PhD-FSTC-4-2009

Faculté des Sciences, de la Technologie et de la Communication

THÈSE

Soutenue le 07/05/2009 à Luxembourg

En vue de l'obtention du grade académique de

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

par

Benoît Ries
né le 21 février 1979 à Jaunay-Clan (France)

SESAME: A MODEL-DRIVEN PROCESS FOR THE TEST SELECTION OF SMALL-SIZE SAFETY-RELATED EMBEDDED SOFTWARE

Jury de thèse

To my angel,

"When someone is searching," said Siddhartha, "then it might easily happen that the only thing his eyes still see is that what he searches for, that he is unable to find anything, to let anything enter his mind, because he always thinks of nothing but the object of his search, because he has a goal, because he is obsessed by the goal. Searching means: having a goal. But finding means: being free, being open, having no goal. You, oh venerable one, are perhaps indeed a searcher, because, striving for your goal, there are many things you don't see, which are directly in front of your eyes."

Herman Hesse
Siddhartha

ACKNOWLEDGEMENTS

A particular thank goes to Nicolas Guelfi, my thesis director. Your sharp criticism and loose supervision have helped me constructing my own opinion, your stimulation and encouragments have been very welcomed during the hard times. I will particulary rembember our philosophical discussions on models and other meta-discussions that have colored our meetings.

I would also like to warmly thank David Wiseman, for accepting to supervize the industrial-side of this work. Your indepth expertise in software process definition and software quality are invaluable. Our relaxed, yet professional, meetings have greatly helped making this thesis more concrete.

I would like to thank Pascal Bouvry, Didier Buchs, Ina Schieferdecker and Yves Le Traon for accepting to be part of the thesis jury and providing constructive feedback during the defense.

I am particularly indebted to Aloyse Schoos for allowing me to perform this industrial PhD thesis at IEE and his flexibility during the numerous administrative obstacles.

A phd thesis could not be achieved without a team. I would like to thank all the LASSY team members of the University of Luxembourg for providing a good work environment. In particular Alfredo, Barbara, Cedric, Jacques, thank you for being there during the difficult part of the writing of the thesis... I would also like to thank the embedded software team, for sharing their industrial concerns with me and bringing an enjoyable work environment.

I would like to warmly thank my family, my parents who have always trusted in me and believed in my potential since my very young age, my sister who supported me and my little brothers whose sole presence made me happy. My grand parents have also been very understanding and I thank them for that. Merci mamie et papie.

Finally, I would like to thank the person without which this thesis would have never ended, my loving and lovely wife Angela. Thank you for bearing my mood swings and depressive attitude during this long journey. This journey is now over and a new life is awaiting us.

ABSTRACT

Embedded software applications are part of our daily lives (home appliances, means of transport, etc.), some of which have safety implications on human beings. It is thus important to trust this type of software. In our industrial context, testing is the most widely-spread technique used for that purpose. The state of practice is divided in two main categories. Explicit test selection techniques that enumerate the test cases to be exercised, but do not offer large coverage of the system's behavior; and implicit techniques that encompass fully automated test generation techniques, which tend to hide the test selection information from engineers. The problem that we aim at solving in this thesis is the definition of a test selection approach that is capable of helping test engineers to better reach a delimited and verifiable test set with respect to some given test requirements and project specificities taking into account different test stakeholders.

Model-Driven Engineering (MDE) is a part of software engineering in which models are not only used as documentation but also as a building tool. Models become the backbone of the software development process. In this thesis, we define a lightweight model-driven test selection process (called SESAME) based on formal grounds that is particularly adapted to small-size and safety-related embedded software systems. The SESAME process makes a systematic and coherent use of meta models and model transformations, the two fundamental concepts of MDE. Meta models are used to precisely define the different types of SESAME models and model transformations are used to give meanings to these models.

The SESAME process is composed of the four following tasks: specification of the system for testing purposes; specification of a test selection that constrains the system specification; evaluation of the test selection; generation of test cases derived from the constrained system specification.

To perform these tasks, two modeling languages are defined: FOREST, defined from a subset of UML, for the specification of the System Under Test and its Test System as black-boxes. Its concise meta model is particularly well-suited to engineering teams for its rapid adoption; and SERELA, a domain-specific language targeted at test selection specification. SERELA precisely delimits the test set to be exercised by constraining FOREST models. The interpretation of SERELA on FOREST models is defined as model transformations.

Lastly, in order to provide verification of the test selection, FOREST models are given semantics in terms of the existing Alloy formal language. A generic model transformation defines the semantics of FOREST models. The advantage of these Alloy formal models is that they can be precisely analyzed. The formal analysis is used to perform evaluation of the test selection against some metrics; it is also used for generating test cases.

As a validation of our approach, we apply the SESAME process to a case study taken from an industrial project. This industrial application has shown the applicability of our approach on a small-size safety-related embedded system.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Abstract

In the introduction chapter, we explain the motivation behind this thesis. The industrial context, in which this thesis has been performed, is presented. We state the current problems encountered in this context and we present the contributions to the solution that this thesis provides. Finally, the organization of this manuscript is presented.

1.1 Motivation

Nowadays, embedded systems are everywhere, from washing machines to cars, trains, and planes, etc....Numerous definitions of *embedded systems* exist, we take the following as a reference [BS05]: "Embedded systems are programmable electronic subsystems which are generally an integral part of a larger heterogeneous system". We consider in this thesis, small-size embedded systems which have a restricted computation capacity as well as storage capacity. A typical example are the electronic control units (ECU) present in cars, for instance, those responsible for the airbag deployment or car breaking systems.

Specification and testing activities are crucial phases of the development of small-size embedded systems. System specification activities (among others activities of software development processes) are human activities. As such they are error-prone, that is why the verification phase is an unavoidable phase of the development process. Verification activities, part of a system's development, are classically divided in two categories, dynamic verification (test) and static verification (analysis). In this thesis, we are interested in the dynamic verification of systems based on their specification; commonly called *specification-based testing* [ROT89] (or conformance testing). During this phase, system's specifications are used as a reference to verify the implemented system, and test cases are exclusively derived from the specification of the System Under Test (SUT)[1]. For many industrials, which subcontract system implementation activities to external companies, these are the only tests that can be accomplished, thus are of particular importance. One of the advantages resulting from specification based testing is that test can be specified earlier in the development process, thus are ready before the end of the system's implementation. Moreover, when tests are specified earlier [OA99], engineers may find possible inconsistencies or ambiguities in the specifications. This may be performed, either by automatic analysis, or by informal inspection of the test specification, which allows improving the system's specification; avoiding potential implementation of incorrect specifications.

[1] Actually, the term "system under development" is more appropriate in our context because the system may not yet exist, still we will use the term SUT in the remaining of this thesis to fit to the usual software testing terminology.

As mentioned previously, in a specification-based testing approach (in contrast with implementation-based testing), test cases are solely generated from specifications of the SUT. Specifications of the system are made of a set of specifications documents. This set of specifications documents focus on different views on the system and on different granularities of details. Specifications of a system may contain, among other specification documents, the following documents: system requirements specifications, system design specifications, software requirements specifications, software module specifications, etc. Each of these documents may specify different kinds of properties (e.g. behavioral, structural, real-time, etc.) that will be of particular interest for testing. In this thesis we focus on the derivation of software requirements specifications of behavioral and structural properties of the SUT. Several modeling languages exist for the behavioral specification of reactive embedded systems [HP85], in the context of this thesis, we focus on modeling languages based on states machines, firstly because they are well-spread in the industry [2], and secondly because state machines are defined within the Unified Modeling Language (UML). In the industrial context of this thesis, it is particularly important that the modeling language can potentially be exchanged easily among stakeholders. In this thesis, we focus solely on the software part of the system's specification, knowing that state machines may also be used for hardware component descriptions [DH89] and that our approach could be adapted to take into account hardware properties.

Exhaustive testing of a system consists in testing all possible behaviors of the SUT. In most cases, and even for small-sized systems, exhaustive testing of the system is impossible. Test cases generated for exhaustive testing are usually too numerous to be completely executed. For instance, let's take the test of sending a 4 bytes message to a communication bus. The amount of test cases required for exhaustive testing of the sending of all possible messages of four bytes is 2^{32}, which makes around 10^9 test cases. Assuming that each test case takes 1ms to be executed, it would take around 10 days to exhaustively test this small part of the system, which may be unreasonable for most companies. More generally, test selection is necessary because of the potential infinitively large test set derived from software specifications, due mainly to the two following reasons:

- Specification can represent an infinite or too large number of traces (e.g. caused by variables ranging over infinite or very large definition domains)
- Specification can contain infinite or too long traces (e.g. caused by never-ending loops)

Thus there is an obvious need for selection of the test cases to be executed out of the potentially huge set of test cases from exhaustive testing. In short, it is all about choosing which of the possible behaviors of the system are going to be tested and which ones are not going to be tested.

