transformations or OCL constraints	- Analysis realized through java code	- Very easy to parse	- Analyses are often allowed during compiling time
Simulation	- Some IDEs allow for the simulation of UML models	- There exist many facilites for simulation of realized models	- Simulation is difficult to perform	- Several engines for simulation

MOF - Meta Object Facility Meta Object Facility (MOF) is a standard produced by OMG which is place at the top layer of Model Driven Architecture. MDA is a consolidate approach in which models and modelling techniques are the main artefacts of software development cycle. The MDA architecture consists of four layers:

- M3, the meta-meta-model layer, which provides a model of the modelling language;
- M2, the meta-model layer, which describes the concepts used by the modelling language to construct the model within M1 layer;
- M1, the model layer, in which there are the models of the element of the system. Layer M1 provides the generalization of concepts in M0 layer;
- M0, the system layer.

MOF provides a language to describe modelling language hence it is defined at the M3 layer. MOF enables to define a new language both from-scratch and by extending/specializing an existing language. In fact, it provides the mechanism of the UML Profiles which allow for extending UML models for specific domains. Such extension could be made by using

- *Stereotypes*, used to extend UML concepts, so providing constructs to build the models. Graphically the application of a Stereotype is identified by the

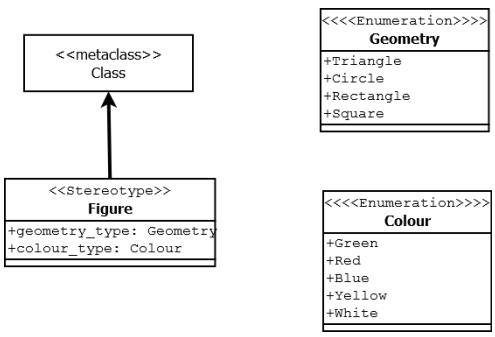


FIGURE 2.5: Example of UML Profile

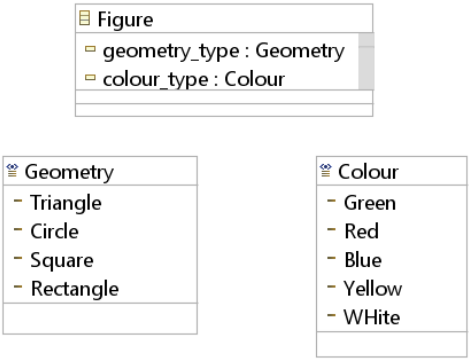


FIGURE 2.6: Example of an Ecore metamodel

label «Stereotype» before its name on a UML element; it is also possible to define icons which are showed after the stereotype application;

- *Constraints*, associated to stereotypes, which are used to impose restrictions to the stereotype. OMG has defined a language named OCL to express a constraint. Any rule associated to the stereotype can be expressed by using a constraint;

- *Tagged values*, which are meta-attributes associated to stereotypes or metaclass. Each tagged value has a name and a type associated to a specific stereotype.

Several UML Profiles can be found in literature; some of them are defined or standardized by OMG itself. Examples of UML profiles from OMG are MARTE (Modeling and Analysis of Real-Time and Embedded systems) [74] and UTP [9]. UTP provides extensions to UML in order to support the design, visualization, specification, analysis, construction, and documentation of the artefacts involved in testing activities. A simple example of an UML Profile is depicted in Figure 2.5; in this example the stereotype *figure* is added to UML metamodel by extending the *class* metaclass. The properties of the figure, i.e. colour and geometry, are described by using homonym tagged values.

This example shows some relevant advantages of the UML Profiles. Firstly they allow to introduce specific concepts of an application domain at metamodel level (UML in fact does not have the concept of "figure"); secondly they allow to add expressive power at modelling language. UML profiles also provide additional information that can be used for M2M or M2T transformation [37].

UML is widespread used during all software engineering processes; it is implemented by a large amount of tools and software development environments.

Finally the interoperability is guaranteed by the XMI format, defined by the OMG group, that allow the interchange of the same models between different tools.

EMF - Eclipse Modelling Framework Eclipse Modelling Framework (EMF) is a stable framework which provides facilities for building toolsets and Java applications based on model manipulation. EMF allows for creating a model and generating code from it with the same level of usability of an UML model. As stated in previous Subsection, a model is an abstract representation of an object. A model can be described using several languages. EMF introduces a new concept which is the passage between different high-level representations. More specifically, as depicted in Figure 5, EMF unifies three relevant technologies: UML, XML and JAVA. As an example, EMF allows to transform a XML schema into a UML class diagram or directly in executable Java code [90]. The EMF is supported by the Ecore format, that is a simple format guaranteeing interoperability between tools. The Ecore format is also an open format, easy to understand and manipulate, also in textual way.

Hence EMF integrates both modelling and programming principles. There are several advantages behind EMF-based solutions:

- it is easy to provide a clear representation of what the system it is supposed to do;
- Code generated from the model is high readable;
- EMF adopts the Ecore format to provide interoperability and information interchange;
- EMF allows for queries on the structure of the model;
- There exist several plugins to generate editors supporting defined languages.

EMF consists of three main parts:

1. EMF.Core that includes a meta model in Ecore format for describing models. Ecore is well supported by large set of Application Programming Interfaces (APIs) for manipulating EMF objects;
2. EMF.edit, that includes generic classes for building editors for EMF models. This framework allows to display EMF model in a standard Eclipse view and provides to manipulate models properties, classes etc.

3. EMF.codegen, which provides the code generation facilities. EMF.codegen is based on the Java Development Tooling to build EMF editor. With EMF.codegen is possible to generate:

   - Classes from model, including factory methods and packages;
   - Adapters that allow editing and display of generated classes;
   - Editor, which allows to customize the model. The editor is show in eclipse-like view.

   The development workflow of EMF is very simple: first a model is created and defined using the Ecore format. Second, when the model is defined, it is possible to automatically generate java code from it, thanks to the plugins provided the same framework.

Ecore includes few and essential concepts such those included in the Essential MOF: some of the basic types included in it are EClass, EAttribute, and EReference which have the same meaning of the Class, Property and Reference of the Essential MOF. As an example, considering the same objects of the UML Profile described in Figure 2.5, it is possible to have the Ecore diagram described in Figure 2.6. The main difference between the two formalism is that the UML profile adds the "figure" concept to UML while the Ecore metamodel reports a new modelling language created from scratch: the usage of this domain model does not allow to use other concepts than the one of "figure". Here the concept of stereotype is replaced by the Eclass concept while tagged values are replaced by the EAttributes. Constraints can be inserted within Ecore diagram by using natural language or structured languages as OCL.

Grammar-based approaches A grammar is a mathematical based system to define a language, as well as a way for giving the sentences in the language a useful structure. They are usually adopted to define a textual-based syntax of a language. The main advantage in the adoption of grammar-based approach is that operations as the parsing and the translation are often made simpler by the structure imparted to the sentences of the language by the same grammar. Grammars are probably the most important class of generators of languages. A particular type of grammar, widely adopted for the definition of concrete syntax of domain-specific modelling language is the context-free grammar (CFG) A context-free grammar is defined as a grammar in which every production rule is in the form $A \rightarrow \alpha$, where A is a grammar variable and α represents a sequence of variables and terminals. The CFG is composed by a set of recursive re-writing

rules (or productions) used to generate patterns of strings. The adoption of CFG is limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a CFG which accepts a superset of the desired language constructs A widely adopted CFG, in the definition of the syntax of a DSML is the Backus–Naur Forms (BNF) or some its variants as the Extended Backus–Naur Form (EBNF)

2.5.3 Model Transformations

Model transformations are a key concept of model-driven principles They allow for obtaining a model from another in an automatic way. The OMG's MDE standards specify the need for moving from platform-independent models to platform-specific models, raising the level of abstraction during the modelling phase, and then reducing it for a specific platform, during the development stages. Model transformations may be useful, for example, to generate a formal model starting from a UML model. Model transformations can be grouped into two categories:

- Model-to-Model (M2M) transformations aiming at transforming source models into other models, also expressed in different formalisms. The main motivation of their need in the context of this work is that the obtained formal model is used to exploit model checking techniques to generate test cases. A M2M transformations allow to translate a source in a target model by defining proper sets of rules between these two. An example of language used to write M2M and M2T transformation is the ATLAS Transformation Language (ATL) defined in the ATLAS Model Management Architecture (AMMA) platform [56]. A deep description of such language is provided in below paragraphs.

- Model-to-Text (M2T) transformations: M2Ts are able to generate text directly from a model (conforms to a specific meta-model). M2Ts have a paramount importance in model driven software development processes since automatic code generation represent a final but a necessary step in

many of such processes. In a wider perspective, M2Ts can be used to generate text, reports, configuration files or to instantiate abstract models according to a specific concrete syntax. This last case can be used when a formal model, expressed for example by means of an Ecore based language, must be translated into a specific data format understandable by existing solvers. M2Ts can be divided into two categories according to the constituting principles:

- Visitor-Based Approaches: the source model is explored and, during the exploration, text is serialized into an output channel (a file). An example is constituted by ATL query [56];
- Template-Based Approaches: the text is organized into templates where "hot-spots" (points that are subject to change according to model structure or values) are calculated by query on the model itself. A widespread example of this technology is constituted by Acceleo [3].

There exist only few solutions enabling model transformations. To the best of our knowledge the technologies surveyed within the context of the present work are summarized in Figure 2.7. With the respect to the figure an arc between a metamodelling solution and a transformation language, means that a model transformation from/to the specified metamodelling language may be implemented by the destination of the considered arc.

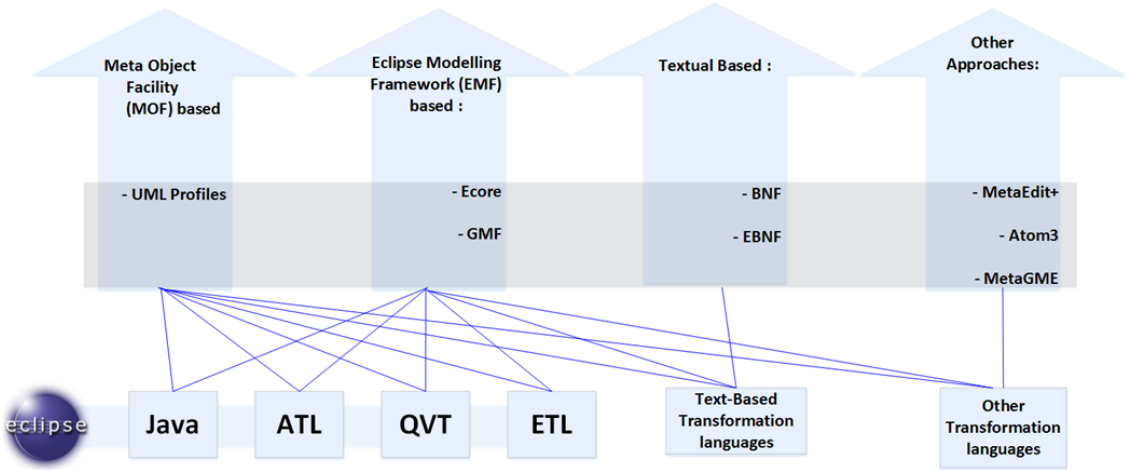


FIGURE 2.7: Transformation technologies

2.5.4 Transformation technologies

This section summarizes the surveyed technologies found in literature.

Java Java is one of the most important programming languages based on object orient paradigm. Java may be also used to implement transformations between source and target languages. Despite the power of Java, specific transformation languages are preferred because they provide a range of useful features which facilitate writing, understood and execution of the transformation.

ATL ATL stands for Atlas Transformation Language, created by the ATLAS INRIA & LINA research group. It is the answer to the OMG MOF and to QVT. It is a model transformation language specified both as a meta-model and as a textual concrete syntax. It is a hybrid language since it is possible to define declarative and imperative statements. Most of the rules written by using ATL are declarative, which means that mappings can be expressed simply. Despite the declarative way is preferred, imperative constructs are provided in order to manage situations too complex to be dealt by means of also declarative rules. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. The structure of an ATL program is composed by four parts: Header, Import, Rules and Helpers. The header contains the transformation name and the declaration of both source and target model. The import section is used to import definition specified by other ATL modules. This can be done by using the keyword "uses" followed by the library name. Rules section is the core of ATL file because it contains the transformation rules. Each rule defines source patterns, the element type of the source model to be transformed, and the target pattern, used to generate a portion of the target model. ATL supports the definition of helpers: a helper is used to declare functions and global variables used by the transformation rules. Helper functions are written by using the OCL language.

In the pattern depicted in Figure 2.8, a source model Ma is transformed into a target model Mb, according to the rules defined in the transformation Mt. The transformation can be seen as model since it is software. Source and target models, as well as the transformation definition are conforming to their respective meta-models: MMa, MMb and MMt. All meta-models in this example are conforming to MOF meta-meta-model (obviously this relationship is not strictly necessary; other meta-meta-models are of course usable). This schema is general enough to be adopted by all other transformation languages. ATL, as mentioned before, is a mixed language which contains declarative and imperative parts, nevertheless, the ATL philosophy encourages the use of the declarative style in

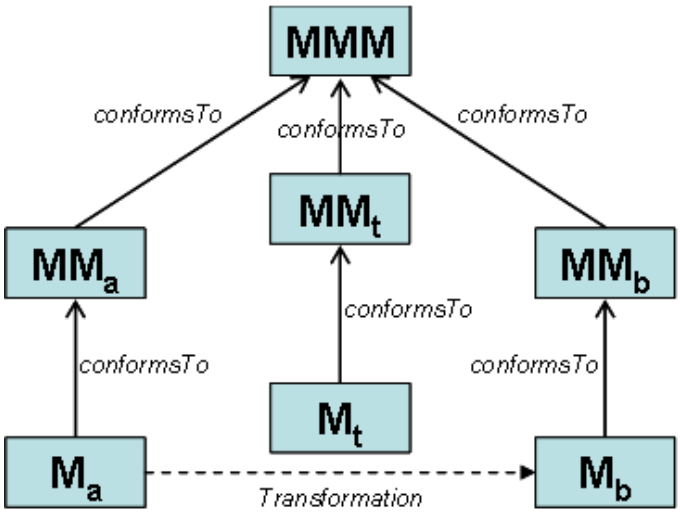


FIGURE 2.8: ATL transformation schema (from http://help.eclipse.org)

specifying transformations. However, sometimes it is difficult to provide a solution completely declarative in a transformation problem. In this case it is possible to use characteristics of the imperative language. ATL transformations are unidirectional, source models are read-only and the transformations produce write-only destination models. The implementation of a bidirectional transformation makes it necessary to realize a pair of transformations, one for each direction. An ATL rule, can be composed by several sections containing declarative or imperative statements. Among these sections, the most important are the *source pattern*, representing the element of the source model from which generate the target one, the *target pattern*, that specifies the way in which the target element is constructed and the *do pattern*, that allows to put the imperative code, when necessary. On the basis of the contained patterns several kind of rules can be distinguished; in particular the widely used are the **matched rules**, the **called rules** and the **lazy rules**. A matched rule specifies the way target model elements can be generated from source model. They are triggered every time the elements specified in the *source pattern* are matched in the source model. Matched rules, can be composed by both of the above mentioned patterns. The called rules, are instead rule that can contain only *target* and *do patterns*. In fact, they are similar to the methods of a general purpose language: a called rule allows to explicitly generate target model elements from imperative code inserted in a *do pattern*. Finally lazy rules are matched rules whose *to pattern* is called from another ATL rule.

QVT QVT stands for Query/View/Transformation and it is a language for model transformation reated and standardized by OMG. Due to OMG derivation, QVT includes both MOF 2.0 and OCL 2.0 specifications. As ATL previously described, this language is hybrid (declarative and imperative). With QVT are provided three different transformation languages:

- QVT-Core which is a declarative language designed to be simple and to be used on QVT Relations target model. Due to the not fully specified nature of QVT-Core, QVT-Relational is more expressive.
- QVT –Operational which is imperative transformation defined for writing unidirectional transformations;
- QVT – Relations, which provide declarative transformations. The transformation written by using QVT-Relational can be both unidirectional and bidirectional.

Compared to ATL QVT languages do not permit M2T transformations, since each model must conform to MOF 2.0 metamodel. M2T transformations are being standardised separately by OMG in MOFM2T standard [75].

ETL ETL stands for Eclipse Transformation Languages and provides M2M transformation language features to Epsilon. ETL allows standard operations of a transformation language but also other advanced features like manipulation, navigation and query of both source and target model. ETL is a hybrid language that implements a mechanism composed by declarative definition of rules but also inherits the imperative features to handle complex transformation which can't be addressed with a declarative language. ETL is similar to ATL because it is organized in modules (named EtlModule). A module contains a number of transformation rules. Each rule has a unique name and has specified both source and target model. A transformation rule can extend other transformation rules with different mechanisms (called lazy, primary or abstract). EtlTransformation defines a block statement in which is collocated the logic for populating the property values of the target model elements. ETL allows defining pre e post statements that can be executed before or after the transformation rule [58].

Text-based transformations The expression "text-based transformation" denotes a transformation which transforms an input document written according a text-format in an output document written according a different format. An

example of text-based transformation language is XML transformation language which allows transforming an XML document in another XML document or in HTML document. Examples of XML transformation language are XSLT, which is a W3C recommendation or XQuery which is a standard de facto used by Oracle, Microsoft etc.

Acceleo Despite Acceleo is not properly a technology for model-to-model transformation, it has become a leading technology for generation of text from a model conformant both to Ecore based language and both to annotated UML.It is based on the template-based paradigm in which the user creates text templates in which some parts can be dynamically defined on the base of some model query. Since its launch, Acceleo has become a widespread solution in model driven engineering processes [3].

Chapter 3

An interoperable framework for testing automation

Verification of functional requirements of safety-critical control systems requires an high effort, in order address recommendations of international standards. According to the discussion in Chapter 1, to support verification processes, by automated solutions, is a key factor for achieving lower effort and costs and reducing time to market.

This chapter describes the methodology defined and followed for to the development of the automatic test case generation approach. In particular, such methodology can be collocated in a wider verification process related to the *system level testing* of railway control systems. The general verification process has been defined in order to address the needs of the ARTEMIS CRYSTAL project, which tackles the challenge to define Interoperability Specifications (IOSs) for the development of an European standard supporting the entire life cycle of safety-critical control systems. According to this, the chapter presents also an overview of the CRYSTAL project clarifying how thesis purposes are partially involved in the verification process defined by the CRYSTAL project. Moreover the high-level architecture of the test case generation framework resulting from the methodology is provided and discussed.

3.1 CRYSTAL project

The ARTEMIS Joint Undertaking project CRYSTAL (CRitical sYSTem engineering AcceLeration) aims at realizing a Reference Technology Platform (RTP), based on common Interoperability Specifications (IOS), able to support the design, the development and the deployment of interoperable safety-critical control systems. The development of a standard RTP can have a significant impact in the

European competitiveness, since it may increase the efficiency in critical control systems development by encouraging the emergence of new market applications. CRYSTAL activities exploit domain-specific insights, into embedded safety-critical control system design and safety process, to investigate and establish cross-domain synergies. CRYSTAL is, in fact, strongly industry-oriented and provides ready-to-use integrated *tool chains* having a mature technology-readiness-level (TRL)[1]. Technical innovations can be achieved by adopting a user-driven approach based on the application of engineering methods to industrial relevant *Use Cases*. According to this, in a first step, user stories are collected and described as work flows. Then these user stories are refined in a set of concrete use cases, applying to them significant improvements. For each concrete use case a set of tools and methods, named *technology brick* is derived. A *technology brick* is in fact in charge of integrate the related use case in the overall RTP. The RTP is based on a generic model-based tool integration platform, composed by a set of interoperable tools, methods and processes, designed to improve the development of safety-critical embedded control systems. The technology bricks shall be designed according to the IOS, implemented using the RTP and finally validated in their domains. The application domains considered within CRYSTAL are the automotive, the aerospace, the rail and the healt-care. In addition CRYSTAL aims at enhancing the maturity of existing concepts achieved in previous European and national projects (as CESAR and MBAT, described in Section 1.4.3) with innovative techniques, methods and tools developed in other research projects, in order to bring them to a level of maturity that is compatible with an industrial pre-deployment stage. The approach defined in the thesis is related to the railway domain, and specifically, addresses the need expressed by Ansaldo STS (ASTS), an international transportation leader in the field of signalling and integrated transport systems for passenger traffic (Railway/Mass Transit) and freight operation. The industry needs expressed by the ASTS's *Use Case* are oriented to improve the quality and the efficiency of existing V&V processes, in particular by reducing the effort to be spent for the definition of *system level* tests.

3.1.1 CRYSTAL V&V process

User needs expressed by ASTS within the CRYSTAL project are oriented to the automation of the system level testing activities, and to the realisation of a tool chain providing full support to interoperable testing. This section describes the

[1]Readiness Levels (TRL) are a type of system measurements used to assess the maturity level of a particular technology.

complete work flow V&V process defined by the CRYSTAL activities of the considered *Use Case* as well as the components of the tool chain and their relationships.

The approach complies with the ASTS *Use Case* requirements and allows to improve the ASTS current testing process, starting from the definition of the system specification to the generation of test reports [10]. In detail, it enables semi-automatic generation of test cases from a set of test specifications, relying on a Model-Driven methodology. The generated test cases are then automatically transformed into executable test scripts, which can be executed on the real system or on simulation environments. Test logs are then analysed and test reports are automatically generated.

Figure 3.1 describes the overall approach of defined within the project context. Lets note that the execution of the script on the real system is out of the scope of such project activities.

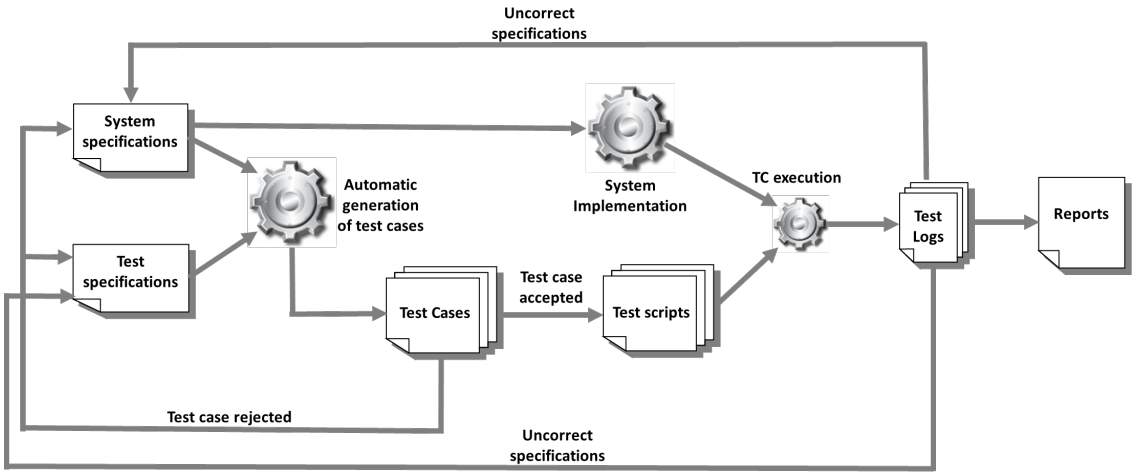


FIGURE 3.1: ASTS CRYSTAL approach

The depicted approach introduces three main advantages:

- *Automatic generation of test cases from test specification*: in the current process, adopted by ASTS, test cases are manually generated by domain experts which are able to control the high complexity of these systems. This activity is heavy and error prone, in addition the training of new testers is very expensive as they must have a great experience, that could be acquired only working on several different projects.

- *Generation of test script in IOP notation*: due to the heterogeneity of simulation environments, system level testing requires an interoperable testing environment where different simulators can exchange information. The

generation of test scripts in IOP notation, under development by UNISIG (Union Industry of Signaling), allows for the execution of interoperable tests in a multi-suppliers environment.

- *Tool supporting analysis of test logs*: the decision about the outcomes of test cases is currently performed by inspecting big log files; the adoption of a tool which is able to parse generated logs, reduces the efforts of navigating them and generating reports.

The first activity of the testing process is the realization of a system specification, performed manually by V&V engineers by using a proper modelling language and a graphical modelling environment. The specification of the system is given by a high-level description of the system behaviour and by the set of functional requirements the system shall satisfy. By the same environment, the test specifications are defined in order to describe the essential features that a test case must accomplish (e.g., the sequence of transitions that the test case shall stress). Test cases are generated from the system and test specifications in semi-automatic way.

According to the proposed work flow, test cases can be analysed and, if they are rejected by engineers, some updates on the source specifications can be performed. Test cases are then transformed into executable test scripts, through a transformation in the IOP notation. This language supports the creation of interoperable and multi-supplier testing environments.

Test scripts are executed and proper test logs are generated. The execution of test cases has not been considered in the CRYSTAL project, since each railway operator is interested in using proprietary testing environment. However test logs can be parsed in order to detect possible inconsistencies between planned test case and test execution. These inconsistencies can be due to wrong specification (and it is necessary the feedback to the source model), otherwise they trace bugs in the developed system. Finally test reports can be generated.

Each component of the framework interacts with the RTP, which is a generic platform for the integration of model-based tools. It is composed by a set of interoperable tools, methods and processes designed to increase the quality of development processes of safety critical embedded systems. This integration platform will host tools coming from different stakeholders (vendors, industrial & academic partners, etc.) that realise *Bricks* within the project. Therefore there is the need for a common non-proprietary standard to realise this interoperability functionality within the RTP. The CRYSTAL IOS would accomplish this task by adopting the Open Services for Lifecycle Collaboration (OSLC), a framework

that moves to the integration of data, workflows and processes among product lifecycles. OSLC is divided in several workgroups each of which addressing specific integration scenarios. The set of scenarios and specifications are named *OSLC Domains*. The presence of different domains introduces the necessity to manage the coherence among them. This need is satisfied by a set of standard rules and patterns, contained in the OSLC Core Specification, and all the domain groups must adopt these rules for the specifications. The union of a OSLC Core Specification and a OSLC Domain constitutes a OSLC protocol that is used in order to add interoperability to a specific tool chain, as in CRYSTAL. Some work packages in CRYSTAL are devoted to the study of existing standards and to the proposition of proper technological solutions in order to integrate the Technology Bricks with the RTP/IOS. Three main technological bricks interacting over the RTP have been during the activities of the CRYSTAL project; they are: *Rail Model*, *IOP Test Writer* and *Log Analyzer*. Next subsections briefly describe such bricks.

3.1.2 Rail Model

The Rail Model brick, is the core of the approach, because is in charge of performing automatic test case generation. Such brick, is the composition of a graphical user interface (GUI) and an engine for the automatic generation of test cases. It in fact, provides the following functionalities:

- Creation and editing of formal state-based models from specifications of system behaviour by using a user-friendly graphical editor. This functionality is based on the definition of a DSML for the modelling of such kind of system. The language definition is deep addressed in Chapter 4.

- Definition of a proper set of Test Specification Patterns providing general reusable models for recurrent classes of requirements. The same environment in fact allows for the modelling of test specifications in order to describe essential features that a test case must accomplish (e.g., the sequence of transitions that the test case shall stress). Test cases are therefore generated from the system and test specifications in semi-automatic way.

- Generation of test sequences. The engine that realize the generation step, is called Test Case Generator (TCG) an relies on the results presented in this thesis. The TCG exploits model-driven principles, previously described, by

combining the advantages of defining proper DSMLs and model transformations to automatically derive formal models from the high-level specification of the system and used them to exploit model checking techniques, chosen as a test case generation strategy (Section 2.3.1).

3.1.3 IOP Test Writer

The *Rail Model* brick allows to generate the sequence of the steps specifying a test case. These traces need to be translated into an concrete notation in order to be executed on simulated environments. Since railway-based infrastructures are composed by different subsystems that can be supplied by different technology providers, the testing and simulation environments reflect this heterogeneity being a federation of different simulators developed by different teams. One of the requirements for a an interoperable testing environment is that each component must speak a common "testing" language. The IOP Notation is developed by the UNISIG. Properly interpreted by vendor-specific adapters, IOP can support the creation of integrated testing environments. The aim of the IOP notation is not limited to simulated environments as it can be used to give commands and interpreting the states of real systems. Test steps, written in this general common language, shall be properly understood by different system implementing proper *adaptors*. The usage of this standard language reduces the risk of misunderstanding/incoherence and enables the execution of interoperability tests in laboratory. Obviously, since each testing environment is built in its own language, all the companies/suppliers have to develop several adaptors for the interoperable testing environment.

Within the CRYSTAL project, IOP Test Writer can significantly increase the level of interoperability of industrial vendors products. According to this, the brick of the IOP Test Writer tool consists of two modules according to software engineering best practices: the *TestWriter* and the *Load/Store Manager*. Each module performs some specific tasks and shows a set of interfaces used to interact with the other module. More specifically:

- the *Load/Store manager* module provides the interfaces to interact with the external modules (for example the RailModel) and with the other technologies within the Crystal IOS. *Load/Store manager* module loads the test sequences produced by the Rail Model Brick and stores their results in the system: such interaction is accomplished by means of IOS/RTP;

- the *IOP Writer* module provides the transformation of the test cases expressed in a Ecore language to a test cases expressed in IOP notation by means of a Model-to-Text (M2T) transformation.

3.1.4 Log Analyzer

The Log Analyzer brick supports the V&V engineer to state if the execution of a specific test passes or fails. Hence, the Log Analyzer has two different sources: (1) a test case as generated by the Rail Model brick, (2) the logs created by the execution of a test case (after its translation into the IOP notation) on the specific testing environment. According to such inputs, the Log Analyzer may find if logs and the test case match. Such operation would be pointed out to the V&V engineer who is able to decide if the test passes, fails for an error in the system model or fails for a misinterpretation of the requirements.

The Log Analyzer has a modular architecture according to software engineering best practices:

- the *Load manager* module which provides the interfaces to interact with the external modules and with the other technologies within the Crystal IOS. The *Load manager* loads the test sequences produced by the Rail Model Brick: such interaction is accomplished by means of IOS/RTP;
- the *Parsing and Analysis* module that is used to parse the logs of the test executions. Moreover, each log is analyzed according to the inputs and then the fail/pass decision is taken for the single log.
- the *Report* module focusing on building up summary information about the entire testing campaign and in generating supporting tables for traceability, coverage, etc.;
- the *Export* module that has the role to export the report of a testing campaign and the related execution logs. The target report will be conform to widespread document formats (e.g. pdf files or spreadsheets).

3.2 Model-driven methodology for the automatic test generation

Within the context of the CRYSTAL project, the work described in the thesis is focused on the definition of a methodology for the automatic generation of test cases. According to this, the methodology relies on model-driven principles and

exploits model checking techniques, in order to generate test cases automatically from the SUT specifications. The activities composing the model-driven methodology are summarized in Figure 3.2 and are below discussed.

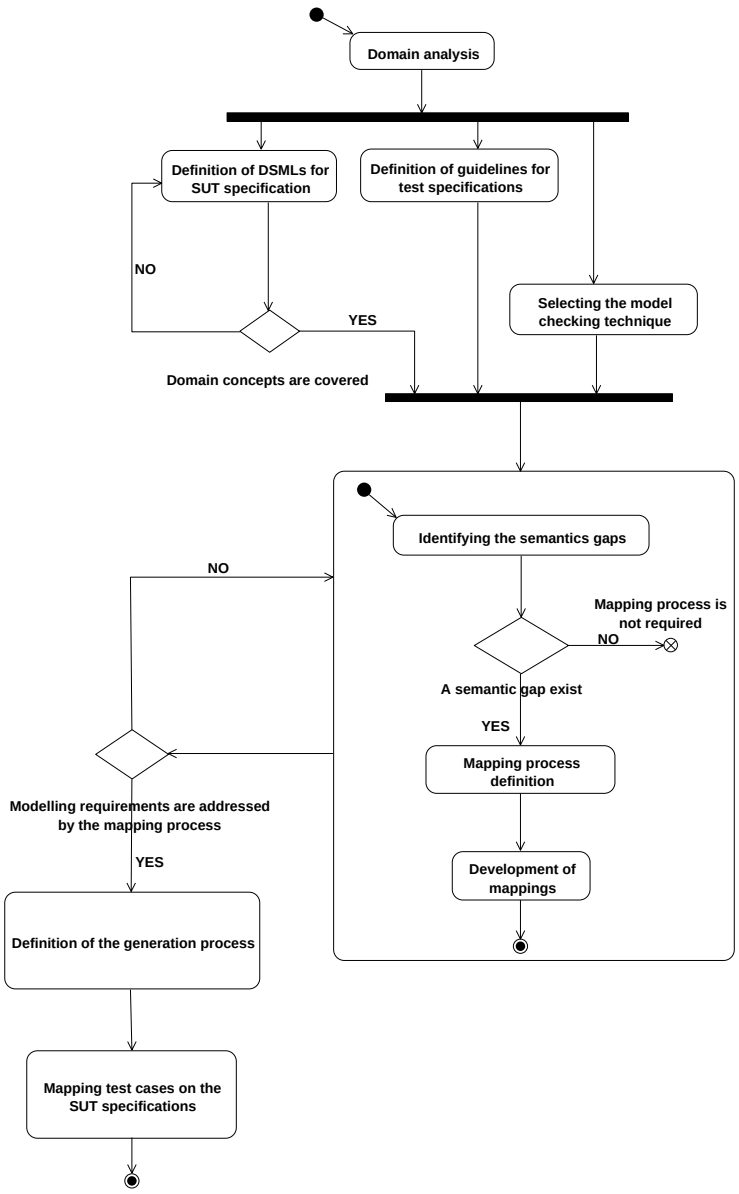


FIGURE 3.2: Methodology for automatic test generation

Domain analysis The first step is the analysis of the domain of interest. The domain analysis is necessary to identify the requirements that a domain-specific language should address, in order to describe the SUT. The domain analysis enables three concurrent activities: (1) the definition of proper language to model

the SUT, (2) the definition of guidelines to model the specifications of tests and (3) the choice of the model checking technique to adopt.

Definition of modelling languages and guidelines In order to fulfil recommendations of international standards applicable in railway domain and according to industrial needs, a state-based formalism for modelling the SUT specification, named DSTM, has been defined.

According to this, during thesis activities, a language extending Hierarchical State Machines [5], named Dynamic STate Machine (DSTM), has been defined. Its peculiarity mainly resides in the semantics of fork-and-join which allows dynamic (bounded) instantiation of machines (processes) and parallel execution of machines inside a box. Each state machine may be parametric over a finite set of dynamically evaluated parameters; in addition the same machine may be dynamically instantiated many times without explicitly replicating its entire structure. DSTM allows for the description of the system structure in terms of its states, possible inputs and obtained actions. In this way, the system behaviour can be completely specified for every input in every state. Such activity can be concluded only if all requirements identified during the domain analysis have been addressed. The complete list of the domain requirements and the definition process of the language is addressed in Chapter 4.

DSTM also allows to model the requirements and add proper information on the behavioural models for implementing requirements traceability. In fact modelling activities are not only related to the modelling of the SUT since also test specifications shall be considered. Thus, a set of guidelines to support modellers in their definition has been defined. Guidelines can allow for a better understanding of the same test specifications and can enable to reuse all the experience maturated in the field of the software testing. According to this, the guidelines have been resulted in a set of Test Specification Patterns (TSPs) whose definition is addressed in the Chapter 4.

Selecting Model Checking strategy According to the Section 2.2, a wide adopted strategy for the automatic generation of test cases, is the one based on model checking techniques. In fact, generation can be enable by modelling a requirement as a property of the system. The negation of such property, allows the model checker to generate a counterexample, if the property is verified on the system. Such counterexample represents the sequence of steps composing a test case. In order to reuse an existing model checker and to avoid the development of a custom one, there is the need to find the model checking language that best

fits characteristics of the defined DSML. The choice of an existing model checker however, could lead to a possible semantic gap between the specification formalism and the concepts provided by the model checking language adopted by it. In case of semantic gap, a nested process is the overall methodology, must be followed in order to resolve such semantic gap. The mapping process is a relevant macro-activity that can be further divided in:

- **Analysis phase**, in which, target language (i.e. the one used by the chosen Model Checker) is analysed in order to identify all the semantic gaps, e.g. features of the high-level language that cannot be natively addressed by corresponding target features;
- **Mapping process** that is in charge of fill the gaps, by mapping abstract concepts of the source language in the corresponding of the target language. This mapping process is addressed in Chapter 5.
- **Development process** which is realized by model transformations, a key feature of model-driven approaches. The realization of the overall framework, named *Test Case Generator (TCG)* is addressed in Chapter 6

Generation process and integration of results On the basis of the mapping process and of the selected model checking technique, a generation process is defined. During this step, existing tools (i.e. the Model Checker) and methods are reused to generate the test cases. The process does not terminate with the generation: in fact, results shall be integrated in the overall V&V process, in order to perform further activities (e.g. coverage measurements, support to test execution, log analysis and reporting).

3.2.1 The proposed framework for test generation

The above discussed methodology, shows the process that allows to the definition of a test case generation strategy based on model-driven principles. According to such methodology, an high-level architecture, showing principal elements of the automatic test case generator here described, is discussed. In particular the input of the methodology is the SUT specification, that can be modelled by means of proper domain-specific languages, defined according to the domain analysis. A formal model derived from the high-level SUT specification is produced and is given as in input to the Model Checker for the test cases generation. The formal model is obtained by means of model transformations, according to the model-driven nature of the approach. Then a model checker is exploited for

the generation of the test cases. The outcomes (i.e. the test cases) can be then integrated in the overall V&V process. This step is realized by modelling them with a proper domain specific language. Such language should provide the same level of abstraction of the language used to model the SUT, in order to allow modellers to map the test cases on the SUT specification. The resulting high-level architecture of the framework is depicted in Figure 3.1.

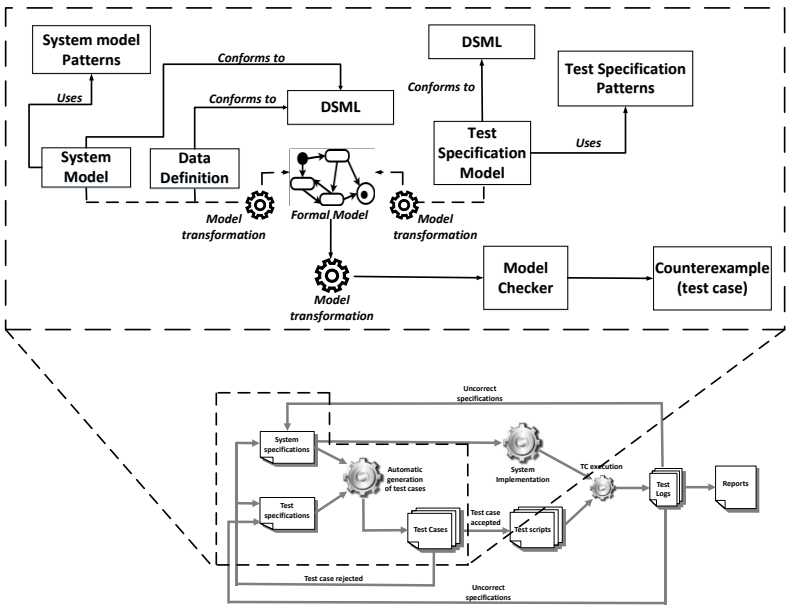


FIGURE 3.3: High-level architecture of Test Case Generation framework

The architecture in Figure 3.3, shows also how the methodology discussed in this work is integrated in the overall process of the CRYSTAL project. It in fact covers only the activities related to the test generation process. Next chapters discuss each aspect of the proposed framework. In particular, the Chapter 4 addresses the definition of domain-specific modelling languages and the guidelines recommended for the modelling of the test specifications. Chapter 5 discusses the conceptual gaps between the source language and the language of the chosen model checker. The Chapter 6 then shows the development of the framework and finally the Chapter 7 describes the application of the approach a real word systems related to the railway control systems.

Chapter 4

Domain-Specific Modelling Languages enabling test case generation

As discussed in Chapter 1, the adoption of a state-based formalism is highly recommended by international standards and industrial needs. International standards, as the CENELEC [16], explicitly recommend the usage of state-based formalisms, based on a sequential computation, to abstract the dynamic of a state-transition control system. Despite the great number of works addressing the usage of state machine and their extensions, the railway industry expressed the need for a concise formal modelling notation, able to easily capture some characteristic features of the specific domain, to be used in model-driven test automation environments. In particular, the railway domain in which the present work is collocated, keeps out the possibility of using several UML diagrams and prefers an ad-hoc formal language, developed from scratch, with the objective to be as simple as possible and as rich as needed for modelling the behaviour and the requirements of a railway control system for system testing purposes. State-based formalisms, in fact, are able to define the control structure of a system in terms of its states, possible inputs, and obtained actions in order to guarantee that the control system behaviour is completely specified for every input in every state. Thought their usage is not widely-accepted in industrial settings, Finite State Machines (FSMs) are widely used in modelling systems where control handling aspects are predominant. FSMs capture how the system moves from one state to another as resulting from the reception of an input in a given state. In order to enable a model-driven generation of test cases, new DSMLs have been defined (Figure 4.1). This work is part of a wider research activity aiming at the development of an interoperable testing environment for railway control system [10]. In this context the test generation methodology discussed, is focused on the definition of an automatic test generation strategy. With the respect to this,

state-based modelling languages have been defined, aiming at describing:

- The behaviour of an embedded critical control systems (Section 4.1)
- The generated test sequences or, in other words, the sequence of steps to be performed on a real system which prove the conformance between the system behaviour and its requirements (Section 4.2).
- Moreover, since the test generation process needs both the system model and the test specification, a set of guidelines to model such specifications has been also defined. These guidelines, named *Test Specification Patterns*, are deeply described in Section 4.2.

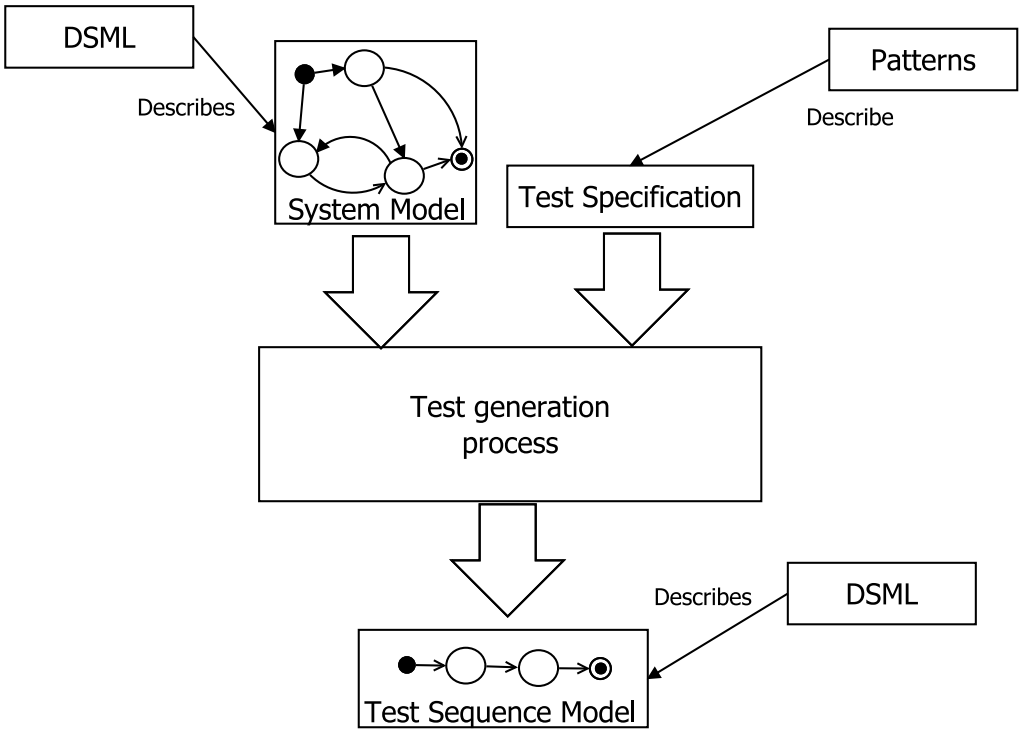


FIGURE 4.1: Languages and patterns defined for the test case generation process

4.1 DSTM: Dynamic STate Machines

Within industrial settings, modelling languages need to be as simple as possible and as rich as needed [43]. To achieve this, a modelling language should provide a set of primitive constructs to allow for a natural modelling of the considered system leading to an high-level of simplicity, without neglecting the avoidance

of ambiguities during the control system specification. However there is a unavoidable gap between the design specification and the actual implementation, which results less useful the result of the design analysis. Thus, the complexity of real industrial systems, entails the need for a formal specification framework supported by a rigorous semantics. In this way, is possible to realize the systems design according to recommendations of international standards and well-accepted formal analysis processes.

The first modelling language described in this chapter addresses the needs expressed by a railway industry within the context of the project CRYSTAL. The modelling language, named Dynamic STate Machine (DSTM) is a state-based formalism, supported by a formal syntax and a well-defined semantics.

DSTMs are an extension of Hierarchical State Machines, originally proposed by Alur et al. in [5]. The key features provided by the DSTM language can be summarized in: a novel semantics for fork and join constructs that allows for the dynamic instantiation of machines, the introduction of preemptive termination and the possibility of passing parameters to machines, at activation time. Next subsections, describe the modelling requirement whose satisfaction drove the development of the language. Then the formal syntax and some hints of the well-defined semantics are provided.

4.1.1 Domain and Modelling Requirements

The analysis of the application domain raised some relevant needs to satisfy, in order to address the modelling of a railway control system. First the demand for a state-based language, with a formal syntax and semantics. The modelling language, in fact, has to provide a high level of expressiveness and be easy to understand and use in an industrial setting.

It must provide *primitives* to express directly specific modelling issues, avoiding counter-intuitive behaviours. In brief, the language must provide primitives in order to model concurrent threads, dynamic instantiation and recursive execution of processes, parameters passing, preemptive termination, messages reception and sending. In addition it must allows for the definition of enumeration types (in order to represent different message classes and their content) and the evaluation of a single field of an enumeration variable. Logical operators must be provided too, in order to express conditions. At the same time the language must be *small*, also considering the usage of the models in the context of the test case generation process (i.e., their transformation towards a model checker).

In this context, UML2 [**UMLv2**] represents one of the most adopted solutions. In fact, UML2 provides State Machines which admit parallel execution through the usage of composite states and regions. Using UML State Machine, the fork (and join) is used in order to split (and merge) an incoming transition into two or more transitions terminating on orthogonal target vertexes (i.e., vertexes in different regions of a composite state). Recursive activation and dynamic instantiation is not admitted. However UML2 has been discarded since dynamic instantiation is only possible by combined usage of sequence, class and state machine diagrams, similarly other characteristics such as preemptive termination or passing parameters to machines would be very hard to implement without using several diagrams. Other solution have been reviewed: the Communicating Hierarchical Machines (CHMs), a variant of Statecharts, introduced for succinctness reasons. The idea behind CHMs, is to have a collection of finite state machines (modules) having nodes and boxes. A transition entering a box represents a call to one or more instances of another module. In a Statechart there is no notion of module and instance. If multiple instances of the same module are required by the specification, each instance has to be explicitly defined. On the other way the introduction of modules allows to define Recursive State Machines (RSMs) where a module can recursively call itself [**alur**]. Lets note that Recursive State Machines are not in the category of Finite State Machines. In [**LMP03**] CHMs has been extended introducing Dynamic Hierarchical Machines (DHMs) which allow the dynamic activation of machines: any DHM M_1 can send to a concurrent DHM M_2 a third DHM M_3, which starts running either in parallel with M_1 and M_2, or inside M_2, depending on contextual information. According to the above literature review, with the respect to the key issue of the *dynamic* instantiation of machines, a new language has been defined. Such new language extends the well-know formalism of the Hierarchical State Machines (HSMs) introduced by Harel [47] and formalized by Alur [4].

A HSM H is a tuple $K = \langle K1, ..., Kn \rangle$ of modules, where each module Ki has the followings elements

- N_i that is the set of Nodes;
- B_i that is the set of Boxes;
- En_i is a subset of N_i of entering nodes;
- Ex_i is a subset of N_i of exiting nodes;

- $Y_i : Bx_i \rightarrow \{1,\ldots,n\}^\star$ assigns to every box a sequence (list) of machine indexes;
- An edge relation Ei consisting of pairs (s; d) where the source s can be either a node or a return of Ki , and the destination d is either a node or a call of Ki.

The definition describes the main syntactical elements of an HSM. However, in order to satisfy the requirement, raised from the domain analysis, we extended HSMs by adding:

- Instantiation and modularity: a process may spread and handle multiple threads of execution that run concurrently. Both the cases of blocking and non-blocking call have to be modelled, as well as dynamic instantiation of more instances of the same execution thread. In addition, it should be possible to use divide-and-conquer approaches in modelling the system behaviour by defining proper sub-models and encapsulating them into machines.
- Preemptive termination: a number of different situations may occur and error handling is a vital part of the control process. Preemptive transitions, which lead to the immediate abortion of machines. Preemption (i.e., the possibility for a running machine to interrupt the execution of another machine) should be modelled.
- Parameters passing: communication between running machines and their caller is necessary and it is performed by message passing (through bounded channels) and global variables. Nevertheless, in some cases it is necessary that the calling machine provides a different value of a parameter to concurrent execution threads.
- Triggers and Conditions: reception of messages and timers expiration[1] may enable state transitions if guard conditions evaluate to true.
- Broadcast communication: it allows to model situations in which different machines have to be triggered by the same event.
- Types and Variables: simple types (boolean, integer), string and enumeration are needed. Types of messages can be modelled as enumerations of integers.

[1]Timers are not explicitly addressed in this work. For testing purposes the expiration of a timer is modelled by the reception of a special message.

Next section describes main features of DSTM by discussing its metamodel and some running example.

4.1.2 DSTM metamodel

The DSTM metamodel, depicted in Figure 4.2, has been realized with an Ecore diagram, according to the technology adopted to generate the model editor and the graphical interface [73]. The metamodel combines the both control flow elements (left-hand side) and types and data types specification (right-hand side).

The core of the depicted metamodel is the Dynamic StaTe Machine *(DSTM)*, which represents the whole specification model. A DSTM is composed of different *Machine*s, *Type*s, *Channel*s and *Variable*s. *Channel*s and *Variable*s allow for the machines communication. *Channel*s are further divided in *Internal* and *External* channels; the formers allow for asynchronous communication via a bounded size buffer the latters are used for broadcast communication from/to the environment (a message has a lifetime of only one step and can be sensed the next step after its generation). Channels scope is global, in fact they are declared at the same level of the machines. Local Channels are not allowed by the language. At the same way, DSTM provides *Global Variable*s that are instantaneously updated as consequence of a transition firing and the other concurrently executing machines can sense the updated values instantaneously (i.e., in the same step). *Global Variable*s can be viewed as a shared memory among the machines.

According to the state-based nature of DSTM, each *Machine* is composed by

- *Vertex*: a vertex is each element of the model. Machine must have at least two vertexes. *Vertexes* are abstract concepts, since different kinds can be used in a machine. According to this, DSTM provides several kinds of *Vertexes*:
  - **node**: nodes of a machine;
  - **exiting node**: exit (or final) nodes of a machine (a machine may specify more than one exit node in order to model different termination conditions);
  - **entering node**: entry nodes of a machine (a machine may specify more than one entry node in order to model different entering conditions);
  - **initial node**: default entry node of a machine (exactly one for each machine);

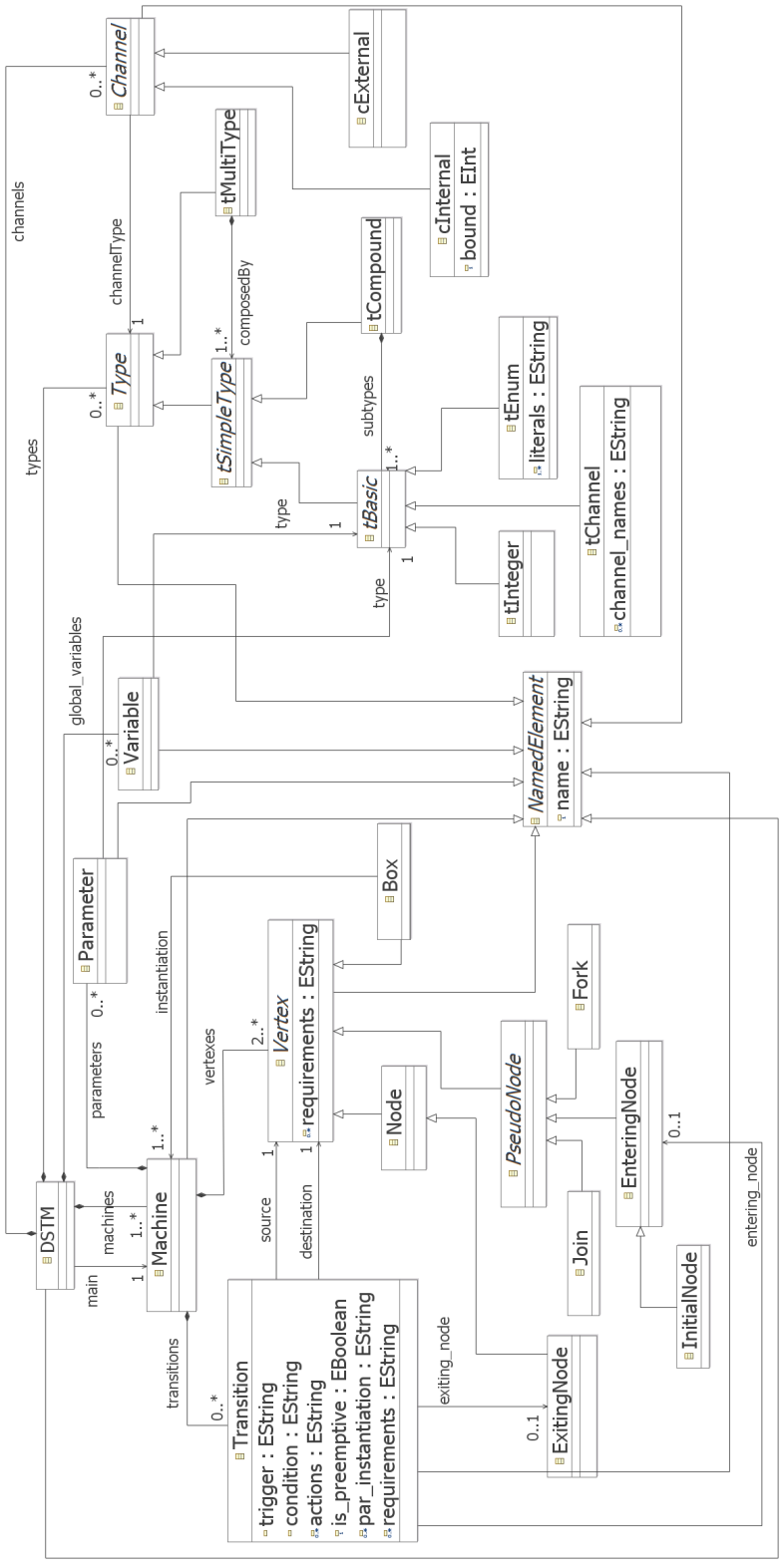


FIGURE 4.2: DSTM metamodel.

– **fork**: splits an incoming control flow into more outgoing flows; it allows for instantiating one or more processes either synchronously or

asynchronously with the currently executing process;

- **join**: merges outgoing control flows from concurrently executing processes; it synchronizes the termination of concurrently executing processes or allows to force the termination when a process is able to perform a *preemptive* termination;
- **box**: it is associated to one or more machines. A transition entering a box models the invocation of the machine(s) associated with the box, and a transition leaving a box corresponds to a return from that machine.

Each kind of vertex is realized by a related class in the metamodel; specifically the classes *Fork*, *Join*, *Entering Node* and *Initial Node* are inherited from the abstract class *PseudoNode*, which encompasses the different kinds of transient vertexes in the machine. Entering and exiting nodes define the *interface* of a machine, in particular the first (pseudo) node of a process is either the initial node or an entering node (when explicitly expressed in the higher level box instantiating the machine); on the contrary, the last state of a machine is an exiting node (which can also be used as returning condition). The association between *Box* and *Machine* indicates the set of machines, which are concurrently instantiated when entering the *Box*.

- *Parameter*: to address the need for the machine parametrization, a machine can own a *Parameter*. *Parameter*s can be used in the decoration of transitions as parametric names for channels or variables, they are actualized when the machine is instantiated.

- *Transition*: a transition represent a connection between two vertexes. In fact the related classes *Vertex* and *Transition* are connected by two association indicating both the source and the target vertex of a considered transition. The class *Transition* is characterized by the following attributes: *trigger*, *condition* and *actions* specify the decoration of the transition; *is_preemptive* defines if a transition, entering a join pseudonode, is preemptive or not, and *par_instantiation* is used to specify the substitution for the parameters, when a machine is called and instantiated. Two associations from the *Transition* class allow to specify a *Entering Node* (resp. a *Exiting Node*) for the transitions, which enters (resp. exits from) *Box*es. Moreover, as the *Vertex* class, *Transition* provides the attribute *requirements* that allows for the annotation of system requirement on the model. Requirements are treated as strings,

since the above attribute records only identifiers of requirements. Constraints are defined on *Transition*, not represented in Figure. 4.2 for readability purpose, in order to allow only the following kinds of *Transition*s as follows:

- **implicit transition**: a transition from an entering pseudo-node to one node of a machine (only one of such transitions is allowed from the same entering node); the specification of trigger and condition is not allowed;
- **internal transition**: a transition between two nodes of the same machine;
- **internal entering fork**: a transition from a node to a fork pseudonode of a machine;
- **internal asynchronous fork**: an asynchronous transition from a fork pseudonode to a node of a machine; the specification of trigger and condition is not allowed;
- **internal entering join**: an internal transition from a node to a join pseudo-node of a machine; the specification of actions is not allowed;
- **internal exiting join**: a transition from a join pseudonode to a node of a machine; the specification of trigger and condition is not allowed;
- **return by default**: a transition exiting from a box; the specification of trigger and condition is not allowed and, if it enters into a join pseudonode, actions are not allowed either;
- **return by exiting**: a transition exiting from a box, which specifies a specific exiting node of the instantiated machine; the specification of trigger and condition is not allowed and, if it enters into a join pseudonode, actions are also not allowed;
- **return by interrupt**: a transition exiting from a box via an interrupt trigger event; the specification of the trigger is required, while the specification of condition is not allowed and, if it enters into a join pseudonode, actions are not allowed either;
- **call by default**: a transition entering into a box (instantiation via the initial node); if the source is a fork or a join pseudonode then the specification of trigger and condition is not allowed;
- **call by entering**: a transition entering into a box, which specifies a specific entering node of the instantiated machine; if the source is a fork

or a join pseudonode then the specification of trigger and condition is not allowed.

Note that instantiation of parameters is allowed only for *call by default* and *call by entering* transitions, namely the only kinds of transitions instantiating machines. Furthermore, in case of *call by entering* transitions, the entered boxes can instantiate only one machine, while in case of *call by default* transitions the entered boxes can instantiate multiple machines.

With the respect to the data-flow, DSTM allows for the definition of own datatypes. The *Type* class, inheriting from the *NamedElement* is composed by *tSimpleType* and *tMultiType*. A *tSimpleType* can be further divided into basic types (*tBasic*) or a compound types (*tCompound*). Basic types are *tInteger*s (the usual integer numbers), *tChannel*s, (channel names) and enumeration types *tEnum*s. A compound type (*tCompound*) is a structured type composed of *tBasic* types (i.e., a record of basic types). Notice that the same basic type may occur more than once in the compound type. *tMultiType* can be composed by different simple types. Global *Variable*s can only be typed as basic types. According to the metamodel (4.2), *Channel* can be either a *cInternal* or a *cExternal* type. With the respect to a channels, is possible to distinguish between its type, which is the basic type *tChannel*, and the type of messages conveyed by the channel. Messages conveyed by a channel (see the association between *Channel* and *Type*) can have any instance of *tBasic*, *tCompound* or *tmulti-type* as type. Note that all the containment relations are bidirectional: a contained element has a pointer to its container; in this way, for example, each *Vertex* and each *Transition* has visibility of the container *Machine*.

The aim of DSTM is the modelling of the internal behaviour of embedded critical control systems. Below some running examples, representing abstract machines, are provided, in order to show the expressive power of the language.

In Figure 4.3 a machine $M0$ is represented. $M0$ has the set of nodes (depicted as round boxes) $\{a0, b0, c0, d0,$
$e0, f0\}$, a box $bx01$ (depicted as a rectangle), an initial node (depicted as a filled bullet), an entering node $en01$ (depicted as a circle) and a set of exiting nodes $\{ex01, ex02, ex03\}$ (depicted as crossed circles). Transition $t01$ has trigger $\tau 01$, condition $\gamma 01$ (between square brackets) and action $\alpha 01$. The transition from the initial node to node $a0$ is an implicit transition; the transition $t01$ is internal, the transition $t03$ is a call by default transition (it leads to a box); the transition $t04$ is a call by entering transition (it leads to an entering node of $bx01$); the transition $t05$ is a return by default transition (without trigger); the transition $t06$ is a return

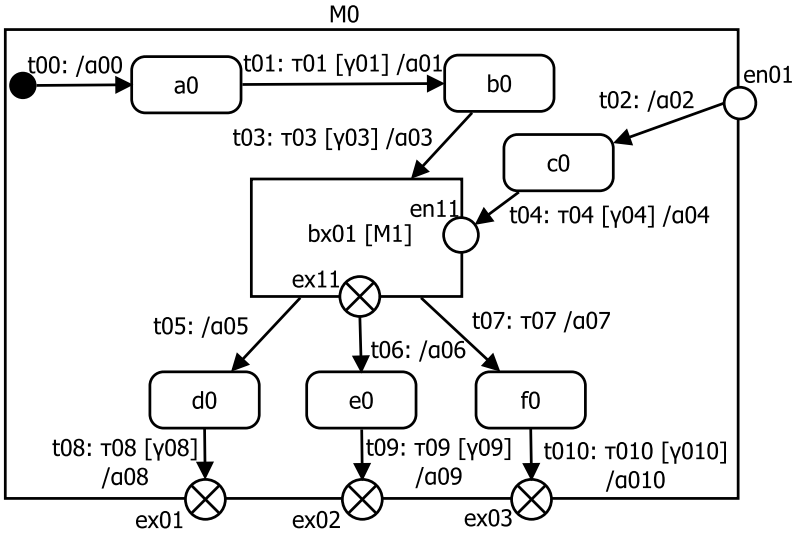


FIGURE 4.3: Example 1

by exiting transition (leading from an exit node of $bx01$); the transition $t07$ is a return by interrupt transition (with trigger). The firing of $t03$ or $t04$ instantiates the machine $M1$ depicted in Figure 4.4. The machine $M1$ is proposed to exemplify fork and join transitions. In particular, the transition $t11$ is an internal fork transition (leading to a horizontal bar representing the fork pseudonode); the transition $t12$ is an asynchronous fork (it leads to a node); the transition $t13$ and $t14$ are both call by default transitions; the transition $t16$ is an internal entering join (taking from a node to a join pseudonode, represented as an horizontal bar); the transitions $t17$ and $t18$ are return by default transitions joining with transition $t16$; transition $t17$ leads to a crossed circle overlapping the join bar qualifying the transition $t17$ as a preemptive transition.

In Figures 4.5, a machine $M4$ is shown, which exemplifies the (dynamic) instantiation of a parametric machine $M5$, associated with the box $bx41$. The internal entering fork transition $t40$ is triggered when a message is received along channel c. When the transition fires, the received message on c is stored in the variable x and counter $cont$ is incremented by 1. The call by default transition $t42$ instantiates the parametric machine $M5$, associated with parameter $Par1$. At call time, parameter $Par1$ is substituted with the current value of x (i.e., a name of a channel) previously received on channel c.

Transition $t41$ is an internal asynchronous fork transition (its target is a node), which, combined with $t40$, determines a loop containing a fork, which is not followed by a join. This pattern of transitions, which is usually not allowed in standard state transition diagrams, allows to dynamically generate a (potentially unbounded) number of instances of the parametric machine $M5$, whose parameters are assigned to values by possibly different parameter instantiations. In the

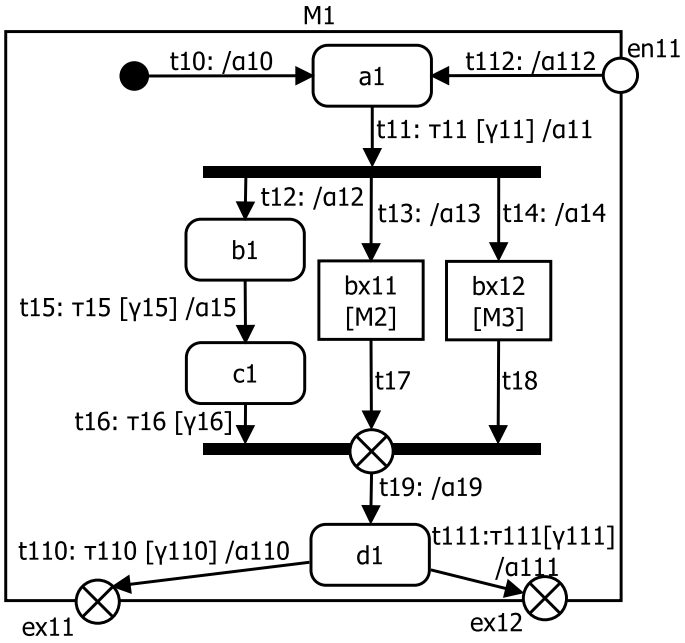


FIGURE 4.4: Example 2

example of Figure 4.5, each call to box $bx41$ executed by transition $t42$ may substitute a different value to $Par1$, depending on the current value of variable x, which is assigned by transition $t40$ on entering the fork node.

Machine $M5$, depicted in Figure 4.6, can perform some local activities sending a value along the channel name associated to $Par1$ at instantiation time, or can instantiate machines $M6$ and $M7$ in order to complete its activities. In this case, at instantiation time, Machine $M5$ assigns the current value of $Par1$ to $Par2$ of machine $M6$, and to $Par3$ of machine $M7$.

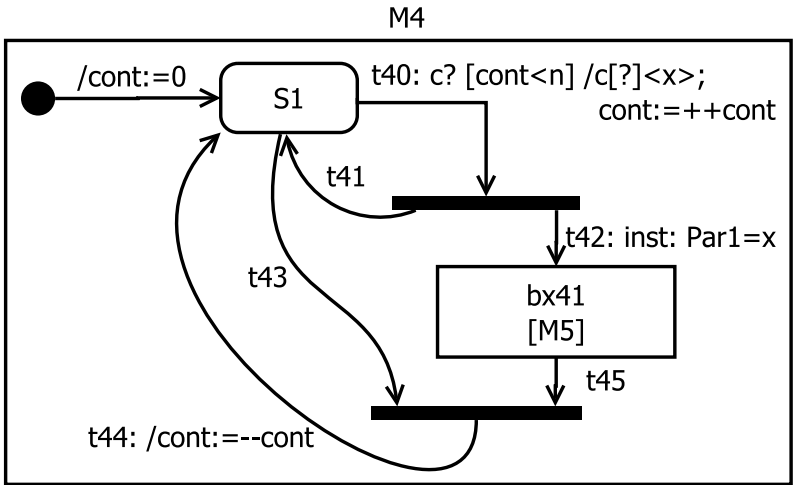


FIGURE 4.5: Example 3

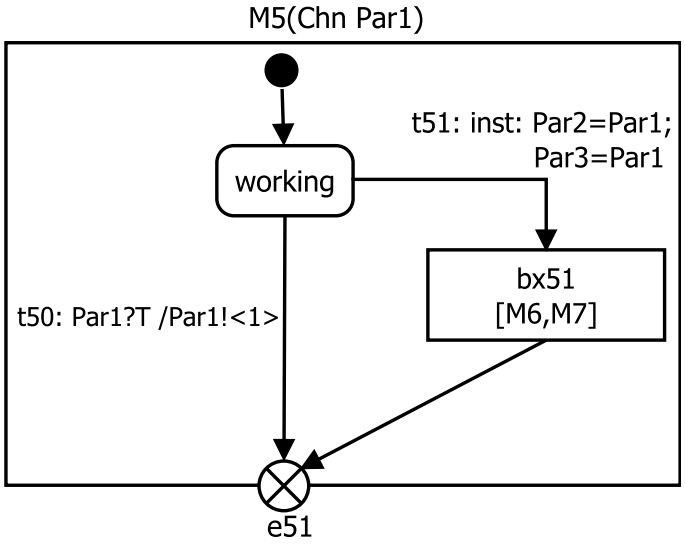


FIGURE 4.6: Example 4

4.1.3 Formal syntax

As discussed in the chapter introduction, a modelling language for embedded critical control system, needs to be supported by a formal syntax and a well-defined semantics, in order to avoid ambiguousness during the system modelling activities. This subsection, provides the formal syntax, defined for DSTM, both for the control and the data flow part of the language.

As stated in Subsection 4.1.2, DSTM provides simple and multi types. Simple types, are the composition of:

- *Basic types*: the set of basic type, $\mathbf{BT} = \{\mathbf{Int}, BT_1, \ldots, BT_k, \mathbf{Chn}\}$, provides a type **Int** for the integers, a type **Chn** for channels names, and a set of user defined enumeration types $BT_1, \ldots, BT_k$. The domain $\mathcal{D}(\mathbf{Int})$ of integers is $\mathbb{Z}$, the domain $\mathcal{D}(\mathbf{Chn})$ is a set of channel names $C = \{c_1, \ldots, c_h\}$, and the domain $\mathcal{D}(BT_i)$ of enumeration type BT_i is a set of labels $\{l^i_1, \ldots, l^i_{s_i}\}$. The union of the domains of the basic type is denoted as $\mathcal{D}(\mathbf{BT})$. For each basic type a default value is provided in the corresponding domain. For this purpose the language introduce a function $default : \mathbf{BT} \rightarrow \mathcal{D}(\mathbf{BT})$.

- *Compound types*: A compound type $CT = (BT_{j_1}, \cdots, BT_{j_k})$ is defined as a tuple of basic types BT_{j_i}, for $i \in [1, \ldots, k]$. Each element of the domain $\mathcal{D}(CT)$ of CT is a tuple of the form $\langle d_1, \ldots, d_k \rangle$, where $d_z \in \mathcal{D}(BT_{j_z})$ for $z \in [1, \ldots, k]$. In other words, the domain $\mathcal{D}(\mathbf{CT})$ is the set of tuples of elements of the basic types composing it.

Multi-types, instead, are the composition, by means of a union operator, of simple types. A multi-type $MT = \{ST_{j_1}, \cdots, ST_{j_z}\}$ groups together a set of simple

types, thus allowing the domain of the resulting type for elements belonging to the domain of any of the simple types ST_{j_i}, with $z \in [1, \ldots, k]$. Hence, the domain of MT corresponds to the union of the domains of the composing types. The set of all type is denoted as $\mathbf{T}$.

The communication among the machines, is allowed by *Global Variables* and *Channels*. Each channel name c is associated with a concrete channel $\widehat{c}$. The set of concrete channels is denoted by $\widehat{C}$. Channels allow for asynchronous communication both with the environment and among internal components via a bounded buffer. Therefore, there is a function $bd : C \to N$ assigning the bound of the buffer associated with any channel name. Each concrete channel has an associated type, either a simple type or a multi-type. Formally, $type : \widehat{C} \to \mathbf{T}$. The type of a concrete channel is actually the type of the message conveyed by the concrete channel. The domain of the contents of a concrete channel $\widehat{c} \in \widehat{C}$ (i.e. th contents of the bounded length buffer associated with the channel) is the set of sequences of length at most $bd(c)$ of elements of its associated type, namely $\mathcal{D}(\widehat{c}) = (\mathcal{D}(type(\widehat{c})))^{\leq bd(c)}$. In order to keep track of the type of the corresponding concrete channel, the type $\mathbf{Chn}$ of the channels names can be further decorated with the type of the named channel. Given a channel $\hat{c}$ of type T, we denote with $\mathbf{Chn}[T]$ the decorated type of the name c of $\hat{c}$. We denote with $\mathbf{BT}^+$ the set of *decorated basic type*, containing type $\mathbf{Int}$, the enumeration types BT_i and all the decorated types for channel names of the form $\mathbf{Chn}[T]$, for $T \in \mathbf{T}$.

As stated above, Channels, can be further divided in *internal* and *external*. channels. Internal channels are used for communication among internal components and the set of their names is $C_I \subseteq C$. External channels are used to interact with the environment and the set of their names is $C_E \subseteq C$.

The set of internal and external channels are mutually disjoint and form a partition of C ($C = C_I \cup C_E$) In addition, the bound of external channels is restricted to 1, i.e., $bd(c) = 1$, for every $c \in C_E$.

With the respect to the variable, let X be the set of (global) variables and P a set of parameters. A typing function $type : X \cup P \to \mathbf{BT}^+$ assigns a decorated basic type to each variable and parameter: variables and parameters of compound types are not allowed. For each variable parameter x (parameter p), the domain of x (p), in symbols $\mathcal{D}(x)$ ($\mathcal{D}(p)$), coincides with the domain $\mathcal{D}(type(x))$ ($\mathcal{D}(type(p))$) of its type.

In the following we introduce the syntax of triggers, guards and actions. Preliminary, we define the notion of terms, a notion which occurs both in definition of guards and actions. Terms are freely constructed over variables, parameters, domains of basic types (i.e., basic types literals) and a set Op_1 of unary operators

(e.g., ++, −−, ...) and a set Op_2 of binary operators (like, e.g., +, −, ∗, /, ...).

More formally, the set of terms over the parameters in P, in symbols Trm_P, is defined as follows

$$trm ::= x \mid p \mid T_i :: l \mid \mathbf{Chn} :: c \mid d \mid \mathit{len}(c) \mid \boxdot_1 trm \mid trm \ \boxdot_2 \ trm$$

where $x \in X$, $p \in P$, $l \in \mathcal{D}(T_i)$, $c \in \mathcal{D}(\mathbf{Chn})$, $d \in \mathcal{D}(\mathbf{Int})$, $\boxdot_1 \in Op_1$ and $\boxdot_2 \in Op_2$. Notice that an enumerative literal is prefixed by its type, and a channel name is prefixed by type **Chn**. The term $\mathit{len}(c)$ denotes the actual length of the buffer associated with channel c.

In order to have admissible term constructions using unary and binary operators (which, for simplicity, are assumed to have only integer operands), each term has a type. Terms with undefined type are not admissible. The type $Type(trm)$ of a term trm is defined as follows:

- $Type(x) = type(x)$ and $Type(p) = type(p)$;
- $Type(T_i :: l) = T_i$, $Type(\mathbf{Chn} :: c) = \mathbf{Chn}$, $Type(d) = Type(len(c)) = \mathbf{Int}$;
- $Type(trm_1 \boxdot_2 trm_2) = Type(\boxdot_1 trm_1) = \mathbf{Int}$, if trm_1 and trm_2 are of type **Int**, undefined otherwise.

A term is *well-typed* if its type is defined. In the following an assumption stating that all terms under consideration are well-typed is made.

The definition of action as a sequence (possible empty) of atomic actions is provided. An atomic action can take one of the following forms: assignment of a term to a variable; the sending of a tuple of values over a channel ($\gamma\,!\,\langle trm_1, \ldots, trm_{k_\gamma}\rangle$); reading (and removing) a tuple of values from a channel and storing them into variables ($\gamma\,?\,\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$); reading a tuple of values from a channel and storing them into variables without altering the content of the channel ($\gamma\,[?]\,\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$). Each element of the tuple $\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$ is either a variable, to which the corresponding element within the message is assigned, or the don't-care symbol _ to skip the corresponding element within the message. The set $\mathcal{A}_P$ of atomic actions over the set of parameters P is, therefore, defined as follows:

$$act ::= x := trm \mid \gamma\,!\,\langle trm_1, \ldots, trm_{k_\gamma}\rangle \mid \gamma\,?\,\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle \mid \gamma\,[?]\,\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$$

where $x \in X$, $type(x) = Type(trm)$, $\gamma \in P \cup C$, $type(\gamma) = \mathbf{Chn}$, $\eta_i \in X \cup \{\_\}$ and if $\gamma \in C$, then $type(\widehat{\gamma}) = (Type(trm_1), \ldots, Type(trm_{k_c}))$. Notice that, since parameters can be assigned to a channel name only at runtime no static type-checking can

be defined for these cases. In addition, parameters cannot be assigned a value by an action.

An action is a sequence of atomic actions, with ϵ denoting the empty sequence.

$$\alpha ::= \epsilon \mid a;\alpha \qquad \text{with } a \in act.$$

Now is possible to defined the set of triggers. A trigger over P is a boolean expression freely constructed from a set of events (an event is essentially the availability of a message on a channel) by means of standard connectives using parameters in the set P. The definition is as follows:

$$\xi ::= \tau \mid \gamma? \mid \gamma?T \mid \xi \wedge \xi \mid \xi \vee \xi \mid \neg\xi \qquad \text{with } \gamma \in C \cup P$$

where τ is the silent trigger (no event is required for triggering), while a trigger of the form $\gamma?$ (resp., $\gamma?T$) signals the presence of a message (resp, a message of type T) in channel γ. An event of the form $\gamma?$ is sensed when a message is conveyed in channel γ, while the event $\gamma?T$ allows to sense whether a message of a specific type is present in the channel, in particular when the type of the channel is a multi-type.

Lets consider now the syntax of guards. A guard over a set of parameters P is a boolean expression freely constructed from a set of atomic guards by means of Boolean connectives. Lets consider a concrete channel $\widehat{c}$ of (simple) type $(BT_1, \ldots, BT_k)$, atomic guards of the form $c[?\top]$ and $c[?\bot]$ check whether the (buffer of) channel $\widehat{c}$ is full or empty, respectively. Moreover, if $trm_1, \ldots, trm_k \in Trm \cup \{\_\}$ are terms, an atomic guard of the form $c[?\langle trm_1, \ldots, trm_k\rangle]$ checks the content of the message contained in the head of (the buffer of) the channel, namely, for all $i \in [1, \ldots, k]$, either $trm_i = \_$ (the i-th component of the structured message is ignored) or the i-th component of the message at the head of channel $\widehat{c}$ is equal to the value of term trm_i, which must be of type BT_i. In addition, an atomic guard can compare the values of two terms with respect to standard equality and ordering relations (if ordering relations are considered, terms are of integer type).

The formal syntax of guards is as follows:

$$\phi ::= True \mid \gamma[?\top] \mid \gamma[?\bot] \mid \gamma[?\langle trm_1, \ldots trm_{k_c}\rangle] \mid trm \odot trm \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$$

where $\gamma \in C \cup P$ and $\odot \in \{\geq, \leq, =\}$.

Let Ξ_P, Φ_P and $\mathcal{A}_P$ be the syntactic categories of *triggers*, *conditions* and *actions*,

respectively, over the set P of parameters. An element of Ξ_P (resp. Φ_P, $\mathcal{A}_P$) is a trigger (resp. condition, action) expression.

Finally, the syntactic category of parameter substitutions has to be introduced. A *parameter substitution* function over $\bar{P} \subseteq P$, $\ell_{\bar{P}} : P \rightharpoonup Trm_{\bar{P}}$, is a partial function whose domain is the set of parameters P and the range is the set of a terms over $\bar{P}$. For instance, the parameter substitution function associated with transition $t51$ in Figure 4.6 assigns the (parametric) term $Par1$ to both parameters $Par2$ and $Par3$. In this case, the domain P of the parameter substitution function contains both $Par2$ and $Par3$, while the range is the set of terms $Trm_{\bar{P}}$, where $Par1 \subseteq \bar{P}$. Let $\Upsilon_{\bar{P}}$ denote the set of possible parameter substitutions over $\bar{P}$.

The definition given above, allow to formally define the notion of Dynamic State Machine:

Definition 1 (Dynamic State Machine)**.** *A DSTM D is a tuple $\langle M_1, \ldots, M_n, X, C, P \rangle$, where:*

- *X (resp. P and C) is a (finite) set of variables (resp. parameters and channels);*
- *M_1 is the* initial machine *over X, C (no parameters are allows in the initial machine);*
- *M_i, with $i \in \{2, ..., n\}$, is a machine over X, C and P of the form*

$$\langle P_i, N_i, En_i, df_i, Ex_i, Bx_i, Y_i, Fk_i, Jn_i, \Lambda_i \rangle \text{ , where:}$$

  - *$P_i \subseteq P$ is the local set of parameters of machine i;*
  - *N_i is a (finite) set of nodes and $Ex_i \subseteq N_i$ is a set of exiting nodes;*
  - *En_i is a (finite) set of entering nodes;*
  - *$df_i \in En_i$ is the initial node (default);*
  - *Bx_i is a (finite) set of boxes;*
  - *$Y_i : Bx_i \rightarrow \{1, \ldots, n\}^\star$ assigns to every box a sequence (list) of machine indexes;*
  - *Fk_i is a (finite) set of fork (pseudo) nodes;*
  - *Jn_i is a (finite) set of join (pseudo) nodes;*
  - *$\Lambda_i = \langle T_i, \mathrm{Src}_i, \mathrm{Dec}_i, \mathrm{Trg}_i, \mathrm{Inst}_i \rangle$ is the structure defining the set of transitions of M_i, where:*
    - *T_i is a (finite) set of transition labels;*

* $\mathrm{Src}_i : T_i \to Source_i$ *associates a source to each transition label, where* $Source_i = (N_i \setminus Ex_i) \cup En_i \cup Bx_i \cup (Bx_i \times Ex(D)) \cup Fk_i \cup (Fk_i \times \{\downarrow\}) \cup Jn_i$ *and* $Ex(D) = \bigcup_{1 \le j \le n} Ex_j$*;*
* $\mathrm{Dec}_i : T_i \to \Xi_{P_i} \times \Phi_{P_i} \times \mathcal{A}_{P_i}$ *associates each transition with its decoration, namely the trigger, condition and action of* Λ_i*;*
* $\mathrm{Trg}_i : T_i \to Target_i$ *assigns a target to each transition, where* $Target_i = N_i \cup Bx_i \cup (Bx_i \times En(D)) \cup Fk_i \cup Jn_i \cup (Jn_i \times \{\otimes\})$ *and* $En(D) = \bigcup_{1 \le j \le n} En_j$*;*
* $\mathrm{Inst}_i : T_i \rightharpoonup (\Upsilon_{P_i})^*$ *is a partial function assigning a sequence of parameter substitutions over* P_i *to a transition.*

Notice that all the elements of a machine instantiate a corresponding class in the class diagram of Figure 4.2. In particular, N_i (resp., Ex_i, En_i, Bx_i, Fk_i, Jn_i, T_i) contains instances of the class *Node* (resp., *ExitingNode*, *EnteringNode*, *Box*, *Fork*, *Join* and *Transition*) associated with the instance M_i of the class *Machine*. Moreover, the pairing of Fk_i with symbol $\downarrow$ (resp. Jn_i with symbol $\otimes$) is used to qualify a entering fork as asynchronous (resp. an exiting join as preemptive).

As an example, lets consider the DSTM $D = \langle M_4, M_5, M_6, M_7, X, C, P \rangle$ in Figure 4.5 with $X = \{x, cont\}$, $C = \{c, c1, c2\}$ and $P = \{Par1, Par2, Par3\}$ (where $c1$ and $c2$ are two possible channel names, which are supposed to be sent over channel c and assigned to variable x by transition $t40$).

$$M_4 = \langle P_4, N_4, En_4, df_4, Ex_4, Bx_4, Y_4, Fk_4, Jn_4, \Lambda_4 \rangle \text{ , where:}$$

- $P_4 = \{Par1\}$; $N_4 = \{S1\}$; $Ex_4 = \varnothing$; $En_4 = \{df_4\}$; $Bx_4 = \{bx41\}$; $Y_4 = \{(bx41, 5)\}$; $Fk_4 = \{fk\}$; $Jn_4 = \{jn\}$;
- $\Lambda_4 = \langle T_4, \mathrm{Src}_4, \mathrm{Dec}_4, \mathrm{Trg}_4, \mathrm{Inst}_4 \rangle$ is:
  - $T_4 = \{t_d, t_{40}, t_{41}, t_{42}, t_{43}, t_{44}, t_{45}\}$;
  - $\mathrm{Src}_4 = \{(t_d, df_4), (t_{40}, S1), (t_{41}, fk), (t_{42}, fk), (t_{43}, S1), (t_{44}, jn), (t_{45}, bx41)\}$;
  - $\mathrm{Dec}_4 = \{(t_d, (\tau, True, cont := 0)), (t_{40}, (c?, cont < n, c[?]\langle x \rangle)),$ $(t_{41}, (\tau, True, \epsilon)), (t_{42}, (\tau, True, \epsilon)),$ $(t_{43}, (\tau, True, \epsilon)), (t_{44}, (\tau, True, cont := -cont)), (t_{45}, (\tau, True, \epsilon))\}$;
  - $\mathrm{Trg}_4 = \{(t_d, S1), (t_{40}, fk), ((t_{41}, \downarrow), S1), (t_{42}, bx1), (t_{43}, jn), (t_{44}, S1), (t_{45}, jn)\}$;
  - $\mathrm{Inst}_4 := \{(t_{42}, \{(Par1, x)\})\}$.

The definition of M_5 can be given analogously, while M_6 and M_7 are left unspecified and not reported in the example.

As shown in the metamodel, transitions allowed in a DSTM belong to predefined typologies, based on the types of their sources, targets and decorations. Specific typologies do not allow for triggers, actions or conditions in their decoration. To this end, a machine M_i has *well-formed transitions* if each transition $t \in T_i$ complies to one of the following forms (all the constraints described in the metamodel apply):

implicit transition: a transition from an entering pseudonode to a node:

- the source $\mathrm{Src}_i(t) \in En_i$, the target $\mathrm{Trg}_i(t) \in N_i$;
- the decoration $\mathrm{Dec}_i(t) = \langle \tau, \mathit{True}, \alpha \rangle$, with $\alpha \in \mathcal{A}_{P_i}$;
- $\mathrm{Inst}(t) = \epsilon$ (no parameter instantiation is allowed)

internal transition: a transition between two nodes of a machine:

- $\mathrm{Src}_i(t) \in N_i$ and $\mathrm{Trg}_i(t) \in N_i$;
- $\mathrm{Inst}(t) = \epsilon$;

internal entering fork: a transition from a node to a fork pseudonode of a machine:

- $\mathrm{Src}_i(t) \in N_i$ and $\mathrm{Trg}_i(t) \in Fk_i$;
- $\mathrm{Inst}(t) = \epsilon$;

internal asynchronous fork: an asynchronous transition from a fork pseudonode to a node of a machine

- $\mathrm{Src}_i(t) \in (Fk_i \times \{\downarrow\})$, $\mathrm{Trg}_i(t) \in N_i$;
- $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \alpha)$, with $\alpha \in \mathcal{A}_{P_i}$;
- $\mathrm{Inst}(t) = \epsilon$;

internal entering join: an internal transition from a node to a join pseudonode of a machine:

- $\mathrm{Src}_i(t) \in N_i$, $\mathrm{Trg}_i(t) \in Jn_i \cup (Jn_i \times \{\otimes\})$;
- $\mathrm{Dec}_i(t) = (\xi, \phi, \epsilon)$, with $\xi \in \Xi$ and $\phi \in \Phi_{P_i}$;
- $\mathrm{Inst}(t) = \epsilon$;

internal exiting join: a transition from a join pseudonode to a node of a machine:

- $\mathrm{Src}_i(t) \in Jn_i$, $\mathrm{Trg}_i(t) \in N_i$;

- $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \alpha)$, with $\alpha \in \mathcal{A}_{P_i}$;
- $\mathrm{Inst}(t) = \epsilon$;

return by default: a transition exiting from a box:

- $\mathrm{Src}_i(t) \in \mathit{Bx}_i$ and $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \alpha)$, with $\alpha \in \mathcal{A}_{P_i}$;
- if $\mathrm{Trg}_i(t) \in (\mathit{Jn}_i \cup (\mathit{Jn}_i \times \{\otimes\}))$, then $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \epsilon)$;
- $\mathrm{Inst}(t) = \epsilon$;

return by exiting: a transition exiting from a box, which specifies a specific exiting node of the instantiated machine:

- $\mathrm{Src}_i(t)$ is of the form (bx, ex), where $bx \in \mathit{Bx}_i$ and $ex \in \mathit{Ex}_j$ with $j = \mathrm{Y}_i(bx)$;
- $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \alpha)$, with $\alpha \in \mathcal{A}_{P_i}$;
- if $\mathrm{Trg}_i(t) \in (\mathit{Jn}_i \cup (\mathit{Jn}_i \times \{\otimes\}))$, then $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \epsilon)$;
- $\mathrm{Inst}(t) = \epsilon$;

return by interrupt: a transition exiting from a box via an interrupt trigger event:

- $\mathrm{Src}_i(t) \in \mathit{Bx}_i$ and $\mathrm{Dec}_i(t) = (\xi, \mathit{True}, \alpha)$, with $\xi \in \Xi_{P_i} \setminus \{\tau\}$, and $\alpha \in \mathcal{A}_{P_i}$;
- if $\mathrm{Trg}_i(t) \in (\mathit{Jn}_i \times \{\otimes\})$, then $\mathrm{Dec}_i(t) = (\xi, \mathit{True}, \epsilon)$;
- $\mathrm{Inst}(t) = \epsilon$;

call by default: a transition entering into a box (instantiation via the initial node):

- $\mathrm{Trg}_i(t) \in \mathit{Bx}_i$;
- if $\mathrm{Src}_i(t) \in \mathit{Fk}_i$, then $Y_i(\mathrm{Trg}_i(t)) = j$ for some $j \in \{1, ..., n\}$ $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \alpha)$, with $\alpha \in \mathcal{A}_{P_i}$;
- if $\mathrm{Src}_i(t) \in \mathit{Jn}_i$, then $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \alpha)$, with $\alpha \in \mathcal{A}_{P_i}$;
- if $Y_i(\mathrm{Trg}_i(t)) = j_1 \cdots j_s$, then $\mathrm{Inst}(t) = \ell_1 \cdots \ell_s$ for some parameter substitutions $\ell_1, \ldots, \ell_s \in \Upsilon_{P_i}$ and, for all $i \in [1, \ldots, s]$, ℓ_i is defined on all the parameters in P_{j_i};

call by entering: a transition entering into a box, which specifies a specific entering node of the instantiated machine:

- $\mathrm{Trg}_i(t)$ is of the form (bx, en), where $bx \in \mathit{Bx}_i$ and $en \in \mathit{En}_j$ with $j = Y_i(bx)$;
- if $\mathrm{Src}_i(t) \in \mathit{Fk}_i \cup \mathit{Jn}_i$, then $\mathrm{Dec}_i(t) = (\tau, \mathit{True}, \alpha)$, with $\alpha \in \mathcal{A}_{P_i}$;

- $\mathrm{Inst}(t) = \ell$ for some $\ell \in \Upsilon_{P_i}$ defined on all the parameters in P_j.