In practice, test cases selection is rarely done in a systematic manner; test engineers usually select the test cases of interest based on the textual system specifications. Test engineers read the specifications and choose the input values that exercise important software behaviors based on past experiences or intuition.

The main objective of this thesis is to improve the efficiency of activities performed by test engineers during the phase of specification-based testing. This approach should be, on the one hand, based on formal grounds, and on the other hand, be usable by test engineers.

[2] STATEMATE is a lodestar-tool of systems' behavior specifications in the automobile industry

1.2 Industrial context

This PhD project has been partly performed within the I.E.E. company [IEE]. I.E.E. is an international company leading in the offering of sensor-based systems. Products range from car seat pressure mats to 3D vision sensing solutions.

Originally, I.E.E. was focused on the development of non programmable physical systems, i.e. systems without embedded software. The company put in place a process for the development of system, including the testing of hardware attributes of the developed systems. The developed hardware products are safety-related and as such benefit from a rigorous test process that is tool-supported. Hardware-related tests mainly focused on putting the products within extreme climatic conditions (i.e. low/high temperature and humidity) and under varying electro-magnetic fields. Functional tests are performed at IEE but are of lower priority than hardware-related tests. The performed functional tests have an unknown coverage, and the amount of testing performed is usually defined by the amount of time available in the schedule.

Due to the growing complexity of systems to be developed and the growing number of customers, a (rather quick) shift from hardware-systems to embedded software-systems has been made within the company in order to produce more flexible systems tightly adapted to the customers needs. This resulted in a biased situation. The first bias concerns the test activity which is focused on physical attributes and thus do not cover the attributes introduced for the purpose of the embedded software. The second bias is similar but concerns the requirements activity. Requirements specifications tend to pay more attention to physical attributes of the system. In our context of specification-based testing, this may have important consequences, in particular, the supplier may neglect its own software requirements (and thus test cases) coming from its software development (and test) platforms.

We believe that there are a number of companies that share similarities with IEE system development history. That is why the SESAME approach presented in this thesis and developed for the I.E.E. company is indeed also applicable to a broad range of companies. The following characteristics of companies for which the SESAME process is applicable are:

- Concerning the process in place at companies targeted by the SESAME approach, their process should typically have the following characteristics[3]:
  - *low capability level* of the company for its software testing processes. This could be software testing processes assessed as incomplete processes (level 0 of SPICE process assessment standard [ISO04]) or performed processes (SPICE level 1). In these cases, software testing processes are not fully managed nor fully defined and their work products are not completely established.
  - *model-based software development.* The process in place *may* be already based on (UML) models either for documentation purposes or for generating code. But the model-based development in place does not cover the testing activities. As the modeling language used within the SESAME approach is fully compatible with UML, it is possible to integrate the SESAME test selection process within an existing model-based development process covering other activities of the software development life cycle. In deed, the SESAME process can also be used if the software development in place is not based on models at all.

[3] The first characteristic is required and the second one is optional

- The SESAME approach may be used for test selection of a broad range of software which share the same characteristics than software developed at I.E.E., as follows:
  - *small-size* in terms of the number of events exchanged between the software and its environment, *and* in terms of their physical distribution. Systems are restricted to be hosted by a single hardware unit (typically an electronic control unit) part of a larger system. As a result, the SESAME process model use UML class diagrams restricted to a small number of concepts and protocol state machines which suit the modeling of small-size software.
  - *safety-related.* The software functionalities are safety-related, i.e. if software fails at performing its functionalities according to its requirements, it may result in endangering human beings. That is why, the SESAME process model is based on formal grounds, in order to provide precise verification of the test selection.
  - *embedded software.* Software targeted for test selection are embedded in hardware, so they are usually already compiled and their source code is not accessible from systems outside the boundary of the software (e.g. Test System). That is why the SESAME test selection process model focuses on the black-box testing of the embedded software, i.e. testing of the interface of the software with its environment.

1.3 Problem statement

Various techniques [Mat08, PTV96, RC81, WC80] are used to select test cases from a usually very large number of data and behaviors of the SUT. Some of these techniques follow a scenario-driven test selection approach [BDG$^+$08], where the test cases are enumerated from critical behaviors specified in the requirements. The main advantage of this first type of approach is that test specifications focus on "important behaviors" of the SUT but often suffer from a poor coverage of the input domain.

The second type of test selection approach focuses on the automatic generation of test cases [OA99, Hes06], based on coverage criteria. The main advantages of this type of approach are that they cover a larger part of the SUT behaviors and are usually supported by tools. But in an industrial context, this approach often results in test engineers not knowing what have been effectively tested. Another drawback is that if the generated test cases are not satisfactory (e.g. their execution time is too high) then there is no adequate means to limit the test case generation. In practice, it is performed by hand on the list of generated test cases, which is far from being optimal.

The problem that we are striving to solve in this thesis is the definition of a test selection approach that is capable of helping test engineers to better reach a delimited and verifiable test set with respect to some given test requirements and project specificities taking into account different test stakeholders.

1.4 Contribution

In this thesis, we present the SESAME approach for the selection of test in the context of model-driven engineering. The contributions of this thesis are the following ones:

- We introduce the notion of *model-driven test selection process*. Model-driven test selection processes are based on the two fundamental concepts of model-driven engineering, namely meta models and model transformations. We provide a software development process model (SESAME) that precisely defines, with the SPEM notation, a generic process for model-driven test selection processes. The main contribution of the SESAME process model is the coherent and systematic use of model transformations for test selection activities. We also identified a number of roles typically taking part of the specification of test requirements in the test selection context of SESAME. These roles are presented in a 5+2 View Model integrating multiple views.
- In model-driven test selection processes, test selection is defined on models having a precise meta model. We define a modeling language based on a subset of UML2 concepts restricted to the purpose of describing the System Under Test and its Test System as black-boxes. This small meta model is particularly well-suited to engineering teams with low-level of expertise in UML modeling.
- Model-driven test selection processes specify test selections with the help of a domain-specific test selection language. In this thesis, we define a domain-specific language (SERELA) targeted at the description of test selections.
- Production of a test model. The semantics of the test selection language is given in terms model transformations. In particular, model transformations enrich UML models with test selection constraints derived from test selection specifications. So that, models resulting from the test selection activity and input models share the exact same meta model.
- A first step toward an evaluation activity is proposed that uses metrics to measure the test selections. In the context of this activity, we define a set of metrics related to the semantics of UML models, which are computed with the help of formal analysis tool based on Boolean satisfiability (SAT solving technique).

1.5 Document organization

Chapter 2 introduces necessary background concepts related to the definition of model-driven test selection processes. It presents existing work related to features of the SESAME approach and presents the choices made in the SESAME approach with respect to the literature.

Chapter 3 describes the SESAME process model with the SPEM [OMG08] standardized notation. The description of SESAME is composed of three parts: work products required and produced during test selection; human roles involved in the performance of test selection; and lastly ordered set of activities to follow.

Chapter 4 describes the FOREST modeling notation that is used for the specification of analysis models. The abstract syntax is described as a meta model, and concrete syntax as a subset of UML [OMG07b] modeling notation. The semantics of FOREST is described as a model transformation into the Alloy [Jac06] formal language.

Chapter 5 describes SERELA, a domain-specific language that is focused on the specification of test selections on FOREST models. The syntax of SERELA is described with a set of BNF grammar rules. Its semantics is described in terms of model transformations from FOREST models into (constrained) FOREST models.

Chapter 6 describes a set of model-based metrics that aims at evaluating test selections by comparing metrics on its input and output models. The syntax if the metrics is described as a set of BNF grammar rules. The semantics of the metrics is given in terms of two languages depending on their nature. Metrics measuring structural attributes of FOREST models are given semantics in terms of KerMeta program. Metrics measuring attributes related to the semantics of FOREST models are given semantics in terms of Alloy expressions.

Chapter 7 illustrates the use of the SESAME approach with an industrial case study. The case study is a "visual occupancy classification system" that is based on a real project from the I.E.E. company with which I collaborated during the PhD project.

Chapter 8 concludes the dissertation by summarizing the SESAME approach and its current limitations. Chapter 9 presents potential extensions and/or improvements of the SESAME approach, as presented in this thesis.

A number of additional materials is also appended to this thesis. Appendix A gives the KerMeta program (i.e. model transformation) that implements FOREST semantics. Appendix B gives the KerMeta program that implements SERELA semantics. Appendix C and D show the semantics of two FOREST models (an initial model and a constrained model) within the context of the industrial case study of this thesis. Appendix E shows the Alloy semantics of the metrics defined within the industrial case study. Appendix F gives uses-cases used as input for the industrial case study. Appendix G gives the additional class diagrams of the static view of the industrial case study that were not included within the main body of Chapter 7.

2. BACKGROUND AND RELATED WORK

Abstract

In this thesis, we present the SESAME approach that is a model-based test selection process for small-sized safety-related embedded systems. This approach makes use of a number of existing concepts. In this chapter, we present the various general concepts used within the SESAME approach as well as the existing work related to these concepts. This chapter is structured as the remaining of the thesis: there are four sections for each of the following chapters 3, 4, 5 and 6. We start by presenting the concepts related to the definition of model-driven testing processes: model, model transformation, model-driven engineering and model-driven testing. Then we present, the existing work on analysis modeling language that match the requirements of the SESAME approach. The third section presents the existing test selection techniques. Finally, the concept and existing work on metrics for the test selection activity are presented.

2.1 Model-driven test selection processes

The SESAME approach, presented in this thesis, is a model-driven test selection process. In the following, we present the background and underlying concepts of the SESAME model-driven test selection process. Then we present some related approaches and conclude by summarizing the similarities and dissimilarities between the approaches and SESAME.

2.1.1 Background

Software engineering

Software engineering, as defined by Sommerville [Som04], is an engineering discipline that is concerned with all aspects of software development from the early stages of system specification to maintaining the system after it has gone into use. Software engineering faces three key challenges:

1. *Coping with increasing diversity.* In our context of testing small-size embedded systems, the diversity of software products may appear in different ways. There are numerous possible hardware platforms on which software may be embedded. The communication between small-size embedded systems is also heterogeneous. Thus software engineering techniques must be able to cope with this diversity of platforms and communication protocols.

2. *Demands for reduced delivery times.* Time is money, and in today's increasing concurrence between suppliers, the demands for reducing software development cost are increasing.

3. *Developing trustworthy software.* In our context of safety-related systems, faults in safety-related systems may endanger life or cause severe consequence on the environment, as such is paramount to be able to trust this kind of software systems.

Software development process model

The international standard ISO 12207 [ISO95], a reference for software practitioners, defines a common framework for the definition of software processes. In this standard, a *process*[1], is defined as "a set of interrelated activities, which transform inputs into outputs". More generally, a *life cycle model* is a framework containing the processes activities and tasks involved in the development, operation and maintenance of a software product, spanning the life of the system from the definition of its requirements to the termination of its use.

In this thesis, the term *software development process model*[2] will be used to refer to life cycle models covering activities and tasks of the development of a software product, while leaving out activities and tasks involved in the operation and maintenance of a software product. As an illustration, the model shown in Figure 2.1 is a software development process model giving an overview on specification, development and validation activities of the development of a software product.

The definition and use of software development process models is a first step to deal with the challenges aforementioned. Well-organized software development increase the trust level that software customers have in the software product delivered to them. The improvement of process models in place will help reducing delivery times, for instance, by increasing the automation of repetitive activities, or re-using work products throughout the development of different software products.

Typical software development processes comprise the activities of Analysis, Design, Implementation, Integration & Unit-level Testing and System-level Testing. Figure 2.1 shows when the SESAME process activities may be performed with respect to other activities of a typical software development process model.

The first activity in the process of developing software is the analysis of the requirements of the software product to be developed. During this phase the identification and description of *what* the system must do is performed. The software requirements may be based on discussion with customers, existing documents, etc. The analysis activity attempts on producing a complete and precise description, as a set of models, of the requirements. This activity results in a software requirements specification (SRS) document.

The design activity defines *how* the software behavior, specified in the SRS document, is obtained. The design activity consists in describing the architecture of the software product, i.e. the set of software units it comprises and how they inter-relate to each other.

The implementation activity constructs the software product based on the software design document. The result from this activity is code written in a programming language.

Integration & Unit-level testing activities aim at validating that the software code units and their integration match the specification of the software design document.

The system-level testing activity aims at validating that the software product as a whole (i.e. the software system) matches the specification of the software requirements specification. The

[1] In software engineering context, the term *software development process* is commonly used to refer to a process whose purpose is the development of software.

[2] We will also simply use the shorter term *process model* to refer to software development process model.

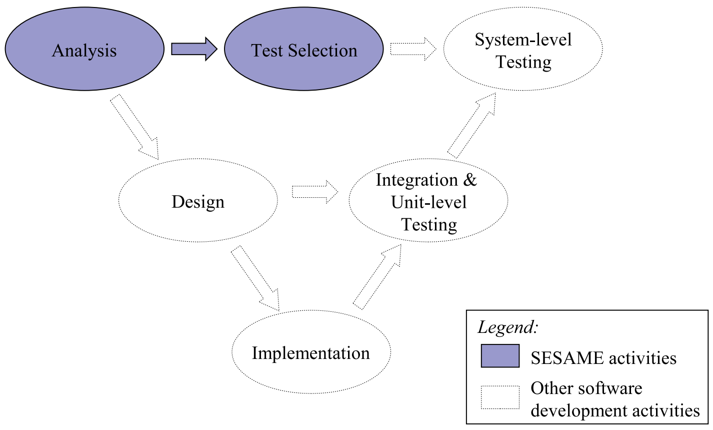


Fig. 2.1: SESAME activities within the V-model

SESAME process, defined in the following Chapter 3, focuses on the test selection-related sub-activities of the "System-level Testing" activity. That is why, in this figure, the "Test Selection" activity is separated from the "System-level Testing" activity.

Model-driven engineering

Engineering disciplines make extensive use of models, e.g. house construction plans are composed of a set of models of the house to be built (electrical models, plumbing models, masonry models, etc.).

Software engineering, as an engineering discipline, also makes use of models. In the software engineering community, the concepts of model have been standardized by the Object Management Group, as follows: "A *model* is an abstraction of the physical system with a certain purpose" [OMG07b]. The OMG formalized the concept of models [OMG07a] within a four layer hierarchy, as illustrated by Figure 2.2:

- M0 (the bottom layer) corresponds to the (concrete) elements of the system under study. As the system is software, concrete elements are logical elements present in the memory of the system running the software.
- M1 is the "user-model" layer. It comprises all the elements modeling the system under study specified by users. The UML concrete syntax is defined at this layer.
- M2 Meta-model [AK03, AK02, AK06]. The meta-model elements describe the kinds of elements which may be present in the models of layer M1. This layer allows formalizing user-models. The UML meta model (or abstract syntax) is described at this layer.
- M3 is the meta-meta-model layer. This layer is defined in the Meta-Object Facility (MOF) standard [ISO05]. MOF is designed to define modeling languages. Elements of the layer M3 describes the kinds of elements that may be used within the lower layer M2.