Note that every transition entering into a join pseudonode cannot perform actions (the decoration allows only the empty action). Actions associated with a join transition are allowed only for the transition exiting from the corresponding join pseudonode.

Finally, some restrictions in the use of fork and join transitions are introduced, in order to guarantee that at each time instant there is at most one node in which the control of a machine can be located. The following well formedness constraints enforce a correspondence between join pseudonodes and fork pseudonodes, requiring the synchronization of all the processes activated by the corresponding fork transition. The correspondence from join to fork is total (i.e., every join has a corresponding fork) but neither onto nor injective. Indeed, a fork may have no corresponding join (synchronization is not mandatory) or may have more than one corresponding join (multiple synchronization forms). More precisely, a machine M_i is *well-formed* if all its transitions are well formed and the following constraints are satisfied:

1. If a box $bx \in Bx_i$ is the target of either a call by entering or a call by default transition having as source a fork pseudonode, then
   - there is no other transition entering in bx;
   - each transition exiting from bx_i (return by default, return by exiting or return by interrupt) has a join pseudonode as target.
2. Each join pseudonode $jn \in Jn_i$ there is a single corresponding fork pseudonode $fk \in Fk_i$ and
   - if there is an asynchronous fork transition exiting from fk, then there is an internal join transition entering into jn;
   - if a box $bx \in Bx_i$ is the target of a call transition (call by entering or call by default) exiting from fk, then either there is no exiting transition from bx or there is at least a return transitions (return by default, return by exiting or return by interrupt) from bx to jn.

The first constraint ensures that a box called by a fork transition cannot be called in any other way and can only have exiting transitions leading to a join. The second constraint enforces the correspondence mentioned above between a join pseudonode and some fork pseudonode. It also requires that a machine M performing an asynchronous fork, which gives the control back to the M, must

participate to all the possible join in correspondence with that fork, so as to ensure that a single control state of M results from the join. Moreover, a box bx called by a fork either participates with an exit transition in all the join corresponding to that fork, or it never exists.

For eample, the machine in Figure 4.5 is well formed. The transitions are all well-formed and, in addition, the fork and join pseudonodes are in correspondence: the asynchronous fork transition $t41$ corresponds to the internal join transition $t43$ and and the entering by default transition $t42$ is in correspondence with the exiting by default transition $t44$.

4.1.4 Formal semantics

To define the formal semantics of a DSML, several approaches can be followed. According to aim of define a framework for the automatic generation of language-based artefact, two main approaches are adopted in literature (BRYANT). The first is the translation semantics in which the abstract syntax of the considered DSML is mapped in the concrete syntax of the target language with a well-defined semantics. This method, despite allows to reuse the experience of a well-know and used existing language, implies two main problems: first the different level of abstraction of the source and target language, and second its difficult to drawback the outcomes of the target language to the level of the source language. The second approach is to insert the semantics inside the abstract syntax of the DSTM. Two relevant examples are Kermeta and several UML-based DSMLs. Kermeta [57], in fact, allows to define operation on the concepts of the metamodel. The generation is than supported by tool related to the Eclipse Framework. With the respect to UML, instead, there are some attempts to insert semantics, most of them relying on the introduction of constraints written using Object Constraint Language (OCL) [18]. This work focuses on the first approach. In fact, in order to automatically generate test cases model checking capabilities to generate counterexamples have been exploited. This means that a language equipped with a well-defined semantics is introduced. However, in order to address the issues related to such approaches, a formal semantics, supporting the defined domain-specific language, has been defined. The formal semantics, in fact, allow to constraint possible behaviours of DSTM machines, in order to avoid ambiguous ones.
Thus, the evolution of DSTM is described by means of Labelled Transition Systems (LTS). An LTS is a 4-tuple $L = \langle S, \Sigma, \Delta, S_0 \rangle$, where:

- S is a non-empty set of states;

- Σ is a non-empty alphabet of labels;
- Δ is a transition relation, i.e., a subset of $S \times \Sigma \times S$;
- $S_0 \subseteq S$ is a set of initial states.

With reference to a DSTM D (see Definition 1), $s \in S$ represents the *current* state of D including: (a) the current control locations, (b) the values of variables, (c) the content of channels, (d) the set of events produced by actions. Thus, the LTS in each step is always organized with a root node that represent the current node of the main machine, intermediate nodes which can be: (i) the name of a box, (ii) the name of a machine instantiated by the box addressed in its parent node, or (iii) the name of a vertex which represents the current node of a currently executing machine, addressed in its parent node. According to this organization, the leafs of the LTS can be the nodes of the currently executing machines.

The formal definition of the semantics is not the main focus of this thesis work. However, its definition is at the basis of the mapping process described in the Chapter 5. This section provide the necessary background to support the discussion of the above mentioned chapter and to better state the constraints defined for DSTM.

According to this, there are some relevant aspects, of the DSTM formal semantics, that can be summarized as follows:

DSTM Machine evolution The evolution context of a DSTM machine is a sequence of *steps*. A *step* is a maximal set of transitions which are triggered by the current set of available events, with some constraints:

- a node or a box cannot be entered and exited simultaneously in the same step (this is instead possible for pseudo nodes);
- events generated by the firing of a transition cannot trigger other transitions in the same step but only in the next one.

Parametric instantiation DSTM allows for dynamic instantiation of parametric machines. This means that parameters are instantiated at execution time, when call transition are performed. Thus, parameters, do not hold values during the execution, but serve only as place holders for the actual values that are dynamically substituted when a call operation is performed. To allow such behaviour a substitution function, associated with the corresponding call transition has been defined. As a consequence, in order to define the semantics of a DSTM, the concept of *ground machine* has been introduced. A *ground machine* is

a machine where terms occurring in actions, triggers and guards do not contain parameters. Ground machines are obtained from parametric ones by applying the appropriate parameter substitutions.

Message exchange with the environment As stated above, the events generated by the firing of a transition cannot trigger other transitions in the same step but only in the next one. This means that the messages generated during a step and sent over an external channel cannot trigger other transitions in the same step. This means that sequential firings of transitions are not allowed within a step and only transitions affecting concurrent processes can be performed within the same step.

Transition semantics A specific semantics for the transitions has been defined. In particular, there are some transitions whose source and/or destination are pseudonodes. When such transitions fire, the machine lead itself in a non-stable state. Thus, in order to avoid the ambiguousness, there is the need to specify system behaviour with the respect to these cases. The transition of the language, that could lead to a non-stable state are:

- transitions exiting from an entering node;
- transitions exiting from the initial node;
- transitions entering in a fork or join.

In particular, the set of transitions crossing a fork can be seen as an hyper transition taking from a source and leading to many targets simultaneously, and, analogously, the set of transitions crossing a join can be seen as an hyper transition taking from many sources simultaneously and leading to a target. To manage such transitions the concept of *Compound transition* has been introduced. A *Compound transition*(CT) is a pair of two sets of transitions: *incoming transitions* and *outgoing transitions*; it is used to represent the evolution of the machine in the cases above discussed. According to compound transition, three different cases exist:

- Simple case: when a transition is between two nodes (or boxes), in the related CT both the incoming transition and the outgoing transition sets are made uniquely by the considered transition. The trigger, condition and actions of the CT are give by those of the considered transition.

- Fork case: the transition entering a fork and the corresponding transitions outgoing the same fork are grouped in the CT as follows: the incoming transitions set coincides with the transition entering the fork while the outgoing transitions set is give by the set of the transitions exiting the fork. In this the CT is executed as follows: when the transition belonging to the incoming transition set is executable (trigger and condition are evaluated true) it is taken and possible actions are hence executed; the outgoing set, instead, is executed immediately after the previous one as well as the related actions, that are however executed in interleaving (that means with non-determinism in the execution order).

- Join case: the transitions entering a join and the corresponding transition exiting, are grouped in the CT as follows: the incoming transitions set coincides with the transitions entering the join while the outgoing set coincides with the transition exiting the considered join. The incoming transitions set is executable when all the triggers and the condition of the transitions in it, are evaluated true. When it happens, these transition are taken and the corresponding actions are executed in interleaving. After that, the transition belonging to the outgoing set is taken and the related actions are executed.

Lets note that the outgoing transitions set is always executable since the transitions composing it cannot have trigger and condition (constrained by the syntax). In the simple case, actions are removed in the incoming transitions set and put in the outgoing one, while, with the respect to the outgoing set trigger and condition are removed. In addition lets note that when more than one transition enter a fork (respectively exit a join) different CTs are obtained, each one corresponding to a single of these transitions.

4.2 TESQEL: Test SeQuEnce Language

The generation process described in this work aims at producing a set of test sequences, describing the test specification given as input. These outcomes can be, in turn, modelled using a state-based modelling language (Figure 4.1). In fact test sequences can be viewed as an ordered list of state changes, named *firings*, which can be grouped in a compound firings related to the semantic evolution of the DSTM model. Next subsection describes the metamodel of the TESQEL language.

4.2.1 The TESQEL metamodel

The TESQEL metamodel, depicted in Figure 4.7, is created using the Ecore language. The main concept of the metamodel, is the *TestSequence* which corresponds to a given test specification. A *TestSequence* is characterized by *name*, refers to a set of *initial* vertexes of the DSTM model, a list of *initial* condition, each of them stating the values of a set of system variables before the start of the model analysis/execution and the related specification. Each *TestSequence* is composed of a list of *Compound Firing*s, representing a single atomic passage of the DSTM model evolution: it is a sequence of single *Firing*s.

The concept of *Firing* is related to the one of transition execution. A fire consists in a passage from two vertexes belonging both to the same *Box*. Each firing, in fact, has three traceability references to DSTM structural elements: the current DSTM Vertexes, the DSTM transitions, crossed in the firing and the next DSTM Vertexes. As the *CompoundFiring*s, *Firing*s own an attribute *Order* that is used to trace the progressive number of the fired items. This attribute is also in charge to support the graphical realization of the considered test sequence.

A can be further divided in several phases, each of them describing a specific aspect of the firing itself. Moreover, a *Phase*, owns a *order* attribute, that allows to trace the progressive number of the phases in the *phases* list of a *Firing*; Phases are abstract concepts, that can be specialized in the following kinds:

- **Trigger**: representing the arrival of a message on a channel; while the channel name is specified in the mandatory field *channelName*, two cases are considered: if *message* is null, it means that the trigger fires when any message arrive on the channel; otherwise, a specific message type is waited.
- **Condition**: representing a condition that a variable fulfils which can belong to the following cases:
  - *isFull*: the condition is true when the channel specified in the *Lvalue* field is full;
  - *isEmpty*: the condition is true when the channel specified in the *Lvalue* field is empty;
  - *variable*: the condition is true when the variable *Lvalue* has the value shown in *Rvalue*;
  - *message*: the condition is true when a message arriving on the channel *Lvalue* has the value specified in *Rvalue*.
- **Action**: representing the execution of a statement occurring during the passage through a transition; it belongs to the following cases:

- *send*: a message whose value is specified in *Lvalue* is send on the channel whose name is in *Lvalue*;
- *assignment*: the value of the variable *Lvalue* is set to *Rvalue*;
- *read*: the action reads the value of the first message present on the *Lvalue* channel (consuming the message itself);
- *check*: the action reads the value of the first message present on the *Lvalue* channel (without consuming the message itself).

With the respect to the Action, *CompoundFiring* can be composed by several *IOmessage*s in order to differentiate internal messages from the messages exchanged with the environment.

The CFOrder (FOrder) class has been added to the metamodel, in order to relate consecutive compound Firings (Firing) by means of an arc. This, allow the graphically realization of a model instance of of a test sequence, modelled with TESQEL

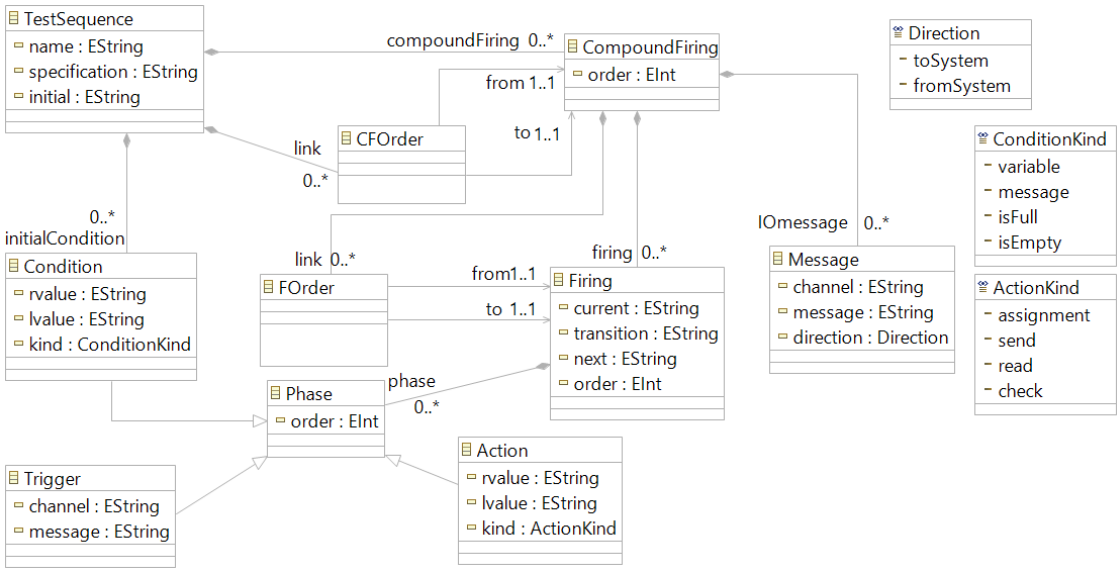


FIGURE 4.7: The TESQEL Metamodel

4.3 Test Specification Patterns

The scientific literature related to model driven techniques, mainly addresses the definition of languages and methodologies for model development and test case generation algorithms. Only few works, instead, address methods for define test specifications, in order to reduce the complexity of the modelling activities. According to this, the concept Test Specification Pattern (TSP) has been introduced

[41]. TSPs are a way to gather the previous and consolidated experiences in testing and provide a set of guidelines for test specification. In order to define a set of TSPs two main source have been considered: (1) requirement specifications of critical system (and in particular, since the work is partially involved in the CRYSTAL project 3, ERTMS/ETCS specifications are considered) and (2) scientific literature (e.g., [30, 1]). With the respect to this second, the TSPs are inspired to the works of Dwyer at al. [30]. Dwyer at al., in fact, define a set of patterns to reuse property specifications for finite state verification where properties are specified with temporal logics. TSPs, instead, are expressed by state-based models and translated into Promela never claims in order to enabling the automatic test generation. Despite the words assonance, the concept of TSP deeply differs from "test pattern" or design pattern in software testing: testing patterns, in fact, are used to define testing strategies; the work in [87] defines a set of testing patterns from classical design patterns in order to find "recurring" errors. There exist also other works extending the Dwyer's property specification patterns; [44] extends Dwyer patterns to probabilistic properties, Tsai et al [**1463211**] define Validation Patterns for the verification of embedded systems and Mondragon et al. ([69]) apply specification patterns for run-time monitoring of the system behaviour. With respect to the above mentioned specification patterns, TSPs are oriented to a testing-based form of verification. This feature is the main semantic difference between TSPs and the specification patterns in [30].

The set of TSP described below does not want to be exhaustive since they are part of a work in progress.

TSPs have been divided into two categories and are listed in Table 4.1: (1) *Control Patterns* address properties related to the evolution of the system. Related concerns are parallelism, sequence and/or loops; (2) *Data Patterns* refer to the management of data within the property to verify (e.g., set and evaluation of variables inside the test specification). In the following a model element can be both a transition or a state. Indeed, the meaning for covering a model element indicates respectively the passage through a transition and the reaching of a state.

In order to show the process that allows to define test specifications using TSPs, a simplified version of a car braking system is provided in Figure 4.8.

The car braking system has two Boolean inputs which represent the decision to accelerate or brake. Upon acceleration, the car starts to move, with either slow or fast velocity. Upon braking the car immediately stops. Among all TSPs, one of the most important is the *Cover pattern*. The *Cover pattern* is adopted to verify that a given step, that could be a transition or a state, is crossed within a test sequence. With the respect to the considered system, Figure 4.9 represent test

TABLE 4.1: List of Test Specification Patterns

Name	Pattern
Control Patterns	
Sequence	Specifies an ordered sequence of steps related to the test specification. The test sequence generated according by this pattern must shows the the sequence of steps in the specified order.
Cover	Specifies that a model element is covered at least once within a specific point of the test specification.
NotCover	Specifies that a model element is not reached before the fulfillment of the following step of the test specification (within a specified Sequence).
Next	Specifies that two model elements are reached in a close succession.
And	Checks that two or more sub-steps are accomplished regardless of their order.
DetChoice	Allows for choosing between two alternative sub-steps with respect to a condition.
Memory	Considers some alternative steps that, according to which step is fulfilled, address the future steps of the test specification to fulfill.
Loop	Specifies that a model element is reached for a defined number of times.
Any	Allows for verifying that at least one of the model elements specified within this pattern is reached at least once.
Data Patterns	
Test	Checks the value of a variable (of the model of the systems or of the test specification) to address a choice in the specification future behaviour.
Set	Assigns the value of an internal variable of test specification.
Assert	Allows to wait until a variable assumes a specified value.

specification of the requirement “When brakes are pressed a car must stop”.

Such specification requires that the test specification should if the brakes are pressed within the "FAST" state, the consecutive state reached is the "STOP" state. This requirement suggest the adoption of a *Next pattern*. The *Next pattern* indicates that considered steps must be cover consecutively. According to this, three covers are used: the first cover for "FAST" state, the second to cover the transition "T8" and the third to cover the state "STOP". Indeed, if the requirement is “During its movements car can accelerate or stop", the test specification can be made by means of *AND pattern* In fact, the requirement states that if the system is in a "SLOW" state (basic movement of the car) it can reach the "FAST" state or the "STOP" is any order. The *AND* pattern, representing such requirement is

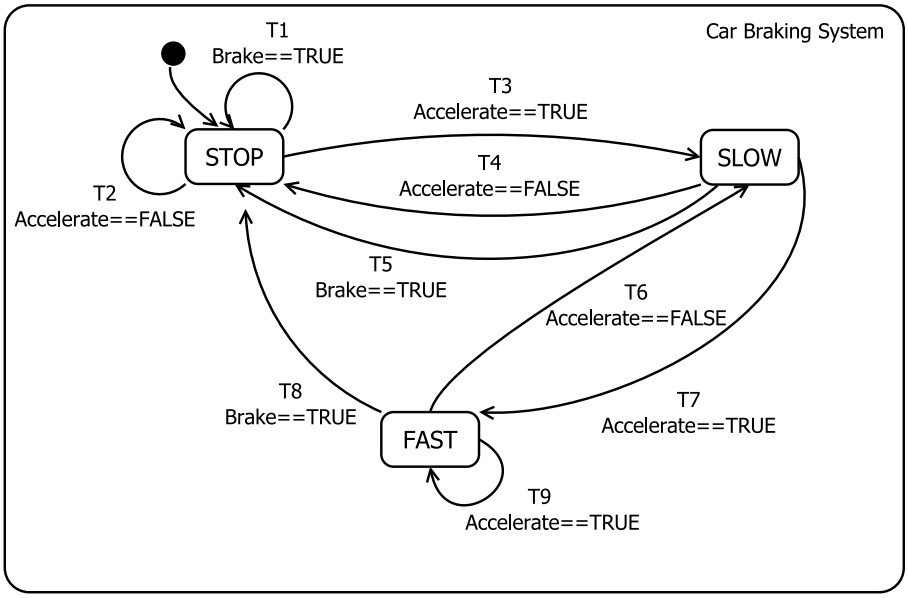


FIGURE 4.8: Car braking systems

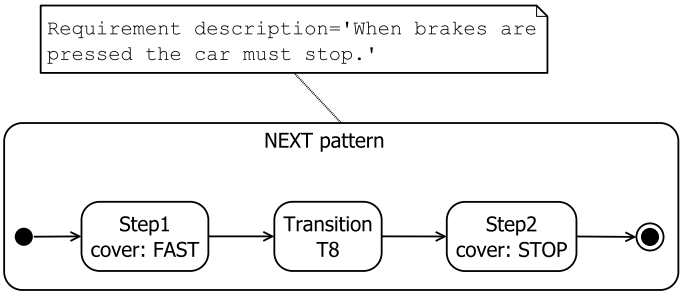


FIGURE 4.9: A next pattern for car braking requirement

depicted in Figure 4.10

In this work, the only TSP adopted is the *Cover pattern*. In fact, according to the needs of CRYSTAL project, in which this thesis work is partially involved, the only elements on which requirements can be tagged are the states and the transitions. Thus, since no order is specified, TSP can be modelled using only *Cover patterns*. Next chapter addresses the mapping process on which the testing strategy, described by the present work, relies. Among the mappings also ones related to the gap between the high level representation provided by TSPs and the concrete concepts of Promela, are discussed.

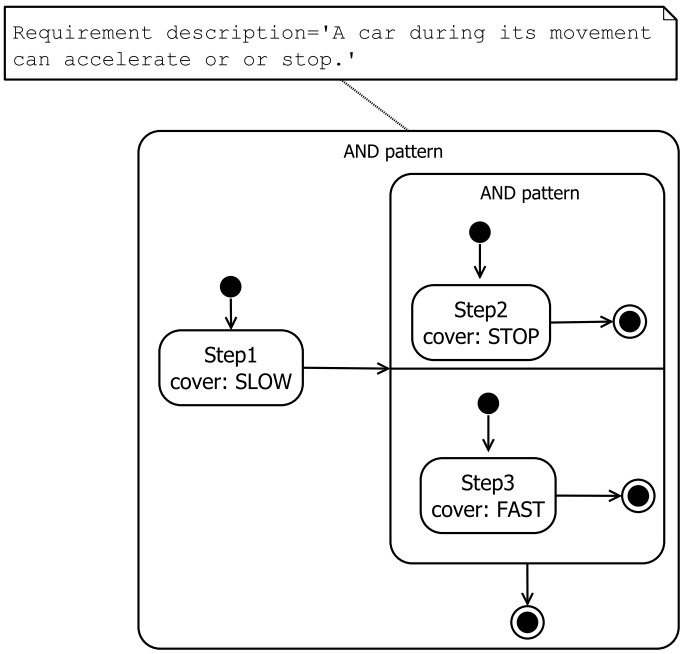


FIGURE 4.10: The use of AND pattern for a car braking requirement

Chapter 5

From DSTM to Promela

In the previous chapter the formal definition of DSTM has been addressed. DSTM extends Hierarchical State Machine by introducing a novel semantic for fork and join, in order to allow for dynamic instantiation, preemptive termination and the possibility to pass parameters to machines at activation time. This chapter is focused on the mapping process, between the DSTM language and the target model checking language, chosen according to objective of automatically generate test cases. The mapping process is conceived to be as general as possible, even some hypotheses, specific for the chosen model checking language, are introduced. With the respect to this last, the chosen model checking language is Promela (Process or Protocol Meta Language) introduced with the Spin Model Checker by Holtzmann [**Holzmann1997**].

The Spin Model Checker supports many of the control and data flow constructs specified by the DSTM language, as buffered message passing, communication via shared memory and dynamic instantiation of processes. These features make Spin a logical choice to tackle the problem at hand. In addition, Spin is a well known on-the-fly model checker, able to handle efficiently problems of large size supporting the high level specification of system models with the Promela language. Despite some similarities exist between the two languages, a non-straightforward mapping process has been defined during the definition of the approach. Clark [20] defines the concept of mapping as relationships or transformations between models or programs written in the same or different languages. This is a key point of model driven techniques, since mapping processes are often at the basis of the model transformations. Address a mapping process related to the specification of a SUT can lead to several issues: (1) the the different level of abstraction between the source and the target language [15], which can lead to a semantic gap and (2) and the level of abstraction of the outcomes of the transformation process, which should be defined according to knowledge of the SUT specification modeller. This can allow the modeller to easy analyse the outcomes with the respect to the specified system. According to this, several approaches

exist to perform a mapping process. Among them, this work focused on a *unidirectional mapping* [20] process which means that no dependencies are considered between the source and the target models. Thus, every time a change is made on the source language, the mapping process has to be executed again. The sections of this chapter address each aspects of the mapping process between features of DSTM and the constructs provided by Promela.

5.1 Mapping process: an overview

In order to better understand the overall mapping process, this section summarizes the principal modelling issues which have been addressed in the translation of DSTM machines in Promela.

Machine structure As already stated in Section 4.1.1, DSTM extends the Hierarchical State Machine formalism introduced by Harel [47]. This means that the structure of the machine modelled with DSTM is strongly hierarchical. A DSTM machine, in fact, consists of sets nodes, pseudonodes, boxes and transitions for which there is the need to find corresponding concepts in Promela. According to this, a first fundamental step has been to find the way to map a generic HSM-based structure into Promela. With the respect to this, Promela provides the concept of Process, that is the basic constructs to execute model behaviours. A process, in fact, allows the syntactical structure of a DSTM machine, even if this is not a straightforward process. The syntactical mapping that allows to translate the machine structure of the DSTM in the process structure of Promela is discussed of the Section 5.2. Next paragraph discuss how DSTM addresses the mechanism of instantiation, already provided by HSMs.

Dynamic Instantiation The original formulation of HSMs already provide the concept of machine instantiation. However, according to the work of Alur [6], the kind of instantiation provided static. In fact, the instantiation process, consists in a systematic substitution of the boxes, with the entire model corresponding to them. The main consequence of such process is that it can lead to a state-space explosion even for medium-complex system. Moreover, approaches found in literature, related to HSMs instantiation, provide that the caller machine is always suspended in order to wait the callee machine termination. DSTM overcomes this mechanism by introducing an innovative approach of the dynamic instantiation of modelled machines. Furthermore, DSTM provides also the recursive instantiation and parallel execution of machines, giving to fork and join a novel

semantics. Hence this formalism is substantially different from Hierarchical Machine (which not permit recursion) and also from other HSMs extensions as Recursive Machines (which not permit parallelism) and Communicating Machines (which not permit recursion and dynamic instantiation). Concepts of fork, join and boxes introduce a hierarchical structure inside the models. At the contrary Promela, as the largely part of model checkers languages, relies on the concept of Kripke structure, that is a way to represent reactive systems for model checkers.

Let AP be a set of atomic propositions [21]. A Kripke structure M over AP is a four tuple $M = \langle S, S0, R, L \rangle$ where:

1. S is a finite set of states;
2. $S0 \subseteq S$ is the set of initial states;
3. $R \subseteq SxS$ is a total transition relation, that means that for every state $s \in S$ there is a state $s' \in S$ such that is defined $R(s, s')$;
4. $L : S \rightarrow 2^A P$ is a function that assigns a label to propositions that are true in the state S.

The definition of Kripke structure clearly shows the assonance between HSM-based formalism and model checking language but also underlines that there is crucial difference in the hierarchical structure. As discussed above, several approaches can be used to flatten a hierarchical model:

- Replacing each box with the related machine. This approach, however could lead to an high state explosion: in fact, when multiple entry or exiting points are defined, there is the risk to repeat the same machine several times
- Removing hierarchical structures with some simple nodes and/or transitions, with a special semantics. These approaches could lead to concurrency problems, due to the fact that they are conceived to execute simultaneously more machines.

The approach followed in this thesis is the second one, and its deep addressed in Section 5.4.

Termination and preemption Another feature of the proposed specification formalism is the possibility to terminate the machines, using some nodes with a specific semantics (i.e. exiting nodes). Moreover, with the respect to HSMs in

which the caller machine suspends its execution when a new machine in instantiated, in DSTM there is the possibility to concurrently execute both the caller and the callee machines. This introduces a synchronization problem since the caller must know the termination state of its callee. Furthermore, DSTM provides the possibility of preemptive join, which means that when a machine terminates with preemption, ask for the termination of one of the instantiated machines even if they are not complete their execution. Promela does not provide any primitive to terminate processes. Thus, in mapping process, the need to realized the termination mechanism of DSTM in Promela has been addressed.

Machine communication DSTM provide a novel communication mechanism with the respect to the HSMs. These last, in fact, allow for broadcast communication by means of events, put in the actions of a transition decoration that is sensed by all the other machines. Nevertheless, two different kind of communication mechanisms have been addressed: a broadcast internal communication between DSTM machines and a communication between machines and the environment. The communication mechanisms can be realized also thanks to the introduction of a type system able to model complex data structures. In fact, within the type system, shared variables, internal and external channels and data types have been defined. According to this, shared variables and internal and external channels are involved in the communication mechanisms. The choice is not casual: in fact Promela provides a similar mechanism for the communication of processes. Mappings related to communication mechanisms based on type system are described in Section 5.3.

The step semantic In order to obtain a suitable Promela specification, there is the need to correctly implement the step semantics of DSTM and its policy for the firing of transitions. The transitions firing is constrained by the enabledness condition, defined in Chapter 4 for compound transitions. This condition requires that for a transition of an active machine to fire, none of its ancestors in the control tree must be currently enabled to interrupt the execution of the that machine (either by an interrupt transition or by a preemptive join). This condition will be guaranteed by imposing an execution priority among the machines, which is compliant with the hierarchical activation of machines. Moreover the step semantics prevents sequential firing of transitions within the same execution step. Since in the flat model compound transitions have been removed, there is

the need to guarantee that at most one enabled transition can fire for each active process. The mapping process addressing the step semantics is described in Section 5.5.

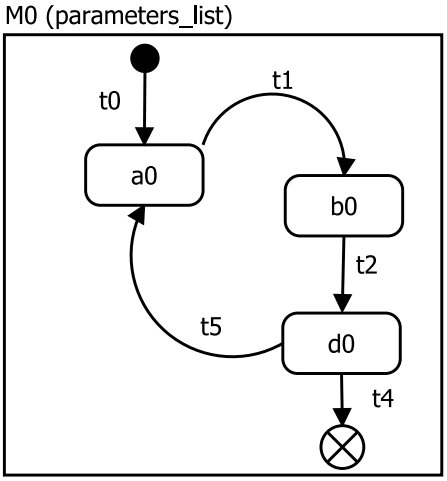
Table 5.1 summarizes the overall mapping process between DSTM and Promela. In particular on the left-side of the table are listed the features of DSTM while in the right-side Promela concepts which allow to address the modelling issues above discussed. Next sections, provide a deep analysis of the mapping process, with particular focus on the solution proposed.

		PROMELA					
DSTM		Process	Iterative statement	Selection statement	Data types	Variables	Global channels
Machine Structure	Machine	X	X				
	Nodes					X	
	Transition			X	X		
	PseudoNodes						
	Boxes						
Dynamic Instantiation		X					
Termination and Preemption						X	X
System Type	Data Type				X		
	Channels						X
	Variables					X	
Step Semantics	Environment communication		X	X			X
	Machine instantiation	X	X	X			X

TABLE 5.1: Mappings between DSTM features and Promela

5.2 Syntactical mapping

As previously discussed, DSTM Machines can be easily mapped in Promela processes. In particular, the internal behaviour of a machine can be represented by using iterative constructs allowed within a Promela process. This choice is not straightforward, since the semantics provided by an iterative construct internal to a process is quite different from the corresponding concept in a general purpose languages [88]. In fact the iterative construct in Promela does not reach the exit state when none of the condition guards can be true but simply blocks its execution, waiting for the truth of a guard. In this way, processes (and therefore machines) can remain executable. Figure 5.1 represents the syntactical mapping between the DSTM machine depicted in the left-side and the Promela process in the right-side.




```
proctype M0(parameters_list){
  state=a0;
  do
   :: (state == a0) ->
       atomic{
       }
   :: (state == b0) ->
      atomic{
      }
   ::(state == c0) ->
      atomic{
      }
  od unless{ }
}
```

(b)

FIGURE 5.1: Mapping between DSTM machine (a) and Promela process (b)

5.3 Data-flow mapping

In order to address communications of DSTM Machines and some specific features as the termination, To address the communication mechanisms and also the machine termination a complex type system has been introduced in DSTM. In particular three main elements are here considered: (1) Data Types, (2) Channels and (3) Variables. Below mappings related to each of them are discussed.

5.3.1 Data types

With the respect to the Data Types, it has to be considered that Promela provides seven different predefined simple data types: *bit*, *bool*, *pid*, *byte*, *short*, *int* and *unsigned*. With the respect to the composite type, instead, Promela provides the concept of *typedef* which allows for user-defined data types. Also DSTM provides simple and multi types and the mapping process is different in the two cases. In fact, for simple types the mapping process is quite straightforward since it is a *one-by-one* mapping: each tInteger is mapped in a int, each tChannel, in a global channel and each tEnum, in an mtype which is a declaration for the introduction of symbolic names for constant values. Table 5.2 summarize mappings between *Data Types*.

The process to map channels and variables, and in particular the multi type ones, is very different and is treated in the following subsections.

TABLE 5.2: Mapping of types

DSTM	Promela equivalent
Data Types	
tInteger	int
tEnum	$mtype = \{literals_0, literals_1, ..., literals_n\};$
tChannel	$mtype = \{channel\_names_0, ..., channel\_names_m\};$

5.3.2 Channels

Channels represent the way with which DSTM machines can communicate. The problem in mapping channels with corresponding Promela is that DSTM introduces two fundamental characteristics which are not present in Promela:

1. first, DSTM distinguishes between internal and external channels; the former are used for internal DSTM machine communication while the latter are used for the exchange of messages with the environment. At the contrary, Promela, provides only one kind of channel, that is the *Global Channel*.

2. DSTM provides the concept of "channels of channels", which can be expressed in the form Chn[T] (Section 4.1.3), where T is used to trace the type of the contained channels;

3. furthermore, DSTM channels can be of simple and multi type. The first ones are simple to translate in Promela, while for the second the process is more complex.

In order to address the mapping issues discussed above, Promela channels, are generated according to the below process:

- Internal Channels of simple types, are mapped in a Promela global channel which has the same *bound* specified within the internal channel of the DSTM model. The type of the channel, instead, correspond to the type of messages conveyed in the channel.

- At the same way, external channels of simple types are mapped in Promela global channels, with the type corresponding to the message type, but the bound fixed to one;

- multi type channel, instead, does not have an corresponding concept in Promela. For such reason, they are modelled by creating one Promela channel for each simple type contained within the multi type of the considered channel. The set of Promela channels associated with a multi-type channel are properly managed, so as to ensure that at each moment at most one of them contains a message.

- finally, the Chn[T] are translated in a Promela global channel, containing as many *chan* type, as the subtypes contained in T.

Thus, Table 5.3 reports an example of mappings related to DSTM channels.

TABLE 5.3: Mapping of channels

DSTM	Promela equivalent
Simple type channels	
Chn external c of Int	$chan\ c\ =\ [2]\ of\ \{int\};$
Chn internal c[bound] of Int	$chan\ c\ =\ [bound]\ of\ \{int\};$
Chn external c of enum	$chan\ c\ =\ [2]\ of\ \{mtype\};$
Chn internal c[bound] of enum	$chan\ c\ =\ [bound]\ of\ \{mtype\};$
Chn external c of compound	$chan\ c\ =\ [2]\ of\ \{...\};$
Chn internal c[bound] of compound	$chan\ c\ =\ [bound]\ of\ \{...\};$
Multi type channels	
Mtype MT1 { ST1,ST2,ST3 }	
Chn external c of MT1	$chan\ c\_ST1\ =\ [2]\ of\ \{...\};$ $chan\ c\_ST2\ =\ [2]\ of\ \{...\};$ $chan\ c\_ST3\ =\ [2]\ of\ \{...\};$
Chn internal c [bound] of MT1;	$chan\ c\_ST1\ =\ [bound]\ of\ \{...\};$ $chan\ c\_ST2\ =\ [bound]\ of\ \{...\};$ $chan\ c\_ST3\ =\ [bound]\ of\ \{...\};$
Chn external c of Chn[MT1]	$chan\ c\ =\ [2]\ of\ \{chan, chan, chan\};$
Chn external c of Chn[MT1]	$chan\ c\ =\ [2]\ of\ \{chan, chan, chan\};$

5.3.3 Variables

DSTM global variable, are translated into the corresponding concept of Promela variable. As for the type the process is quite straightforward. In fact, type and name of a DSTM variable, are mapped in a Promela variable, in a C-like style. The only exception is the multi type variable which does not have a corresponding concept in Promela. As in the case of the channels, many simple type variable as the number of subtypes contained in the considered multi type variable, are generated. Table 5.4 shows some examples of such mappings.

TABLE 5.4: Mapping of channels

DSTM	Promela equivalent
Simple type variables	
Int v **Enum v** **Chn v**	$int\ v;$ $mtype\ v;$ $chan\ v;$
Multi type variable	
Mtype MT1 { ST1,ST2,ST3 }	
Chn[MT1] v	$chan\ v\_ST1;$ $chan\ v\_ST2;$ $chan\ v\_ST3;$

5.3.4 Triggers, Conditions and Actions

Data types, channels and variables allow to define the decoration on the DSTM transitions. As in the HSMs, a decoration can be composed by trigger, conditions and actions and are treated as strings, whose meanings is given by the modeller with the respect to him context. In DSTM triggers, conditions and actions have a formal syntax and semantics, despite, at the modelling level, can be treated as simple strings. Treat decorations as simple strings allows to not increase the complexity of the DSTM metamodel and enables to model complex expressions (as the nested ones) by means of text. Thus, triggers, conditions and actions must be mapped in corresponding Promela concepts, according to the meaning demanded by the formal syntax and the semantics. In particular, triggers and conditions can be specified only as receptions on a channel (external and internal). Thus, the mapping process must be performed according to the mapping defined for them treating separately the simple and the multi type cases. Also the actions provide send on a channel, but in addition they allow to specify variables assignment. In this case the mapping is quite straightforward, since both the left and the right operands are mapped in the corresponding Promela elements. Table 5.5 reports a summary of all possible mappings related to triggers, conditions and actions.

TABLE 5.5: Translating the transitions decorations to Promela

		DSTM	
		Simple Type	Multi type
Trigger	$C?$	$C?[,...,\_]$, if C is internal	$C\_T1?[1,\_,...,\_]\|\|...C\_TN?[1,\_,...,\_]$, if C is internal
		$C?[1,\_,...,]$ if C is external	$C\_T1?[1,\_,...,]\|\|...C\_TN?[1,\_,...,\_]$ if C is external
	$C?T$	Not allowed	$C\_T1?[1,\_,...,\_]\|\|...C\_TN?[1,\_,...,\_]$, if C is internal
			$C\_T1?[1,\_,...,]\|\|...C\_TN?[1,\_,...,\_]$ if C is external
Condition	$C[? < ..., Xt, .. >]$	$C[? < \ldots, x, ... >]$ if C is internal	$C\_T1?[1,...,Xt,...]\|\|...C\_Tn?[1,...,Xp,...]$[1] if C is internal
		$C[? < 1, \ldots, x, ... >]$ if C is external	$C\_T1?[1,...,Xp,...]\|\|...C\_Tn?[1,...,Xt,...]$ if C is internal
	$empty(C)$	$len(C) == 0$	$len(C\_T1) == 0\&\&...\&\&len(C\_Tn) == 0$
	$full(C)$	$len(C) = channel\_bound$	$len(C) = channel\_bound\&\&..\&\&len(C) = channel\_bound$
Action	V=x	$Vp = Xp$	Not allowed
	$C! < x, y, ..., z >$	$C!x, y, ..., z$	$C\_Tj!x, y, ..., z$ where C_Tj is a channel compliant with the type Tj

5.4 Dynamic instantiation and termination of machines

The introduction of hierarchy mainly means that DSTM provides macro-nodes containing other nodes. Hierarchy enables concept of reuse of modelling components and allows for more compact models. HSMs, have been extended in several ways in literature, in order to allow the dynamic instantiation of machines. For example UML State Machines admit parallel execution through the

[1] Xp is the corresponding promela element

usage of composite states and regions. In this formalism, the fork (and join) is used in order to split (and merge) an incoming transition into two or more transitions terminating on orthogonal target vertexes (i.e., vertexes in different regions of a composite state). Recursive activation and dynamic instantiation is not admitted. Communicating Hierarchical Machines (CHMs) are a variant of Statecharts introduced for succinctness reasons. They introduced the idea to have a collection of finite state machines (modules) having nodes and boxes. A transition entering a box represents a call to one or more instances of another module. In a Statechart there is no notion of module and instance. If multiple instances of the same module are required by the specification, each instance has to be explicitly defined. On the other way the introduction of modules allows to define Recursive State Machines (RSMs) where a module can recursively call itself [**alur**]. Notice that, in the case of Recursive State Machines, we are not anymore in the category of Finite State Machines. In [59] CHMs has been extended introducing Dynamic Hierarchical Machines (DHMs) which allow the dynamic activation of machines: any DHM M_1 can send to a concurrent DHM M_2 a third DHM M_3, which starts running either in parallel with M_1 and M_2, or inside M_2, depending on contextual information. In DSTM dynamic instantiation of machines has been realized through the hierarchical structure and the semantics of fork and join. However, as discussed in Section 5.1 in order to map a hierarchical state-based formalism in a Kripke-based one, there is the need to remove the hierarchy through a flattening process.

5.4.1 Flattening process

The flattening process is a relevant step in the mapping process since it shall allow to remove the hierarchical structure maintaining the same modelled semantics. DSTM provides three different ways to enable the dynamic instantiation:

- the use of simple box, activated by a transition entering in it which does not have a fork as a source node (Figure 5.2(a));
- the use of boxes, activated by a asynchronous transition whose source node is a fork;
- the use of boxes, activated by a non-asynchronous transition, whose source node is a fork.

It is evident that fork and boxes must be removed in order to flatten the machines. Below solutions to preserve the semantics of the models, when the hierarchical structure is removed, are discussed.

Case of the simple box In this case the dynamic instantiation is realized by using a transition entering in a box which does not have a fork as a source node (Figure 5.2). According to the DSTM metamodel, boxes are vertexes of model, with the capability to activate other machines, passing them related parameters. The target language, Promela, provides a *run* operator which takes as arguments for the related process, the list of parameters for its activation. With the respect to this mechanism, simple boxes can be removed by performing following steps:

- Each box is substituted by a simple *Node* with the same name;
- The transitions entering in the considered box, are substituted by a transition with the same name, that inherits the same decorations of the source transitions with the exception of the actions, to the which, two special actions are added: (1) a action that call the run operator, passing it the name of the machine to activate and (2) the initialization of the related parameters list.

Figure 5.2(b) shows the result of the flattening process on simple boxes.

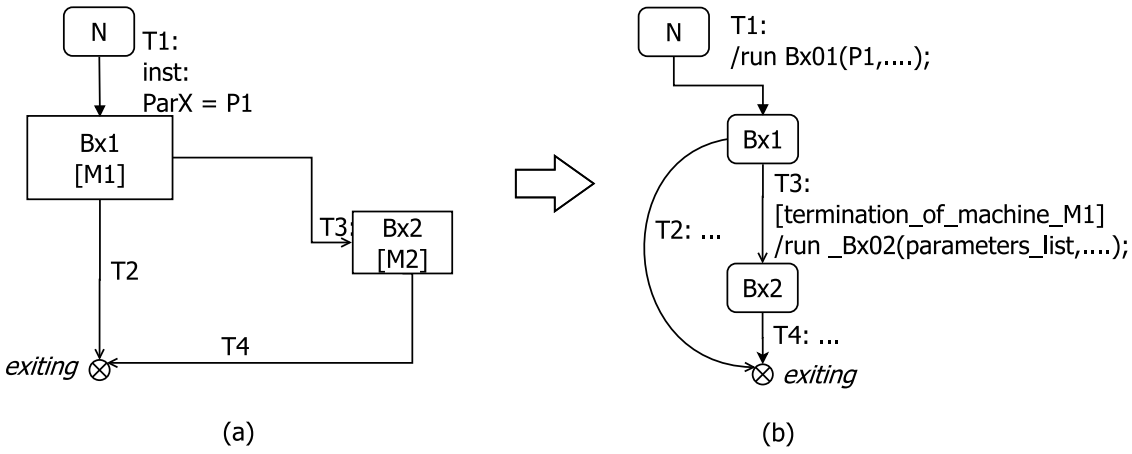


FIGURE 5.2: Flattening process on simple boxes

Case of asynchronous fork In this case the current machine (*caller*) continues to execute concurrently to the new activated machine (*callee*). In order to flatten the model, considered fork and boxes are removed and substituted with internal transitions. In particular, according to the Figure 5.3(a) the *internal entering fork* transition (*T1*), the *call by default* transition(*T3)* and the two *internal asynchronous fork* transitions are substituted by a single transition ($T1_T2_T3_T4$) with the source in the *T1* source and the destination in the node subsequent the termination of

the considered box (in this case is the same, N). The decoration of such transition inherits trigger and condition from the *internal entering fork* while the actions are obtained with a process similar to the simple boxes case. In particular in case of multiple boxes activated by the fork, the actions on the *call by default* transitions are executed in *interleaving* that means with a non-deterministic order. Figure 5.3(b) shows the result of the flattening process described.

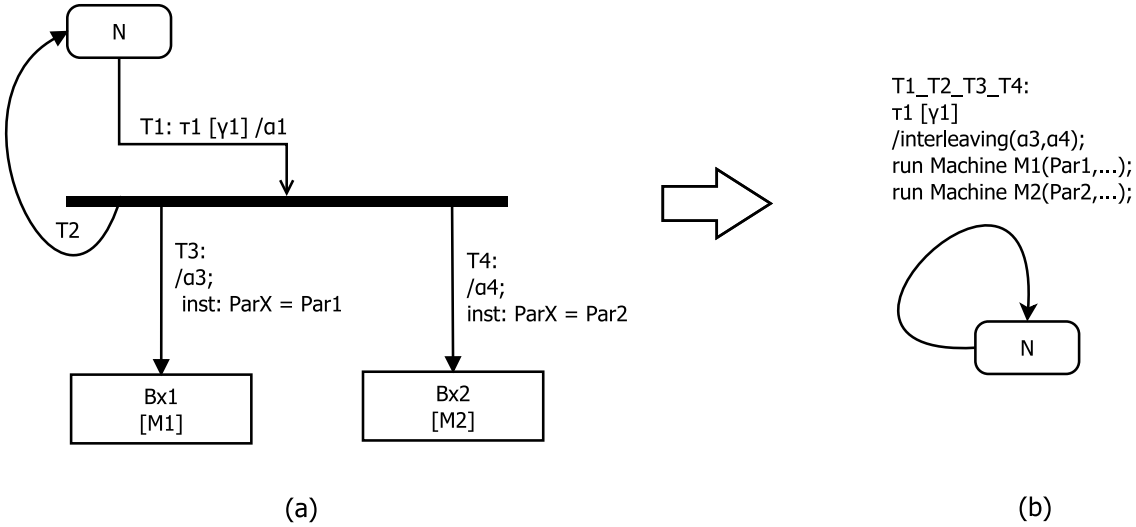


FIGURE 5.3: Flattening of asynchronous fork

Case of synchronous fork With the respect to this case caller machine, suspends its execution in order to wait for the termination of activated boxes. In this case, the entire block, containing the fork, the join and the boxes, is substituted with a new node, (called wait node), which models the state of the suspended machine. Transitions to and from the *wait* node, are substituted with the procedures described in the case of the asynchronous fork, in fact the *internal entering fork T4* and the *call by default* transitions *T5 and T6* are substituted with the compound transition $T4_T5_T6$. The two boxes instead, are substituted by the wait node $wait_Bx2_Bx3$ Figure 5.4(b) describes the result of mapping process for such case.

5.4.2 Termination and preemption

The flattening process above described does not completely resolve the problem of the flattening. In fact, according to the well-formedness constraints discussed in Section 4.1.3, each fork is associated with one or more join and furthermore

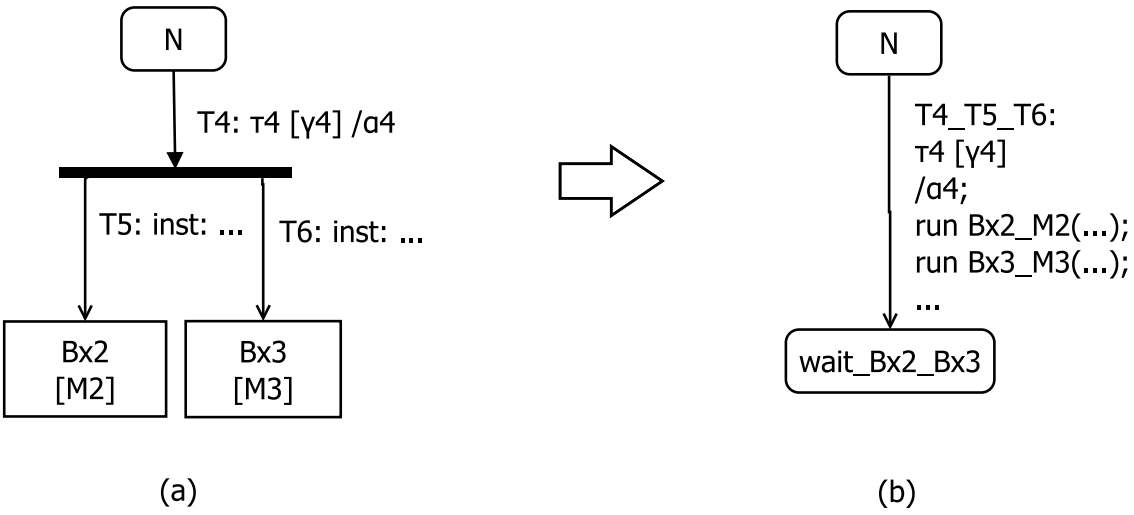


FIGURE 5.4: Flattening for synchronous fork

each box can be associated by zero or more exiting nodes representing the way with which machines can terminate. Thus, the problem of termination should be properly managed since the presence of the dynamic instantiation. Also for the termination issue, Promela does not provide any concept of exiting node neither provides constructs to terminate processes. To guarantee in the mapping process the termination Promela global channels are exploited. In fact, for each box containing *n* exiting nodes, *n+1* Promela channels are generated and passed as parameters in the *run* operator. The first *n* channels are related to the *n* exiting nodes and are used by the instantiated machines to communicate their termination to the caller machine. The *n+1-th* internal channel instead, is used by the caller to communicate to its instantiated machines which is terminated. Figures 5.5 and 5.6 represent two different cases of join. In the first case (Figure 5.5) two boxes *Bx1* and *Bx2* are connected to a join pseudonode by the exiting nodes *ex11* and *ex21*. In order to flatten the block represented by the two *return by exiting* transition *T1* and *T2*, the *internal exiting join T3* and the join itself, the transition $T1_T2_T3$ is generated. Such transition owns a condition that wait for a termination message from both the internal channels associated with the exiting nodes of the boxes. With the respect to the actions, instead, a termination message must be send on the internal channel related to the machine termination.

The second case (Figure 5.6), instead, shows a case in which on the same join are connected a node of the current machine (*N1*) and two boxes. In this case the block is substituted with an internal transition which condition inherits the condition of the *internal entering join* transition and, as the previous case, the

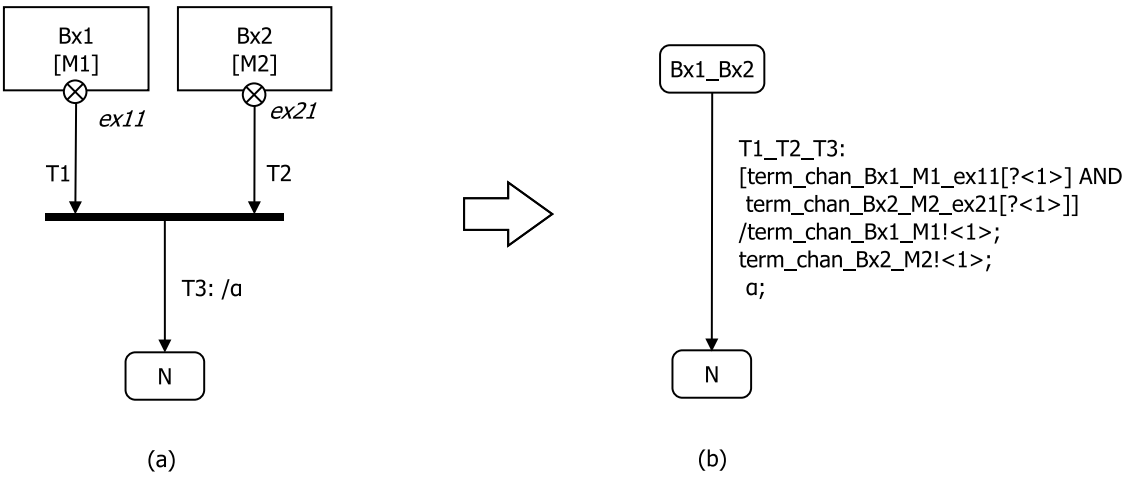


FIGURE 5.5: Flattening of join with only boxes

termination condition of each termination channels of each machines.

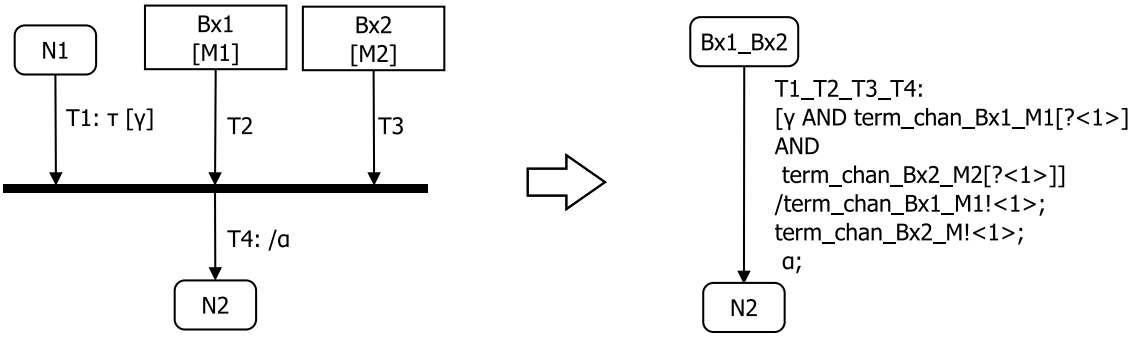


FIGURE 5.6: Flattening of join with current machine node and boxes

The flattening process is different in case of preemption. DSTM, in fact, enables a machine which exit by a preemptive transition to ask for the termination of an instance of each box related to the considered join. In general, in presence of preemptive join, each *entering join* transition can trigger the termination of all the other parallel machines, regardless of their current state. In this case a distinct internal transition for each preemptive entering join is generated. Each transition inherits the same trigger of the original entering join transition while guards and actions are treated as in the non-premptive case.

With the respect to the preemption several cases are possible which are treated in different ways:

- Case 1: Preemptive transitions with source in a box. When join collects the

exiting state of boxes with a preemptive transition, according to the general approach described above, the flattening process substitutes the whole block of boxes, *internal entering join* transitions and *internal exiting join* transitions with as many transitions as the preemptive ones. In particular, with the respect to the Figure 5.7(a) there are two boxes with a preemptive transitions (*T1* and *T2*) and another box with no preemption. The only transition with a trigger is *T1*. In this case, the substitution is made by means of two transitions: *T1_T4* which inherits the trigger of the *T1* transition and the action of *T4* with the addition of termination messages on the channel of the other boxes and *T2_T4* which has no trigger but the waits condition of the termination of machine M2. Figure 5.7(b) depicts the result of the flattening process.

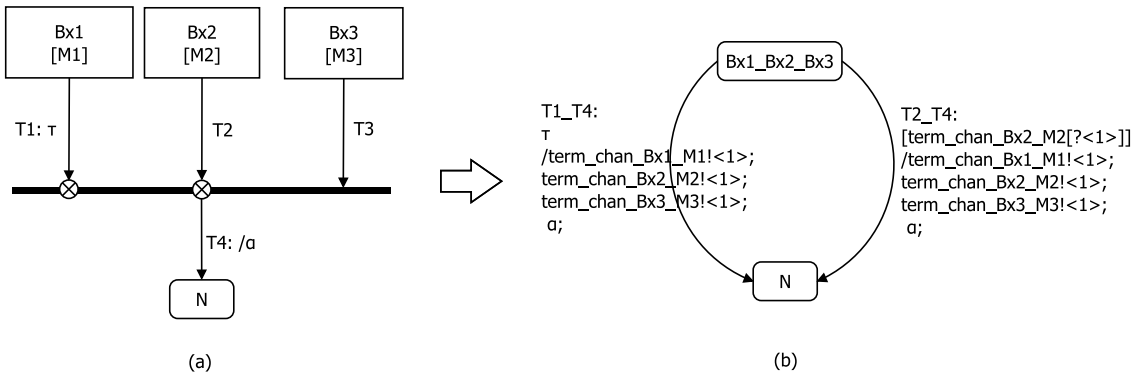


FIGURE 5.7: Flattening of preemption in case 1

- Case 2: Asynchronous case with no preemption. In this case on the join insist preemptive transitions with source in boxes and a transition with source in a node of the current machine decorated with triggers and conditions. In this case, each preemptive transition, must consider the trigger and the condition since they are related to the execution flow of the current machine. Thus, according to the previous case, the *join-block* is flattened by adding two transitions: *T1_T2_T5* that derives from *T2* transition, and inherits trigger and condition from T1 and *T1_T4_T5* that inherits from the preemptive transition *T4* considering also the trigger of *T1*. Figure 5.8(b) summarizes the flattening process.

- Case 3: asynchronous case with preemption on current machine. With the

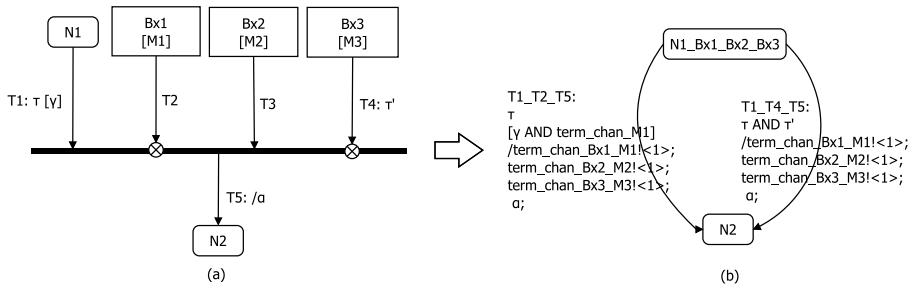


FIGURE 5.8: Flattening of preemption in case 2

respect to the previous, this case has the only preemption on the transition which source a node is a node of the current machine. Since the only preemption is related to the execution flow of the current machine, the *join-block* is substituted by a transition that inherits trigger and condition of the preemptive entering join transition and add in the actions the termination messages for the machines connected to the same join (Figure 5.9(b)).

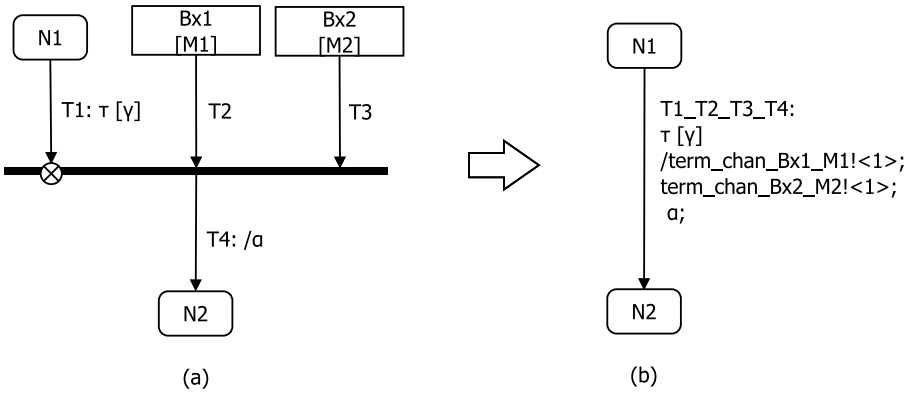


FIGURE 5.9: Flattening of preemption in case 3

- Case 4: asynchronous case with preemption on both current and instantiated machine. This is the complete case, in which both the current execution flow and the flows related to the instantiated machine end in the same join with preemptive transitions. Despite the general approach provides a transition for each preemption, in this case, the execution flow of the current machine is priority with the respect to the other flows. Thus, the *join block* is substituted with a transition that inherits trigger and condition from the entering join transition related to the current machine and has as actions the terminations of the other instantiated machines. Figure 5.10 exemplifies the flatten model of the above discussed case.

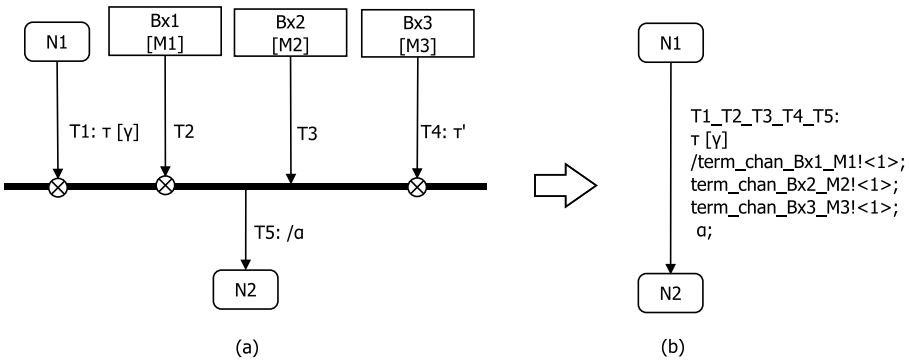


FIGURE 5.10: Flattening of preemption in case 4

5.5 DSTM Step semantics

According to the Subsection 4.1.4 the evolution of a DSTM machine is possible under the satisfaction of two constraints: (1) nodes or boxes cannot be entered and exited simultaneously in the same step and (2) the events generated by the firing of a transition cannot trigger other transitions in the same step (the *enabledness* constraint). This means that if a transition of a machine is enabled in a given step, any of its ancestor machines cannot be enable to fire a transition. To do this, it is necessary to impose a priority among machines executions. Moreover, since the removal of the hierarchical structure delete the compound transitions, to guarantee the enabledness constraint is necessary to guarantee that at most one enabled transition can fire for each active process.

To achieve such objective, the mechanism adopted is the one often used by scheduling algorithms of distributed system [42] that is the *Token mechanism*. In fact, the Promela process derived from the machine, can fire a transition only if owns a token. When a process owns a token is scheduled, it consumes its token and:

- if none of its transitions is enabled, the process generates and sends a token for each of its children;
- otherwise, an enabled transition is selected and executed.

The token propagation is started by a special process which gives the token for first to the initial machine. In this way is possible to avoid the sequential firing of transitions within the same step. Thus, the step semantics, evolves in two steps:

1. *the step execution* in which the token is passed to a machine in order to enable it to execute. The token does not completely resolve the multiple transitions activation problems. In fact a machine can provides more the one enabled transition. In this case non-deterministically one of them can be chosen to fire.

2. *the next step initialization* in which the token is recovered and passed to the initial machine in order to perform another step.

This mechanism is restarted every time no machine can fire a transition. This usually happens when each machine belonging to the system model has consumed its own token, meaning that the current step semantics execution is completed.

The same process that is in charge of manage the token mechanism is used to model the environment. The environment is conditioned by the messages exchanged on the external channels. In fact, according to the language semantics, the subsequent evolution of the machines is driven by a suitable message handling mechanism. In fact, each external channel, has a buffer that stores two messages as depicted in Figure 5.11. The first position C is used to store the message available in the current step, whereas the position N is used to store the message to be delivered in the next step (if any). During the execution of the current step, the processes modelling the SUT can read mes- sages contained in positions C of any channels, without removing them. If a new message is produced by the SUT, it is stored in positions N of the corresponding channel, thus making the messages generated available for the next step.

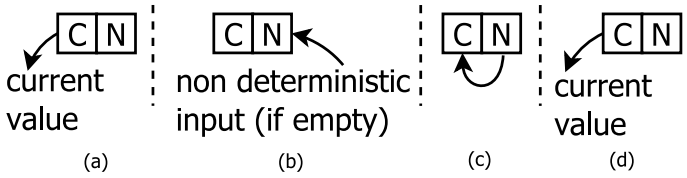


FIGURE 5.11: Message generation for an external channel

The environment, as the DSTM machine, has a behaviour that can be easily described with a state-based formalism. Thus, the obvious choice to map it in Promela is by means of Promela process [72]. Next chapter will deeply address all the characteristics of such special process, that is in charge of "start" the movement of the machines, and that we named *Engine*.

5.6 Test specifications mappings

The automatic generation of test cases, is enabled also by the modelling of test specifications. Ever the test specifications are not expressed with the DSTM language, they must be translated in corresponding concept of the Promela language. In particular, the role of the test specifications is that they are used to exploit the counterexamples generation capability of model checkers. Moreover, according to the discussion in Section 4.3, within the defined set of TSPs only a simple class of them is addressed in this thesis in order to enable the test cases generation process. More specifically, the TSPs considered in this work, focus on the coverage of transitions or nodes, according to the needs of domain needs, discussed in 4. A requirement annotated on a transition or a node, can be considered with a *cover* pattern. This allows to generate a test case consisting in a sequence of steps to perform to reach the considered node/transition. With the respect to Promela, the test specification modelled with the *cover* pattern is mapped in a *never claim*. The aim of a *never claim* is to represent a behaviour that should never happen; in fact is defined as a series of propositions on the system state that must be true in the specified sequence in order to match the expected behaviour [88]. The test specification translated in a Promela *never claim*, models a linear time temporal logic formula, and specifically the crossing of a specified model element which should never happen.

```
never {
step1:
        if
                :: (transition_M0 == t2) ->
                        goto endStep
                :: else -> goto step1
        fi;
endStep:
        skip
}
```

FIGURE 5.12: Mapping of a cover pattern

Figure 5.12 reports a *never claim* to verify that the transition variable of the $M0$ machine reported in Figure 5.1(a) is never equal to $t2$; the model checker verifies if the property is true, and in case it is violated (as expected), provides a counterexample which represents an execution sequence covering transition $t2$, that is a trace that eventually makes the transition fire.

Chapter 6

Development of the test cases generation framework

The high-level architecture of the test case generation framework has been discussed in Section 3.2.1; this chapter if focused on the development phase of the framework named Test Case Generator (TCG). The TCG has been realized with a client/server architecture, in which the client-side provides the facilities to support modelling activities (e.g. a graphical editor) and the server-side is instead in charge of providing the functionalities related to the test case generation: verification of the model correctness, merging of the control and data flow, generation and post processing of the test cases.

6.1 Test cases generation workflow

The work flow of the TCG framework is reported in Figure 6.1. Each swimline of the depicted UML Activity Diagram, represents the related actor/component in charge of perform the activities inside described. In fact, in order to achieve the objective of test case generation, the following activities must be accomplished:

- **SUT and tests specification**: this step is composed by three concurrent activities. In fact, in order to obtain a complete specification of the SUT, both control and data flow of DSTM machines must be modelled. The specification is realized by modellers on the client-side, by using the graphical and textual facilities provided by the TCG framework. At the same time, modellers have to define the test specifications, according to the guidelines provided by the TSPs. The results of the modelling activities are three artefacts: the specification of the control flow of the SUT, its data declarations and the test specifications.

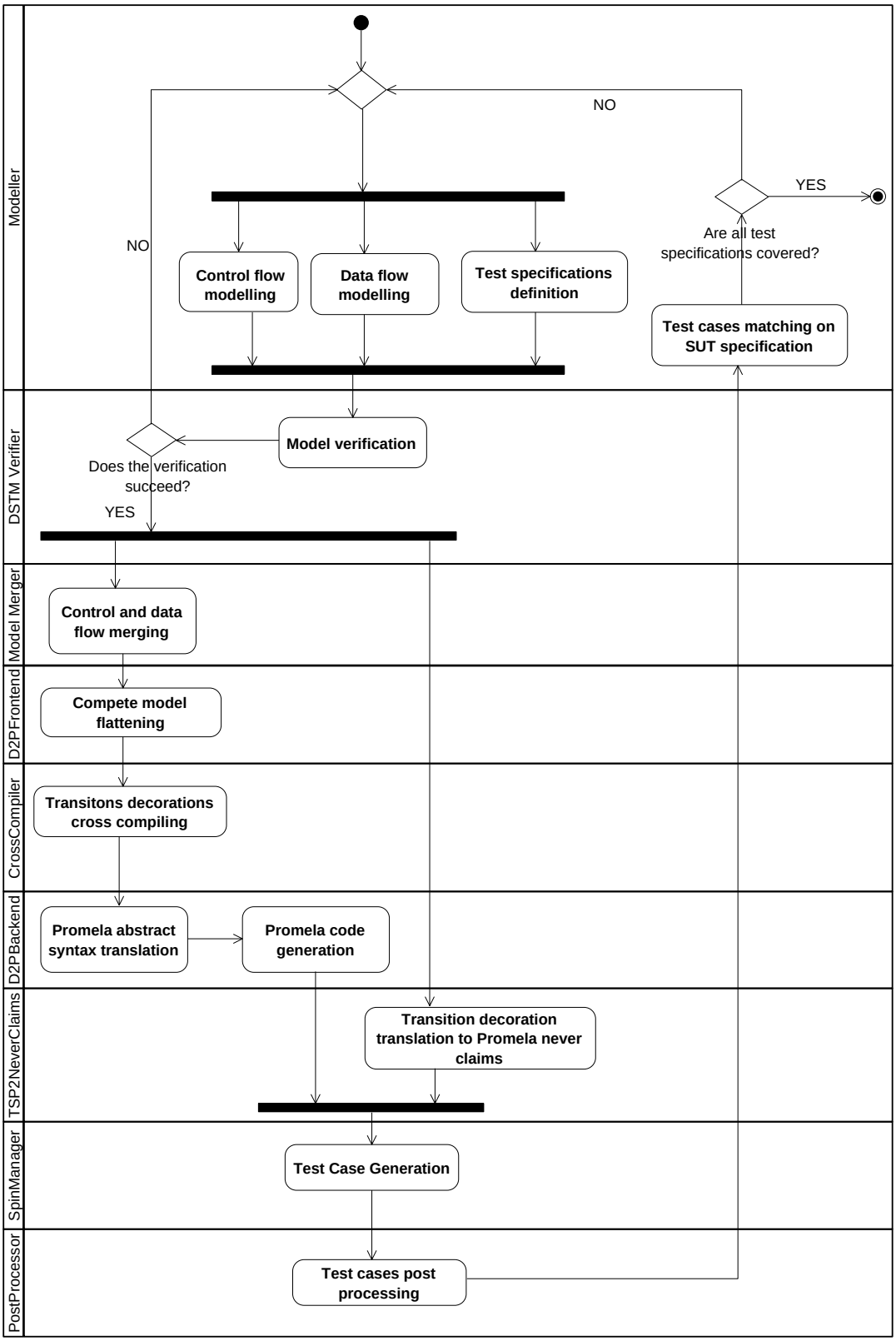


FIGURE 6.1: Workflow of the TCG framework

- **Model verification**: the correctness of the SUT specification (both of the control and data flow) according to the formal syntax and to the well-formedness constraints, must be verified in order to continue the generation process. If the model verification discovers flaws in the specification, problems are reported to the modeller that has to correct the specification.

- **Model merging**, that is performed by a *ModelMerger* component. The *ModelMerger*,in case of positive model verification, allows to merge in a unique DSTM model the control and data flows of a DSTM model, provided in two separated files. This merged model is necessary for the next phases of translation to Promela.

- **Flattening process**: according to the mapping process, the complete DSTM model must be flatten, in order to remove the hierarchical structure. This is made through a M2M transformation, realized by a specific component named *D2PFrontend*

- **Translation to Promela**: the flat DSTM model is transformed in a Promela model. This is made by two subsequent steps: a M2M transformation to translate the abstract syntax of DSTM in the abstract syntax of Promela and a M2T transformation, to generate the Promela code for the Model Checker.

- **Translation of the test specifications**: this is a key point concurrent to the previous since the test specifications must be translated in never claims to enable the model checker to generate counterexamples representing the specification of the test cases.

- **Test cases generation**: the generation is made by a *Spin Manager* component that merge the Promela code, related to the DSTM model and to the never claims. Than the Spin Model Checker is called and the verification of never claims is performed, to generate test cases.

- **Post processing**: the last activities is the post processing of the Spin Model Checker output, that allows to add some informations, in order to their specifications in TESQEL. Using the test cases modelled with TESQEL, modellers can trace them on the SUT specification.

The workflow above described has been realized with the architecture showed in Figure 6.2. The TCG framework has been defined with a service-oriented architecture. The TCG, in fact, communicates over a REST [34] protocol in order to provide all the functionalities as web services and make it highly interoperable

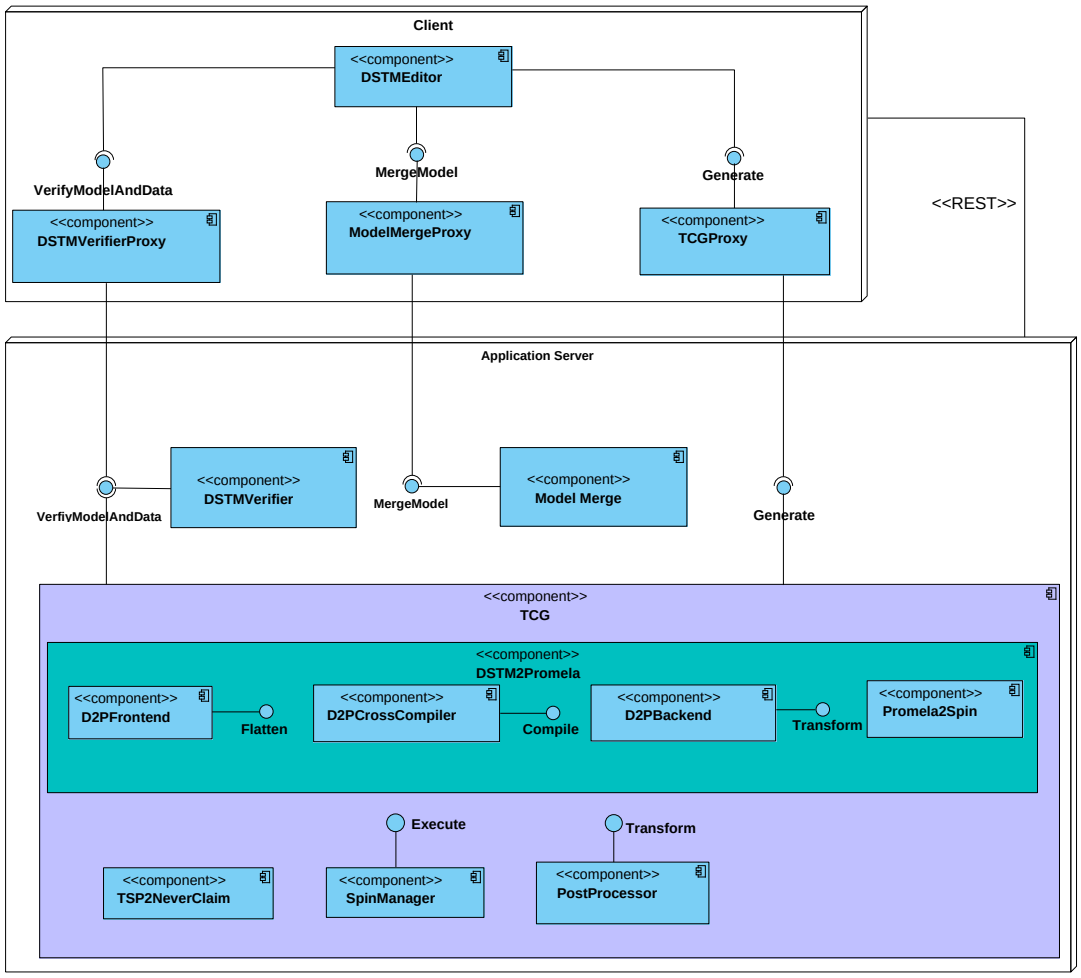


FIGURE 6.2: Component-level view of tcg framework

and integrable in a wider V&V process. The TCG has been organized in three services:

1. **DSTM Verifier service**, that allows to verify the correctness of the edited models, according to the formal syntax in Section 4.1.3 and the well-formedness constraints described in 4;

2. **Model Merger service**, since model control and data flow are edited in different files. The control flow, in fact, can be edited in a graphical form while the data flow, only by means of text. Thus, in order to insert our framework in a wider interoperable platform, we decide to explicitly release such service, in order to give the possibility to merge the two parts of a model that can be used for analyses different from the generation of test cases;

3. **TCG service**, that is the core of the approach since it is in charge of generate the test cases from the high-level model of the system. The TCG, in fact, realizes the transformation chain defined in the model-driven approach and orchestrate the other components in order to allow for the integration of the test cases within the overall V&V process.

Next sections discuss the architecture in Figure 6.2 showing the development process of each component.

6.2 DSTM Editor

As showed in Figure 6.2, the main component of the client-side of the TCG architecture, is a graphical editor, supporting the specification of the SUT. The *DSTMEditor* addresses the need for an user-friendly modelling environment able to simplify modelling activities. The graphical editor has been realized by exploiting functionalities provided by the Eclipse Modelling Framework (EMF) and the Graphical Modelling Framework of Eclipse (GMF) (see Section 2.5.2 for further details on such technologies). EMF and GMF provide several facilities and plugins to build tools, editors or in general Rich Client Platform (RCP). A detailed package diagram of the *DSTMEditor* is depicted in Figure 6.3. In such diagram, the yellow coloured packages are the exploited external libraries.

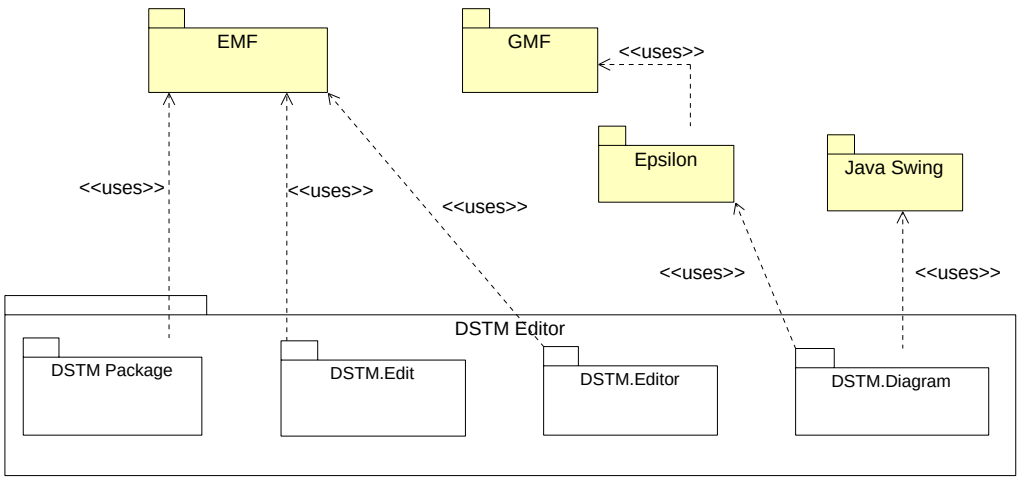


FIGURE 6.3: Architecture of DSTMEditor

The DSTMEditor is composed by four sub-packages:

- *DSTM package* is the object-oriented representation of the DSTM metamodel. In fact, according to the description of the EMF provided in Section 2.5.2,

each concept of the DSTM metamodel is a specialization of concepts provided by the ecore meta metamodel. According to this, the *DSTM* package is a key component for the overall TCG, since provides the necessary java code to manage DSTM models.

- *DSTM.edit*, that is a supporting package that includes generic reusable classes for building editors for EMF models.
- *DSTM.editor*, that allows to display the EMF models using standard desktop viewer (e.g. JFace) or property sheets
- *DSTM.diagram*, which is in charge of realize the graphical part of the considered EMF model editor. This package is based on the facilities provided by the Eugenia plugin, which is a graphical frontend for the Graphical Modelling Editor of Eclipse. In fact, since the GMF plugin is not very user-friendly, other plugins exist to create and customize graphical editor. One of them is Eugenia provided by Epsilon framework [31]. The Eugenia plugin allows to obtain graphical elements from the meta-concepts provided by the considered metamodel in a semi-automatic way, simply by annotating metamodel elements. As showed in the architecture, the *DSTM.Diagram* package uses also the facilities provided by the Java Swing libraries [91] since custom graphical interfaces have been developed, in order to support modellers in the initialization of attributes of models elements.

According to the service-oriented architecture of the framework, the DSTMEditor interacts with some proxies, implements the same interfaces of the related components in the application server. In particular, according to the services described in the previous section, three proxies have been developed for the *DSTMVerifier*, the *ModelMerger* and the *TCG*. Such proxies communicate over a REST protocol and are in charge of perform the marshalling and the unmarshalling of the requests/responses between clients and server (i.e. models, data declarations, test specifications and resulting test sequences).

6.3 DSTM Verifier

A complete DSTM model, is composed by data declarations and a definition of the control flow. Both of them, must be compliant to the syntax and the semantics of DSTM, specified in Chapter 4 In order to address constraints demanded by the syntax and semantics of the language, different approaches can be followed. For

example, the metamodel can be enriched with additional constraints, by exploiting facilities of the metamodelling environment or by using an ad-hoc language for constraints (e.g. OCL). Another approach, instead, is to demand the verification of such constraints to an external tool. The DSTMVerifier component, is based on this second approach; in fact, in order to maintain modelling activities as simple as possible, a specific tools has been developed with the responsibilities of:

- verify the data flow of the model, i.e. the declarations related to custom data types, variables and channels
- verify the decorations of transitions, i.e. their triggers, conditions and actions;
- verify the correctness of the model based on the well-formedness constraints not defined at the metamodel level.

The first two objectives can be achieved by the definition of a syntax analysis process. The syntax analysis, or *parsing* is the process that allows to verify if sequences of symbols (or streams) are conform to a set of formal rules called *grammars*. The syntax analysis process is composed by three phases:

- *Lexical analysis*: streams are examined and grouped on the basis of their meanings. Such groups, called *tokens* are then matched with a set of rules typically given as regular expressions. This phase produce sequences of tokens.
- *Syntactical analysis*: the token streams are matched with another set of rules, called grammars. Each rule in the grammar, is called *production*. A production describes a valid tree-like structure in terms of tokens and, recursively, in terms of other productions. This process is usually performed by a routine called *syntactical analyser*. The rules we consider, are expressed in form of context-free grammars (Subsection 2.5.2). CFGs allow to perform an efficient analysis of recursive tree-structured text, which means that each structure identified can be viewed as a tree node interconnected with each other node to compose the syntax tree. Thus, the syntax tree can be represented in different ways. Two of the most important are the Abstract Syntax Tree (AST) and the Concrete Syntax Tree (CST). CSTs are a more concrete view of the input token stream, since they represent in a tree-like form the input as it is parsed during the syntax analysis. The AST instead is more abstract

then the CST and allows to reproduce the structure of the input in a hierarchical data structure, useful for further analysis and transformations [62]. In the translation between different languages, ASTs are needed because of the nature of languages. Languages are often ambiguous by nature. In order to avoid this ambiguity, CFGs are useful but there are some aspects that they cannot address, such as details which require a context to determine their validity and behaviour.

- *Semantic analysis* since most languages (included the domain-specific ones) cannot be specified completely using a CFG. In fact there is the need to additional rules to define other constraints of the language. Thus, costraints, can be given using natural language or specific languages for constraints.

The realization of the DSTMVerifier, have been performed by following the below steps:

- Definition of the grammars related to data declarations, triggers, conditions and actions of transitions. Grammars have been expressed by using a BNF notation (Subsection 2.5.2) and used to verify the lexical correctness of the model structure.

- Generation, with a parser generator, called SableCC [80], of the packages to check the consistency of the data flow of the DSTM models;

- Analysis of model constraints, trough a java-based software module, in charge of make an exhaustive verification of such constraints on edited models.

An example of BNF grammar, defined for the syntactical verification of triggers on the transition decorations is reported in Listing 6.1.

LISTING 6.1: BNF grammar for Triggers

```
1  Helpers
2  eol = 11 | 9 | 10 | 13 | 10 13;
3  sp = ' ';
4  digit = ['0'..'9'];
5  char = [['a'..'z']+['A'..'Z']];
6  echar = [[['a'..'z']+['A'..'Z']]+'_' ];
7  plus = '+';
8  minus= '-';
9  Tokens
10 int = 'Int';
11 bool = 'Bool';
12 enum = 'Enum';
13 chn = 'Chn';
14 struct= 'Struct';
15 mtype= 'Mtype';
```

```
of = 'of';
true = 'true';
false = 'false';
full='full';
empty='empty';
spaces = eol+;
space = sp+;
external= 'external';
internal='internal';
identifier = char (digit | echar)*;
numbers = digit*;
semi = ';';
comma = ',';
qmark = '?';
colon=':';
plus = '+';
minus = '-';
star='*';
slash='/';
assign = ':=';
lt = '<';
gt = '>';
eq = '==';
not='!';
amp_amp='&&';
bar_bar='||';
lteq = '<=';
gteq = '>=';
plus_plus = '++';
minus_minus = '--';
l_par = '(';
r_par = ')';
l_bkt = '[';
r_bkt = ']';
l_brc = '{';
r_brc = '}';

Ignored Tokens
space,spaces;

Productions
trigger =
 {unique} term;

term =
 {nested} l_par term r_par|
 {unary}  not term |
 {simple} atomic |
 {binary} l_par first binop second r_par;

binop =
 {andop} amp_amp |
 {orop} bar_bar;

first =
 {left} term;

second=
```

```
74  {right} term;
75
76  atomic =
77   {message} var qmark |
78   {messagetype} var qmark simpletype;
79
80  simpletype=
81   {basic} basic|
82   {compound} compound_name;
83
84  compound_name=
85   {id} defined;
86
87  basic=
88   {int} int |
89   {bool} bool |
90   {ename} e_name|
91   {chant} chn l_bkt defined r_bkt |
92   {chn} chn;
93
94  e_name=
95   {enum} defined;
96
97  var=
98   {simple} defined;
99
100 defined =
101  {simple} identifier;
```

As stated before, each rule of the grammar is represented as a *production* in the form $A \rightarrow \alpha$ which states that the symbol on the left-hand side must be replaced by one of the alternatives on the right-hand side. The package structure of the generated parser, from the grammars, is represented in Figure 6.4.

According to the Figure 6.4, a package is created for each defined grammar (data, trigger, condition, action and parameter). However since transition decoration and parameters are related to data declarations, the *Data* package is common. Moreover, each package is further divided in the following subpackages:

- *Lexer* package, which contains the Lexer and LexerException classes, respectively in charge of perform the lexical analysis and rise exceptions if such analysis fails;
- *Parser* package, which contains Parser and ParserException classes with the same aims of the previous;
- *Node* package containing all the classes to define the typed AST. As discussed, the AST consists in an intermediate representation of the model that serves for further analysis or transformations.

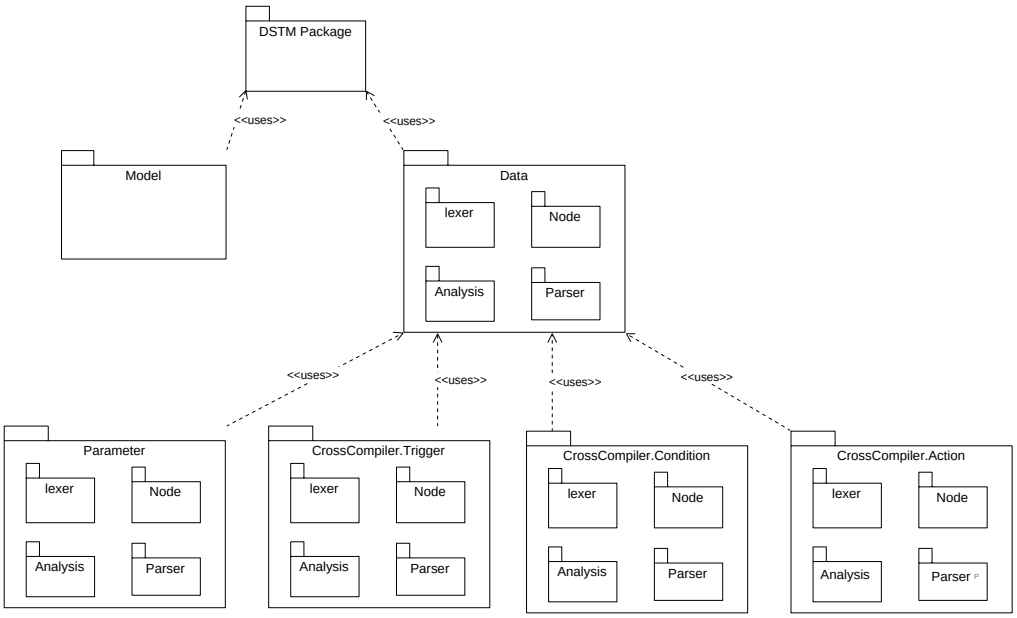


FIGURE 6.4: DSTMVerifier package diagram

- *Analysis* package containing the AST walkers. A tree-walkers is a class that visit all the nodes of an AST in a predefined order. Walkers allow to cross the AST in a depth-first traversal by calling proper in and out methods, when nodes are entered and exited. The adoption of walkers makes the parser easy to reuse and extend because the same tree-walkers can be used as the parents of many different walker classes.