Models are particularly fitted to deal with the growing complexity and diversity today's software (one of the software engineering challenge presented earlier), because they represent abstractions of the software under development. Abstraction allows to focus on particular attributes of

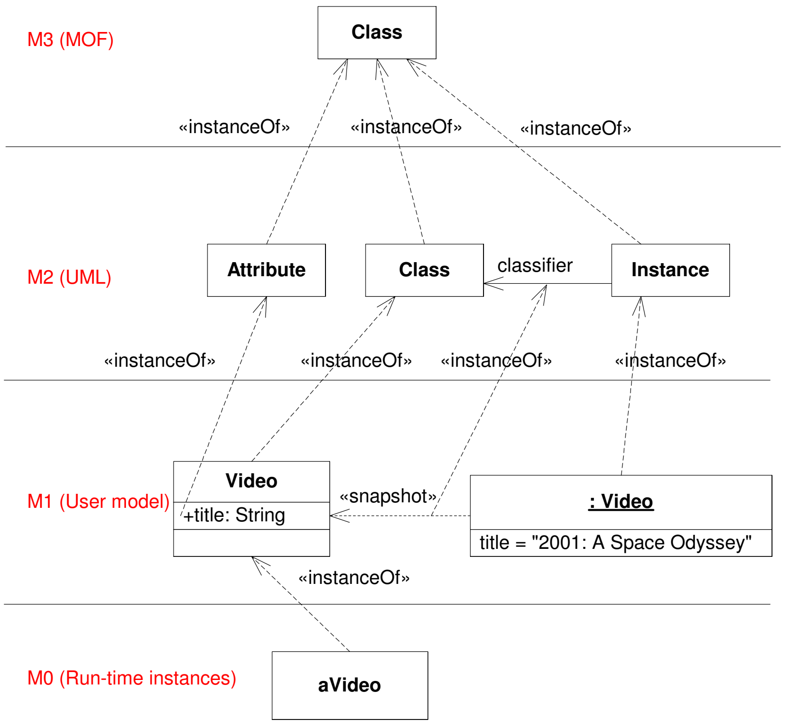


Fig. 2.2: Four layers of UML (from [OMG07a])

the software while ignoring some others. Thus the abstractions underlying models ease the understanding of software products. In turn, the understanding of software products results in trust and enables to cope with diversity.

Model-driven engineering [Ken02, SPHP02, Sch06, Bez04] is a particular subset of software engineering in which models are not only used as documentation but also as a building tool. They become the backbone of the software development process. Ideally, the software system would only be seen through its models. Each activity of the development process takes a number of input models and produces a number of output models. Thus, the process of building a software application can be seen as a sequence of model transformations which, in the end, yields to the final software product.

There are many initiatives that address model-driven engineering, one of the most developed being the OMG's Model Driven Architecture (MDA) [OMG03a, KWB03]. Models may only be first-class artifacts with adequate tool-support. The OMG's standardization efforts in the modeling domain provide an interesting platform for tool support and integration of model-driven technologies. The current relevant OMG industry standards are the Unified Modeling Language (UML) [OMG07a, OMG07b] defining a general-purpose notation for the specification of models within software engineering; and the standards related to UML, among others: the Meta Object Facility (MOF) [ISO05] defining the meta-meta-model used within UML and the XML Metadata Interchange (XMI) [OMG03b] defining a grammar for UML models in a XML textual format [W3C08].

Model transformations

Models specify the system, the application domain, and the requirements from different viewpoints and at different levels of abstraction. There are many interrelationships between these models. As mentioned before, each activity of a development process can be seen as a transformation of a number of input models into a number of output models. Model transformations are therefore a key concept in many emerging methodologies. But applying transformations by hand is a tiresome and error-prone process. Therefore, model transformations should be described in a specialized language, and these descriptions should be used by tools that automate the transformation.

Model transformations [CH03, GLR+02, Sei03, MJ02] can facilitate the reuse of test specifications by supporting a better separation of concerns. As an illustration, in the context of MDA [OMG03a] the system is described by two different models: the Platform Independent Model (PIM) and the Platform Specific Model (PSM). The test specifications are specified independently from the platform, i.e. at the PIM-level. Then, when the platform is known, the test specification specific to the platform may be generated, i.e. at the PSM-level, using model transformations. In the case of embedded systems development, it is common that customers have different implementation platforms; in that case test specifications may be re-generated automatically, for each different platform.

While model transformations are an interesting concept that can guide people in a development process, applying model transformations by hand is an error-prone and time-consuming process. In the last few years, a number of tools have supported model transformations, in order to ease the use of model transformations and increase the confidence levels in the specified transformations. Model transformations are an interesting topic for tool support. Currently there are a number of tools that support model transformations. The tools supporting model transformations are of different kinds [Fle06]. Fleurey identifies four general categories of tools supporting model transformations:

1. *Generic transformations tools* that may be used in particular to perform model transformations. As for instance, XML transformations tools, like XSLT [W3C99] that are designed to transform XML files. Thanks to the XMI [OMG03b] standard, UML models may be stored in XML format, thus enabling the use of such generic transformation tool like XSLT.

2. *CASE tools.* Computer-Aided Software Engineering (CASE) tools have for long provided ways of accessing their models either via proprietary scripting languages, such as the J language within the Objecteering [Obj] CASE tool or via an open interface such as the MagicDraw CASE tool [Mag] with its so-called OpenAPI written in Java.

3. *Model transformation tools.* Tools specifically designed for the transformation of models, like ATL [ABJK06].

4. *Meta-modeling tools* (e.g. KerMeta [MFV+05], MetaEdit+ [TR03]). In this kind of tools a model transformation is specified as a meta program.

In the SESAME approach, we define modeling languages with their associated meta models. We, thus, decide select a tool in the fourth category of tools supporting model transformations. We selected KerMeta because it is open-source (unlike MetaEdit+) and it is integrated in the Eclipse environment.

Viewpoint engineering

The development of software products requires several persons having different skills and viewing the system under development in their own way. The customer will not see the system in the same way than the supplier. The quality manager will be interested in particular attributes of the system and the software development team in some others. In deed, all these ways of seeing the software are not isolated, i.e. they may intersect in some ways.

Software engineering inevitably has to deal with several points of views on the system. In the early 90s, Finkelstein et al. [FKN$^+$92] defined a conceptual framework on Viewpoint-oriented systems engineering (VOSE). This framework was defined for system development and focused on dealing with the different viewpoints related to its specification. Then other research work [KS96, Kot99] have improved this first framework focusing on the requirements engineering phase.

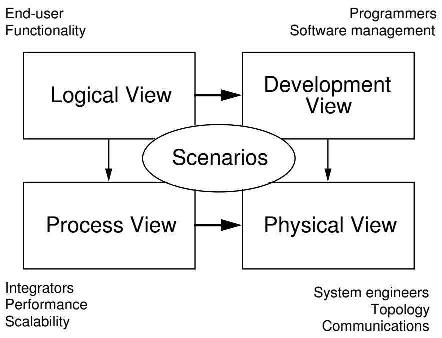


Fig. 2.3: 4+1 views on architecture (from [Kru95])

Another example of a viewpoint engineering approach is the 4+1 view model, defined by Kruchten [Kru95], where fives views are defined in the context of the specification of software architecture. Figure 2.3 illustrates the view model. The software architecture is defined through its four views: Logical, Development, Process, and Physical. The Scenarios view is a transversal view that overlaps the four views.

Model-driven testing

Software testing, as defined in the Software Engineering Body of Knowledge [Ber00, Tri05], consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution set against the expected behavior. Stuart Kent [Ken02] exhibited the benefits of model-driven engineering for software testing. We refer to *model-driven testing*[3] as software testing within the context of model-driven engineering.

Figure 2.4 details activities of a model-driven testing process model. The SESAME approach, presented in this thesis, focuses on the Test Selection Specification, Test Selection Evaluation and Test Generation activities. In the following, activities of the model-driven testing process model are described. The activities Test Set Concretization, Test Set Execution, Test Results Interpretation and Test Results Evaluation are not presented here, as they are out of the scope

[3] The term model-based testing [UPL06] is also used. In our opinion model-based testing is a more general way of seeing software testing with the help of models. The main difference lies in that model-based testing approaches do not require the use of automation, and consequently, may not use model transformations.

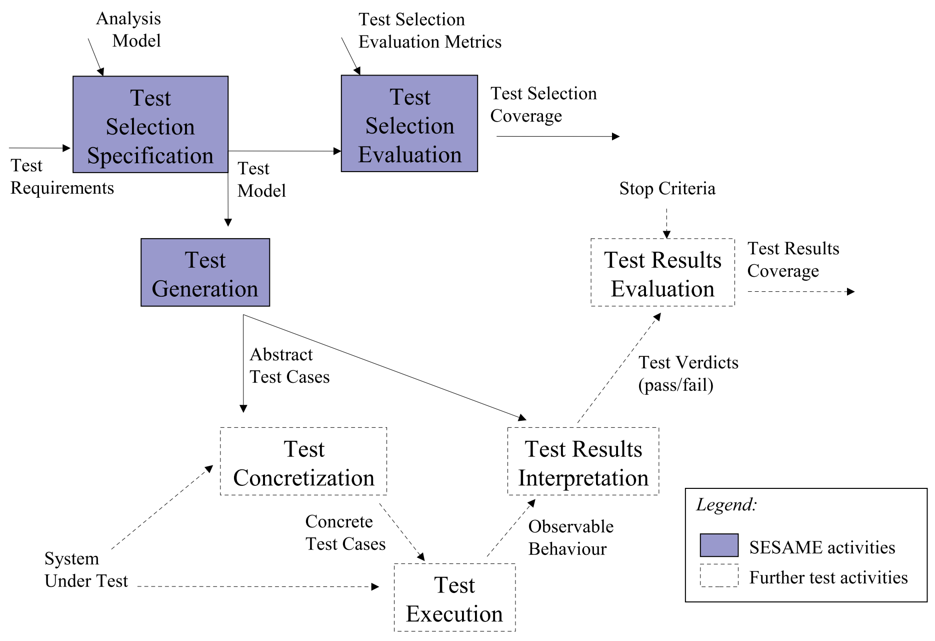


Fig. 2.4: SESAME test selection activities within a typical model-driven testing process

of the SESAME approach described in this thesis. Chapter 9 shortly presents these out-of-scope activities in the context of the perspectives of this work.