The above packages allow to perform a syntax analysis of transition decorations and data. With the respect to the verification well-formedness constraints, a dedicated package has been developed and named *Model* package. Model constraints are expressed by means of java code. As for example, a relevant constraint is that for each machine only one *InitialNode* must exist. Such constraint can be expressed, by exploiting the primitives of the DSTMPackage. Listing 6.2 represents the *uniqueness* constraint.

LISTING 6.2: Uniqueness constraint of InitialNode

```
1  for (Machine machine : model.getMachines()) {
2   int count = 0;
3   for (Vertex vertex : machine.getVertexes()) {
4    if(vertex.getClass().getName().compareTo("DSTM4Rail.impl.
         InitialNodeImpl")==0){
5     count++;
6     if(count>1){
7      Violation v = new ModelViolation("UNQINIT", "Machine", (EObject)
           machine);
8      this.violations.add(v);
```

```
9           retval = false;
10        }
11      }
12    }
13  }
```

6.4 Model Merge

The *DSTMEditor* allows to model separately the model structure (*e.g.* the system control flow) and the data declarations (*e.g.* the data flow). At the same way, the *DSTMVerifier* verifies separately their correctness. However there is the need to merge the two model parts in order to enable the model transformation process. The component that is in charge of merging the control and the data flow of the system model is the *Model Merger*. The structure of the *Model Merger* is represented in Figure 6.5.

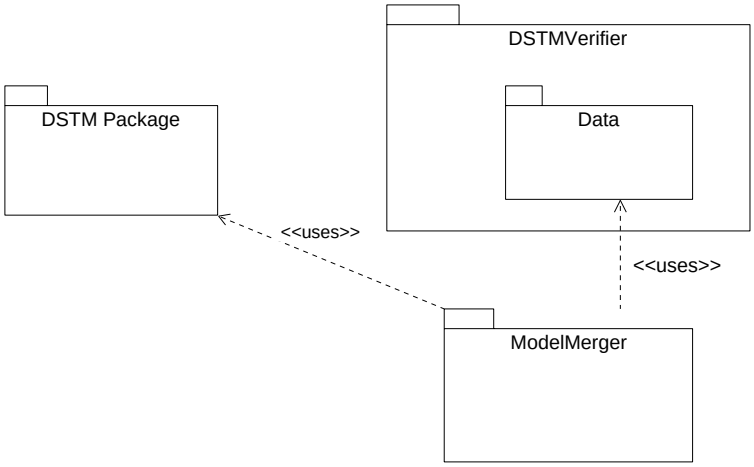


FIGURE 6.5: ModelMerger package diagram

The merging process consists of two phases:

- *Data exploration*, in which AST walkers of the *Data* package of the *DSTMVerifier* are used in order to parse the data declarations, and create the corresponding EObjects to be inserted in the DSTM model. Such EObjects are inserted in proper data structures, at the basis of the next phase;
- *Data insertion*: exploiting data structures created in the previous phase, the merged model is constructed by injecting EObjects created, in the model part containing the control flow of the SUT specification.

The verification of the control and data flow performed before the merging process, guarantee the correctness of the generated merged model.

6.5 DSTM2Promela

The DSTM2Promela component is one of the key components to enable the automatic generation of test case. It is based on the mapping process discussed in Chapter 5. According to such mapping process, the *DSTM2Promela* component has been divided in the following sub-components:

- *D2PFrontend* that is in charge of perform the flattening process finalized to removal of the hierarchical structure of the model. The outcome of such component is still a DSTM model;
- *D2PCrossCompiler* that allows to translate the DSTM syntax on transitions decorations to the Promela syntax;
- *D2PBackend*, that is in charge of translate the cross compiled DSTM model in a Platform Specific Model (PSM) compliant with the Promela abstract syntax, specified by means of a metamodel;
- *D2PPromela2Spin*, that allows for the code generation, from the PSM expressed using the Promela abstract syntax, executable by the Spin Model checker.

Next subsections discuss the realization of the above components.

6.5.1 D2PFrontend

As stated before, the *D2PFrontend* is in charge of perform the flattening process at the basis of the automatic generation approach that removes the hierarchical structure. Conceptually, since the source and the target language of such process is the same, this is a refinement of the PIM. In agreement with model driven principles, described in Section 2.4, this refinement can be performed by means of a model transformation. According to this, the *D2PFrontend* has been realized as an ATL model-to-model transformation. The target DSTM model generated by the flattening process owns the same data types, channels and variables of the source, but is different from the point of view of the machine structure and with the respect to fork, boxes, join and exiting nodes.

Mapping of boxes According to the syntactical mapping of the previous chapter, boxes can be isolated (simple boxes) or connected to a fork (synchronous or asynchronous). In the first case, the mapping process demand that the box has to be substituted by a Node with named constituted by the machine name to which the box belongs and the same box name. The matched rule in charge of realize such mapping is showed in Listing 6.3.

LISTING 6.3: simpleBox2Node rule

```
1  rule simpleBox2Node {
2   from
3    BoxFrom : DSTMRail!Box (
4     not BoxFrom.exitsAFork() and
5     not BoxFrom.entersAJoin()
6    )
7   to
8    NodeTo : DSTMRail!Node (
9     name <- BoxFrom.machine.name + thisModule.separator() + BoxFrom.name,
10    machine <- BoxFrom.machine
11   )
12 }
```

The replacement of a box connected to a fork is more complex. In fact, it needs of a matched rule, triggered by a transition entering in a box and with a source in a fork, and has to be substituted with a single transition, with the same triggers and conditions of the fork exiting transitions, but with the actions, related to the instantiation of the machine of the considered box (e.g. the calls to run operator). An example of matched rule to replace the fork, is reported in Listing 6.4.

LISTING 6.4: ForkCompoundTransition2Transition rule

```
1  rule ForkCompoundTransition2Transition {
2   from
3    TransitionFrom : DSTMRail!Transition (
4     TransitionFrom.destination.oclIsKindOf(DSTMRail!Fork)
5    )
6   to
7    TransitionTo : DSTMRail!Transition (
8     name <- TransitionFrom.machine.name + thisModule.separator() + TransitionFrom.
          name,
9     machine <- TransitionFrom.machine,
10    trigger <- thisModule.returnUndefinedIfEmpty(TransitionFrom.trigger),
11    condition <- thisModule.returnUndefinedIfEmpty(TransitionFrom.condition),
12    actions <- TransitionFrom.actions,
13    source <- TransitionTo.machine.getTransformedVertex(TransitionFrom.source.name),
14    destination <- TransitionTo.machine.getTransformedVertex(TransitionFrom.machine.
          transitions->select(t | t.source=TransitionFrom.destination).first().
          destination.name)
15   )
16  do {
17   for (trans in TransitionFrom.machine.transitions->select(t | t.source=
         TransitionFrom.destination)) {
18    TransitionTo.name <- TransitionTo.name.concat(thisModule.separator() + trans.
          name);
```

```
19      if (trans.destination.oclIsTypeOf(DSTMRail!Box)) {
20       TransitionTo.actions <- TransitionTo.manageMachineRun(trans.destination.
             instantiation.first());
21       TransitionTo.actions <- '#MyChildren[PidTemp]=1';
22       TransitionTo.actions <- '#HasToken[PidTemp]=1';
23      }
24     }
25    }
26   }
```

Termination and preemption Also each join case is treated by the *D2PFrontend*. In fact the join flattening is realized by a set of matched rules triggered by join pseudo nodes in the hierarchical source model. Such matched rule generate nodes and transitions according to the mapping process defined in Section 5.4.2. As for example, Listing 6.5 reports the case of a non preemptive join, on which insist both a node and several boxes. In the listing in fact, the rule is activated by the first transition entering in a join. The generated target transition inherits trigger and condition of the source transition; then a cycle on each source vertex of the considered join is performed in order to add to the transition the termination conditions and the termination actions.

LISTING 6.5: EnteringJoinTransition2Transition rule

```
1
2   rule EnteringJoinTransition2Transition {
3    from
4     TransitionFrom : DSTMRail!Transition (
5      TransitionFrom.destination.oclIsKindOf(DSTMRail!Join)
6     )
7    to
8     TransitionTo : DSTMRail!Transition (
9      name <- TransitionFrom.machine.name + thisModule.separator() + TransitionFrom.
           name,
10     machine <- TransitionFrom.machine,
11     trigger <- thisModule.returnUndefinedIfEmpty(TransitionFrom.trigger),
12     condition <- thisModule.returnUndefinedIfEmpty(TransitionFrom.condition),
13     actions <- TransitionFrom.actions,
14     source <- TransitionTo.machine.getTransformedVertex(TransitionFrom.source.name),
15     destination <- TransitionTo.machine.getTransformedVertex(TransitionFrom.machine.
           transitions->select(t | t.source=TransitionFrom.destination).first().
           destination.name)
16    )
17   do {
18    for (trans in TransitionFrom.machine.transitions->select(t | t.source=
         TransitionFrom.destination)) {
19     TransitionTo.name <- TransitionTo.name.concat(thisModule.separator() + trans.
           name);
20     if (trans.destination.oclIsTypeOf(DSTMRail!Box)) {
21      TransitionTo.condition.concat( '&& chTerm'_'+machine.name+'_exiting?[1])
22      TransitionTo.actions <- '#chTerm_' + trans.destination.name' + '_' + trans.
            machine.name + '!1';
23     }
```

```
24
25      }
26    }
27  }
```

With the respect to the above rule, conditions and actions of the transitions substituted to the join realize the termination mechanism by exploiting the communication over internal channels as described in Chapter 5. Termination channels are named with the prefix *ChTerm* and with the name of the machine to which belongs and the name of the exiting node to which is related. The *D2PFrontend* add a further channel, named with the same prefix and the name of the machine to which belongs. This further channel is used by the caller machine to allow the instance to terminate. The matched rule in charge of realize such mechanism, is triggered by each exiting node and its reported in Listing 6.6. The additional *ChTerm* channel is indeed created by the rule that transform the source DSTM machine in the target DSTM machine.

LISTING 6.6: ExitingNode2ExitingNode rule

```
1  rule ExitingNode2ExitingNode {
2   from
3    ExitingNodeFrom : DSTMRail!ExitingNode
4   to
5    ExitingNodeTo : DSTMRail!ExitingNode (
6     name <- ExitingNodeFrom.machine.name + thisModule.separator() + ExitingNodeFrom.
          name,
7     machine <- ExitingNodeFrom.machine
8    ),
9    chTermParamTo : DSTMRail!Parameter (
10    machine <- ExitingNodeTo.machine,
11    name <- 'chTerm' + thisModule.separator() + ExitingNodeFrom.name,
12    type <- ExitingNodeFrom.machine.DSTM.getChnBoolType()
13   )
14 }
```

6.5.2 D2PCrosscompiler

The flatten model own decorations on transitions expressed with the DSTM syntax. In order to translate these last in the Promela syntax, a specific component named *CrossCompiler* has been developed. The DSTMVerifier component is in charge of verify the correctness of decorations on transitions, with the respect to the constraints demanded by the formal syntax and realized by the grammars. In fact, thanks to the generation of the ASTs, it allows to identify each token of triggers, conditions and actions. The implementation of the ASTs is based on the grammars defined for transition decorations. Thus, the *CrossCompiler* aims at reusing such ASTs structures, in order to identify tokens of triggers, conditions

and actions, and generate the corresponding syntactical Promela elements. The *CrossCompiler* performs its tasks according to the mappings summarized in Table 5.5. The package diagram of the *CrossCompiler* is depicted in Figure 6.6.

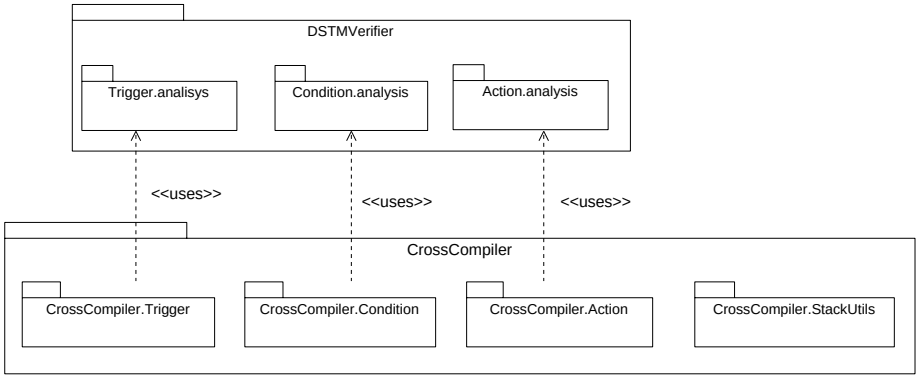


FIGURE 6.6: CrossCompiler package diagram

The main package is the *CrossCompiler* that is in charge to start the recognition of transitions decorations on the merged flatten model. Thus, in order to reuse the ASTs structures generated from the grammars, the packages *CrossCompiler.Trigger*, *CrossCompiler.Condition*and *CrossCompiler.action* reuse the corresponding packages of the *DSTMVerifier*. The additional sub-package *CrossCompiler.StackUtils* contains facilities to support the recognition of simple operations inside complex logical expressions on transitions decorations. The the model resulting from the cross compiling process, is an hybrid model composed by a syntactical structure expressed with the DSTM language, and decorations on transitions expressed in a Promela syntax.

6.5.3 Promela metamodel

As the *DSTM2DSTM* transformation, the DSTM2Promela is realized by using ATL. In order to enable the transformation, the source and the target metamodel must be provided. The metamodel of the source language, DSTM, is described in Section 4.1.2. Nevertheless, another relevant activity of work described in this thesis has been the definition of the abstract syntax of the Promela language, expressed by using an ecore diagram.

The problem of define a Promela metamodel is not a new topic. Nevertheless, at the current state, any standard metamodel to which is possible to refer has been defined. Indeed there are many attempts, specially in scientific literature, focused on the definition of a proper Promela metamodel. McUmber et al [66],

define a framework to translate UML models into Promela formal models, in order to enable simulation and analysis through model checking. The definition of Promela metamodel is not the focus of the work, even if it is used to support the formal definition of the mapping rules. The metamodel in this case, is realized by means of UML Class diagram. Another approach to define a Promela metamodel, is the one proposed by Abdulhameed et al [2] in which the aim is to verify SysML functional requirements using state machine diagrams. In this case, in order to enable the model transformation process with the ATL framework, there is the need to provide an ecore version of the Promela metamodel. The metamodel proposed is very detailed and considers all the syntactical constructs provided by the Promela language. On the same line, dos Santos et al. define an Ecore-based Promela metamodel to enable mappings between an executable version of UML, they propose, and Promela. Also in this case, the need for the metamodel is due to adoption of ETL language for model transformations [58]. The metamodels above discussed are very similar in their objectives. In fact, they aiming at provide all possible concepts of the the Promela syntax, even if they are not all used in the transformation process. The proposed Promela metamodel (Figure 6.7) here described, instead, relies on the attempt to have a metamodel as much possible simple and general. This design choice allows to simplify the implementation of the abstract mappings defined in the previous chapter. Moreover, be more general, will allow to easily modify the metamodel for further formal languages in avoiding the complete re-engineering of the transformations below described.

The main concept of the metamodel depicted in Figure 6.7, is the *Model* which is an abstraction of a Promela module. It is characterized by the name and the maximum number of processes which can be instantiated. Each *Model* can be further composed by:

- *Processes*, characterized by a name, a boolean attribute to establish if is an active process or not, and which is the main process. The concept of main process does not exist in Promela. Nevertheless, such attribute has been defined in order to maintain in the the target model the information related to the main machine of the DSTM model;

- *Enumerations*, characterized by a name and a list of literals;

- *Variables*, that are further divided in *Primitive* and *Channel*. A *Primitive* variable can have a initial value and a *type* chosen from the ones natively provided by Promela. The complete list of the native type provided by Promela is expressed by means of the the data type *DefaultType*. *Channels*

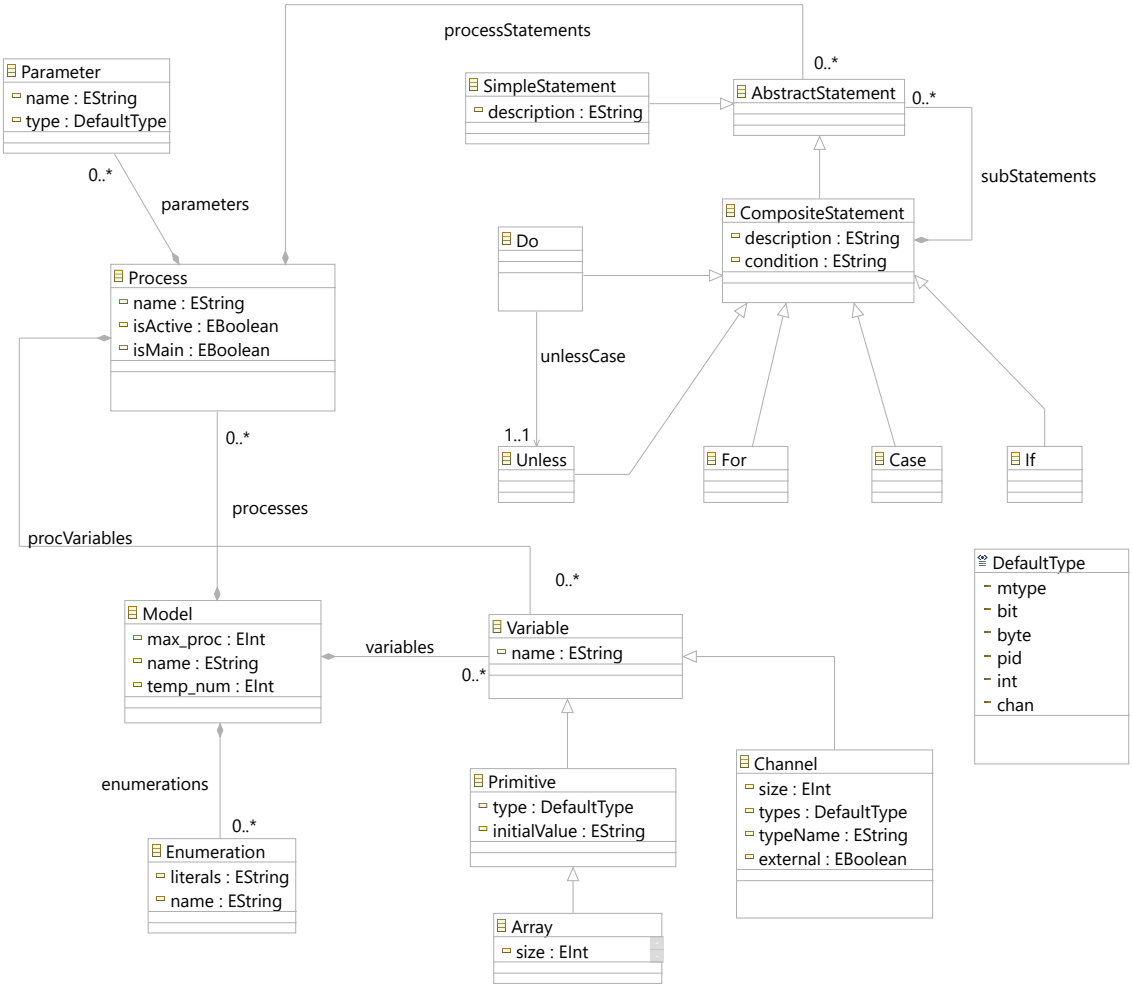


FIGURE 6.7: Promela metamodel

are, indeed, characterized by the *size* (emphi.e. the maximum number of messages that can be exchanged in the channel), a set of *types* (*i.e.* the fields that each message can contains). Moreover a *typeName* has been added, to trace the DSTM type associated to the considered channel, and a boolean attribute to know if the channel is external or not. This last is very important since, as discussed in Section 5.3.2, the Promela language, does not make difference between external or internal channels. *Variables* of the metamodel are further extended to consider Promela arrays that are sets of primitive variables.

- *Parameters* representing the list of parameters that are passed to the *run* operator, when a new process has to be activated. As the *Variable*, a *Parameter* has a name and a type from the defaults provided by Promela.

The right-side part of the metamodel, describes the control structures of the Promela language. Control structure are described by referring to the *Composite*

design pattern [38]. The *Composite design pattern*, in fact, allows to address a tree-like structure in which there is the need to distinguish between nodes and leafs. In the specific case, an *AbstractStatement* can be a *SimpleStatement* (a leaf) or a *CompositeStatement* (a node) characterized by the fact that is a composition of other statements. The *CompositeStatement* is then specialized in order to represent the control structures provided by Promela (*Do*, *If*, *Case* and *For*). Anyway, each statement is characterized by a description while only the composite ones can have a condition. Descriptions and conditions can be simple strings, since the presence of the *CrossCompiler* that is in charge of translate the syntax from the DSTM to the Promela language.

6.5.4 D2PBackend

The metamodel above discussed, allow to enable the transformation process realized by the *D2PBackend* component. The *D2PBackend*, is in charge of realizing the mapping between the flatten version of a DSTM model to an intermediate model, representing the same model in the Promela abstract syntax. The *D2PBackend* is based on a model transformation realized by means of ATL language. At the contrary of the previous subsection describing the *D2PFrontend*, the source and the target formalism in this case are different. Below, rules supporting this transformation step are described.

Type System mappings The Type System, provided by DSTM, is composed by *tSimple* and *tMulti* types. The former are further divided in *tBasic* and *tCompound* types. With the respect to the mapping process, *tBasic* are mapped in the corresponding *DefaultType* provided by the Promela metamodel. The only exception is represented by the *tEnum* since the presence of the set of literals. In fact, the enumerative type provided by DSTM is mapped in a set of Promela *mtypes*, used to introduce symbolic names for constant values. The rule that allows to realize such mapping is a matched rule (see Section 2.5.4) and is reported in Listing 6.7.

LISTING 6.7: tEnum2Mtype

```
1  rule tEnum2Enumerative {
2  from
3   tEnum: DSTMRail!tEnum (
4    enum.tCompound.oclIsUndefined() and enum.tMultiType.oclIsUndefined()
5   )
6  to
7   enumeration: PROMELA!Enumeration (
8    name <- enum.name,
9    literals <- enum.literals
10  )
11
```

```
12 }
```

A different approach has been followed for *tMultiType*. In fact, *tMultiType* is not explicitly transformed since Promela does not provide a corresponding concept. Thus multi types are considered only with the respect to the transformations related to channel and to the *Chn[T]*. Such transformations are described described below in this section.

Global variables DSTM global variables are key elements of the language data flow. Global variable can be only of a simple type, with an exception on the *Chn[T]* that can contain a multi type and must be carefully transformed. In general transformation of global variables is quite straightforward since they are mapped in Promela variable with the same name and the corresponding DSTM simple type in Promela. Such correspondence is obtained by using a supporting helper, defined with the same ATL language. In the Listing 6.8 the type match is obtained by using an ATL helper, that returns the Promela type corresponding to the type of the DSTM global variable.

LISTING 6.8: simpleGlobal2primitive rule

```
1 rule global2primitive {
2  from
3   var: DSTMRail!Variable (not var.type.name.contains('Chn['))
4  to
5   var_global: PROMELA!Primitive (
6    name <- var.name
7    type <- thisModule.correspondingTypeOf(var.type.name)
8   )
9 }
```

The transformation of simple variables of *Chn[T]* type is more complex if the type T contained is of multi type. The mapping process for such kind of variables in described in Section 5.3.3. They are of course of *chan* type but, in order to realize such mapping a dedicated called rule is adopted. The main rule (e.g. the *DSTM2Promela* rule) is in charge of recognize the eventual multi type inside the *Chn[T]* and call such rule, as many times as the number of subtypes in T. The called rule for *Chn[t]* variables is represented in Listing 6.9.

LISTING 6.9: primitiveChnVariable rule

```
1 rule primitiveChnVariable(name : String, type: String, model : PROMELA!Model){
2  to
3   multiVar_global: PROMELA!Primitive (
4    name <- name+'_'+type,
5    type <- 'chan'
6
7   )
8  do{
```

```
9     model.variables<-multiVar_global;
10   }
11 }
```

Channels The transformation of DSTM channels in Promela is another complex task, for several reasons: (i) rule transforming external channels, are partially involved in the generation of the process describing the *Step semantics*, (ii) Promela does not make a difference between external or internal channels and (iii) DSTM channels can be of simple and multi type. This paragraph in focused only on the syntactical generation process of Promela channels. The step semantics is treated separately, below in this section.

Promela channels of simple type (both internal and external) are generated by using matched rules, composed by:

- a source pattern which indicates the source DSTM channel considered (i.e. external or internal)
- a target pattern, that creates the corresponding Promela channel, by setting the name, the bound, in the case of the internal channels, and the type of the channel;
- a do pattern, that contains the necessary imperative code to construct the set of messages types of the channel. According to this *tBasic* and *tCompound* channel type are treated separately. In fact, the case of *tBasic*, the channel contains only one message of a predefined Promela type. Otherwise, for *tCompound*, an iteration must be performed in order add each message type, contained in the compound. The rule in Listing 6.10 clarifies the transformation process. Since the generation is the same for both external and internal channel, the listing shows the generation only for an internal one.

LISTING 6.10: simpleInternalToPromelaChannel rule

```
1  rule simpleInternalToPromelaChannel {
2   from
3    intChan: DSTMRail!cInternal (
4     intChan.channelType.oclIsKindOf(DSTMRail!tSimpleType) and not intChan.name.
5       contains('chTerm') and not intChan.name.contains('#')
6    )
7   to
8    promelaIntChan: PROMELA!Channel (
9     name <- intChan.name,
10    size <- intChan.bound,
11    typeName <- intChan.channelType.name
12   )
```

```
13  do{
14    -- case tBasic
15    if (intChan.channelType.oclIsKindOf(DSTMRail!tBasic)) {
16     if(intChan.channelType.oclIsTypeOf(DSTMRail!tInteger)) {
17      promelaIntChan.types<-'int';
18     }else if (intChan.channelType.oclIsTypeOf(DSTMRail!tEnum) and not
19       (intChan.channelType.name = 'Bool')){
20
21      promelaIntChan.types <- 'mtype';
22     }else if (intChan.channelType.oclIsTypeOf(DSTMRail!tEnum) and intChan.
23       channelType.name = 'Bool'){
24      promelaIntChan.types <- 'bit';
25     }else if (intChan.channelType.oclIsTypeOf(DSTMRail!tChannel)){
26      if(intChan.channelType.name.contains('Chn[') and intChan.channelType.name.
            contains(']')){
27       for(component in thisModule.getTypeFromName(intChan.channelType.name.toString
            ().substring(5, intChan.channelType.name.toString().size()-1)).
            composedBy){
28        promelaIntChan.types<-'chan';
29       }
30       }else{
31        promelaIntChan.types<-'chan';
32       }
33     }
34    -- case tCompound
35    }else if (intChan.channelType.oclIsKindOf(DSTMRail!tCompound)){
36     intChan.channelType.name;
37     for(p in  intChan.channelType.subtypes) {
38      if(p.oclIsTypeOf(DSTMRail!tInteger)) {
39       promelaIntChan.types<-'int';
40      }else if (p.oclIsTypeOf(DSTMRail!tEnum) and not (p.name = 'Bool')){
41       promelaIntChan.types<-'mtype';
42      }else if (p.oclIsTypeOf(DSTMRail!tEnum) and p.name = 'Bool'){
43       promelaIntChan.types<-'bit';
44      }else if (p.oclIsTypeOf(DSTMRail!tChannel)){
45       if(p.name.contains('Chn[') and p.name.contains(']')){
46        for(component in thisModule.getTypeFromName(p.name.toString().substring(5, p
            .name.toString().size()-1)).composedBy){
47         promelaIntChan.types<-'chan';
48       }
49       }else{
50        promelaIntChan.types<-'chan';
51       }
52
53      }
54     }
55    }
56  }
57 }
```

There is only one difference in case of multi type channels. In fact, according to the mapping process in Section 5.3.2, in order to generate single-type channels from the multi type one, called rules are adopted, and activated by the main rule. Such called rules miss the source pattern, but have the same target and do patterns of the matched rule of the simple case.

Processes Promela processes are key concepts in the transformation of DSTM machines in Promela. A *Process* is composed by process variables, process parameters and statements. The set of process variables consists of two enumerations containing the names of all the states and transitions belonging to the corresponding DSTM machine and a list of parameters. Such parameters are obtained by exploiting the do pattern of the rule, that activates a called rule for each parameter of the Machine (Listing 6.13).

The set of statements, instead, allows to realize the external behaviour of a process. In particular statements are grouped in two blocks: a *Do repetition* that represents the main loop inside the process. The remaining behaviour of the process, is generated by using the *resolveTemp* function, from the considered rule. The *resolveTemp* function provided by ATL, allows to point from an ATL rule to any target pattern of other matched rules, that can be generated from a given source model element [56]. The second block is the *Unless*. The *Unless* block allows to realize the mechanism of termination, described in Section 5.4.2, consisting in the send of messages over the termination channels. Termination channels are created by the *D2PFrontend* transformation and added by using a proper called rule, activated by the do pattern of the *machine2process* rule. Listing 6.11 represents the *machine2process* rule described in this paragraph. Listing 6.12 shows the called rule to add the termination channels in the process and finally Listing 6.13 shows the called rule to add parameter to the set of process variables.

LISTING 6.11: machine2process rule

```
1  rule machine2process {
2   from
3    machine: DSTMRail!Machine
4   to
5    process: PROMELA!Process (
6     name <- machine.name,
7     processStatements <- mainDo,
8     processStatements <- unlessDo, -- riabilitare l'unless
9     procVariables <- states
10
11   ),
12   state_enumeration: PROMELA!Enumeration (
13    name <- machine.name + '_states',
14    literals <- machine.vertexes -> collect(p | p.name)
15   ),
16   transition_enumeration: PROMELA!Enumeration (
17    name <- machine.name + '_transitions',
18    literals <- machine.transitions -> collect(p | p.name)
19   ),
20   states: PROMELA!Primitive (
21    name <- 'state',
22    type <- 'mtype',
23    initialValue <- 'initial'
```

```
24    ),
25
26    mainDo: PROMELA!Do (
27     subStatements <- machine.vertexes -> collect(p | thisModule.resolveTemp(p,'
          loopCase')),
28    ),
29    unlessDo: PROMELA!Unless (
30     subStatements <- statementUnless
31    ),
32    statementUnless: PROMELA!SimpleStatement (
33     description <- '(chTerm?[1] ||  dyingPid==parent);',
34     description <- 'if',
35     description <- '::chTerm?[1] -> chTerm?1;',
36     description <- 'fi;',
37     description <- 'dyingPid = _pid;',
38    )
39   do{
40    for (param in machine.parameters){
41     if(param.type.oclIsTypeOf(DSTMRail!tChannel)){
42      thisModule.addParameter(param, 'chan', process);
43     }else if(param.type.oclIsTypeOf(DSTMRail!tInteger)){
44       thisModule.addParameter(param, 'int', process);
45     }else if(param.type.oclIsTypeOf(DSTMRail!tEnum) and not (param.name = 'Bool')){
46       thisModule.addParameter(param, 'mtype', process);
47     }else if(param.type.oclIsTypeOf(DSTMRail!tEnum) and (param.name = 'Bool')){
48       thisModule.addParameter(param, 'bit', process);
49     }
50
51    }
52     if(machine.DSTM.main.name.equalsIgnoreCase(machine.name))
53     process.isMain <- true;
54
55    for (chTerm in machine.DSTM.channels -> select(ch | ch.name.contains(machine.
56      name))) {
57     if(chTerm.name.split('#').at(2) = machine.name){
58      chTerm.name.split('#').at(1).debug();
59      thisModule.addChTerm(chTerm, process);
60     }
61    }
62   }
63   }
```

LISTING 6.12: addChTerm rule

```
1   rule addChTerm(chan: DSTMRail!cInternal, proc: PROMELA!Process){
2    to
3     chTerm: PROMELA!Channel (
4      name <- chan.name.split('#').at(1),
5      size <- chan.bound,
6      types <- 'bit'
7     )
8    do{
9     proc.procVariables<-chTerm;
10   }
11  }
```

LISTING 6.13: The addParameter rule

```
rule addParameter (inParam: DSTMRail!Parameter, inType: String, proc: PROMELA!
    Process){
 to
  promelaParam: PROMELA!Parameter (
   name <- inParam.name,
   type <- inType
  )
 do{
  proc.parameters<-promelaParam;
 }
}
```

As stated before, to complete the behaviour of a Promela process, a *resolveTemp* function is called from the target pattern of the *machine2process* rule. The rule activated by the *resolveTemp* is the *vertex2case*. Such rule is activated as many time as the number of vertexes of the machine. The rule, set the condition of the case, by considering the current vertex and the enabledeness condition (i.e. the owning of the token). Vertexes of a machine, can have zero or more exiting transitions; for each of them, a called rule, named *transition2If* is activated. The *transition2If* called rule, is in charge of creating an *If* statement, inside the *Case*, whose condition is the logic *And*, between trigger and condition of the corresponding transition, and whose other statements are its actions and the print statements containing the TESQEL directives. Actions and TESQEL directives are added by using the do pattern of the rule. Listing 6.14 shows the *vertex2case* rule while the listing 6.15 shows the *transition2If* called rule.

LISTING 6.14: Vertex2Case rule

```
rule Vertex2Case {
 from
  vertex: DSTMRail!Vertex
 to
  loopCase: PROMELA!Case (
   condition <- 'state == ' + vertex.name + ''+ ' && HasToken[_pid]==1',
   subStatements <- has_token
  ),
  has_token: PROMELA!SimpleStatement (
   description <- 'HasToken[_pid]=0'
  )
 do {
  for (p in vertex.machine.transitions -> select(tr | tr.source.name = vertex.name)
      ) {
   thisModule.transitionToIf(p, loopCase);
  }
 }
}
```

LISTING 6.15: transition2If rule

```
rule transition2If (trans: DSTMRail!Transition, loopCase: PROMELA!Case){

```

```
3   to
4    transitionToif: PROMELA!If (
5     condition <- trans.trigger + '&&' + trans.trigger
6    )
7
8   do{
9    for (action in trans.actions) {
10    transitionToif.description <- action.toString();
11   }
12   transitionToif.description <- 'printf("<current>'+trans.source.name+'</current>\\
         n")';
13   transitionToif.description <- 'printf("<transition>'+trans.name+'</transition>\\n
         ")';
14   transitionToif.description <- 'printf("<next>'+trans.destination.name+'</next>\\n
         ")';
15   transitionToif.description <- 'printf("</firing>\\n")';
16   transitionToif.description <- 'state = ' + trans.destination.name;
17   transitionToif.description <- 'LastTransition=' + trans.name;
18   transitionToif.description <- 'NoFirings=0';
19
20   --insertion in the promela Case
21   loopCase.subStatements<-transitionToif;
22  }
23 }
24
```

6.5.5 Engine generation

The *Step semantics* of DSTM by generating a proper Promela process named *Engine* process. Main features of the step semantics are discussed in Section 5.5. This paragraph describes how the step semantics is realized in Promela. As stated in the Section 5.5, the state-based nature of the step semantics, lead us to model it by a special Promela process, the *Engine*. As already discussed its main responsibilities are: (1) instantiation of the main process, (2) non-deterministic generation of messages on the external channels and (3) assignment of the token to the main machine, in order to start the step execution. The *Engine* is the first process to be activated and is the only process required to be running in the initial state. Then, it is activated whenever no statements is executable in any process belonging to the system model, that means that the step execution is terminated. The handling of the step execution termination can be catch thanks to the presence of a *timeout* variable being true. The run of the main machine, is realized by a call to the *run* operator provided by Promela on the main process, that stores the pid of the process in a *_pid* variable (Listing 6.16, line 22-24). Then the *Engine* can proceeds to initialize the channels.

The generation of the *Engine* process is realized by using a called rule, activated by the target pattern of the main rule (the *DSTM2Promela*). A dedicated

matched rule is not possible, since there is no an equivalent machine in the DSTM model. Listing 6.16 shows the generation above discussed.

LISTING 6.16: Generation of the Engine process

```
1  ....
2
3  engine: PROMELA!Process (
4   name <- 'Engine',
5   isActive <- true,
6   procVariables <- DSTM.channels -> select(ch | ch.name.contains('Engine')) ->
        collect(p | thisModule.resolveTemp(p, 'chTermOut')),
7   procVariables <- pid_main,
8   processStatements <- engineDo
9  ),
10 engineDo: PROMELA!Do (
11  subStatements <- timeout,
12  subStatements <- DSTM.channels -> collect(p | thisModule.resolveTemp(p,'
        extChanFullClause'))
13 ),
14 timeout: PROMELA!Case (
15  condition <- 'timeout',
16  subStatements <- generation
17 ),
18 generation: PROMELA!SimpleStatement (
19  description <- 'goto generation'
20 ),
21 pid_main: PROMELA!Primitive (
22  name <- 'PidMain',
23  type <- 'pid',
24  initialValue <- 'run ' + DSTM.main.name + '('_pid',thisModule.getParametersList(
        DSTM.main.name))'
25 )
```

In order to guarantee that the *Engine* is the first process to be executed, the attribute *isActive*, provided in the Promela metamodel, is set to true. Moreover, the *timeout* variable is generated (Listing 6.16, lines 14-17) and all the statements to call the run operator and register the process pid. An ATL helper is used to retrieve the list of parameters, that the run operator passes to the main process when instantiate it.

After the run of the main process, the *Engine* is in charge of initialize the external channels. In order to perform such initialization, the *Engine* uses local variables to non-deterministically generate new messages in them. These local variables correspond to the fields of the compound types exchanged over those channels. In particular a temp variable is created for each field contained in each external channel. The Figure 6.8 clarifies the concept: the showed channel contains a set of two types, *type1* and *type2*. For each of them, a temporary variable is created, in order to generate non-deterministically messages over it.

The generation of the temp variables is made with a called rule (Listing 6.18) activated from the transformation rules of both simple and multi type channels.

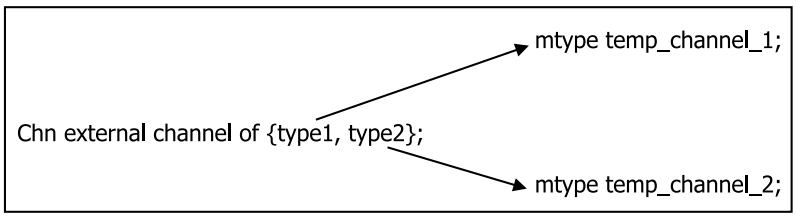


FIGURE 6.8: Generation of temp variables from an external channel

LISTING 6.17: Called rule for temporary variable generation

```
1  rule createEngineVariable(engine: PROMELA!Process, name: String){
2   to
3    var: PROMELA!Primitive (
4     type <- 'mtype',
5     name <- name
6    )
7   do{
8    engine.procVariables<-var;
9   }
10 }
```

Once temp variables are created, the *Engine* process starts an atomic block in which it non-deterministically generates the message to be sent over the channels. The starting statement of this block is identified by the label *generation*. The evolution of the *Engine* process is made by generating a *Case* in its main loop for each external channel. The condition to be checked is if the channel is not full (or that all sub channel are not contemporary full in case of multi type external channels). Then, by using another called rule, named *addEngineStatement*, an *If* statement is generated for each subtypes contained by the considered channel. In case of multi type channels, the *If* statement must take into account all possible permutations of the values in the sub channels. Listing *lst:engine4* shows the code of the *addEngineStatement* and *addIfLenSubstatement* called rules.

LISTING 6.18: Do pattern of an external channel rule to generate temporary variables

```
1  --Do pattern
2   ...
3   for(p in thisModule.getLiteralsFromChannelName(promelaExtChan)){
4      thisModule.counter <- thisModule.counter + 1;
5      thisModule.createEngineVariable(thisModule.getEngineHandler(), ('temp_' +
           promelaExtChan.name + '_' + thisModule.counter));
6      thisModule.addEngineStatement(extChanLenClause, ('temp_' +promelaExtChan.name +
           '_' + thisModule.counter), thisModule.getLiteralsFromEnumName(p));
7     }
8   ...
```

LISTING 6.19: Called rule to generate Cases and If of the Engine main loop

```
1  rule addEngineStatement(clause: PROMELA!If, chan_name: String, values : PROMELA!
       Enumeration){
2
3   to
4    mainIfOfChannelLen : PROMELA!If(
5    )
6   do{
7    for(p in values){
8     thisModule.addIfLenSubstatement(mainIfOfChannelLen, chan_name, p);
9    }
10  clause.subStatements<-mainIfOfChannelLen;
11  }
12 }
13
14 rule addIfLenSubstatement(mainIf: PROMELA!If, chan_name : String, value : String){
15  to
16   subIfStatementEngine: PROMELA!If (
17    condition <- chan_name + '=' + value
18   )
19  do{
20   mainIf.subStatements<-subIfStatementEngine;
21  }
22 }
```

The generation block ends by assigning the token to the main process. Then, the Engine process enters the do construct, where it waits until the Promela global variable *timeout* is evaluated to true. This happens when no statement is executable in the active processes, hence when all the SUT processes have consumed their token. In this case, Engine executes a jump to the generation label, starting a new step.

6.6 Promela2Spin: generate the Spin code

The *D2PBackend* transformation, according to model-driven principles discussed in Section 2.4.1, generates the Platform Specific Model in terms of abstract syntax of Promela language. However, in order to enable Spin to execute the test case generation, the concrete syntax (e.g. the Spin Model code) must be generated from the abstract one. The generation of the concrete syntax is performed by a M2T transformation starting from the PSM derived from the previous step. In agreement with model driven principles, the complexity of the passage between the PIM and the PSM, is embed in the M2M transformation. The M2T, indeed, is in charge of replicate the same structure, imposed by the M2M transformation, to the generated code. According to this, some *Promela2Spin* mappings are quite straightforward. However, in order to show the models structure, two relevant rules, are below described: the rule to generate a process, and the rule to generate the *Engine*.

Process generation According to the intermediate Promela model, a process can contain local variables, a main loop consisting in a *Do repetition* and eventually a set of other statements not included in the main loop. The M2T rule, that generates the spin code from the PSM model of a process, is reported in Listing 6.20.

LISTING 6.20: Acceleo rule to generate a Process code

```
1  [template public generateProcess(proc: Process)]
2  [if proc.isActive]active[/if] proctype [proc.name/](pid parent; mtype initial;[for
       (par : Parameter | proc.parameters)separator(';')][par.type/] [par.name/][/for
       ]) {
3  bit MyChildren['[MAX_PROC]'/];
4  [for (c : Channel | proc.procVariables->select(p | p.oclIsTypeOf(Channel)) )]
5  [ generateChannel(c) /]
6  [/for]
7  [for (v : Primitive | proc.procVariables->select(p | p.oclIsTypeOf(Primitive)) )]
8  [ generatePrimitiveVariable(v) /]
9  [/for]
10 [if not proc.name.equalsIgnoreCase('Engine')][generateDo(do, proc)/][/if]
11 [if proc.name.equalsIgnoreCase('Engine')][generateEngineDo(do, proc)/][/if]
12 [for (ifStmt : If | proc.processStatements->select(p | p.oclIsTypeOf(If)))]
13 [generateIf(ifStmt, proc)/]
14 [/for]
15 }
16 [/template]
```

The first part of the rule in the listing, performs the necessary iterations, to generate the code of process-local variables and channels. Then the statements related to the internal behaviour of a process are performed. In particular, the *generateDo* rule is called in case of a normal process, otherwise, is called the *generateDoEngine*. The *generateDo* template rule, is the core of the code generation, since is in charge of iterate over the statements of a process and of add the necessary elements that complete the code generation. Lines 1-14 of Listing 6.21 show the template rule to generate a process loop.

LISTING 6.21: Acceleo rule to generate the internal code of a process

```
1  [template public generateDo(do : Do, proc : Process)]
2  do
3  [for (subStmt : SimpleStatement | do.subStatements->select(oclIsKindOf(
       SimpleStatement)))]
4
5    [for(description : String | subStmt.description)]
6    [description/];
7    [/for]
8  [/for]
9  [for (case : Case | do.subStatements->select(oclIsKindOf(Case)))]
10  [generateDoCases(case)/]
11
12 [/for]
```

```
13 od [if not proc.name.equalsIgnoreCase('Engine')][generateUnless(proc)/][/
       if]
14 [/template]
15
16
17 [template public generateDoCases(case: Case)]
18  :: ([case.condition.toString()/]) ->
19 atomic{
20 [for (simpleStmt : SimpleStatement | case.subStatements->select(p | p.
       oclIsTypeOf(SimpleStatement)))]
21   [for(description : String | simpleStmt.description)]
22   [description/];
23   [/for]
24 [/for]
25 if
26 [for (ifStmt : If | case.subStatements ->select(p | p.oclIsTypeOf(If)))]
27  [generateIfConstruct(ifStmt)/]
28 [/for]
29 :: else ->
30    for (i : 0 .. MAX_PROC-1) {
31     HasToken['['/]i[']'/]=MyChildren['['/]i[']'/];
32    }
33 fi;
34 }
35 [/template]
36
37
38 [template public generateIfConstruct(ifStmt : If)]
39 :: ([ifStmt.condition.toString()/]) ->
40   [for(description : String | ifStmt.description)]
41   [description/];
42   [/for]
43 [/template]
```

Since a main loop can contain both simple and composite statements, such rule is composed by two parts: the writing of the simple statements (Listing 6.21, lines 5-7) and a new iteration over all possible sub-statements (lines 9-12). Such iteration call another template rule, named *generateDoCases*. The *generateDoCases* rule is in charge of write, in the Spin syntax, the condition of each case and its simple statements. Moreover, if the considered *Case* corresponds to a vertex with more than one exiting transition, a new iteration is performed to generate the *If statements*. Moreover to generate the code of *If statement* a template rule, called *generateIfConstruct* (Listing 6.21, lines 38-43) is used. As stated before, M2T transformation add some information not present in the PSM. As an example, the *generateDoCases* add the code necessary to exchange the token from a process to its children processes when it terminates (Listing 6.23, lines 32-39). Thus, the M2T transformation allows to describe the general structure of a process in which a DSTM machine is mapped. Such structure is reported in Listing 6.23.

LISTING 6.22: general structure of the Promela code of a DSTM machine

```
proctype process(pid parent; parameters list) {
byte i;
pid PidTemp;
bit MyChildren[MAX_PROC];
mtype state=initial;
do

 //case1
 :: (state && HasToken[_pid]==1) ->
  atomic{
   simple statements;
  //in case of composite statements
  if
  :: (condition) ->
   Simple statements;
  if
  :: (condition) ->
   Simple statements;
  if
  :: (condition) ->
   Simple statements;
  :: else ->
  //token is passed to children processes
  for (i : 0 .. MAX_PROC-1) {
   HasToken[i]=MyChildren[i];
  }
  fi;
  }
  ...
  //case N
  ...
od unless{
 //statements to send messages of termination of chTerm channels
 (chTerm?[1] ||  dyingPid==parent);
 if
 ::chTerm?[1] -> chTerm?1;
 fi;
 dyingPid = _pid;
     }
}
```

Engine code generation A different structure is owned by the *Engine* process. In fact, the *Engine* is in charge of initialize the external channels and produce non-deterministically messages over them. In order to realize such structure, the M2T transformation in Listing 6.23 is used.

LISTING 6.23: M2T rule to generate the engine

```
[template public generateEngine(proc: Process)]
[if proc.isActive]active[/if] proctype [proc.name/]() {
mtype temp;
//local channel and variables
[for (c : Channel | proc.procVariables->select(p | p.oclIsTypeOf(Channel))
     )]
[ generateChannel(c) /]
```

```
7  [/for]
8
9  [for (v : Primitive | proc.procVariables->select(p | p.oclIsTypeOf(
       Primitive)) )]
10 [if not (v.name='PidMain')][ generatePrimitiveVariable(v) /]
11 [else]
12 [v.type/] [v.name/];
13 [v.name/]=[v.initialValue/]
14 [/if]
15 [/for]
16
17 //generation of first messages
18 [for (c : Channel | proc.model.variables->select(p | p.oclIsTypeOf(Channel
       )) )]
19 [if (c.external=true)]
20 [c.name/]!0,[for (val : DefaultType | c.types )separator (',')]0[/for]
21 [/if]
22 [/for]
23
24
25 //generation block
26 generation:
27  atomic{
28
29    if
30    [comment]:: chterm_[for(pro : Process | proc.model.processes->select(p |
          p.isMain))][pro.name/][/for]?['['/]1[']'/]) -> goto abort;[/comment
         ]
31
32     [for (ifStmt : If | proc.processStatements->select(p | p.oclIsTypeOf(If
          )))]
33      [generateEngineStatements(ifStmt, proc)/]
34     [/for]
35    :: (NoFirings==1) -> goto abort;
36    :: else -> skip;
37    fi;
38    NoFirings=1;
39
40 //GIVE TOKEN TO THE MAIN PROCESS
41    HasToken['['/]PidMain[']'/]=1;
42    printf("<ENGINE: end execution>\n");
43
44 }
45 //generate the main loop
46 [for (do : Do | proc.processStatements->select(p | p.oclIsTypeOf(Do)))]
47  [generateDoOfEngine(do, proc)/]
48 [/for]
49 abort:
50  dyingPid=_pid;
51 }
52 [/template]
```

The initialization of the external channels, is made thanks to the presence of the attribute *external* in the Promela metamodel. In fact, iterating over external channels, all the statements to initialize them are generated. At the same way, in order to perform the non-deterministic messages generation statements,

an iteration is performed over the process statements, generated by the corresponding M2M transformation (template rule *generateEngineStatements* at line 33). Lets note that, the *generateEngineStatements* rule, produces the code of non-deterministic messages generation on the basis of the external channels of the DSTM model.

6.7 Spin Manager: generation of test cases

Once the Promela model has been generated, the test cases are obtained by exploiting model checking. According to this three components are necessary: (1) the TSP2NeverClaim, the (2) Spin Manager and the (3) Post Processor.

The Spin Manager is the component in charge of execute the Spin Model checker in order to automatically generate test case from the SUT specification. The Spin Manager takes as input the model expressed in Promela, generated by the M2T transformation and the never claims, generated by the *TSP2NeverClaim* component. This last, with the respect to the previous step, is realized by means of Java. In fact, according to the Section 4.3, the only TSP taken into account by the TCG framework, is the cover, and in particular the cover of requirements tagged on transitions. Thus, the *TSP2NeverClaim* components, for each transitions of the DSTM model, generates a never claim, according to the structure in Listing 6.24.

LISTING 6.24: Structure of a never claim for transition covering

```
1 never n_i {
2 never_step:
3  if
4   :: (LastTransition==transition_name) -> goto end_never;
5   :: else -> goto never_step;
6  fi;
7 end_never:
8  skip;
9 }
```

Phases and artefacts involved in the execution of the Spin manager, are represented in Figure 6.9.

First the Spin Manager, merges the DSTM model and the generated nerver claims in a unique merged Promela code file. Then, invokes the Spin Model Checker facilities, in order to produce the test cases. According to this, the first phase is the generation of the verifier, or in other words of a specific model checker for the provided input model. The verifier, can be generated by invoking the following command of the Spin Model Checker:

spin -a model_name.pml

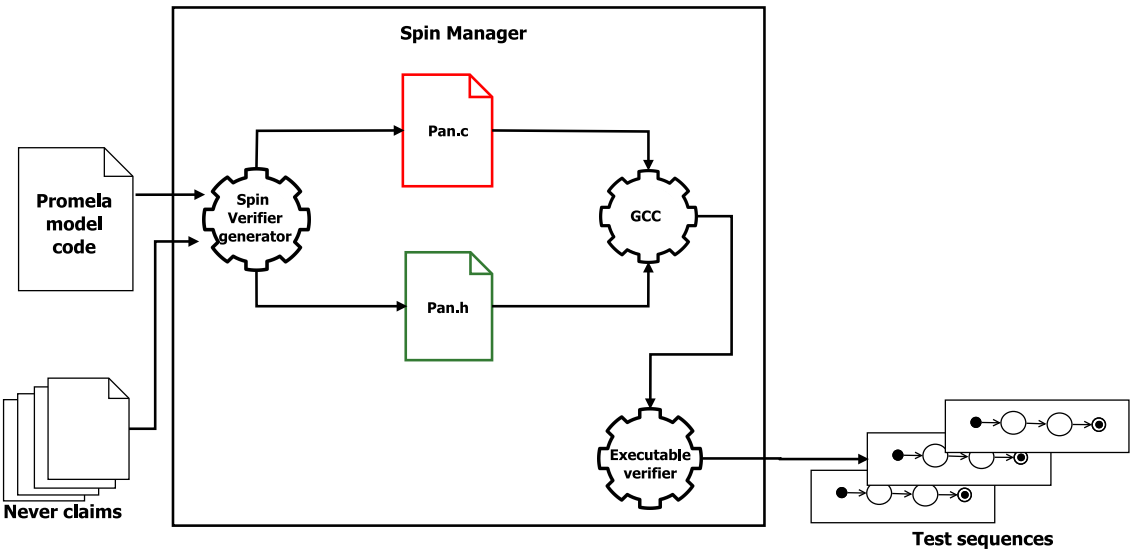


FIGURE 6.9: Spin Manager steps and artefacts

Such command performs a syntactical checking of the Promela code, and can produce a list of errors, if some flaws are found. Otherwise the header and implementation files describing the verifier in C language is generated. The header file, contains declarations of global variables, of channels and of all the proctypes. The implementation file, instead, contains the algorithms for the computation of the labelled transition system and of the state-space of the model. Such files are given as inputs to a gcc compiler [46] in order to produce an executable verifier. The command used for the compiling phase is the following:

gcc -DMEMCNT=32 -DVECTORSZ=4112 -DSAFETY -DBITSTATE -o pan pan.c

Where gcc is the command to invoke the C/C++ compiler and the pan.c is the implementation file in the C language, containing the verifier generated on the basis of the provided model. The remaining options adopted in the command, are compiling directives inserted in order to generate a optimized and more efficient executable verifier [50]. More in detail, such options are:

- *-DMEMCNT=32* indicates a limitation on the usage of memory. More specifically it specifies that the maximum memory that the verification can use is of 2^{32}. This option should be set in order to avoid the exceeding of memory. In fact, not specify this option, can lead the verifier to saturate the physical memory and using the virtual memory until a crash.

- *DVECTORSZ=4112*, represents an option to drive the compiler behaviour if some errors are found during the compiling phase. In particular the DVECTORSZ, represents the maximum size for the vector state (i.e. the vector

used for the internal representation of model states). The default value is set to 1024 byte, even if, when larger models exceed this size, the compiler can ask for a recompiling with a different maximum size.

- *DSAFETY* belongs to the options for the performances enhancement of the verifier. In particular the DSAFETY, optimize the code, when no cycle detection is needed.

- *DBITSTATE* changes the default policy for the depth search algorithm adopted by the generated verifier. In particular the default algorithm provides the construction of a state space, in order to avoid to re-visit a state already visited. Normally, in order to register states in such space, an hash table is adopted. The bitstate algorithm acts on the memorization of the states. In fact, instead of memorize the whole state representation in the arrays of the hash table, each state is represented as number. Thus, when the hash function computes the number, have only to know if the number is present or not in the hash table, without register the entire state representation.

- *-o*, is a gcc option that specify the output file in which memorize the compiling process results.

The verifier is generated on the basis of the Promela model and is executed to verify each never claims. Such process produces a set of traces, each of them representing a test case, e.g. a sequence of steps to verify that the related transition is covered. The obtained test sequence contains the description of the steps, made by means of TESQEL language (Section 4.2). However, a post processing must be performed in order to refine such representation and obtain the final set of test case, described with TESQUEL, that will be the starting point for other testing phases, as the implementation in executable scripts and their execution on the real system.

Chapter 7

Automatic generation of functional test cases in the railway domain

This chapter discusses the application of the automatic test cases generation approach, described in the previous chapters. DSTM is used to provide the specification of a railway control systems, related to the CRYSTAL use case, in which this work is partially involved. More specifically, the use case is related to the Radio Block Centre (RBC) which is a key component of the ERTMS/ETCS standard. The application of the approach considers a complete scenario which refers to the procedure performed by the RBC to manage the communication with the trains on the tracks under its supervision. This procedure has been named *Communication Procedure*. Next sections provides an overview of the European Rail Traffic Management System/European Train Control System (ERTMS/ETCS) standard and a description of the RBC system. The automatic system-level test case generation of *Communication Procedure* is discussed.

7.1 ERTMS/ETCS standard

The ERTMS is an initiative of the European Union to enhance cross-border interoperability and the procurement of signalling equipment by creating a single Europe-wide standard for train control and command systems. In the 1989 the European committee for transportations conceives a strategy for the development of a common train control system standard, to be applied to the European railway infrastructure. According to this, the first Interoperability Directive, was issued by the European Commission in the 1993, reporting the decision to define a set of Technical Specifications for Interoperability (TSI). Two years after, in the 1995, a plan, describing the first version of the ERTMS system, was released for the development of such systems. Only in the 1996, the EU decided that ERTMS would become the standard for all railway high-speed lines. The

two EU Council Directive 96/48/EC [33] and 2001/16/EC [32] defined the interoperability of the trans-European high-speed rail systems and the conventional rail systems specifications. As a response to the same directives the first specification of the European Train Control System was defined as a part of the ERTMS initiative. The ETCS is one of the main systems composing the ERTMS. It consists in a signalling, control and train protection system designed to replace the many incompatible safety systems adopted by European railways, especially on high-speed lines. The ETCS allows a train equipped with it to travel without signal system boundaries within the ERTMS/ETCS fitted infrastructure network, regardless of the country the train is travelling in, the legal nature of the infrastructure manager or the supplier providing the ERTMS/ETCS system. In 2004, the ERA (European Railway Agency) was created and designated as the ERTMS system authority, and thus is in charge of managing system specifications. The ERA, with the regulation (EC) N° 881/2004 officially starts the development of the ERTMS/ETCS system that is actually used within the European high-speed railway network.

7.1.1 ERTMS/ETCS Safety Integrity Level

The ERTMS/ETCS provides the safe movement of trains and the optimal traffic regulation of high-speed trains. Thus, the whole system shall be classified as safety critical although complex system: it shall guarantee the safety of the trains movement, preventing collisions in any case, also in situations of breakdowns and human errors. As discussed in Section 1.3, several international standards are applicable in this context, since the reliability, the safety evaluation and the management are mandatory for these kind of systems. The basic standard adopted for such kind of railway systems is the IEC/EN 61508 (Section 1.3.1) which defines the SIL, as an indicator of the integrity level of safety functions provided by the considered system. This definition is applicable to all kind of industries for both hardware and software components. Lets note that the SIL is related to a single safety function and not to the entire system or individual components: within a given system a lot of safety features will exist, each of them related to a particular hazard to which an appropriate SIL will be associated. The whole set of components of each security system must agree with the specific SILs defined by the related standards.

With the respect to the railway application domain, other reference standards must be taken into account. In particular three of them are nowadays adopted: the EN 50126, EN 50128 and EN 50129 (Section 1.3.4) produced by CENELEC, the

European Committee for Electrotechnical Standardization, which is responsible for standardization in the electrotechnical engineering field.

The three aforementioned CENELEC standards represent the backbone of the RAMS (Reliability, Availability, Maintainability, and Safety) demonstration process of a railway system: possible failures and hazards are identified during the overall lifecycle and are properly corrected or mitigated considering their occurrence rate and the effort to spend; finally the risk is evaluated. In detail EN 50126 describes the processes and methods that are used to specify the most essential and important aspects for operability and safety in the rail domain; the EN50128 and the EN 50129 give a set of requirements which have to be satisfied during the safety-critical software (the former) / hardware (the latter) development, deployment and maintenance phases. The lifecycle suggested for these systems is the 'V' lifecycle (Section 1.3.5) where the design is implemented during the descendent activities which correspond to the verification and validation activities performed during the ascending branch.

According to the CENELEC standards, the ERTMS/ETCS implements functions classified as SIL 4, which is the highest dependable level. This is a key factor in system verification and validation: in fact some techniques listed in standards are highly recommended for these systems. Among all, the adoption of a structured methodology, based on formal methods and a modelling approach are needed during the requirements specification as well as during design and implementation.

A modelling methodology is required at system level to have a clear and complete understand of the system behaviour. Furthermore the evaluation of some coverage metrics is necessary: specific test cases which stress a predefined portion of the system shall be defined in order to individuate which portions are active and which ones are unreachable.

7.1.2 The Radio Block Centre

The ERTMS is based on two main components: the ETCS and the Global System for Mobile Communication - Railway (GSM-R). This second, is the trackside communication Network used for exchange of information between the onboard sub-system and trackside equipment. The ERTMS/ETC system has been organized in four different functional levels, depending on system architecture. Level 0, Level 1 and Level 2 are already implemented and used while Level 3

is currently under development. The definition of the ERTMS/ETCS levels, depends on how the considered route is equipped and the way with which the information are exchanged between the system and the trains. The four functional levels are:

- **Level 0**: the Level 0 regards ERTMS/ETCS-compliant locomotives or rolling stock interact with line-side equipment that is non-ERTMS/ETCS compliant. Technically this is not part of the ETCS, since the train driver shall observe the physical signals encountered along the route, knowing the specific meaning of those signals on the railway;

- **Level 1**: this level consists of a cab signalling system that can be superimposed to the existing conventional signalling system leaving the fixed signal system (national signalling and track-release system) in place. The on-board equipment monitors and calculates the maximum speed and the braking curve relying on the data received from the beacons at fixed points;

- **Level 2**: the Level 2 is a train protection system based on continuous communication of variable data between the RBC and the trains via a radio system (some additional information are received on board via fixed beacons);

- **Level 3**: Level 3 implements a full radio-based train control and spacing hence fixed block track equipment is no longer required. This technology allows for detecting the current position of each train always in time, hence it is possible to send continuously line-clear authorization to each train.

Level 2 and Level 3 are actually the two more cutting-edge solutions than Level 1. In particular Level 2 is the most widespread choice, according to the current deployment statistics [1]. A reference architecture for Level 2 is represented in Figure 7.1. It consists of three main subsystems: (1) the on-board system which is the core of the control activities located on the train; (2) the line-side subsystem which is responsible for providing geographical position information to the on-board subsystem; (3) the trackside subsystem which is in charge of monitoring the movement of the trains. Within the track side subsystem the most important component is the Radio Block Centre (RBC). RBC is a computing system whose aim is to guarantee a safe inter-train distance on the track area under its supervision. As shown in Figure 7.1 RBC interacts with the on-board system by managing a Communication Session using the EURORADIO protocol and the

[1] http://www.ertms.net

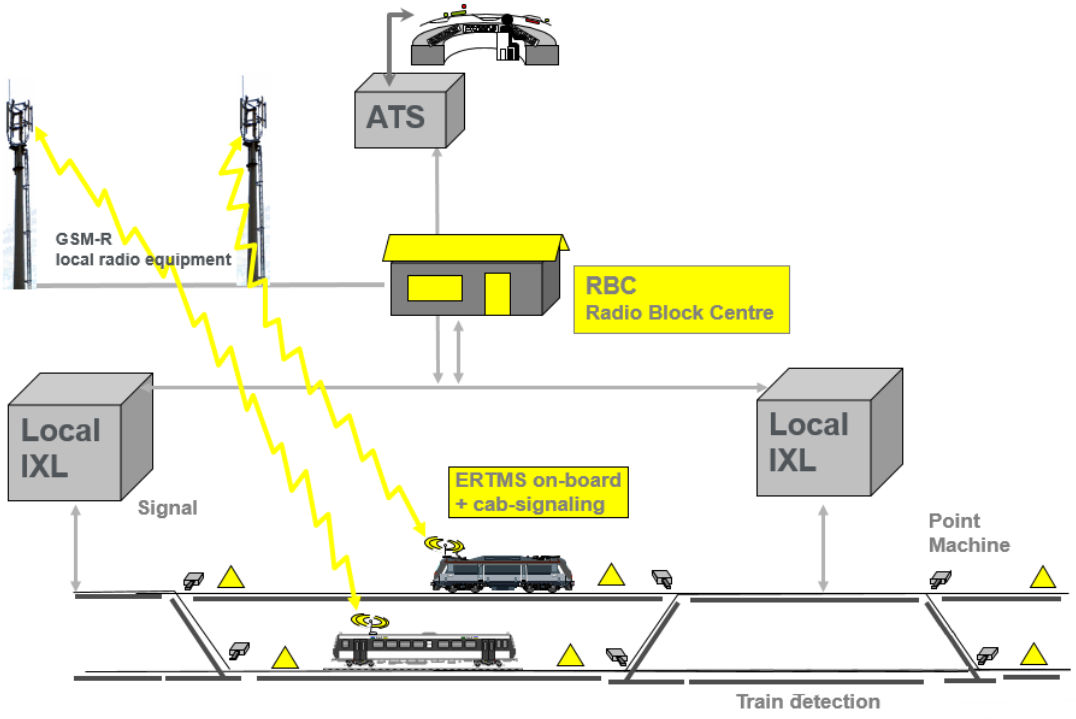


FIGURE 7.1: ERTMS/ETCS - Level 2

GSM-R network. A single RBC is in charge of concurrently and continuously controlling a fixed maximum number of connections with trains, depending on physical characteristics of the GSM-R network. The main objective of the train control system is to timely transmit to each train its up-to-date Movement Authority (MA) and the related speed profile. The MA contains information about the distance the train may safely cover, depending on the status of the forward track. RBC is also in charge of managing emergency situations if the communication with one or more trains is compromised. Specifically, when a train is approaching the area supervised by a RBC, it sends a connection request. If the request may be accepted, RBC tries to establish and manage the connection until the train remains under its control. This is performed by dynamically instantiating a thread of execution for each connection. Each of them manages the communication with a specific train and performs a number of data checks and actions based on the content of the messages that a train receives from the RBC. Since a communication can be lost at any time, each thread handles several errors and termination conditions. In particular, if a communication is definitely lost, all pending processes which refer to the MA delivery are terminated and an emergency procedure has started.

7.2 The Communication procedure

The communication procedure of the RBC is articulated in three ordered steps, starting from the entering of a train on the track area controlled by the RBC, to its leaving. The aforementioned steps are below described.

Step1: Communication Establishment When a train is going to establish a safe connection with an RBC, it sends a request message. RBC may accept connection requests only from a limited number of trains. This number depends on physical features of the communication radio channel; over this value, RBC refuses new connection requests, by sending to the train a proper answer message (refused). Otherwise it sends to the train the acceptance notification (accepted). Hence, it is necessary to model a process that accepts or refuses these requests by checking the number of already accepted connections, and instantiates all the processes in charge of managing the communication with the trains. After the acceptance of a request, this process remains active and waits for other communication requests.

Step2: Session establishment Once a connection request is accepted, the communication session between the related train and RBC can be established. If the procedure succeeds RBC authorizes the train to start the mission (Start of Mission, SoM) or to perform the set of actions required to enter the high-speed area from a not high-speed area. Ultimately, RBC sends the System Version message to the train. The train answers with an Acknowledgement (Ack) and a Session Established message. A specific value in the Session Established message (the area field) is used by RBC to distinguish between a train that needs to start its mission (L0 area) and a train that is coming from a non high-speed area (L1 area). In fact, two different procedures shall be performed: the Start of Mission and the Entry actions respectively. If a message different from the expected one is received during this protocol, the session establishment procedure is aborted and the communication with the train is closed.

Step 3: Management of train movement RBC periodically sends the Movement Authority (MA) to each train and checks for the reception of commands from the Centralized Traffic Control (CTC) where a human operator may raise alarms which requires that the train has to brake: in this case an Unconditional Emergency Stop (UES) message is sent to the train. On the other hand, when the train successfully ends its trip, RBC performs the End of Mission (EoM) procedure.

The entire Communication procedure of the RBC is composed by ten machines and by the related data declarations, described in the next section. Since some of these machine are quite straightforward, only the four most complex are here discussed. For each of the discussed machine both the DSTM specification and the flat model are provided. The Promela formal model, automatically obtained from the flat, is provided only for the *M_CommunicationEstablishment* since the other machines are obtained with the same process, and present the same structure. The complete DSTM specification related to both data flow and control flow is reported in Appendix A. In the present chapter also the generated *Engine* process is discussed and some test cases, obtained by the set of functional requirements of the RBC Communication Procedure are showed.

7.2.1 Data declarations for the Communication Procedure

In order to model the specification of the Communication Procedure of the RBC, several data declarations, modelling the data flow of the model, are necessary. Such data declarations are below discussed.

A set of enumeration have been defined within data declarations:

- *answer* defining the possible values of RBC answer messages to a connection request (i.e., *accepted* and *refused*);
- *version* defining the possible values of the version field inside a System Version message (e.g., *V0*, *V1*, *V2*);
- *registration* containing the possible values of the registration field inside a Train Registration message (e.g., *registered*);
- *area* allowing to define the possible values of the area field inside a Session Established message (e.g., *L0*, *L1*);
- *msgId* defining the possible values of the message identifiers specified in all the messages (e.g., *SessionEstablished*, *Ack*).

The set of structured messages are modelled by several compound types:

- *M_Request: {Chn, Chn}* models the connection request message;
- *M_Answer: {answer}* models the answer message;
- *M_SessionEstablished: {msgId, area, Int, Int}* models the Session Established message;

- *M_Ack: {msgId}* models the Ack message;
- *M_TrainRegistration: {msgId, registration}* models the Train Registration message;
- *M_SystemVersion: {msgId, version}* models the System Version message;
- *M_MovementAuthority: {msgId, Int}* models the Movement Authority message.

Compound types are grouped into two multi-types:

- *MT_from: {M_SessionEstablished, M_Ack, M_TrainRegistration}* models a set of message types received from a train;
- *MT_to: {M_SystemVersion, M_MovementAuthority}* models a set of message types sent to a train.

Three global variables are used in the model: *V_cont* typed as *Int*, *V_chFrom* typed as *Chn[MT_from]* and *V_chTo* types as *Chn[MT_to]*, their role is described in the following.

Different channels exist, one of which (*C_request*) is used by RBC to receive the connection requests from the trains. Each train conveys on *C_request* the names of the two channels used to communicate with RBC; a further channel is used by RBC to answer the train (*C_answer*) and communicate if its request is accepted or refused. In addition, a pair of multi-type channels for each train is defined: *C_fromTrain_i* (typed as *MT_from*) and *C_toTrain_i* (typed as *MT_to*); they are used to receive and send messages to the i-th train, respectively. Data declarations are partially generated by transformation rules related to the data flow (e.g as for the simple types) and partially generated by the rules addressing the control flow (e.g. as for the multi types).

7.2.2 M_CommunicationEstablishment

The most important machine specifying the behaviour of the Communication Procedure is the *M_CommunicationEstablishment* that manages the establishment of the safe connection between the RBC and the train board, according to the **Step1** of the procedure. As described at the beginning of the this section, a train which is going to establish a safe connection with an RBC, sends a proper message to it. RBC may accept or not, depending on the number of already activated trains. A specific variable, has to trace the number of activated train.

DSTM model The DSTM specification of the *M_CommunicationEstablishment* machine, can be modelled by referring to a specific model pattern, named *Finite Thread Pool*. The *Finite Thread Pool* pattern consists of an idle state modelling the system that waits for a request. On the arrival of a request, the considered machine, enters a box and starts a new instance of the worker machine. The worker handles the received request, by assigning proper values to the parameters of the new machine instance. Figure 7.2 shows the graphical representation of the considered machine.

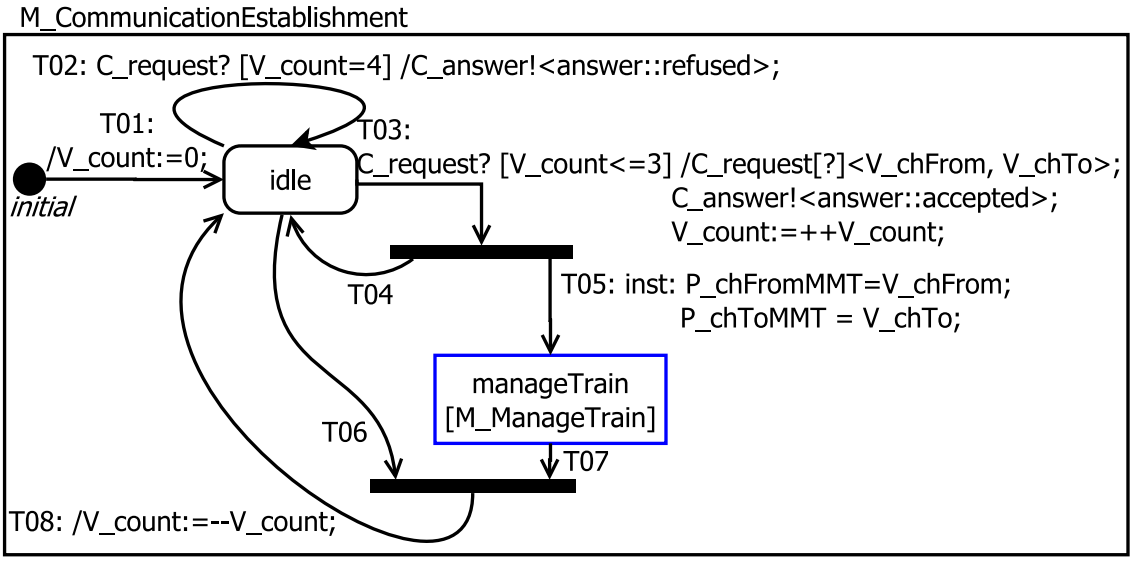


FIGURE 7.2: M_CommunicationEstablishment

According to *Finite Thread Pool* pattern, the *M_CommunicationEstablishment* machine is composed by an idle state, modelling the wait for a request and the check of the current number of activated trains, memorized in the *V_cont* variable. In the specific case, the maximum number of trains which can be accepted is 4. When a request arrives, if activated trains are less then 4, the *M_CommunicationEstablishment* instantiates a *M_ManageTrain* machine by entering the *manageTrain* box. Every time a request is accepted, the *V_count* variable is incremented. The *idle* state is the source of the *T03* transition that enters in the fork and is triggered by the presence of a message on the channel *C_request*, and guarded by the condition *V_cont<=3*. The actions annotated on this transition are the send of the acceptance message over the channel *C_answer*, the increase of the counter *V_cont* and the memorization of the names of the multi-type channels, on which the communication with the train will continue (i.e. over the variables *V_chFrom* and *V_chTo*). The *M_ManageTrain* termination is managed by the join pseudonode, through the transition *T06*. The transition T08, exiting the join, instead, decrements the counter *V_cont*. Finally, the transition T02, exiting the

node idle, is instead activated on the reception of a connection request, when the counter has reached its maximum value. The action annotated on *T02* transition regards the sending of a proper refusal message over the channel *C_answer*.

Flatten model The DSTM specification has to be flatten in order to enable the test case generation process. The machine resulting from the flattening process is depicted in Figure 7.3.

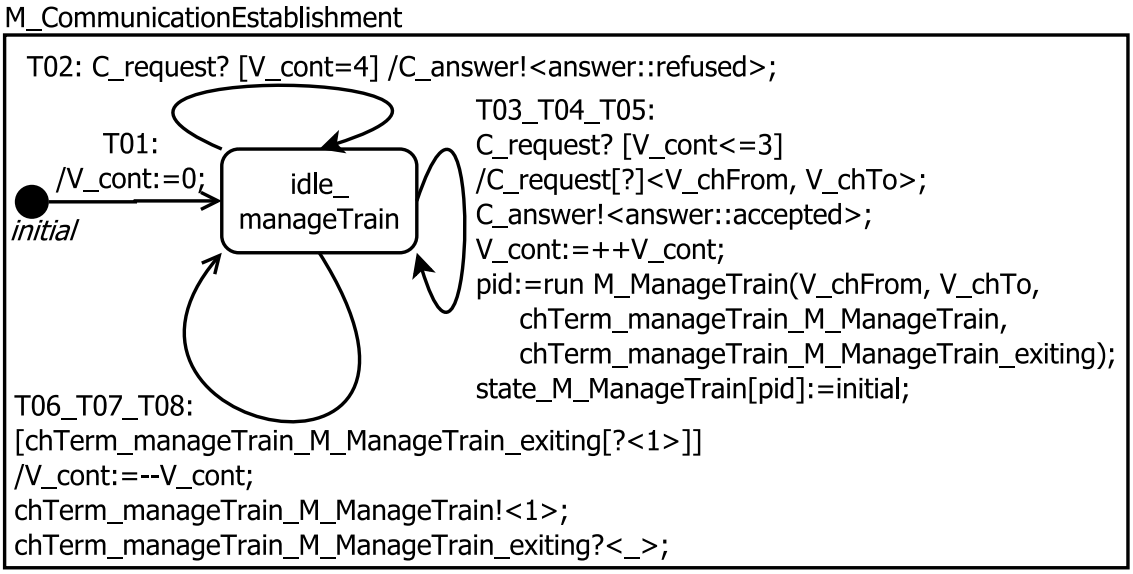


FIGURE 7.3: Flattened M_CommunicationEstablishment machine

The flat machine presents two of the cases treated in Section 5.4: the asynchronous fork and the simple join. The box *manageTrain* of the DSTM model is removed and the entering and exiting fork transitions *T03*, *T04* and *T05* are replaced by the single transition *T03_T04_T05*. Such transition has the same source of the *T03* transition and the same destination of the asynchronous fork transition *T04*. Moreover the decoration of such transitions inherits trigger and condition from the *T03* transition and the actions of all original transitions with the additional actions related to the instantiation of the *M_ManageTrain* machine. The *idle* node is renamed as *idle_manageTrain* in order to keep track of the removed box. Since the idle node is both the source and the target node of this transition, the new transition forms a self-loop on that node. With the respect to the join, lets note that an exiting join transition can specify neither a trigger nor a condition, but it may have an associated action. Since in this case the join is non-preemptive, it is replaced by a single internal transition with no trigger and a guard checking for the termination of all the machines associated with the joined boxes. The action of the transition must include the action of exiting join transition as well as the actions necessary to deal with the termination

of the joined boxes. According to this the transition *T06_T07_T08* replaces the non-preemptive join transitions *T06*, *T07* and the exiting join transition *T08*. The actions on this new transition include those specified on the exiting join transition T08. The guard is $chTerm\_manageTrain\_M\_ManageTrain_exiting[? < 1 >]$, checking for the reception of the exiting message from machine *M_ManageTrain*. The action associated with transition *T06_T07_T08* is the action of transition T08 (V_Cont:= ++V_Cont) followed by the consumption of the exiting message from *M_ManageTrain* and the dispatch of the termination message to *M_ManageTrain* along channel *chTerm_manageTrain_M_ManageTrain*.

Promela model In order to show a complete generation process the Promela model related to the *M_CommunicationEstablishment* machine, is provided in Listing 7.1. Such Promela model is structured according to the schema, discussed in Section 5.2.

LISTING 7.1: The DSTM model of the Communication Establishment Machine

```
1  proctype MCommunicationEstablishment(pid parent; mtype initial;chan
       chTerm) {
2  byte i;
3  pid PidTemp;
4  bit MyChildren[MAX_PROC];
5
6  chan chTerm_manageTrain_MManageTrain = [MAX_PROC] of {bit};
7  chan chTerm_manageTrain_MManageTrain_exiting = [MAX_PROC] of {bit};
8  mtype state=initial;
9  do
10   :: (state == MCommunicationEstablishment_initial && HasToken[_pid]==1)
       ->
11  atomic{
12    printf("<current node [%d] = MCommunicationEstablishment_initial>\n",
        _pid);
13    HasToken[_pid]=0;
14  if
15   :: (1) ->
16     Vcont=0;
17     printf("<firing transition[%d] = MCommunicationEstablishment_T01>\n",
         _pid);
18     state = MCommunicationEstablishment_idle_manageTrain;
19     printf("<next node[%d] = MCommunicationEstablishment_idle_manageTrain
         >\n",_pid);
20     LastTransition=MCommunicationEstablishment_T01;
21     NoFirings=0;
22  :: else ->
23     for (i : 0 .. MAX_PROC-1) {
24      HasToken[i]=MyChildren[i];
25     }
26  fi;
27  }
28
```

```
29    :: (state == MCommunicationEstablishment_idle_manageTrain && HasToken[
          _pid]==1) ->
30   atomic{
31     printf("<current node [%d] =
           MCommunicationEstablishment_idle_manageTrain>\n",_pid);
32     HasToken[_pid]=0;
33
34   if
35    :: (Crequest?[_,_,_,_,_]&&Vcont==4) ->
36
37      Canswer!1,refused;
38      Crequest?_,_,_,_,_;
39      printf("<firing transition[%d] = MCommunicationEstablishment_T02>\n",
            _pid);
40      state = MCommunicationEstablishment_idle_manageTrain;
41      printf("<next node[%d] = MCommunicationEstablishment_idle_manageTrain
            >\n",_pid);
42      LastTransition=MCommunicationEstablishment_T02;
43      NoFirings=0;
44    :: (Crequest?[_,_,_,_,_]&&Vcont<=3) ->
45      Crequest?VchFrom_MSessionEstablished,VchFrom_MAck,
            VchFrom_MTrainRegistration,VchTo_MSystemVersion,
            VchTo_MMovementAuthority;
46      Canswer!1,accepted;
47      Vcont++;
48      PidTemp=run MManageTrain(_pid,MManageTrain_initial,
            VchFrom_MSessionEstablished,VchFrom_MAck,
            VchFrom_MTrainRegistration,VchTo_MSystemVersion,
            VchTo_MMovementAuthority,chTerm_manageTrain_MManageTrain,
            chTerm_manageTrain_MManageTrain_exiting);
49      MyChildren[PidTemp]=1;
50      HasToken[PidTemp]=1;
51      printf("<firing transition[%d] =
            MCommunicationEstablishment_T03_T04_T05>\n",_pid);
52      state = MCommunicationEstablishment_idle_manageTrain;
53      printf("<next node[%d] = MCommunicationEstablishment_idle_manageTrain
            >\n",_pid);
54      LastTransition=MCommunicationEstablishment_T03_T04_T05;
55      NoFirings=0;
56    :: ((chTerm_manageTrain_MManageTrain_exiting?[1])) ->
57      chTerm_manageTrain_MManageTrain!1;
58      chTerm_manageTrain_MManageTrain_exiting?_;
59      Vcont--;
60      printf("<firing transition[%d] =
            MCommunicationEstablishment_T06_T07_T08>\n",_pid);
61      state = MCommunicationEstablishment_idle_manageTrain;
62      printf("<next node[%d] = MCommunicationEstablishment_idle_manageTrain
            >\n",_pid);
63      LastTransition=MCommunicationEstablishment_T06_T07_T08;
64      NoFirings=0;
65   :: else ->
66      for (i : 0 .. MAX_PROC-1) {
67       HasToken[i]=MyChildren[i];
68      }
69   fi;
70   }
71  od unless{
72    (chTerm?[1] ||  dyingPid==parent);
```

```
73    if
74    ::chTerm?[1] -> chTerm?1;
75    fi;
76    dyingPid = _pid;
77    printf("<Machine MCommunicationEstablishment[%d] terminated>\n",_pid);
78  }
79  }
```

The process reported in the above listing is composed by two enumerations types which introduce the symbolic names for nodes an transitions. A global variable *HasToken*, typed as a bit array, is used to simulate the step semantics as described is Section 5.5. Specifically, the i-th position of this array is set to 1 if the process, whose _pid is equal to i, currently holds the token, meaning that it can evolve. Lets note that this array is unique for the entire Promela model. The body of the process consists in an iteration (a do-repetition construct) repeated until the termination message is received from the calling process over the channel chTerm (exchanged as parameter). The termination message is then propagated to all the children processes of the current process, if any. Each node of the machine is translated into a guarded statement. The guard is satisfied if the specified node is the current state of the process and the process holds the token. The atomic statement associated to the guard contains a sequence of statements executed indivisibly. The first statement in the sequence consumes the token (e.g., HasToken[_pid] = 0). Then a conditional statement (a selection construct - if) contains one guarded statement for each transition exiting from that node, where the guard correspond to the enabling condition of the transitions (presence of the trigger and truth of the condition) and the associated statements translate the actions (if any). Each action is translated into basic Promela statements and operators, and executed when the associated guarded statement is selected for the execution. If more than one guarded statement is executable, one of them is non-deterministically selected. The else branch in the conditional statement guarantees from one hand that the process is not blocked if no transition can fire, from the other hand it executes a block of statements which propagates the token to the children of the current process.

7.2.3 M_ManageTrain

The machine activated by the *M_CommunicationEstablishment* is the *M_ManageTrain* that is in charge of modelling the acceptance of a train on the track area under the supervision of the RBC.

The DSTM model The *M_ManageTrain* models the management of the communication procedure with a specific train. The parameters taken into account by this machine are the names of the multi-type channels on which the train and the RBC will communicate. This machine enters the node *idle* and then instantiates the machine *M_SessionEstablishment*, which models the session establishment protocol, by entering the proper box. This last machine can terminate its execution through different exiting nodes: exiting through *entry* means that the machine *M_Entry* shall be subsequently instantiated, while *som* represents the necessity to instantiate the machine *M_StartOfMission* (the models of these last are provided in Appendix A). If the session establishment protocol is aborted (exiting through *aborted*), this machine also terminates its execution. After both the machines *M_entry* and the *M_StartOfMission*, the machine *M_MovAuth* is instantiated in order to give periodically MA to the train. Note that, if the entry actions have been performed, the machine *M_MovAuth* shall be instantiated through its *afterEntry* entering node.

Figure 7.4 shows the graphical representation of the *M_ManageTrain*.

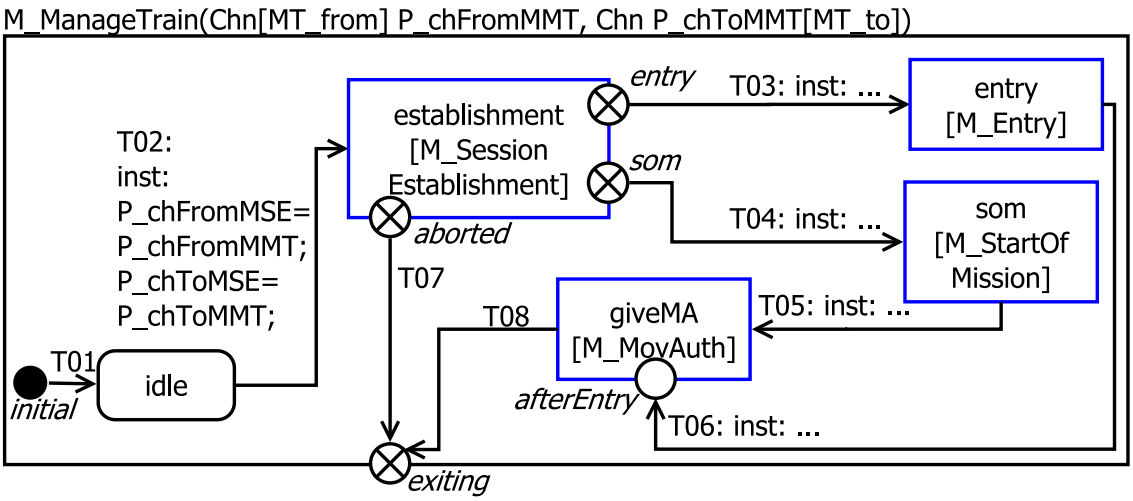


FIGURE 7.4: M_ManageTrain machine

Flatten model The *M_ManageTrain* machine presents several simple box cases (Section 5.4.1). In fact, the boxes of the *M_ManageTrain* DSTM specification are removed and replaced by nodes having the same names. Transitions entering and exiting from the considered boxes are substituted with corresponding transitions entering and exiting from the new generated nodes. Such transitions inherit triggers, conditions and actions of the original ones, adding the actions related to the call and termination operations of the machines associated to the removed boxes. According to the Figure 7.5 the transition *T02* activates the

M_SessionEstablishment machine, hence its decoration is modified by adding the action
pid = run(M_SessionEstablishment(parameter list).

The parameters list is obtained from the list of parameters of the original box. Additional parameters are however needed to correctly model the termination of the called machines: the *ChTerm* channels, related to each specified exiting node of the *M_SessionEstablishment* machine, and the additional *ChTerm* channel, used by the caller to signal its termination.

The transitions exiting from the boxes, are substituted with other transitions exiting from the substituting nodes. There are two possible cases: (1) if the original transition is a *return by exiting*, then the transition must be taken only if the called machine has terminated. Therefore, the decoration of the replacing transition, must also check that an exiting message has been sent by the callee over the channel associated with the considered exiting node. Hence, the guard $chTerm\_BoxName\_MachineName\_ExitingNodeName? < 1 >$ is added to the guard of the original replaced transition. Indeed, if the transition is a *return by default* then it can be taken as soon as the callee has reached any exiting node. In this case, the guard of the transition is augmented with the disjunction of all the guards of the form $chTerm\_BoxName\_MachineName\_ExitingNodeName? < 1 >$, one for each exiting node. In either case, when the transition is taken, the callee must terminate. In the Figure 7.5 such conditions are added to the *T03* transition which checks for the reception of the notification message from the *M_SessionEstablishment* over the channel associated to the *entry* exiting node. Thus, two actions are added on the *T03*: the first models the transmission of the termination message from the *M_ManageTrain* to the *M_SessionEstablishment*, the latter is needed to consume the message on the same channel. The same flattening process is adopted for the *T06* transition.

7.2.4 The M_MovAuth machine

Machine *M_MovAuth* models the activities of Step 3 of the Communication Procedure. Below its DSTM specification and the related flat model are discussed.