Test Selection Specification [GG75, Bar97] is the activity of specifying which behaviors of the system are going to be tested for a given test purpose. In this thesis, we define a test selection language (SERELA) providing a set of test selection instructions (described in the further Chapter 5). Test selection instructions can be seen as generic *test selection criteria* [Mye79, Mat08, BJK$^+$05] that must be instantiated. During the Test Selection Specification activity, domain experts select the test selection instructions of interest for the current test purpose and instantiate them depending on the Analysis Model of the system under test given as input of the activity, as shown in Figure 2.4. The test selection instructions may come from several different views (see the subsection Viewpoint-Engineering). Thus, the test selection specification is the result from the integration of the test selection instructions of the different views into a coherent set. When a consistent set of instructions is defined, a Test Model that complies with the specified test selection instructions is created.

Test Selection Evaluation is the activity that validate a given test selection. In order to validate a test selection, the Test Model resulting from the previous activity is evaluated given some metrics. In this thesis, we define a number of different metrics, described in Chapter 6, that help evaluating the effect of the test selection. The evaluation is performed by computing the values of the metrics on the Test Model and comparing them with the values of the metrics on the Analysis Model.

Test Generation is the activity that produces a set of test cases from the Test Model generated in the Test Selection Specification activity. The Test Model is parsed in order to create a set of abstract test cases [BJK$^+$05]. Abstract Test Cases is the set of possible simulations of the Test Model. As the Test Model is specified at the analysis-level, the test set is said to be abstract because it is not expressed with respect to the concrete element of the system under test but rather with abstract elements of the analysis model.

2.1.2 Related work

A large number of testing processes have been defined in the literature. In this section, we focus on three software testing processes based on UML models: an approach by Baker et al. based on the UML Testing Profile, an approach by Utting and Legeard which defines a model-based testing process based on models[4], and a test synthesis approach based on UML models by Pickin et al.

Model-driven testing using the UML testing profile

The UML Testing Profile (UTP) [OMG05], standardized by the OMG in 2005, provides concepts that target the pragmatic development of concise test specifications and test models for testing. In particular, the profile introduces concepts covering *test architecture*, *test behavior*, *test data* and *test time*. Together, these concepts define a modeling language for visualizing, specifying, analyzing, constructing and documenting the artifacts of a Test System.

The book [BDG+08] by Baker et al. describes systematic, model-based test processes in the context of UML. UML is used for specifying models of the system under test and the UML Testing Profile (UTP) standard [OMG05] is the formalism used to describe the Test System, i.e. the environment of the system under test. This approach is defined within a model-based testing context. This book describes the modeling of a wide range of testing artifacts with UTP.

Model-based testing with UTP is illustrated with a Library example, a set of UML diagrams describing the example are provided: a use case diagram to describe the functional requirements of the Library system; a number of class diagrams to describe the data related requirements; and a number of interaction diagrams to describe the behavioral requirements.

Black-box testing techniques are presented to derive test from the UML models. Three different levels of testing are presented: unit-level testing, integration-level testing and system-level testing. Separately from the black-box testing techniques, a so-called data-driven testing technique is presented that focuses solely on data-related testing. Other chapters present the following techniques out of the scope of this thesis: real-time and performance testing, user-interface testing, testing service-oriented architecture applications.

As a tool-support for the execution of the test cases, it is suggested two test languages: JUnit [JUn] and TTCN-3 [ETS03]. The methodology recommends defining a mapping from UTP models to JUnit when performing black-box testing at the unit-level and recommends defining a mapping from UTP to TTCN-3 [ZDSD05, Dai06] when performing black-box testing at the integration- and system-levels.

The underlying process presented in this approach is not formally described, i.e. the list of input and output work products required for each activity, the roles associated with the activities, etc. Five main activities are considered. Firstly, Test Design during which test cases are derived and/or selected. Secondly, the Test Specification during which concrete data is added to the test design and test procedure specifications. Thirdly, Test Validation is an activity that checks the correctness of the tests. Fourthly, Test Execution comprises all the steps, either manual or automated, necessary to perform the test on the system under test. Lastly, Test Result Analysis encompasses the comparison of expected versus received responses from the SUT along the tests.

[4] UML models being one of the models supported by the approach

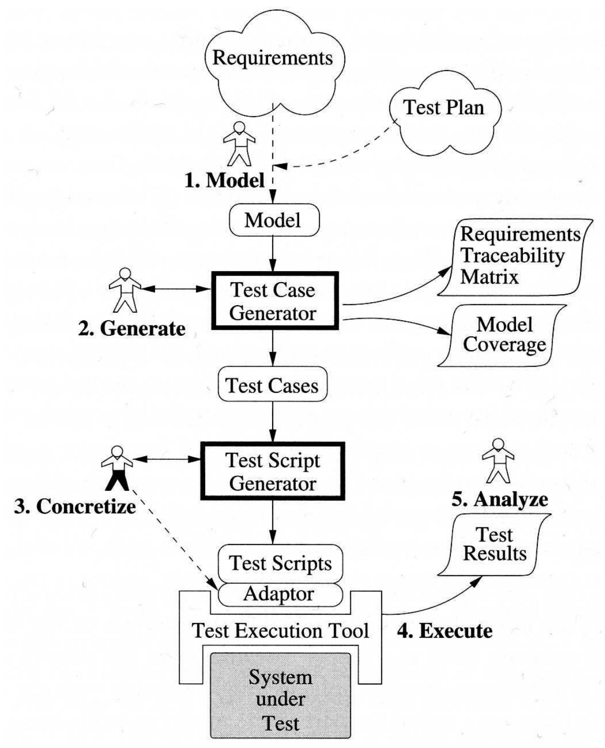


Fig. 2.5: Overview of the model-based testing process by Utting et al. (from [UL07])

Practical model-based testing

Utting and Legeard [UL07] present in their book a practical introduction to model-based testing showing how to write models for testing purposes and how to use model-based testing tools to generate test suites. It is focused on functional black-box testing. It presents black-box testing with two kinds of models: transition-based models and pre/post models [UPL06]. Examples illustrates model-based testing with the two kinds of models using UML state machines for the first kind and B formal specification models [Abr96] for the second kind. Their model-based testing process as illustrated in Figure 2.5, comprise five activities:

1. Model the SUT and/or its environment
2. Generate abstract tests from the model
3. Concretize the abstract tests to make them executable
4. Execute the tests on the SUT and assign verdicts
5. Analyze the tests results.

The generation of the abstract test cases is performed by a proprietary tool (Leirios [Lei] test generator) that works with the two kinds of models transition-based (LTG/UML) and pre/post models (LTG/B). The test generator takes as input: the model of the SUT and the model coverage criteria to be used. A set of model coverage criteria are defined (all-transitions, all-states, etc.) within the approach.