DSTM model The *M_MovAuth* is modelled according to the *Alarm Management* pattern in which a machine worker is executed in parallel to a machine alarm that checks for the presence of alerts. In case any alarm is raised, the execution of the worker is immediately stopped. According to this, as depicted in Figure 7.6, the fork instantiates two concurrent workers by entering in the *centralManag* and *periodicMA* boxes. If the former ends its execution, the join on the left is

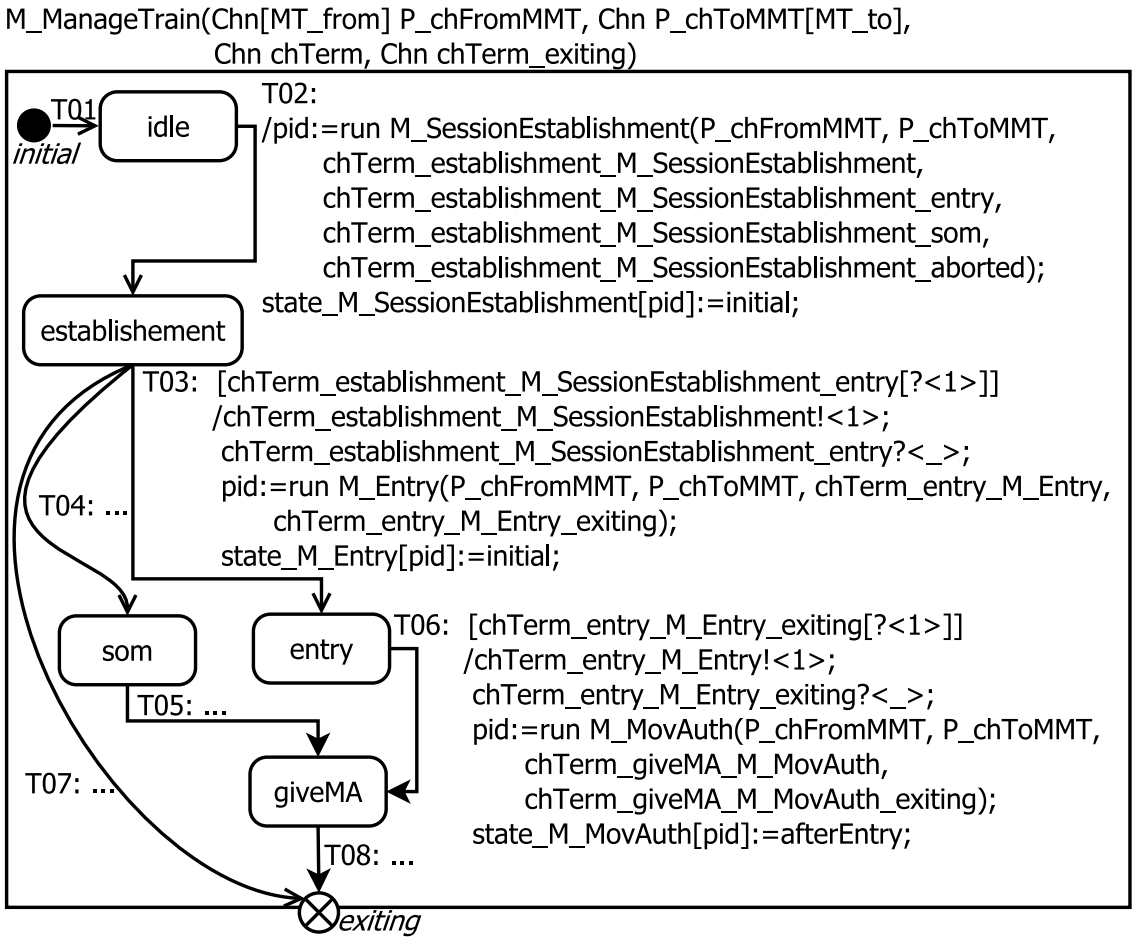


FIGURE 7.5: Flattened M_ManageTrain machine

taken and immediately the latter machine is interrupted (in fact the transition *T07* is preemptive). Indeed, if an alarm is raised, it forces the termination of the *M_PeriodicMA* and activates an instance of the machine *M_EmergencyManagement*. In case the train successfully ends the trip, the *M_PeriodicMA* machine terminates, causing the termination of *M_CentralControl* machine. Moreover it causes the activation of an instance of *M_EndOfMission* machine. This last machine serves to successfully terminate the activities of the *M_MovAuth* machine. Finally the termination of the *M_MovAuth* causes, in turn, the termination of the caller instance of machine *M_ManageTrain*.

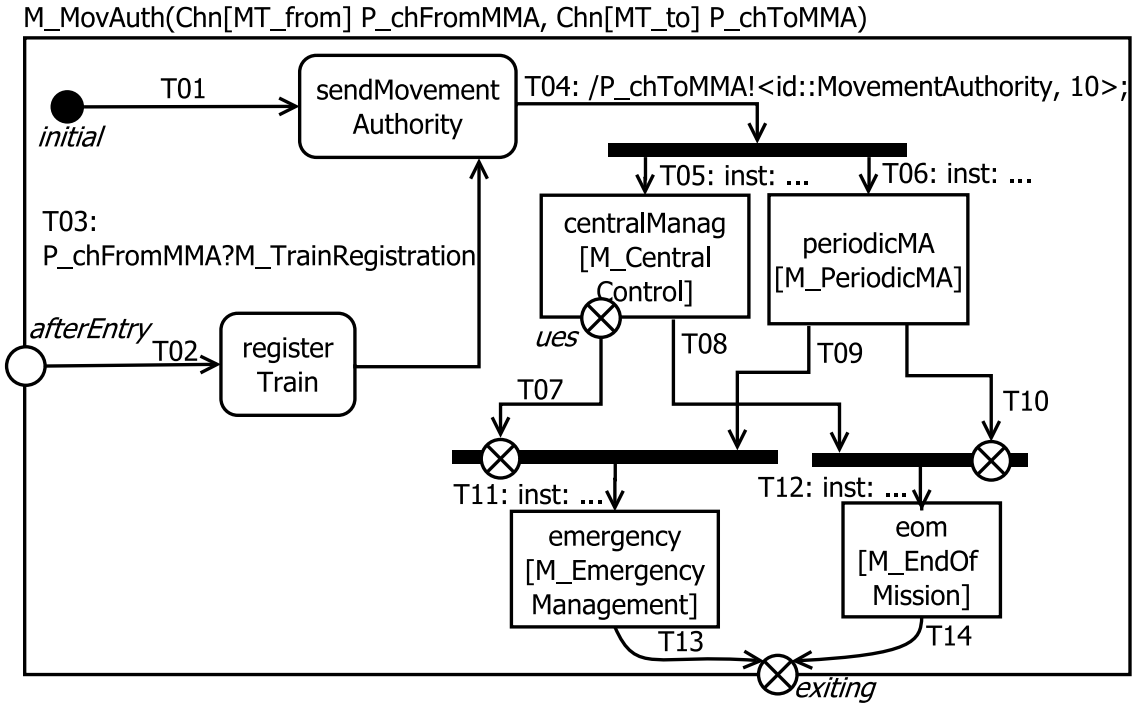


FIGURE 7.6: M_MovAuth machine

Flatten model The *M_MovAuth* machine is interesting since it shows the realization of a flattening process related to a machine containing synchronous fork and preemptive join transitions. According to the mapping process of the synchronous fork the currently executing process is suspended to wait for the termination of the called machines. In this case, the entire block, containing the fork, the join and the boxes, is substituted with a new node which models the state of the suspended calling process. In addition, the transitions in the block are replaced by a set of internal transitions to and from this wait node. The flat version of the *M_MovAuth* is depicted in Figure 7.7. The two boxes *M_CentralControl* and *M_PeriodicMA* are removed and replaced by the wait node

centralManag_periodicMA. Transitions T04, T05 and T06 are replaced by transition T04_T05_T06 whose actions perform the instantiation of the two machines *M_CentralControl* and *M_PeriodicMA* by means of suitable run constructs, according to what explained in the mapping process. Transitions T07, T09 and T11 are replaced by the internal transition T07_T09_T11, while transitions T08, T10 and T12 are replaced by transition T08_T10_T12. This example allows to illustrate an instance of a preemptive join. In fact, the transition T07 (and respectively T10) in Figure 7.6 is preemptive. The guard of transition T07_T09_T11 (and respectively T08_T10_T12) holds when the exiting message is received from the machine M_CentralControl (and respectively, M_PeriodicMA) on channel chTerm_centralManag_M_CentralControl_ues. The suspended caller wakes up and sends the termination message to both machines M_CentralControl and M_PeriodicMA, on the channels chTerm_centralManag_M_CentralControl and chTerm_periodicMA_M_PeriodicMA, respectively.

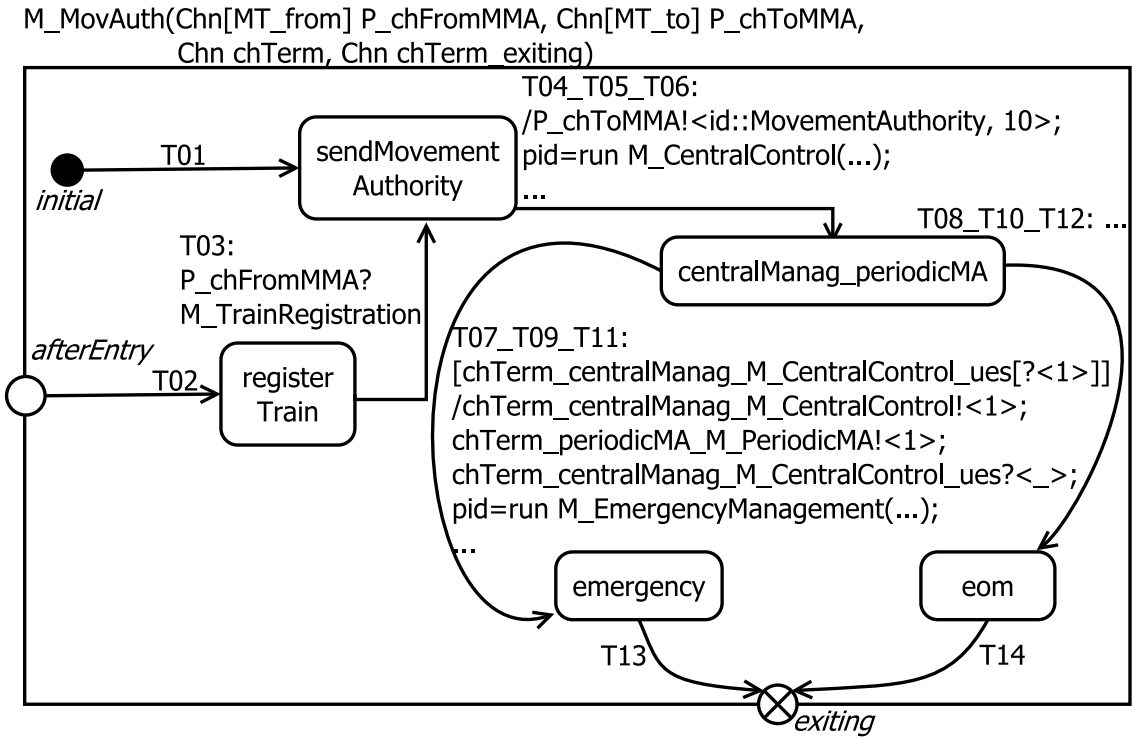


FIGURE 7.7: Flattened M_MovAuth machine

7.2.5 M_SessionEstablishment

The *M_SessionEstablishment* machine allows for the modelling of the session establishment protocol. This is a key feature of the Communication Procedure of the RBC, since is the machine the concretely instantiates the communication

channel between the RBC and a train. The *M_SessionEstablishment* does not provide a hierarchical structure. Thus, only the DSTM model is below discussed.

DSTM model The DSTM model of the *M_SessionEstablishment* is made according to the *Protocol Execution* system pattern. This system pattern provides that a machine waits for the reception of a correct sequence of messages from an external entity. On the reception of a correct message, it can answer with a proper message. If a message different from the expected one is received, the protocol is aborted by exiting a proper exiting node. This pattern requires the definition of n compound types (M1, . . . , Mn), each of them representing a possible message. According to this the *M_SessionEstablishment* machine has two parametric channels: *P_chFromMSE* and *P_chToMSE*. These parametric channels are instantiated by the transition *T02* with the concrete channels names associated with the parameters *P_chFromMMT* and *P_chToMMT*. The called machine *M_SessionEstablishment* sends the System Version message to the train over the parametric output channel *P_chToMSE*. Then, it waits for an acknowledgement message from the train on the parametric input channel *P_chFromMSE* (node waitForAck).

According to the *Protocol Execution* pattern, the channel *P_chFromMSE* is a multi-type channel of the *MT_from* that includes the type of the acknowledgement message *M_Ack*. Therefore, in order to check the presence of an acknowledgement message, the trigger used by transition T03 is $P_chFromMSE?M\_Ack$, requiring that the multi-type channel currently contains a message of the specific type *M_Ack*. If the currently delivered message is not of type *M_Ack* the protocol terminates reaching the abort exiting node (transition T02). After the reception of the acknowledgement, the machine waits for the communication of a session establishment message in the node *waitForSessEstab*. To check the presence of the communication of session establishment message, transitions T05 and T06 use the trigger $P\_chFromMSE?M\_SessionEstablished$, requiring that the corresponding multi-type channel currently containing a message of type *M_SessionEstablished*. If the currently delivered message is not of type *M_SessionEstablished*, the protocol terminates, reaching the abort exit node (transition T04). At this point, if the session establishment protocol terminates in the abort exiting node, the instance of *M_ManageTrain* terminates its activity as well. Termination of the session establishment protocol in the *entry* or *som* exit nodes, leads to the activations of machines *M_Entry* and *M_StartOfMission*, respectively. This allows for specifying the different procedures for trains coming from non high-speed area and trains starting their mission, respectively. After termination

of the entry procedures, Step 2 of the the communication management is concluded and machine *M_MovAuth* is instantiated to periodically send the MA to the train. Lest note that, after the termination of the *M_Entry* procedure, the *M_MovAuth* machine is activated in its entering node *afterEntry*, through the transition *T06*. Otherwise the activation is made on its default node that allows to record the area of the connection establishment.

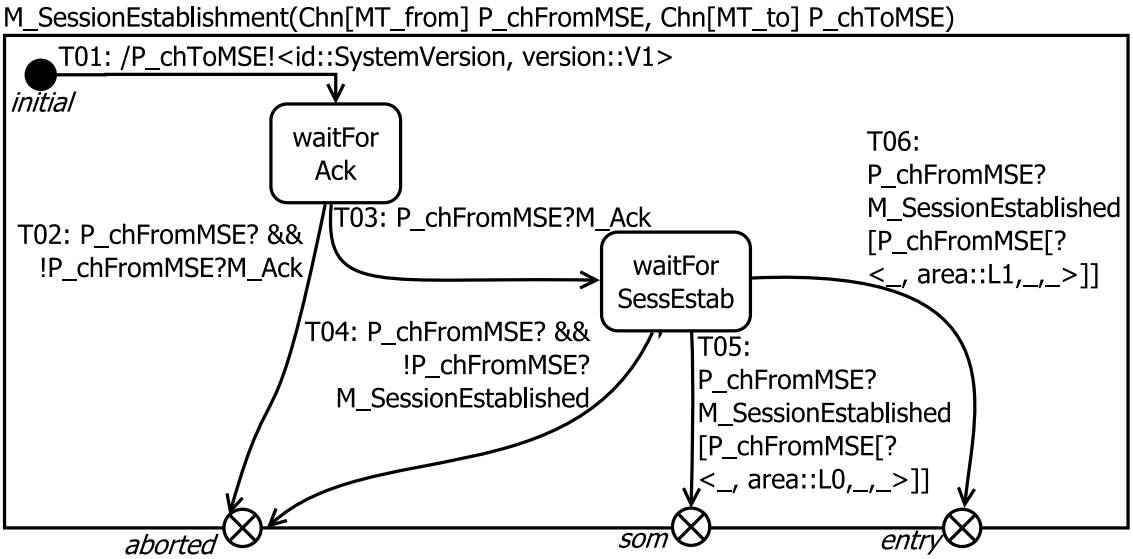


FIGURE 7.8: M_SessionEstablishment machine

7.3 Communication Procedure step semantics

A relevant feature of DSTM is that it owns a strictly formalized step semantics. The step semantics and the external environment are modelled by means of a special Promela process named *Engine*. The *Engine* process initializes the external channels, simulates the non-deterministic generation of messages over them and controls the step semantics by assigning the token for the transitions firing. Below the *Engine* process of the RBC Communication Procedure is described.

Channels initialization The exchange of messages between machines and the environment is performed using external channels defined in the data declarations. The external channels of the Communication Procedure are multi type. This means that for each of them, the generation of the single-type sub channels must be performed, for both the initialization process and the non-deterministic messages generation. As for example the a generic *ChFrom* channel (i.e. a channel managing messages coming from a train) is of *MtFrom* multi type. The *MtFrom*

multi type is composed by three sub-types related to the messages that can convey in the channels of this type: (*MSessionEstablished*, *MAck*, *MTrainRegistration*).

Thus, for each subtype a simple-type external channel is generated. In the specific case, considering the *ChFrom* channels, the simple type external channels generated are *CfromTrain_MSessionEstablished*, *CfromTrain_MAck* and *CfromTrain_MTrainRegistration*. Furthermore, strictly related to this, local variables for the non-deterministic generation are generated. Such local variables, named *temp* variables, are generated according to the process described in Section 6.5.5. Listing 7.2 shows the *temp* variables for the Communication Procedure and the initialization of the external channels.

LISTING 7.2: The DSTM model of the Communication Establishment Machine

```
active proctype Engine() {
pid PidMain;
//Temp Variables
mtype temp;
mtype temp_Canswer_1;
mtype temp_CfromTrain1_MSessionEstablished_1;
mtype temp_CfromTrain1_MSessionEstablished_2;
int temp_CfromTrain1_MSessionEstablished_3;
int temp_CfromTrain1_MSessionEstablished_4;
mtype temp_CfromTrain1_MAck_1;
mtype temp_CfromTrain1_MTrainRegistration_1;
mtype temp_CfromTrain1_MTrainRegistration_2;
mtype temp_CtoTrain1_MSystemVersion_1;
mtype temp_CtoTrain1_MSystemVersion_2;
mtype temp_CtoTrain1_MMovementAuthority_1;

...

mtype temp_CfromTrain4_MSessionEstablished_1;
mtype temp_CfromTrain4_MSessionEstablished_2;
int temp_CfromTrain4_MSessionEstablished_3;
int temp_CfromTrain4_MSessionEstablished_4;
mtype temp_CfromTrain4_MAck_1;
mtype temp_CfromTrain4_MTrainRegistration_1;
mtype temp_CfromTrain4_MTrainRegistration_2;
mtype temp_CtoTrain4_MSystemVersion_1;
mtype temp_CtoTrain4_MSystemVersion_2;
mtype temp_CtoTrain4_MMovementAuthority_1;
int temp_CtoTrain4_MMovementAuthority_2;
chan chTerm_MMain = [MAX_PROC] of {bit};
PidMain = run MMain(_pid,MMain_initial,chTerm_MMain);

//Generation of first messages
Canswer!0,0;
CfromTrain1_MSessionEstablished!0,0,0,0,0;
CfromTrain1_MAck!0,0;
CfromTrain1_MTrainRegistration!0,0,0;
CtoTrain1_MSystemVersion!0,0,0;
CtoTrain1_MMovementAuthority!0,0,0;

```

```
41 ...
42
43 CfromTrain4_MSessionEstablished!0,0,0,0,0;
44 CfromTrain4_MAck!0,0;
45 CfromTrain4_MTrainRegistration!0,0,0;
46 CtoTrain4_MSystemVersion!0,0,0;
47 CtoTrain4_MMovementAuthority!0,0,0;
```

Non-deterministic generation The non-deterministic generation is in charge of dispatching messages over the external channels. As reported in the Listing 7.3 in the *generation* block this task is performed by an iteration. First, the length of each external channel is checked in order to verify that is not full. This condition is checked by using the *len()* function natively provided by Promela (Listing 7.3, line 11). If the considered channel is of multi type in the corresponding DSTM model, the length check must be applied to each sub-channel (Listing 7.3, lines 30-31). In case the considered channel is full, the control is passed to the else statement, that add the TESQUEL directives to signal, in the resulting test case, such situation. Otherwise the non-deterministic generation, is performed, according to the structure of the external channels: each channel has two positions, one in which is memorized the current available value and one in which generated values of the current step are memorized for the next step execution. Listing 7.3 shows an excerpt of the non-deterministic generation of messages, related to the *Engine* process for a single train. The structure in the Listing 7.3, is repeated as many times as the number of trains that can be accepted by the considered RBC.

LISTING 7.3: The DSTM model of the Communication Establishment Machine

```
1  generation:
2   atomic {
3    if
4    :: (NoFirings==1) -> goto abort;
5    :: else -> skip;
6    fi;
7    NoFirings=1;
8    printf("<ENGINE: message generation>\n");
9    //MESSAGES ON Canswer
10   if
11   :: (len(Canswer)==1) ->
12    if
13    :: (1) ->
14     if
15     :: temp_Canswer_1=accepted;
16     :: temp_Canswer_1=refused;
17     fi;
18     printf("<ENGINE: Canswer - generated <%e>>\n",temp_Canswer_1);
19     Canswer!1,temp_Canswer_1;
```

```
20      :: (1) ->
21       printf("<ENGINE: Canswer - generated EMPTY>\n");
22       Canswer!0,0;
23      fi;
24     :: else ->
25      printf("<ENGINE: Canswer - generated BY SYSTEM>\n");
26     fi;
27     Canswer?temp,temp_Canswer_1;
28
29     //MESSAGES ON CfromTrain1
30     if
31     :: (len(CfromTrain1_MSessionEstablished) + len(CfromTrain1_MAck) + len(
           CfromTrain1_MTrainRegistration) == 3) ->
32      if
33      :: (1) ->
34       if
35       :: temp_CfromTrain1_MSessionEstablished_1=SessionEstablished;
36       :: temp_CfromTrain1_MSessionEstablished_1=Ack;
37       :: temp_CfromTrain1_MSessionEstablished_1=TrainRegistration;
38       :: temp_CfromTrain1_MSessionEstablished_1=SystemVersion;
39       :: temp_CfromTrain1_MSessionEstablished_1=MovementAuthority;
40       fi;
41       if
42       :: temp_CfromTrain1_MSessionEstablished_2=L0;
43       :: temp_CfromTrain1_MSessionEstablished_2=L1;
44       fi;
45       temp_CfromTrain1_MSessionEstablished_3 = 1
46       temp_CfromTrain1_MSessionEstablished_4 = 1
47       printf("<ENGINE: CfromTrain1_MSessionEstablished - generated <%e,%e,%d
             ,%d>>\n",temp_CfromTrain1_MSessionEstablished_1,
             temp_CfromTrain1_MSessionEstablished_2,
             temp_CfromTrain1_MSessionEstablished_3,
             temp_CfromTrain1_MSessionEstablished_4);
48       CfromTrain1_MSessionEstablished!1,
             temp_CfromTrain1_MSessionEstablished_1,
             temp_CfromTrain1_MSessionEstablished_2,
             temp_CfromTrain1_MSessionEstablished_3,
             temp_CfromTrain1_MSessionEstablished_4;
49       CfromTrain1_MAck!0,0;
50       CfromTrain1_MTrainRegistration!0,0,0;
51      :: (1) ->
52       if
53       :: temp_CfromTrain1_MAck_1=SessionEstablished;
54       :: temp_CfromTrain1_MAck_1=Ack;
55       :: temp_CfromTrain1_MAck_1=TrainRegistration;
56       :: temp_CfromTrain1_MAck_1=SystemVersion;
57       :: temp_CfromTrain1_MAck_1=MovementAuthority;
58       fi;
59       printf("<ENGINE: CfromTrain1_MAck - generated <%e>>\n",
             temp_CfromTrain1_MAck_1);
60       CfromTrain1_MAck!1,temp_CfromTrain1_MAck_1;
61       CfromTrain1_MSessionEstablished!0,0,0,0,0;
62       CfromTrain1_MTrainRegistration!0,0,0;
63      :: (1) ->
64       if
65       :: temp_CfromTrain1_MTrainRegistration_1=SessionEstablished;
66       :: temp_CfromTrain1_MTrainRegistration_1=Ack;
67       :: temp_CfromTrain1_MTrainRegistration_1=TrainRegistration;
```

```
68      :: temp_CfromTrain1_MTrainRegistration_1=SystemVersion;
69      :: temp_CfromTrain1_MTrainRegistration_1=MovementAuthority;
70      fi;
71      if
72      :: temp_CfromTrain1_MTrainRegistration_2=registered;
73      fi;
74      printf("<ENGINE: CfromTrain1_MTrainRegistration - generated <%e,%e>>\n
            ",temp_CfromTrain1_MTrainRegistration_1,
            temp_CfromTrain1_MTrainRegistration_2);
75      CfromTrain1_MTrainRegistration!1,temp_CfromTrain1_MTrainRegistration_1
            ,temp_CfromTrain1_MTrainRegistration_2;
76      CfromTrain1_MSessionEstablished!0,0,0,0,0;
77      CfromTrain1_MAck!0,0;
78     :: (1) ->
79      printf("<ENGINE: CfromTrain1 - generated EMPTY>\n");
80      CfromTrain1_MSessionEstablished!0,0,0,0,0;
81      CfromTrain1_MAck!0,0;
82      CfromTrain1_MTrainRegistration!0,0,0;
83     fi;
84    :: else ->
85     printf("<ENGINE: CfromTrain1 - generated BY SYSTEM>\n");
86    fi;
87    CfromTrain1_MSessionEstablished?temp,
          temp_CfromTrain1_MSessionEstablished_1,
          temp_CfromTrain1_MSessionEstablished_2,
          temp_CfromTrain1_MSessionEstablished_3,
          temp_CfromTrain1_MSessionEstablished_4;
88    CfromTrain1_MAck?temp,temp_CfromTrain1_MAck_1;
89    CfromTrain1_MTrainRegistration?temp,
          temp_CfromTrain1_MTrainRegistration_1,
          temp_CfromTrain1_MTrainRegistration_2;
90
91
92    //MESSAGES ON CtoTrain1
93    if
94    :: (len(CtoTrain1_MSystemVersion) + len(CtoTrain1_MMovementAuthority) ==
          2) ->
95     if
96     :: (1) ->
97      if
98      :: temp_CtoTrain1_MSystemVersion_1=SessionEstablished;
99      :: temp_CtoTrain1_MSystemVersion_1=Ack;
100     :: temp_CtoTrain1_MSystemVersion_1=TrainRegistration;
101     :: temp_CtoTrain1_MSystemVersion_1=SystemVersion;
102     :: temp_CtoTrain1_MSystemVersion_1=MovementAuthority;
103     fi;
104     if
105     :: temp_CtoTrain1_MSystemVersion_2=V0;
106     :: temp_CtoTrain1_MSystemVersion_2=V1;
107     :: temp_CtoTrain1_MSystemVersion_2=V2;
108     fi;
109     printf("<ENGINE: CtoTrain1_MSystemVersion - generated <%e,%e>>\n",
            temp_CtoTrain1_MSystemVersion_1,temp_CtoTrain1_MSystemVersion_2);
110     CtoTrain1_MSystemVersion!1,temp_CtoTrain1_MSystemVersion_1,
            temp_CtoTrain1_MSystemVersion_2;
111     CtoTrain1_MMovementAuthority!0,0,0;
112    :: (1) ->
113     if
```

```
114       :: temp_CtoTrain1_MMovementAuthority_1=SessionEstablished;
115       :: temp_CtoTrain1_MMovementAuthority_1=Ack;
116       :: temp_CtoTrain1_MMovementAuthority_1=TrainRegistration;
117       :: temp_CtoTrain1_MMovementAuthority_1=SystemVersion;
118       :: temp_CtoTrain1_MMovementAuthority_1=MovementAuthority;
119       fi;
120       temp_CtoTrain1_MMovementAuthority_2 = 10
121       printf("<ENGINE: CtoTrain1_MMovementAuthority - generated <%e,%d>>\n",
              temp_CtoTrain1_MMovementAuthority_1,
              temp_CtoTrain1_MMovementAuthority_2);
122       CtoTrain1_MMovementAuthority!1,temp_CtoTrain1_MMovementAuthority_1,
              temp_CtoTrain1_MMovementAuthority_2;
123       CtoTrain1_MSystemVersion!0,0,0;
124      :: (1) ->
125       printf("<ENGINE: CtoTrain1 - generated EMPTY>\n");
126       CtoTrain1_MSystemVersion!0,0,0;
127       CtoTrain1_MMovementAuthority!0,0,0;
128      fi;
129     :: else ->
130      printf("<ENGINE: CtoTrain1 - generated BY SYSTEM>\n");
131     fi;
132     CtoTrain1_MSystemVersion?temp,temp_CtoTrain1_MSystemVersion_1,
          temp_CtoTrain1_MSystemVersion_2;
133     CtoTrain1_MMovementAuthority?temp,temp_CtoTrain1_MMovementAuthority_1,
          temp_CtoTrain1_MMovementAuthority_2;
134
```

The generation block ends by assigning the token to the main process (Listing 7.4, line 2). Then, the Engine process enters the do construct, where it waits until the Promela global variable *timeout* is evaluated true. This happens when no statement can be executable in the active processes, hence when all the SUT processes have consumed their token. In this case, Engine executes a jump to the generation label and starts a new step.

LISTING 7.4: The DSTM model of the Communication Establishment Machine

```
1   //GIVE TOKEN TO THE MAIN PROCESS
2   HasToken[PidMain]=1;
3  }
4  do
5   :: timeout -> goto generation;
6  od;
7 abort:
8  dyingPid=_pid;
9 }
```

7.4 Test cases generation

Exploiting the Promela model described in the previous section and the Spin Model Checker, test cases have been generated. The generation process has been

based on the coverage of the SUT transitions. This kind of test specification is modelled by using the *Cover* pattern. According to a set of *Cover patterns*, the TCG generates a set of never claims. Each never claim, owns a variable named *LastTransition* that stores, for each instant of time, the name of the last transition entered. The never claim defines a behaviour in which the *LastTransition* variable does never assume the value of the element to cover. This property in fact, should be evaluated as false by the Spin Model Checker which returns back the required test case as a counterexample of the violated property. With the respect to this, two test cases are below discussed. The test cases are chosen among those obtained by the set of functional requirements of the Communication Procedure of the RBC. Each test case is described with the TESQEL language, whose statements are partially generated by outcomes of the model checker and then refined by the *Post Processor* component, described in Chapter 6. Some remarks on the executions time of the TCG framework are provided at the end of the section.

Req01 The **Req01** states that when the *SessionEstablished* message is received from a train, and the area field of the received message is *L1*, RBC shall perform the *Entry* action. Such requirement, has been annotated on the *T06* transition of the machine *M_SessionEstablishment*, depicted in Figure 7.8. The never claim describing such cover is made according to the general structure showed in Listing 6.24 of the previous chapter. The test case, resulting from the property verification made by Spin, is reported in Listing 7.5. Lets note that some steps of the test cases report the firing of more that one source transitions, when an intermediate pseudonode is crossed. As an example, the step1 of Table 5 reports the firing of T03_T04_T05 of *M_CommunicationEstablishment*, since a fork pseudonode is crossed in the that step.

LISTING 7.5: The DSTM model of the Communication Establishment Machine

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <tesqel:TestSequence xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
       xmlns:tesqel="tesqel" name="never_19" specification="never_19">
3    <initial>MMain_initial</initial>
4    <compoundFiring>
5      <firing>
6        <current>MMain_initial</current>
7        <current>MMain_idle</current>
8        <current>MMain_T01</current>
9      </firing>
10     <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
           SessionEstablished,L0,1,1>"/>
11   </compoundFiring>
12   <compoundFiring order="1">
13     <firing>
```

```
14        <current>MMain_idle</current>
15        <current>MMain_box</current>
16        <current>MMain_T02</current>
17      </firing>
18      <firing order="1">
19        <current>MEnvConstraints_initial</current>
20        <current>MEnvConstraints_waitFor1Train</current>
21        <current>MEnvConstraints_T01</current>
22      </firing>
23      <firing order="2">
24        <current>MCommunicationEstablishment_initial</current>
25        <current>MCommunicationEstablishment_idle</current>
26        <current>MCommunicationEstablishment_manageTrain</current>
27        <current>MCommunicationEstablishment_T01</current>
28      </firing>
29      <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
30      <link from="//@compoundFiring.1/@firing.0" to="//@compoundFiring.1/
            @firing.1"/>
31      <link from="//@compoundFiring.1/@firing.1" to="//@compoundFiring.1/
            @firing.2"/>
32    </compoundFiring>
33    <compoundFiring order="2">
34      <firing>
35        <current>MEnvConstraints_waitFor1Train</current>
36        <current>MEnvConstraints_endOfRequests</current>
37        <current>MEnvConstraints_T02</current>
38      </firing>
39      <firing order="1">
40        <current>MCommunicationEstablishment_idle</current>
41        <current>MCommunicationEstablishment_manageTrain</current>
42        <current>MCommunicationEstablishment_idle</current>
43        <current>MCommunicationEstablishment_manageTrain</current>
44        <current>MCommunicationEstablishment_T03</current>
45        <current>MCommunicationEstablishment_T04</current>
46        <current>MCommunicationEstablishment_T05</current>
47      </firing>
48      <firing order="2">
49        <current>MManageTrain_initial</current>
50        <current>MManageTrain_idle</current>
51        <current>MManageTrain_T01</current>
52      </firing>
53      <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
54      <link from="//@compoundFiring.2/@firing.0" to="//@compoundFiring.2/
            @firing.1"/>
55      <link from="//@compoundFiring.2/@firing.1" to="//@compoundFiring.2/
            @firing.2"/>
56    </compoundFiring>
57    <compoundFiring order="3">
58      <firing>
59        <current>MManageTrain_idle</current>
60        <current>MManageTrain_establishment</current>
61        <current>MManageTrain_T02</current>
62      </firing>
63      <firing order="1">
64        <current>MSessionEstablishment_initial</current>
65        <current>MSessionEstablishment_waitForAck</current>
```

```
66        <current>MSessionEstablishment_T01</current>
67      </firing>
68      <IOmessage channel="Canswer" direction="fromSystem"/>
69      <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
70      <link from="//@compoundFiring.3/@firing.0" to="//@compoundFiring.3/
            @firing.1"/>
71    </compoundFiring>
72    <compoundFiring order="4">
73      <firing>
74        <current>MSessionEstablishment_waitForAck</current>
75        <current>MSessionEstablishment_waitForSessEstab</current>
76        <current>MSessionEstablishment_T03</current>
77      </firing>
78      <IOmessage channel="CtoTrain1" direction="fromSystem"/>
79      <IOmessage channel="CfromTrain1_MAck" message="&lt;SessionEstablished
            >"/>
80    </compoundFiring>
81    <compoundFiring order="5">
82      <firing>
83        <current>MSessionEstablishment_waitForSessEstab</current>
84        <current>MSessionEstablishment_som</current>
85        <current>MSessionEstablishment_T05</current>
86      </firing>
87      <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
88    </compoundFiring>
89    <compoundFiring order="6">
90      <firing>
91        <current>MManageTrain_establishment</current>
92        <current>MManageTrain_som</current>
93        <current>MManageTrain_T04</current>
94      </firing>
95      <firing order="1">
96        <current>MStartOfMission_initial</current>
97        <current>MStartOfMission_working</current>
98        <current>MStartOfMission_T01</current>
99      </firing>
100     <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
101     <link from="//@compoundFiring.6/@firing.0" to="//@compoundFiring.6/
            @firing.1"/>
102   </compoundFiring>
103   <compoundFiring order="7">
104     <firing>
105       <current>MStartOfMission_working</current>
106       <current>MStartOfMission_exiting</current>
107       <current>MStartOfMission_T02</current>
108     </firing>
109     <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
110   </compoundFiring>
111   <compoundFiring order="8">
112     <firing>
113       <current>MManageTrain_som</current>
114       <current>MManageTrain_giveMA</current>
115       <current>MManageTrain_T05</current>
116     </firing>
```

```
117     <firing order="1">
118       <current>MMovAuth_initial</current>
119       <current>MMovAuth_sendMovementAuthority</current>
120       <current>MMovAuth_T01</current>
121     </firing>
122     <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
123     <link from="//@compoundFiring.8/@firing.0" to="//@compoundFiring.8/
            @firing.1"/>
124   </compoundFiring>
125   <compoundFiring order="9">
126     <firing>
127       <current>MMovAuth_sendMovementAuthority</current>
128       <current>MMovAuth_centralManag</current>
129       <current>MMovAuth_periodicMA</current>
130       <current>MMovAuth_T04</current>
131       <current>MMovAuth_T05</current>
132       <current>MMovAuth_T06</current>
133     </firing>
134     <firing order="1">
135       <current>MPeriodicMA_initial</current>
136       <current>MPeriodicMA_working</current>
137       <current>MPeriodicMA_T01</current>
138     </firing>
139     <firing order="2">
140       <current>MCentralControl_initial</current>
141       <current>MCentralControl_working</current>
142       <current>MCentralControl_T01</current>
143     </firing>
144     <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
145     <link from="//@compoundFiring.9/@firing.0" to="//@compoundFiring.9/
            @firing.1"/>
146     <link from="//@compoundFiring.9/@firing.1" to="//@compoundFiring.9/
            @firing.2"/>
147   </compoundFiring>
148   <compoundFiring order="10">
149     <firing>
150       <current>MPeriodicMA_working</current>
151       <current>MPeriodicMA_eom</current>
152       <current>MPeriodicMA_T02</current>
153     </firing>
154     <firing order="1">
155       <current>MCentralControl_working</current>
156       <current>MCentralControl_ues</current>
157       <current>MCentralControl_T03</current>
158     </firing>
159     <IOmessage channel="CtoTrain1" direction="fromSystem"/>
160     <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
161     <link from="//@compoundFiring.10/@firing.0" to="//@compoundFiring.10/
            @firing.1"/>
162   </compoundFiring>
163   <compoundFiring order="11">
164     <firing>
165       <current>MMovAuth_centralManag</current>
166       <current>MMovAuth_periodicMA</current>
167       <current>MMovAuth_emergency</current>
```

```
168         <current>MMovAuth_T07</current>
169         <current>MMovAuth_T09</current>
170         <current>MMovAuth_T11</current>
171       </firing>
172       <firing order="1">
173         <current>MEmergencyManagement_initial</current>
174         <current>MEmergencyManagement_working</current>
175         <current>MEmergencyManagement_T01</current>
176       </firing>
177       <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
              SessionEstablished,L0,1,1>"/>
178       <link from="//@compoundFiring.11/@firing.0" to="//@compoundFiring.11/
              @firing.1"/>
179     </compoundFiring>
180     <compoundFiring order="12">
181       <firing>
182         <current>MEmergencyManagement_working</current>
183         <current>MEmergencyManagement_exiting</current>
184         <current>MEmergencyManagement_T02</current>
185       </firing>
186       <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
              SessionEstablished,L0,1,1>"/>
187     </compoundFiring>
188     <compoundFiring order="13">
189       <firing>
190         <current>MMovAuth_emergency</current>
191         <current>MMovAuth_exiting</current>
192         <current>MMovAuth_T13</current>
193       </firing>
194       <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
              SessionEstablished,L0,1,1>"/>
195     </compoundFiring>
196     <link from="//@compoundFiring.0" to="//@compoundFiring.1"/>
197     <link from="//@compoundFiring.1" to="//@compoundFiring.2"/>
198     <link from="//@compoundFiring.2" to="//@compoundFiring.3"/>
199     <link from="//@compoundFiring.3" to="//@compoundFiring.4"/>
200     <link from="//@compoundFiring.4" to="//@compoundFiring.5"/>
201     <link from="//@compoundFiring.5" to="//@compoundFiring.6"/>
202     <link from="//@compoundFiring.6" to="//@compoundFiring.7"/>
203     <link from="//@compoundFiring.7" to="//@compoundFiring.8"/>
204     <link from="//@compoundFiring.8" to="//@compoundFiring.9"/>
205     <link from="//@compoundFiring.9" to="//@compoundFiring.10"/>
206     <link from="//@compoundFiring.10" to="//@compoundFiring.11"/>
207     <link from="//@compoundFiring.11" to="//@compoundFiring.12"/>
208     <link from="//@compoundFiring.12" to="//@compoundFiring.13"/>
209 </tesqel:TestSequence>
210
```

Req02 The **Req02** demands that when End of Mission message (EoM) message is received from a train, the RBC shall initiate the de-registration of the train by performing the EoM procedure. The **Req02** requirement is annotated on the *T10* transition of the *M_MovAuth* machine, showed in Figure 7.6. The complete test cases is showed in Listing 7.6.

LISTING 7.6: The DSTM model of the Communication Establishment Machine

```
<?xml version="1.0" encoding="ASCII"?>
<tesqel:TestSequence xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:tesqel="tesqel" name="never_25" specification="never_25">
  <initial>MMain_initial</initial>
  <compoundFiring>
    <firing>
      <current>MMain_initial</current>
      <current>MMain_idle</current>
      <current>MMain_T01</current>
    </firing>
    <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
        SessionEstablished,L0,1,1>"/>
  </compoundFiring>
  <compoundFiring order="1">
    <firing>
      <current>MMain_idle</current>
      <current>MMain_box</current>
      <current>MMain_T02</current>
    </firing>
    <firing order="1">
      <current>MEnvConstraints_initial</current>
      <current>MEnvConstraints_waitFor1Train</current>
      <current>MEnvConstraints_T01</current>
    </firing>
    <firing order="2">
      <current>MCommunicationEstablishment_initial</current>
      <current>MCommunicationEstablishment_idle</current>
      <current>MCommunicationEstablishment_manageTrain</current>
      <current>MCommunicationEstablishment_T01</current>
    </firing>
    <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
        SessionEstablished,L0,1,1>"/>
    <link from="//@compoundFiring.1/@firing.0" to="//@compoundFiring.1/
        @firing.1"/>
    <link from="//@compoundFiring.1/@firing.1" to="//@compoundFiring.1/
        @firing.2"/>
  </compoundFiring>
  <compoundFiring order="2">
    <firing>
      <current>MEnvConstraints_waitFor1Train</current>
      <current>MEnvConstraints_endOfRequests</current>
      <current>MEnvConstraints_T02</current>
    </firing>
    <firing order="1">
      <current>MCommunicationEstablishment_idle</current>
      <current>MCommunicationEstablishment_manageTrain</current>
      <current>MCommunicationEstablishment_idle</current>
      <current>MCommunicationEstablishment_manageTrain</current>
      <current>MCommunicationEstablishment_T03</current>
      <current>MCommunicationEstablishment_T04</current>
      <current>MCommunicationEstablishment_T05</current>
    </firing>
    <firing order="2">
      <current>MManageTrain_initial</current>
      <current>MManageTrain_idle</current>
```

```
51        <current>MManageTrain_T01</current>
52      </firing>
53      <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
54      <link from="//@compoundFiring.2/@firing.0" to="//@compoundFiring.2/
            @firing.1"/>
55      <link from="//@compoundFiring.2/@firing.1" to="//@compoundFiring.2/
            @firing.2"/>
56    </compoundFiring>
57    <compoundFiring order="3">
58      <firing>
59        <current>MManageTrain_idle</current>
60        <current>MManageTrain_establishment</current>
61        <current>MManageTrain_T02</current>
62      </firing>
63      <firing order="1">
64        <current>MSessionEstablishment_initial</current>
65        <current>MSessionEstablishment_waitForAck</current>
66        <current>MSessionEstablishment_T01</current>
67      </firing>
68      <IOmessage channel="Canswer" direction="fromSystem"/>
69      <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
70      <link from="//@compoundFiring.3/@firing.0" to="//@compoundFiring.3/
            @firing.1"/>
71    </compoundFiring>
72    <compoundFiring order="4">
73      <firing>
74        <current>MSessionEstablishment_waitForAck</current>
75        <current>MSessionEstablishment_waitForSessEstab</current>
76        <current>MSessionEstablishment_T03</current>
77      </firing>
78      <IOmessage channel="CtoTrain1" direction="fromSystem"/>
79      <IOmessage channel="CfromTrain1_MAck" message="&lt;SessionEstablished
            >"/>
80    </compoundFiring>
81    <compoundFiring order="5">
82      <firing>
83        <current>MSessionEstablishment_waitForSessEstab</current>
84        <current>MSessionEstablishment_som</current>
85        <current>MSessionEstablishment_T05</current>
86      </firing>
87      <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
88    </compoundFiring>
89    <compoundFiring order="6">
90      <firing>
91        <current>MManageTrain_establishment</current>
92        <current>MManageTrain_som</current>
93        <current>MManageTrain_T04</current>
94      </firing>
95      <firing order="1">
96        <current>MStartOfMission_initial</current>
97        <current>MStartOfMission_working</current>
98        <current>MStartOfMission_T01</current>
99      </firing>
100     <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
```