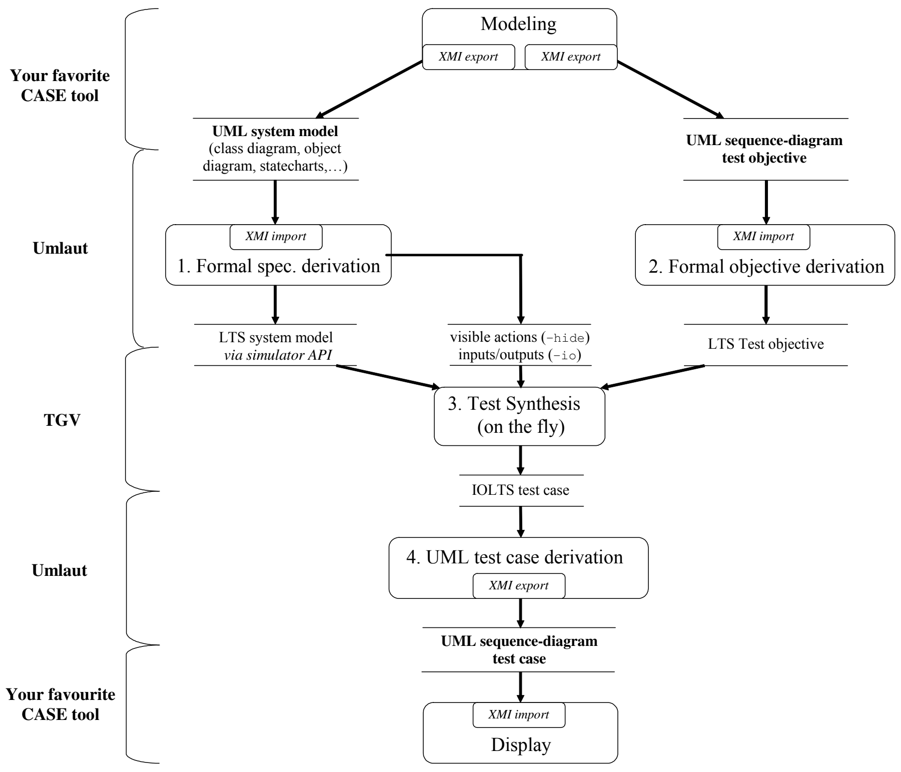

Your favorite CASE tool
Umlaut
TGV
Umlaut
Your favourite CASE tool
Modeling
XMI export
XMI export
UML system model (class diagram, object diagram, statecharts,...)
UML sequence-diagram test objective
XMI import
1. Formal spec. derivation
XMI import
2. Formal objective derivation
LTS system model via simulator API
visible actions (-hide) inputs/outputs (-io)
LTS Test objective
3. Test Synthesis (on the fly)
IOLTS test case
4. UML test case derivation
XMI export
UML sequence-diagram test case
XMI import
Display

Fig. 2.6: Test synthesis process overview by Pickin et al. (from [PJJ$^+$07])

Test synthesis by Pickin et al.

Pickin et al. [PJJ$^+$07] define a process that is targeted at the specification of test selection based on models. The process comprises four main activities, as illustrated in Figure 2.6:

1. *Formal specification derivation* consists in producing the semantic model in terms of labeled transition systems (LTS) from the system model written in UML.
2. *Formal objective derivation* consists in providing the semantic model in LTS from the test selection specification written in a language based on UML sequence diagrams.
3. *Test synthesis* consists in analyzing both formal models produced in the two previous activities in order to create a single formal model taking into account the test selection specification and the input model.
4. *UML test case derivation* consists in translating the formal test model (produced in activity 3.) into an "engineer-readable" model based on UML sequence diagrams.

2.1.3 Conclusion

The first related approach "Model-driven testing using UTP" [BDG$^+$08] by Baker et al. describes a model-driven testing approach at the system-level. The approach describes the derivation of the test model from UML requirements specifications by following a set of guidelines. An important difference with our approach is that our test model is obtained by model transformation of the analysis model and of the test selection model, and also we aim at providing a more concise and generic characterization of the test selection, via our test selection language. The main advantages of this approach are:

- The approach is defined within model-driven engineering
- A test model, in this approach, comprises on the one hand the SUT which is modeled with UML elements and on the other hand its related Test System (SUT environment) modeled with elements from the UML Testing Profile.
- The approach focuses on black-box testing; it comprises a test selection activity[5] and an evaluation of the test selection specification[6].

The main drawbacks of this approach, with respect to the context of our thesis, are:

- Model transformations are not used throughout the approach. The different steps are described informally.
- The Test Generation activity is not covered in the approach; as a consequence no attention is paid on the formalization of the models.
- Roles involved in the process are not precisely defined, in particular multiple stakeholders, as in our 5+2 View Model, are not considered.

The "Practical model-based testing" approach by Utting et al. [UL07] describe a comprehensive approach to model-based testing. This approach differs from ours mainly on the two following points. Firstly, while we suggest in our approach to derive the test model from the analysis model by model transformations, in this approach the test model is manually created either from some models of the SUT or from the SUT itself with the help of high-level guidelines. Secondly, there is neither a language nor modeling notations for the specification of the extensive set of test selection criteria given by the authors, resulting in separating the test model and the test selection criteria. The main advantages of this approach are:

- It is a comprehensive approach to model-based testing which comprises a test selection activity.
- The authors advocate the need for a test model[7]. The test model may describe the SUT and/or its environment.
- A tool is provided that generates the abstract test cases from the test model.

The main drawbacks of this approach, with respect to the context of our thesis, are:

- The approach does not define a dedicated activity for the evaluation of test selections.
- The test model is manually created from models of the SUT or from the SUT implementation itself with the help of guidelines.[8]

[5] called Test Design in the approach
[6] called Test Validation in the approach
[7] The notion of Test Model is different than the one that we introduce in this thesis.
[8] In this thesis, the Test Model is derived from an initial analysis model.

- Roles involved in the process are not precisely defined, in particular multiple stakeholders, as in our 5+2 View Model, are not considered.

The last approach explored in this section related to the SESAME approach, is the test synthesis process from Pickin et al. [PJJ+07]. The main advantages of this approach are:

- The approach is defined within model-driven engineering and based on formal grounds. As in the SESAME approach, model transformations are used to derive UML models (system model and test selection model) into formal models describing their semantics.
- User-models are specified with UML, e.g. class diagrams, statecharts, etc.
- Test selections[9] are explicitly specified with the UML notation, i.e. the test selection language of the approach is UML.
- The process is supported by tools a model transformation tool (UMLaut) and a test generation tool (TGV).

The main drawbacks of this approach, with respect to the context of our thesis, are:

- The test selection language is not specific to the test selection domain.
- The process is focused on the test generation but does not consider the evaluation of Test Models.
- Roles involved in the process are not precisely defined, in particular multiple stakeholders, as in our 5+2 View Model, are not considered.

[9] called Test Objectives in the approach.

2.2 Analysis modeling languages

In our model-driven engineering context, models are first-class artifacts of the software development process. In the SESAME approach, the black-box testing of small-size embedded systems is based on analysis models. In the following, we present the background and underlying concepts of analysis models of the SESAME approach. Then, we present the different UML diagrams and conclude by selecting a subset of diagrams fitting to our context.

2.2.1 Background

Analysis and design models for model-based testing

Models, in a model-driven engineering software development process, are used to deal with complexity at different levels of abstractions. We make the distinction between two kinds of models used within model-driven testing:

- *Analysis models* results from the clarification of the requirements of the system under test. The analysis model is specified based on the requirements expressed by the customer. As such, analysis models elements are problem-oriented and their level of abstraction is high. The model elements are independent from the design decision, and implementation details.
- *Design models* results from the decisions taken on how the system will implement[10] its requirements. Design models elements are solution-oriented and their level of abstraction is lower than for analysis models. The model elements are most likely closer to the technical details of the solution, e.g. its platform, and programming language.

In model-based testing approaches, there are two strategies of producing analysis and design models:

- By derivation of a higher-level model[11]. Analysis models for the purpose of testing may be derived from more general analysis models. If no general analysis models exist, then the derivation is not feasible and models should be produced from "scratch". As for design models, they should be derived from analysis models. The advantages from producing models by derivation of a higher-level model are numerous. Among other advantages, traceability between the specification at different level of abstraction and most importantly description that the models are earlier in the software development process. In particular, they are present before the implementation of the system under test.
- Abstraction of the SUT[12]. During this production strategy, the models are produced by analyzing the implementation of the SUT. The main drawback from this approach is that the models may be produced only after the SUT's implementation; resulting in a later description of the test specification than with the first strategy.

[10] in an abstract way

[11] This is in deed, the strategy promoted and required to perform model-driven engineering.

[12] Some model-based testing approaches like [PP04] accept this kind of technique. In model-driven testing approaches, like the SESAME approach described in this thesis, this kind of technique is discouraged. As models are first-class artifacts, the models should be present before the implementation of the system, thus this technique could not be used.

In our context of black-box testing, analysis models are particularly well-fitted to black-box testing, as they abstract the implementation details and focus on the specification of the software interface with its environment. That is why, in this thesis, we focus on model-driven testing based on analysis models as shown in Figure 2.1. The Test Selection activity, in the SESAME approach, is performed based on analysis models.