```
101     <link from="//@compoundFiring.6/@firing.0" to="//@compoundFiring.6/
            @firing.1"/>
102   </compoundFiring>
103   <compoundFiring order="7">
104     <firing>
105       <current>MStartOfMission_working</current>
106       <current>MStartOfMission_exiting</current>
107       <current>MStartOfMission_T02</current>
108     </firing>
109     <IOmessage channel="CfromTrain1_MSessionEstablished" message="&lt;
            SessionEstablished,L0,1,1>"/>
110   </compoundFiring>
111   <link from="//@compoundFiring.0" to="//@compoundFiring.1"/>
112   <link from="//@compoundFiring.1" to="//@compoundFiring.2"/>
113   <link from="//@compoundFiring.2" to="//@compoundFiring.3"/>
114   <link from="//@compoundFiring.3" to="//@compoundFiring.4"/>
115   <link from="//@compoundFiring.4" to="//@compoundFiring.5"/>
116   <link from="//@compoundFiring.5" to="//@compoundFiring.6"/>
117   <link from="//@compoundFiring.6" to="//@compoundFiring.7"/>
118 </tesqel:TestSequence>
```

Execution times A brief analysis of the execution times, deriving from the generation of the test cases of the Communication Procedure, has been performed. In particular, execution times derive from the printing of timestamps taken from the Linux system shell, on which the generation has been performed. More specifically the considered Linux system is equipped with an Intel Quad-core i7-2677M CPU 1.80GHz with 4 GB of RAM. According to the Section 6.7, describing the Spin compiling options, the only optimization considered is the *DSAFETY* option, that allows for the optimization of the memory.

The result depicted in Figure 7.9 demonstrates that test cases are generated in a feasible time. The histogram in Figure reports on the x-axis the times in milliseconds and on the y-axis the percentage of experiments whose execution times are related to that time value. It can be seen that the most of the executions keeps below four seconds, a groups of executions (about 18% are centred around 10 seconds) while a little percentage of the executions (less than 8%) are executed over 12 seconds depending on state-space, generated by the Spin Model Checker during the production of the counterexamples.

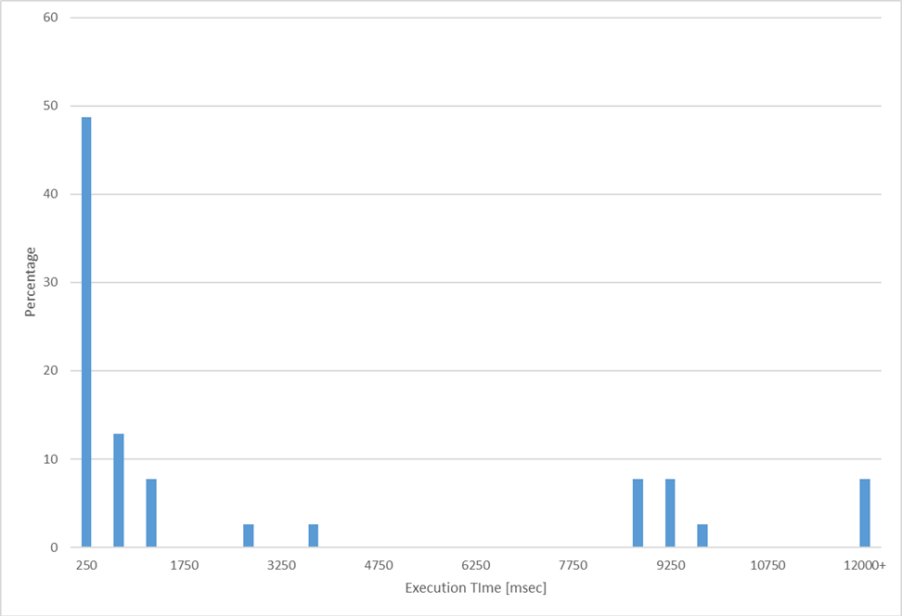


FIGURE 7.9: Generation times of the test cases of the Communication Procedure

Chapter 8

Conclusions

This thesis has been proposed a model-driven approach for the automated generation of test sequences for black-box system level testing. The approach is based on model checking techniques enabled by a behavioural model of the system under test and of the specifications of test, according to the system requirements.

According to the model-driven nature of the approach, and with the respect to the domain need, a domain-specific modelling language, strongly formalized, has been performed. The formalization is due to the critical nature of the treated systems. Moreover, in order to integrate the approach in a wider V&V process, the language has been defined as simple as possible and designed with the objective of achieve an high-level of usability. DSTM has been defined extending the well-know formalism of the Hierarchical State Machines, by introducing new features as the dynamic instantiation, the preemptive termination of the machines and the explicit modelling of the step semantics of the language. Moreover the concept of Test Specification Pattern has been discussed as an effective and efficient way to specify the desired property to test according to state- based model of a critical system. TSPs may have a relevant role in the automated generation process since they can allow for make easier the process of the test specification, re-using some predefined patterns, widely adopted in a given domain of interest. The automatic generation has been performed by defining proper mappings between the high-level specifications and the constructs provided Promela, the language of the chosen model checker. According to the model-driven principles on which the overall thesis relies, the realization of such mappings has been performed trough the definition of Model-to-Model and Model-to-Text transformations. This is a key point of the defined approach, since model-driven techniques promise a relevant reduction of inconsistencies that can occur between the artefacts created with a manual development process. This make the approach more attractive to be adopted by the industrial settings. The approach has been applied to a real railway safety-critical system, in order to exemplify the expressiveness of the proposed specification language and the potentialities

of the entire framework in test generation. The application in the railway context shows that the test generation approach provides a promising potentialities both from the perspectives of the scalability and the performances.

Future research efforts, will be focused on the definition of proper optimizations for the target formal model and on the extension of the proposed language in order to introduce concepts necessary to cover a wider range of embedded safety critical systems. With the respect for the optimization process, at the current state there is the need to define some heuristics to reduce the complexity of the Promela model used for the generation of the test cases. The heuristic optimization is needed in order to avoid the state explosion of the model checker when high-complex system are consider. In this context, a relevant activity actually under design, is the translation of the proposed high-level language for systems specicification in the alternative model checker environment, NuSMV [19]. Such translation will allow to compare the performances obtained by applying different model checking techniques and tools. Finally investigations on the application of the proposed approach in other domains will be performed. This investigations are encouraged by the promising results deriving from its application in other critical domains as the one of the Smart Energy Grids and the one of the Water Distribution Networks.

Appendix A

DSTM models of the Radio Block Centre system

A.1 The DSTM specification

This appendix provides the entire DSTM model related to the Communication Procedure of the RBC. According to this, both the data declarations and the model of the control flow are below provided.

A.1.1 Data declarations

The data declarations related to the Communication Procedure are reported in Listing A.1.

LISTING A.1: Data declarations for the Communication Procedure

```
1  Enum answer {accepted, refused};
2  Enum version {V0, V1, V2};
3  Enum registration {registered};
4  Enum area {L0, L1};
5  Enum msgId {SessionEstablished, Ack, TrainRegistration, SystemVersion, MovementAuthority};
6
7  Struct MAnswer {answer};
8  Struct MSessionEstablished {msgId, area, Int, Int};
9  Struct MAck {msgId};
10 Struct MTrainRegistration {msgId, registration};
11 Struct MSystemVersion {msgId, version};
12 Struct MMovementAuthority {msgId, Int};
13
14 Mtype MTfrom {MSessionEstablished, MAck, MTrainRegistration};
15 Mtype MTto {MSystemVersion, MMovementAuthority};
16
17 Struct MRequest {Chn [MTfrom], Chn [MTto]};
18
19 Int Vcont;
20 Chn [MTfrom] VchFrom;
21 Chn [MTto] VchTo;
22
23 Chn internal Crequest [1] of MRequest;
24 Chn external Canswer of MAnswer;
25 Chn external CfromTrain1 of MTfrom;
26 Chn external CtoTrain1 of MTto;
27 Chn external CfromTrain2 of MTfrom;
28 Chn external CtoTrain2 of MTto;
29 Chn external CfromTrain3 of MTfrom;
30 Chn external CtoTrain3 of MTto;
31 Chn external CfromTrain4 of MTfrom;
32 Chn external CtoTrain4 of MTto;
```

```
33
34 Param PchFromMMT:Chn [MTfrom] of MManageTrain;
35 Param PchToMMT:Chn [MTto] of MManageTrain;
36 Param PchFromMSE:Chn [MTfrom] of MSessionEstablishment;
37 Param PchToMSE:Chn [MTto] of MSessionEstablishment;
38 Param PchFromMMA:Chn [MTfrom] of MMovAuth;
39 Param PchToMMA:Chn [MTto] of MMovAuth;
40 Param PchFromME:Chn [MTfrom] of MEntry;
41 Param PchToME:Chn [MTto] of MEntry;
42 Param PchFromMSOM:Chn [MTfrom] of MStartOfMission;
43 Param PchToMSOM:Chn [MTto] of MStartOfMission;
44 Param PchFromMCC:Chn [MTfrom] of MCentralControl;
45 Param PchToMCC:Chn [MTto] of MCentralControl;
46 Param PchFromMPMA:Chn [MTfrom] of MPeriodicMA;
47 Param PchToMPMA:Chn [MTto] of MPeriodicMA;
48 Param PchFromMEM:Chn [MTfrom] of MEmergencyManagement;
49 Param PchToMEM:Chn [MTto] of MEmergencyManagement;
50 Param PchFromMEOM:Chn [MTfrom] of MEndOfMission;
51 Param PchToMEOM:Chn [MTto] of MEndOfMission;
```

A.1.2 DSTM Machines

The Listing A.2 shows the complete DSTM model of the control flow of all the ten machines composing the Communication Procedure.

LISTING A.2: The DSTM model of the Communication Procedure

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <dstm4rail:DSTM xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org
       /2001/XMLSchema-instance" xmlns:dstm4rail="dstm4rail" name="CommunicationEstablishment" main
       ="//@machines[name='MMain']">
3  --Main machine
4    <machines name="M_Main">
5      <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
6      <vertexes xsi:type="dstm4rail:Box" name="box" instantiation="//@machines[name='
           MCommunicationEstablishment'] //@machines[name='MEnvConstraints']"/>
7      <vertexes xsi:type="dstm4rail:Node" name="idle"/>
8      <transitions name="T01" source="//@machines[name='MMain']/@vertexes[name='initial']"
           destination="//@machines[name='MMain']/@vertexes[name='idle']"/>
9      <transitions name="T02" source="//@machines[name='MMain']/@vertexes[name='idle']" destination
           ="//@machines[name='MMain']/@vertexes[name='box']"/>
10   </machines>
11
12 --M_CommunicationEstablishment machine
13   <machines name="MCommunicationEstablishment">
14     <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
15     <vertexes xsi:type="dstm4rail:Node" name="idle"/>
16     <vertexes xsi:type="dstm4rail:Fork" name="fork"/>
17     <vertexes xsi:type="dstm4rail:Join" name="join"/>
18     <vertexes xsi:type="dstm4rail:Box" name="manageTrain" instantiation="//@machines[name='
           MManageTrain']"/>
19     <transitions name="T01" condition="" source="//@machines[name='MCommunicationEstablishment']/
           @vertexes[name='initial']" destination="//@machines[name='MCommunicationEstablishment']/
           @vertexes[name='idle']">
20       <actions>Vcont=0;</actions>
21     </transitions>
22     <transitions name="T02" trigger="Crequest?" condition="Vcont==4" source="//@machines[name='
           MCommunicationEstablishment']/@vertexes[name='idle']" destination="//@machines[name='
           MCommunicationEstablishment']/@vertexes[name='idle']">
23       <actions>Canswer!&lt;answer::refused>;</actions>
24       <actions>Crequest?&lt;_,_>;</actions>
25     </transitions>
26     <transitions name="T03" trigger="Crequest?" condition="Vcont&lt;=3" source="//@machines[name='
           MCommunicationEstablishment']/@vertexes[name='idle']" destination="//@machines[name='
           MCommunicationEstablishment']/@vertexes[name='fork']">
27       <actions>Crequest?&lt;VchFrom,VchTo>;</actions>
28       <actions>Canswer!&lt;answer::accepted>;</actions>
29       <actions>Vcont=(Vcont++);</actions>
30     </transitions>
31     <transitions name="T04" source="//@machines[name='MCommunicationEstablishment']/@vertexes[name=
           'fork']" destination="//@machines[name='MCommunicationEstablishment']/@vertexes[name='
           idle']"/>
```

```
32      <transitions name="T05" source="//@machines[name='MCommunicationEstablishment']/@vertexes[name=
            'fork']" destination="//@machines[name='MCommunicationEstablishment']/@vertexes[name='
            manageTrain']">
33        <par_instantiation>PchFromMMT=VchFrom;PchToMMT=VchTo;</par_instantiation>
34      </transitions>
35      <transitions name="T06" source="//@machines[name='MCommunicationEstablishment']/@vertexes[name=
            'idle']" destination="//@machines[name='MCommunicationEstablishment']/@vertexes[name='
            join']"/>
36      <transitions name="T07" source="//@machines[name='MCommunicationEstablishment']/@vertexes[name=
            'manageTrain']" destination="//@machines[name='MCommunicationEstablishment']/@vertexes[
            name='join']"/>
37      <transitions name="T08" source="//@machines[name='MCommunicationEstablishment']/@vertexes[name=
            'join']" destination="//@machines[name='MCommunicationEstablishment']/@vertexes[name='
            idle']">
38        <actions>Vcont=(Vcont--);</actions>
39      </transitions>
40    </machines>
41
42  --M_ManageTrain machine
43    <machines name="MManageTrain">
44      <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
45      <vertexes xsi:type="dstm4rail:ExitingNode" name="exiting"/>
46      <vertexes xsi:type="dstm4rail:Node" name="idle"/>
47      <vertexes xsi:type="dstm4rail:Box" name="establishment" instantiation="//@machines[name='
            MSessionEstablishment']"/>
48      <vertexes xsi:type="dstm4rail:Box" name="entry" instantiation="//@machines[name='MEntry']"/>
49      <vertexes xsi:type="dstm4rail:Box" name="som" instantiation="//@machines[name='MStartOfMission'
            ]"/>
50      <vertexes xsi:type="dstm4rail:Box" name="giveMA" instantiation="//@machines[name='MMovAuth']"/>
51      <transitions name="T01" trigger="" condition="" source="//@machines[name='MManageTrain']/
            @vertexes[name='initial']" destination="//@machines[name='MManageTrain']/@vertexes[name='
            idle']"/>
52      <transitions name="T02" condition="" source="//@machines[name='MManageTrain']/@vertexes[name='
            idle']" destination="//@machines[name='MManageTrain']/@vertexes[name='establishment']">
53        <par_instantiation>PchFromMSE=PchFromMMT;PchToMSE=PchToMMT;</par_instantiation>
54      </transitions>
55      <transitions name="T03" source="//@machines[name='MManageTrain']/@vertexes[name='establishment'
            ]" destination="//@machines[name='MManageTrain']/@vertexes[name='entry']" exiting_node
            ="//@machines[name='MSessionEstablishment']/@vertexes[name='entry']">
56        <par_instantiation>PchFromME=PchFromMMT;PchToME=PchToMMT;</par_instantiation>
57      </transitions>
58      <transitions name="T04" source="//@machines[name='MManageTrain']/@vertexes[name='establishment'
            ]" destination="//@machines[name='MManageTrain']/@vertexes[name='som']" exiting_node="//
            @machines[name='MSessionEstablishment']/@vertexes[name='som']">
59        <par_instantiation>PchFromMSOM=PchFromMMT;PchToMSOM=PchToMMT;</par_instantiation>
60      </transitions>
61      <transitions name="T05" source="//@machines[name='MManageTrain']/@vertexes[name='som']"
            destination="//@machines[name='MManageTrain']/@vertexes[name='giveMA']">
62        <par_instantiation>PchFromMMA=PchFromMMT;PchToMMA=PchToMMT;</par_instantiation>
63      </transitions>
64      <transitions name="T06" source="//@machines[name='MManageTrain']/@vertexes[name='entry']"
            destination="//@machines[name='MManageTrain']/@vertexes[name='giveMA']" entering_node="//
            @machines[name='MMovAuth']/@vertexes[name='afterEntry']">
65        <par_instantiation>PchFromMMA=PchFromMMT;PchToMMA=PchToMMT;</par_instantiation>
66      </transitions>
67      <transitions name="T07" source="//@machines[name='MManageTrain']/@vertexes[name='establishment'
            ]" destination="//@machines[name='MManageTrain']/@vertexes[name='exiting']" exiting_node
            ="//@machines[name='MSessionEstablishment']/@vertexes[name='aborted']"/>
68      <transitions name="T08" source="//@machines[name='MManageTrain']/@vertexes[name='giveMA']"
            destination="//@machines[name='MManageTrain']/@vertexes[name='exiting']">
69        <requirements>Identifier=SRS_MMT_T08;Title=Ending of Train management;Modified=2015-09-25
              12:15:04;Url=null</requirements>
70      </transitions>
71    </machines>
72
73  --M_SessionEstablishment machine
74    <machines name="MSessionEstablishment">
75      <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
76      <vertexes xsi:type="dstm4rail:ExitingNode" name="aborted"/>
77      <vertexes xsi:type="dstm4rail:ExitingNode" name="som"/>
78      <vertexes xsi:type="dstm4rail:ExitingNode" name="entry"/>
79      <vertexes xsi:type="dstm4rail:Node" name="waitForAck"/>
80      <vertexes xsi:type="dstm4rail:Node" name="waitForSessEstab"/>
81      <transitions name="T01" source="//@machines[name='MSessionEstablishment']/@vertexes[name='
            initial']" destination="//@machines[name='MSessionEstablishment']/@vertexes[name='
            waitForAck']">
82        <actions>PchToMSE!&lt;msgId::SystemVersion,version::V1>;</actions>
83      </transitions>
```

```
84    <transitions name="T02" trigger="((PchFromMSE?) &amp;amp;amp; (!PchFromMSE?MAck))" source="//
          @machines[name='MSessionEstablishment']/@vertexes[name='waitForAck']" destination="//
          @machines[name='MSessionEstablishment']/@vertexes[name='aborted']"/>
85    <transitions name="T03" trigger="PchFromMSE?MAck" source="//@machines[name='
          MSessionEstablishment']/@vertexes[name='waitForAck']" destination="//@machines[name='
          MSessionEstablishment']/@vertexes[name='waitForSessEstab']"/>
86    <transitions name="T04" trigger="((PchFromMSE?) &amp;amp;amp; (!PchFromMSE?MSessionEstablished))"
          source="//@machines[name='MSessionEstablishment']/@vertexes[name='waitForSessEstab']"
          destination="//@machines[name='MSessionEstablishment']/@vertexes[name='aborted']"/>
87    <transitions name="T05" trigger="PchFromMSE?MSessionEstablished" condition="PchFromMSE[?&lt;_,
          area::L0,_,_>]" source="//@machines[name='MSessionEstablishment']/@vertexes[name='
          waitForSessEstab']" destination="//@machines[name='MSessionEstablishment']/@vertexes[name
          ='som']"/>
88    <transitions name="T06" trigger="PchFromMSE?MSessionEstablished" condition="PchFromMSE[?&lt;_,
          area::L1,_,_>]" source="//@machines[name='MSessionEstablishment']/@vertexes[name='
          waitForSessEstab']" destination="//@machines[name='MSessionEstablishment']/@vertexes[name
          ='entry']">
89      <requirements>Identifier=SRS_MSE_T06;Title=Session establishment;Modified=2015-09-25
            12:15:04;Url=null</requirements>
90    </transitions>
91  </machines>
92
93  --M_MovAuth machine
94  <machines name="MMovAuth">
95    <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
96    <vertexes xsi:type="dstm4rail:EnteringNode" name="afterEntry"/>
97    <vertexes xsi:type="dstm4rail:ExitingNode" name="exiting"/>
98    <vertexes xsi:type="dstm4rail:Fork" name="fork"/>
99    <vertexes xsi:type="dstm4rail:Join" name="join1"/>
100   <vertexes xsi:type="dstm4rail:Join" name="join2"/>
101   <vertexes xsi:type="dstm4rail:Node" name="sendMovementAuthority"/>
102   <vertexes xsi:type="dstm4rail:Node" name="registerTrain"/>
103   <vertexes xsi:type="dstm4rail:Box" name="centralManag" instantiation="//@machines[name='
          MCentralControl']"/>
104   <vertexes xsi:type="dstm4rail:Box" name="periodicMA" instantiation="//@machines[name='
          MPeriodicMA']"/>
105   <vertexes xsi:type="dstm4rail:Box" name="emergency" instantiation="//@machines[name='
          MEmergencyManagement']"/>
106   <vertexes xsi:type="dstm4rail:Box" name="eom" instantiation="//@machines[name='MEndOfMission'
          ]"/>
107   <transitions name="T01" source="//@machines[name='MMovAuth']/@vertexes[name='initial']"
          destination="//@machines[name='MMovAuth']/@vertexes[name='sendMovementAuthority']"/>
108   <transitions name="T02" source="//@machines[name='MMovAuth']/@vertexes[name='afterEntry']"
          destination="//@machines[name='MMovAuth']/@vertexes[name='registerTrain']"/>
109   <transitions name="T03" trigger="PchFromMMA?MTrainRegistration" source="//@machines[name='
          MMovAuth']/@vertexes[name='registerTrain']" destination="//@machines[name='MMovAuth']/
          @vertexes[name='sendMovementAuthority']"/>
110   <transitions name="T04" source="//@machines[name='MMovAuth']/@vertexes[name='
          sendMovementAuthority']" destination="//@machines[name='MMovAuth']/@vertexes[name='fork'
          ]">
111     <actions>PchToMMA!&lt;msgId::MovementAuthority,10>;</actions>
112   </transitions>
113   <transitions name="T05" source="//@machines[name='MMovAuth']/@vertexes[name='fork']"
          destination="//@machines[name='MMovAuth']/@vertexes[name='centralManag']">
114     <par_instantiation>PchFromMCC=PchFromMMA;PchToMCC=PchToMMA;</par_instantiation>
115   </transitions>
116   <transitions name="T06" source="//@machines[name='MMovAuth']/@vertexes[name='fork']"
          destination="//@machines[name='MMovAuth']/@vertexes[name='periodicMA']">
117     <par_instantiation>PchFromMPMA=PchFromMMA;PchToMPMA=PchToMMA;</par_instantiation>
118   </transitions>
119   <transitions name="T07" is_preemptive="true" source="//@machines[name='MMovAuth']/@vertexes[
          name='centralManag']" destination="//@machines[name='MMovAuth']/@vertexes[name='join1']"
          exiting_node="//@machines[name='MCentralControl']/@vertexes[name='ues']"/>
120   <transitions name="T08" source="//@machines[name='MMovAuth']/@vertexes[name='centralManag']"
          destination="//@machines[name='MMovAuth']/@vertexes[name='join2']"/>
121   <transitions name="T09" condition="" source="//@machines[name='MMovAuth']/@vertexes[name='
          periodicMA']" destination="//@machines[name='MMovAuth']/@vertexes[name='join1']"/>
122   <transitions name="T10" is_preemptive="true" source="//@machines[name='MMovAuth']/@vertexes[
          name='periodicMA']" destination="//@machines[name='MMovAuth']/@vertexes[name='join2']"/>
123   <transitions name="T11" source="//@machines[name='MMovAuth']/@vertexes[name='join1']"
          destination="//@machines[name='MMovAuth']/@vertexes[name='emergency']">
124     <par_instantiation>PchFromMEM=PchFromMMA;PchToMEM=PchToMMA;</par_instantiation>
125   </transitions>
126   <transitions name="T12" source="//@machines[name='MMovAuth']/@vertexes[name='join2']"
          destination="//@machines[name='MMovAuth']/@vertexes[name='eom']">
127     <par_instantiation>PchFromMEOM=PchFromMMA;PchToMEOM=PchToMMA;</par_instantiation>
128   </transitions>
```

```
129     <transitions name="T13" source="//@machines[name='MMovAuth']/@vertexes[name='emergency']"
            destination="//@machines[name='MMovAuth']/@vertexes[name='exiting']"/>
130     <transitions name="T14" source="//@machines[name='MMovAuth']/@vertexes[name='eom']" destination
            ="//@machines[name='MMovAuth']/@vertexes[name='exiting']"/>
131   </machines>
132
133 --M_Entry machine
134   <machines name="MEntry">
135     <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
136     <vertexes xsi:type="dstm4rail:ExitingNode" name="exiting"/>
137     <vertexes xsi:type="dstm4rail:Node" name="working"/>
138     <transitions name="T01" source="//@machines[name='MEntry']/@vertexes[name='initial']"
            destination="//@machines[name='MEntry']/@vertexes[name='working']"/>
139     <transitions name="T02" trigger="" source="//@machines[name='MEntry']/@vertexes[name='working'
            ]" destination="//@machines[name='MEntry']/@vertexes[name='exiting']"/>
140   </machines>
141
142 --M_StartOfMission machine
143   <machines name="MStartOfMission">
144     <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
145     <vertexes xsi:type="dstm4rail:ExitingNode" name="exiting"/>
146     <vertexes xsi:type="dstm4rail:Node" name="working"/>
147     <transitions name="T01" source="//@machines[name='MStartOfMission']/@vertexes[name='initial']"
            destination="//@machines[name='MStartOfMission']/@vertexes[name='working']"/>
148     <transitions name="T02" source="//@machines[name='MStartOfMission']/@vertexes[name='working']"
            destination="//@machines[name='MStartOfMission']/@vertexes[name='exiting']"/>
149   </machines>
150
151 --M_CentralControl machine
152   <machines name="MCentralControl">
153     <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
154     <vertexes xsi:type="dstm4rail:ExitingNode" name="exiting"/>
155     <vertexes xsi:type="dstm4rail:ExitingNode" name="ues"/>
156     <vertexes xsi:type="dstm4rail:Node" name="working"/>
157     <transitions name="T01" source="//@machines[name='MCentralControl']/@vertexes[name='initial']"
            destination="//@machines[name='MCentralControl']/@vertexes[name='working']"/>
158     <transitions name="T02" source="//@machines[name='MCentralControl']/@vertexes[name='working']"
            destination="//@machines[name='MCentralControl']/@vertexes[name='exiting']"/>
159     <transitions name="T03" source="//@machines[name='MCentralControl']/@vertexes[name='working']"
            destination="//@machines[name='MCentralControl']/@vertexes[name='ues']"/>
160   </machines>
161
162 --M_PeriodicMA machine
163   <machines name="MPeriodicMA">
164     <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
165     <vertexes xsi:type="dstm4rail:ExitingNode" name="eom"/>
166     <vertexes xsi:type="dstm4rail:Node" name="working"/>
167     <transitions name="T01" source="//@machines[name='MPeriodicMA']/@vertexes[name='initial']"
            destination="//@machines[name='MPeriodicMA']/@vertexes[name='working']"/>
168     <transitions name="T02" source="//@machines[name='MPeriodicMA']/@vertexes[name='working']"
            destination="//@machines[name='MPeriodicMA']/@vertexes[name='eom']"/>
169   </machines>
170
171 --M_EmergencyManagement machine
172   <machines name="MEmergencyManagement">
173     <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
174     <vertexes xsi:type="dstm4rail:ExitingNode" name="exiting"/>
175     <vertexes xsi:type="dstm4rail:Node" name="working"/>
176     <transitions name="T01" source="//@machines[name='MEmergencyManagement']/@vertexes[name='
            initial']" destination="//@machines[name='MEmergencyManagement']/@vertexes[name='working'
            ]"/>
177     <transitions name="T02" source="//@machines[name='MEmergencyManagement']/@vertexes[name='
            working']" destination="//@machines[name='MEmergencyManagement']/@vertexes[name='exiting'
            ]"/>
178   </machines>
179
180 --M_EndOfMission machine
181   <machines name="MEndOfMission">
182     <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
183     <vertexes xsi:type="dstm4rail:ExitingNode" name="exiting"/>
184     <vertexes xsi:type="dstm4rail:Node" name="working"/>
185     <transitions name="T01" source="//@machines[name='MEndOfMission']/@vertexes[name='initial']"
            destination="//@machines[name='MEndOfMission']/@vertexes[name='working']"/>
186     <transitions name="T02" source="//@machines[name='MEndOfMission']/@vertexes[name='working']"
            destination="//@machines[name='MEndOfMission']/@vertexes[name='exiting']"/>
187   </machines>
188
```

```
189 --M_EnvConstrains machine
190   <machines name="MEnvConstraints">
191     <vertexes xsi:type="dstm4rail:InitialNode" name="initial"/>
192     <vertexes xsi:type="dstm4rail:Node" name="waitFor1Train"/>
193     <vertexes xsi:type="dstm4rail:Node" name="waitFor2Train"/>
194     <vertexes xsi:type="dstm4rail:Node" name="waitFor3Train"/>
195     <vertexes xsi:type="dstm4rail:Node" name="waitFor4Train"/>
196     <vertexes xsi:type="dstm4rail:ExitingNode" name="endOfRequests"/>
197     <transitions name="T01" source="//@machines[name='MEnvConstraints']/@vertexes[name='initial']"
            destination="//@machines[name='MEnvConstraints']/@vertexes[name='waitFor1Train']"/>
198     <transitions name="T02" source="//@machines[name='MEnvConstraints']/@vertexes[name='
            waitFor1Train']" destination="//@machines[name='MEnvConstraints']/@vertexes[name='
            waitFor2Train']">
199       <actions>Crequest!&lt;CfromTrain1,CtoTrain1>;</actions>
200     </transitions>
201     <transitions name="T03" source="//@machines[name='MEnvConstraints']/@vertexes[name='
            waitFor2Train']" destination="//@machines[name='MEnvConstraints']/@vertexes[name='
            waitFor3Train']">
202       <actions>Crequest!&lt;CfromTrain2,CtoTrain2>;</actions>
203     </transitions>
204     <transitions name="T04" condition="" source="//@machines[name='MEnvConstraints']/@vertexes[name
            ='waitFor3Train']" destination="//@machines[name='MEnvConstraints']/@vertexes[name='
            waitFor4Train']">
205       <actions>Crequest!&lt;CfromTrain3,CtoTrain3>;</actions>
206     </transitions>
207     <transitions name="T05" source="//@machines[name='MEnvConstraints']/@vertexes[name='
            waitFor4Train']" destination="//@machines[name='MEnvConstraints']/@vertexes[name='
            endOfRequests']">
208       <actions>Crequest!&lt;CfromTrain4,CtoTrain4>;</actions>
209     </transitions>
210   </machines>
211 </dstm4rail:DSTM>
```

Acknowledgements

In every life experience, there are some people that you feel you want to thank for making part of your journey more pleasant. According to this, the first thanks is for Proff. Valeria Vittorini and Nicola Mazzocca, which give me the possibility to join their research group supervising me in every moment of this experience.
Thanks to Stefano, that has always found the time for give me and advice or simply for make me a laugh.
A big thanks is for Roberto, a great friend and a "super" tutor, that helps me during the whole phd and in particular in the editing of the thesis.
During these years I met wonderful persons, as Alessandra and Carmen that welcome me warmly since the beginning.
Thanks also to the guys of the SECLAB research group with which I shared a lot of funny moments.
I cannot forget to thank my best friends, Giuliana and Antonio for every beer shared in joy and Nadia for every nice evening.
Last but not least thanks to my mother and my father, who have always supported me in my choices, teaching me the ways to make them consciously.
Thanks to my sister for discussions but also for the affection.
Thanks to my ever love Eleonora, who has always been by my side even when she was kilometres away.
Finally thanks to all people that are not mentioned above but that with every small acts contributed to make unbelievable this chapter of my life.
This is not the end of an experience but the begin of a new history waiting to be discovered.

Ugo Gentile

Bibliography

[1] W.M.P. Van der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros. "Workflow patterns". In: *Distributed and Parallel Databases* 14.1 (2003), pp. 5–51.

[2] A. Abdulhameed, A. Hammad, H. Mountassir, and B. Tatibouet. "An approach to verify SysML functional requirements using Promela/SPIN". In: *Programming and Systems (ISPS), 2015 12th International Symposium on*. IEEE Computer Society, 2015, pp. 1–9.

[3] *Acceleo*. http://www.eclipse.org/acceleo. [Online; accessed 25-02-2016]. 2013.

[4] R. Alur. "Formal analysis of hierarchical state machines". In: *Verification: Theory and Practice*. Springer, 2003, pp. 42–66.

[5] R. Alur, S. Kannan, and M. Yannakakis. "Communicating Hierarchical State Machines". In: *Automata, Languages and Programming*. Vol. 1644. Springer Berlin Heidelberg, 1999, pp. 169–178.

[6] R. Alur and M. Yannakakis. "Model checking of hierarchical state machines". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 23. 6. ACM. 1998, pp. 175–188.

[7] S. Baharom and Z. Shukur. "Module documentation based testing using grey-box approach". In: *Information Technology, 2008. ITSim 2008. International Symposium on*. Vol. 2. IEEE. 2008, pp. 1–6.

[8] N. J. Bahr. *System safety engineering and risk assessment: a practical approach*. CRC Press, 2014.

[9] P. Baker, Z. R. Dai, J. Grabowski, I. Schieferdecker, and C. Williams. *Model-driven testing: Using the UML testing profile*. Springer Science & Business Media, 2007.

[10] G. Barberio, B. Di Martino, N. Mazzocca, L. Velardi, A. Amato, R. De Guglielmo, U. Gentile, S. Marrone, R. Nardone, A. Peron, and V. Vittorini. "An interoperable testing environment for ERTMS/ETCS control systems". In: *Computer Safety, Reliability, and Security*. Springer International Publishing, 2014, pp. 147–156.

[11] D. Bjørner. "New results and trends in formal techniques and tools for the development of software for transportation systems: a review". In: *Proceedings of the 4th Symposium on Formal Methods for Railway Operation and Control Systems (FORMS03), LHarmattan Hongrie, Budapest* (2003).

[12] P. E. Black, V. Okun, and Y. Yesha. "Mutation operators for specifications". In: *Automated Software Engineering, The Fifteenth IEEE International Conference on*. IEEE. 2000, pp. 81–88.

[13] B. Boehm. *Software risk management*. Springer, 1989.

[14] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. "A subset of precise UML for model-based testing". In: *Proceedings of the 3rd international workshop on Advances in model-based testing*. ACM. 2007, pp. 95–104.

[15] B. R Bryant, J. Gray, M. Mernik, P. J Clarke, R. B France, and G. Karsai. "Challenges and directions in formalizing the semantics of modeling languages". In: *Computer Science and Information Systems* 8.2 (2011), pp. 225–253.

[16] CENELEC. *CENELEC, EN 50126: Railway applications - Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 1: Generic RAMS process*. 2012.

[17] *CESAR*. http://www.cesarproject.eu/. [Online; accessed 25-02-2016]. 2009.

[18] K. Chen, J. Sztipanovits, and S. Neema. "Toward a Semantic Anchoring Infrastructure for Domain-specific Modeling Languages". In: *Proceedings of the 5th ACM International Conference on Embedded Software*. ACM, 2005, pp. 35–43.

[19] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. "NuSMV: a new symbolic model checker". In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425.

[20] T. Clark, P. Sammut, and J. Willans. "Applied metamodelling: a foundation for language driven development." In: (2008).

[21] E. M Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[22] C. Crichton, A. Cavarra, and J. Davies. "Using UML for automatic test generation". In: *Proc. of the Intern. Conf. On Automated Software Engineering, ASE*. 2001.

[23] *CRYSTAL*. http://www.cystal-artemis.eu/. [Online; accessed 25-03-2016]. 2013.

[24] F. Dadeau and F. Peureux. "Grey-Box Testing and Verification of Java/JML". In: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. 2011, pp. 298–303.

[25] Z. R. Dai. "Model-driven testing with UML 2.0". In: *Computer Science at Kent* (2004), p. 179.

[26] *DECOS*. http://www.ercim.eu/publication/Ercim_News/enw52/donhoffer.html. [Online; accessed 25-02-2016]. 2004.

[27] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. "A Survey on Model-based Testing Approaches: A Systematic Review". In: *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies 2007*. ACM, 2007, pp. 31–36.

[28] Y. Dong, G. Wang, and H. B. Zhao. "A Model-based Testing for AADL Model of Embedded Software". In: *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE. 2009, pp. 185–190.

[29] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[30] M.B. Dwyer, G-S. Avrunin, and J.C. Corbett. "Patterns in Property Specifications for Finite-state Verification". In: *Proc. of the ICSE '99*. ACM, 1999, pp. 411–420.

[31] *Epsilon Book*. 2015. URL: http://www.eclipse.org/epsilon/doc/book/.

[32] *EU Council Directive 2001/16/EC - Interoperability of the trans-European conventional rail system*. 2001. URL: http://eur-lex.europa.eu/legal-content/IT/TXT/?uri=uriserv:l24015.

[33] *EU Council Directive 96/48/EC - Interoperability of the trans-European high-speed rail system*. 1996. URL: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31996L0048:en:HTML.

[34] R. T. Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[35] K. Fowler. *Mission-critical and safety-critical systems handbook: Design and development for embedded applications*. Newnes, 2009.

[36] G. Fraser, F. Wotawa, and P. E. Ammann. "Testing with model checkers: a survey". In: *Software Testing, Verification and Reliability* 19.3 (2009), pp. 215–261.

[37] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. "An introduction to UML profiles". In: *UML and Model Engineering* 2 (2004).

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.

[39] S. R Ganov, C. Killmar, S. Khurshid, and D. E Perry. "Test generation for graphical user interfaces based on symbolic execution". In: *Proceedings of the 3rd international workshop on Automation of software test*. ACM. 2008, pp. 33–40.

[40] A. Gargantini and C. Heitmeyer. "Using model checking to generate tests from requirements specifications". In: *Software Engineering—ESEC/FSE'99*. Springer. 1999, pp. 146–162.

[41] U. Gentile, S. Marrone, G. Mele, R. Nardone, and A. Peron. "Formal Methods for Industrial Critical Systems: 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings". In: Springer International Publishing, 2014. Chap. Test Specification Patterns for Automatic Generation of Test Sequences, pp. 170–184.

[42] S. Ghosh. *Distributed systems: an algorithmic approach*. CRC press, 2014.

[43] M. Glinz. "Statecharts For Requirements Specification - As Simple As Possible, As Rich As Needed". In: *Proceedings of the ICSE 2002 International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*. Orlando, Florida, USA, 2002.

[44] L. Grunske. "Specification Patterns for Probabilistic Quality Properties". In: *Proceeding of ICSE'08*. ACM, 2008, pp. 31–40.

[45] N. Guelfi and B. Ries. "SESAME: A Model-Driven Test Selection Process for Safety-Critical Embedded Systems". In: *ERCIM News* 75 (2008).

[46] W. von Hagen. *The Definitive Guide to GCC, Second Edition (Definitive Guide)*. Berkely, CA, USA: Apress, 2006.

[47] D. Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[48] R. Heckel and M. Lohmann. "Towards model-driven testing". In: *Electronic Notes in Theoretical Computer Science* 82.6 (2003), pp. 33–43.

[49] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. "Formal Approaches to Software Testing: Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003." In: Springer Berlin Heidelberg, 2004. Chap. Auto-generating Test Sequences Using Model Checkers: A Case Study, pp. 42–59.

[50] G. Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[51] D. Huizinga and A. Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[52] "IEEE Standard for Software Verification and Validation Plans". In: *IEEE Std 1012-1986* (1986).

[53] ISO. *ISO/IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. 2012.

[54] *ISO 26262-1:2011 Road vehicles – Functional safety*. 2011. URL: http://www.iso.org/iso/catalogue_detail?csnumber=43464.

[55] A. Z. Javed, P. A Strooper, and GN Watson. "Automated generation of test cases using model-driven architecture". In: *Automation of Software Test, 2007. AST'07. Second International Workshop on*. IEEE. 2007, pp. 3–3.

[56] F. Jouault and I. Kurtev. "Transforming Models with ATL". In: vol. 3844. LNCS. Springer Berlin Heidelberg, 2006, pp. 128–138.

[57] *Kermeta*. 2008. URL: http://www.kermeta.org/.

[58] D. S Kolovos, R. F Paige, and F. AC Polack. "The epsilon transformation language". In: *Theory and practice of model transformations*. Springer, 2008, pp. 46–60.

[59] R. Lanotte, A. Maggiolo-Schettini, A. Peron, and S. Tini. "Dynamic Hierarchical Machines". In: *Fundamenta Informaticae* 54.2-3 (2002). ISSN: 0169-2968.

[60] J.C. Laprie. *Dependability: Basic Concepts and Terminology, Dependable Computing and Fault-Tolerance*. Springer-Verlag, 1992.

[61] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guo. "Generating test cases from UML activity diagram based on Gray-box method". In: *Software Engineering Conference, 2004. 11th Asia-Pacific*. 2004, pp. 284–291.

[62] K. Louden. *Programming languages: principles and practices*. Cengage Learning, 2011.

[63] P. Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[64] *mbat*. http://www.mbat-artemis.eu/home/. [Online; accessed 25-02-2016]. 2014.

[65] P. McMinn. "Search-based software test data generation: A survey". In: *Software Testing Verification and Reliability* 14.2 (2004), pp. 105–156.

[66] W. E. McUmber and B. H. C. Cheng. "A General Framework for Formalizing UML with Formal Languages". In: *Proceedings of the 23rd International Conference on Software Engineering*. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 433–442.

[67] MetaCase. *MetaEdit+*. 1995. URL: http://www.metacase.com/products.html.

[68] C. Mingsong, Q. Xiaokang, and L. Xuandong. "Automatic test case generation for UML activity diagrams". In: *Proceedings of the 2006 international workshop on Automation of software test*. ACM. 2006, pp. 2–8.

[69] O. Mondragon, A.Q. Gates, S. Roach, H. Mendoza, and O. Sokolsky. "Generating Properties for Runtime Monitoring from Software Specification Patterns". In: *International Journal of Software Engineering and Knowledge Engineering* 17.1 (2007), pp. 107–126.

[70] Jean-François Monin. *Understanding formal methods*. Springer Science & Business Media, 2012.

[71] G. J. Myers and C. Sandler. *The Art of Software Testing 3rd Edition*. John Wiley Sons, 2011.

[72] R. Nardone, U. Gentile, M. Benerecetti, A. Peron, V. Vittorini, S. Marrone, and N. Mazzocca. "Modeling Railway Control Systems in Promela". In: *Fourth International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2015)* (), p. 181.

[73] R. Nardone, U. Gentile, A. Peron, M. Benerecetti, V. Vittorini, S. Marrone, R. De Guglielmo, N. Mazzocca, and L. Velardi. "Dynamic state machines for formalizing railway control system specifications". In: *Formal Techniques for Safety-Critical Systems*. Springer International Publishing, 2014, pp. 93–109.

[74] OMG. *MARTE Profile*. 2011. URL: http://www.omg.org/spec/MARTE/1.1/.

[75] OMG. *MOF Model To Text Transformation Language*. 2008. URL: http://www.omg.org/spec/MOFM2T/1.0/.

[76] *OPENCOSS*. http://www.opencoss-project.eu/. [Online; accessed 25-02-2016]. 2015.

[77] A. Petrenko, N. Yevtushenko, and R. Dssouli. "Testing strategies for communicating fsms". In: *Protocol Test Systems*. Springer, 1995, pp. 193–208.

[78] *PRESTO project*. http://www.presto-embedded.eu/. [Online; accessed 25-02-2016]. 2011.

[79] S. C Reid. "An empirical analysis of equivalence partitioning, boundary value analysis and random testing". In: *Software Metrics Symposium, 1997. Proceedings., Fourth International*. IEEE. 1997, pp. 64–73.

[80] SableCC. *SableCC*. 2009. URL: http://www.sablecc.org/.

[81] *SafeCer*. http://www.safecer.eu/. [Online; accessed 25-02-2016]. 2014.

[82] D. C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* (2006), pp. 25–31.

[83] B. Selic. "A systematic approach to domain-specific language design using UML". In: *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*. IEEE. 2007, pp. 2–9.

[84] S. Sendall and W. Kozaczynski. "Model Transformation: The Heart and Soul of Model-Driven Software Development". In: *IEEE Software* 20.5 (2003), pp. 42–45.

[85] R. Soley. "Model driven architecture". In: *OMG white paper* 308 (2000), p. 5.

[86] I. Sommerville. *Software Engineering*. Addison-Wesley, 2011.

[87] N. Soundarajan, J. O. Hallstrom, G. Shu, and A. Delibas. "Patterns: from system design to software testing". In: *Innovations in Systems and Software Engineering* 4.1 (2008), pp. 71–85.

[88] Spinroot. *Spin Model Checker*. 1999. URL: http://spinroot.com/spin/whatispin.html.

[89] N. Srinivas and K. Deb. "Muiltiobjective optimization using nondominated sorting in genetic algorithms". In: *Evolutionary computation* 2.3 (1994), pp. 221–248.

[90] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009.

[91] *The javax.swing package*. 2015. URL: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html.

[92] H. Tómasson and H. Neukirchen. "Distributed Testing of Cloud Computing Applications Using the TTCN-3-based Jata Test Framework". In: *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*. 2013, pp. 22–29.

[93] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[94] J. Zander, I. Schieferdecker, and P. J Mosterman. *Model-based testing for embedded systems*. CRC press, 2011.

[95] Z. Zhang, T. Wu, and J. Zhang. "Boundary value analysis in automatic white-box test generation". In: *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. 2015, pp. 239–249.