Data and behavior specification within analysis models

In general, many types of system properties may be included in analysis models (dynamic, static, and non-functional, etc.). In the context of small size reactive systems, analysis models are usually composed of a static view (also called structural specification) that specifies data and events related to the SUT and a dynamic view (also called behavioral specification) that specifies the legal order, and conditions, for sending events to the SUT.

The static view describes the information of the model that is independent from time. In particular it specifies data-related information. As analysis models are defined at a high-level of abstraction. Data defined in static views of analysis models are abstract data types. An *abstract data type* is "a set of data values and associated operations that are precisely specified independent of any particular implementation" [Bla05].

The dynamic view of analysis models describes how the system behaves over time. The dynamic view comprises the specification of the allowed order of received and sent messages between the system under test and its Test System. The dynamic view describes the behavior of the system at a high level of abstraction. The description of the messages exchanged is independent from the platform and implementation constraints of the system to be developed.

2.2.2 Selection of UML concepts for analysis modeling

Many languages may be used to specify analysis models. On the one hand, in our industrial context, it is particularly important to select a modeling language that is standardized and well-spread in the industry (i.e. that has mature tool-support). On the other hand, in the context of safety-related embedded software, it is important to have a language that is based on formal grounds. Modeling languages may be may have different degrees of formalizations (Statecharts[Har87], UML[OMG07b], Z[ISO02], etc.). Formal definition of the modeling language enables to have a precise understanding of the semantics of the specified models. In our test selection context, this is important so that the test selection is precisely delimited and that test cases may be generated from the models.

The Unified Modeling Language (UML) is a general-purpose graphical modeling notation allowing to model software systems at different levels of details (analysis, design and implementation). UML was originally created by Rational Software as a way to unify the various notations introduced by object-oriented methods (Booch [Boo94], OMT [RBP$^+$91], OOSE [JCJO92] and Harel's statecharts [Har87]). In the following, we explore the different concepts of UML and describe how a subset of these concepts is appropriate to the context of the SESAME approach.

Data modeling with UML

Structure diagrams show the static view of a system. The elements in a structure diagram represent the meaningful concepts of an application, and may include abstract, real-world and

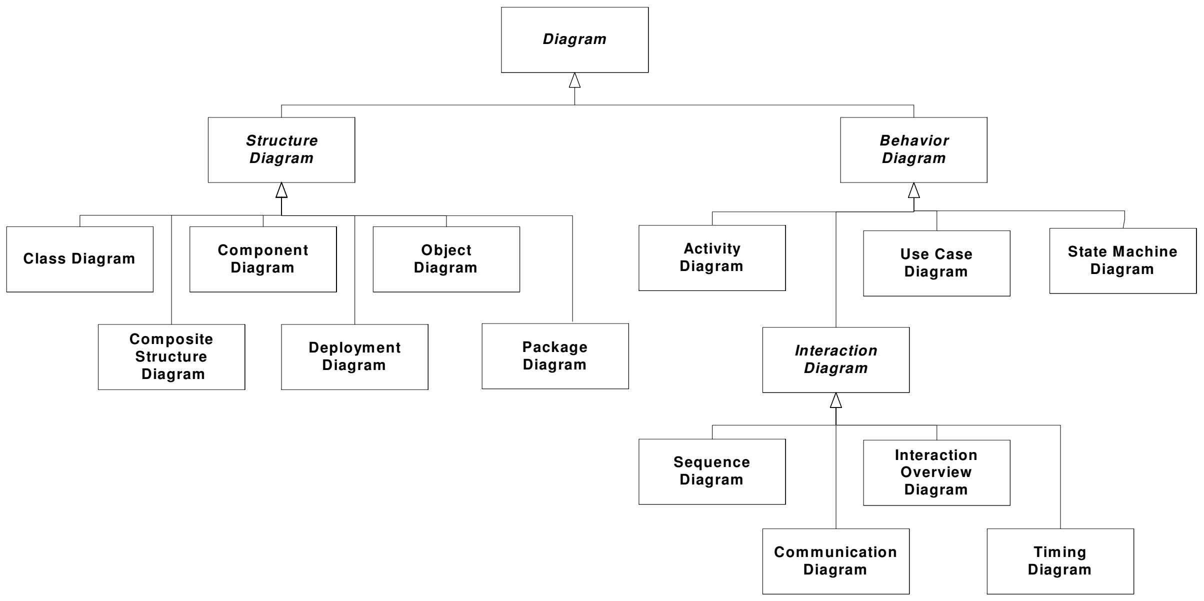


Fig. 2.7: Structure and behavior diagrams of UML2 (from [OMG07b])

implementation concepts[13]. Several variations of structure diagrams are suggested in the UML standard [OMG07b]: Class Diagram, Composite Structure Diagram, Component Diagram, Object Diagram and Package Diagram. Variations of structure diagrams often focus on particular structural aspects, such as relationships between packages, e.g. Package Diagram, showing instance specifications, e.g. Object Diagram, or relationships between classes, e.g. Class Diagram. There are no strict boundaries between different variations; it is possible to display any element you normally display in a structure diagram in any variation.

The UML elements of structure diagrams that we are particularly interested in for data modeling at the analysis-level are: Class, Association, Generalization, and Component. In the SESAME approach, data modeling is specified in a variation of a UML diagram that comprises Components for the specification of the two black-box components, Datatypes (a particular kind of Class) for the specification of abstract data types. Association and Generalization are to specify the relationships between the abstract data types. A detailed presentation of the UML elements selected for this variation of diagram can be found in the further Section 4.3.1.

The UML Testing profile [OMG05], presented earlier, allows the definition of test data by using UML Classes decorated with stereotypes. It defines two particular kinds of UML elements: data pools and data partitions. A *data partition* is a logical value for a parameter used in a stimulus or in an observation. It typically defines an equivalence class for a set of values. A *data pool* is a collection of data partitions or explicit values that may be used for multiple test purposes. A data pool provides a means for re-using data partitions and values for repeated tests. These two concepts are illustrated in Figure 2.8 with a Library example. In this Figure, a data pool ItemPool is defined that comprise two data partitions, named valid and invalid, of type ItemPartition. ItemPartition has three partitions extending it, BookPartition, CDPartition and DVDPartition.

[13] Analysis models focus on the representation of abstract concepts. Real-world and implementation concepts are represented in later activities of the software development process.

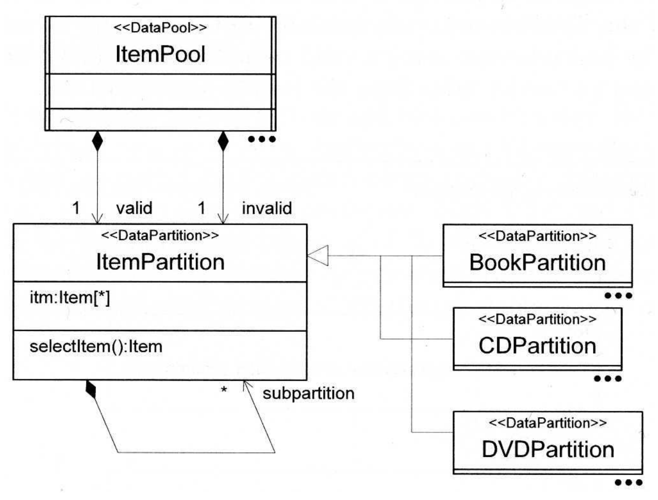


Fig. 2.8: Data modeling with the UML Testing Profile (from [BDG$^+$08])

Behavior modeling with UML

Behavior diagrams enable the specification of the dynamic view of a system. The dynamic view of a system can be described as a series of changes to the system over time. UML [OMG07b] classifies behavior diagrams into the following other kinds, as illustrated in Figure 2.7:

- *Activity Diagram.* Activity modeling emphasizes the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors. These are commonly called control flow and object flow models. The actions coordinated by activity models can be initiated because other actions finish executing, because objects and data become available, or because events occur external to the flow.
- *Interaction Diagram.* Interactions are used in a number of different situations. They are used to get a better grip of an interaction situation for an individual designer or for a group that needs to achieve a common understanding of the situation. Interactions are also used during the more detailed design phase where the precise inter-process communication must be set up according to formal protocols. When testing is performed, the traces of the system can be described as interactions and compared with those of the earlier phases. Interaction diagrams come in different variants: Sequence, Interaction Overview, Communication and Timing diagrams.
- *Use Case Diagram.* Use cases are a means for specifying required usages of a system. Typically, they are used to capture the requirements of a system, that is, what a system is supposed to do. Use Case Diagrams are a specialization of Class Diagrams such that the classifiers shown are restricted to being either Actors or Use Cases.
- *Behavioral State Machine Diagram.* State machines can be used to specify the internal behavior of various model elements.
- *Protocol State Machine Diagram.* Protocol state machines are used to express usage protocols. Protocol state machines express the legal transitions that a classifier can trigger. The state machine notation is a convenient way to define a lifecycle for objects, or an order of the invocation of its operation.

Activity diagrams are aimed at lower-level behaviors and thus do not match with our level of abstractions, i.e. analysis models. Use Case Diagrams specify what are the functionalities provided by the system, at a very high-level of abstraction (i.e. names of the functionalities), and the external actors that may use the functionalities. Interaction diagrams may be used for analysis models; they aim at specifying particular scenarios of the expected system behavior but are not suited for the specification of reactive behavior. Behavioral State Machine Diagram aims at describing white-box behavior of UML model elements, thus are not suited in our black-box approach.

Lastly, UML protocol state machines are very well-suited to the definition of the reactive behavior of the System Under Test reacting to the messages of the Test System. That is the reason why we are considering specifying the dynamic view of analysis models in our approach with protocol state machines.

2.2.3 Conclusion

In the previous sections, we have shortly presented UML and its numerous diagrams. We have selected two UML diagrams that match our goals. Firstly, UML class diagrams have been selected to specify abstract data types. UML Components are meant to specify components as black-boxes and thus perfectly fit the specification of the interface of the SUT and its Test System. Secondly, we have selected UML protocol state machines to specify the reactive behavior of the System Under Test at the analysis-level. Lastly, as UML is the industry de facto standard for modeling, plenty of CASE tools are supporting the UML notation[14] which is of paramount importance within our industrial context.

The syntax of UML is precisely defined by its meta model [OMG07a], but the main disadvantage of UML in the context of our thesis is that its semantics is (purposely) not formalized by the OMG. In order to deal with this known problem, we have decided to use two languages[15]:

- a modeling language for the analyst. This modeling language is going to be used by end-users, typically software analysts. This language, called FOREST in our approach and described in the further Chapter 4, comprises a subset of UML elements from protocol state machine and class diagram that fit to the context of specifying the System Under Test and its Test System as a black-box, both for the data and behavior specification and
- a formal language that will be used to define the semantics of the modeling language[16]. In the SESAME approach, formal analysis is performed during the two following tasks: "Test Selection Evaluation" and "Test Generation". As our approach is defined within model-driven engineering, the FOREST language will be defined in terms of an existing formal specification language using a generic model transformation. This formal specification language is intended to be used for formal analysis of the specifications and is going to be hidden from the end-users.

Thus, now that the modeling language for the analyst has been selected, the formal language that will be used to define its semantics still needs to be selected. In the following Section 2.5,

[14] In fact, as UML protocol state machines have only been recently introduced in the UML standard, only few tools support them for now.

[15] Our approach for the formalization of the subset of UML elements fitted to our approach is similar to the approach in Bordbar et al. [BA05].

[16] We will also refer to this language as the *semantics language*.

we present some related work on the formalization of UML class diagram and protocol state machine and conclude by giving the selected semantics language for the SESAME approach.

2.3 Model coverage criteria

Model coverage criteria [UL07] have been adopted from code coverage criteria [Mye79, Mat08] (e.g. branch coverage, path coverage, etc.) for which the level of abstraction have been raised form the SUT code to the model of the (expected) SUT. For instance, the *all-paths* code coverage criteria refers, in a typical implementation-based testing, to the measurement of whether all the paths of the SUT code may be exercised by a given set of test cases or not; in model-driven testing, the *all-paths* model coverage criteria refers to the coverage of a given set of test cases with respect to all the possible paths of the (behavior) model (e.g. with state machines, this would be all the sequences of state/transitions from the initial state to the final state). In model-driven testing approaches, model coverage criteria may be used for two purposes:

1. *Validate models.* Before generating test cases from analysis models, models must be validated. This validation evaluates if the model is ready for test generation, e.g. if it comprises a reasonable (i.e. not too large) amount of behavior and data to be exercised or if particular paths are present in the model. In that context, we call this kind of model coverage criteria, *model coverage metrics* as these criteria measure attributes of the models.
2. *Constrain test generation.* It is unlikely to be able to perform exhaustive test generation, because models usually comprise an overly large amount of behavior or data to be exercised. In order to cope with this, test generation must be constrained in order to generate a reasonable amount of test cases. Model coverage criteria may be used as constraints on the test generation. In this context, we call them *test selection instructions*.

Model coverage criteria are used within two activities of the SESAME approach, see previous Figure 2.1. Firstly, they are used during the Test Selection Specification activity, which aims at specifying test selection instructions for the further purpose of test generation. Then, they are used during the Test Selection Evaluation activity, which aims at evaluating a given test selection specification by computing the values of some model coverage metrics on analysis models.

As a general statement, model coverage criteria may be used for both purposes. When criteria are used as metrics they check if the model fulfills the criteria and when criteria are used as instructions they ensure that the criteria is fulfilled in the test model. But in some cases, model coverage criteria may not be used for both purposes. For instance, with the *all-states*[17] coverage criteria, it may be used without constraints as a metrics, but if some states of the models are unreachable, it can not be used as an instruction, as no instances of the model will fulfill the *all-states* coverage criteria.

In the following Sections 2.4 and 2.6, we present separately further concepts and some related work on model coverage criteria used to constrain test generation and used to validate models.

2.3.1 Categorization of model coverage criteria by Utting et al.

Utting and Legeard [UL07] define a large number of test selection criteria for models. Many of test selection criteria used in model-driven testing have been adopted from the research field on code coverage, i.e. white-box code coverage. Code coverage test criteria are well described in most text books [Mye79, Bin99, MS01, Mat08] on software testing. Utting and Legeard categorize test selection criteria for models into six categories:

[17] Depending on the behavioral diagrams chosen for the analysis model, this criteria may be interpreted differently. In our case, with protocol state machines, this criteria is fulfilled if all the protocol states are covered.

- *Structural model coverage criteria.* These criteria depend on the meta model describing the structure of the allowed model elements. McQuillan and Power provide an extensive survey of structural model coverage criteria for software testing [MP05] focused on UML models. They present for each type of UML diagrams, model coverage criteria available in the literature. In our context, the criteria concerning UML class diagrams and UML state machine diagrams are of particular interest.
  - Concerning UML class diagrams, Andrews et al. [AFGC03] define test adequacy criteria for UML design models. Their criteria are based on association-end multiplicity, generalization and class attribute UML model elements.
  - Concerning state machine diagrams, a number of work have been performed, among others the works by Offutt et al. [OA99] and Kim et al. [KHDC99]. Suggested criteria comprise: complete sequence coverage, all transitions coverage and full predicate coverage.
- *Data coverage criteria.* Data partitioning is a key technique for black-box testing. Data partitioning consists in splitting the definition domain of a data type in a number of partitions, i.e. disjoint subsets of values. The values contained in a partition are assumed to exercise the System Under Test in the same way, so that the testing of a data type boils down to the testing of one value for each partition.
  - Boundary-value analysis.
  - Equivalence partitioning.
- *Fault-model coverage criteria.* It is also interesting to mention some other test selection techniques [Mor90, LHSC03], coming from the fault-tolerance community, that are based on a fault model (instead of the system specifications in our case), which describes some of the important potential faults of a system. Based on these specified potential faults, a set of test cases is specified that exercises the SUT in order to exhibit failures resulting from the known faults. These techniques may be interesting to define reduction rules that are precisely targeting specific potential faults.
- *Requirements-based coverage criteria.*
- *Explicit test case specifications.*
- *Statistical test generation methods.* Random testing is one of these methods.

2.3.2 Conclusion

In our context, the SESAME approach particularly focuses on explicit test case specifications in order to take advantage of human expertise. Structural model coverage criteria and data coverage criteria are also used. In the SESAME approach, we use model coverage criteria for the purpose of constraining the test generation; this is performed with the help of a test selection language whose instructions are based on model coverage criteria. The SESAME approach also makes use of model coverage criteria for the purpose of validating models; this is performed via the definition of measurement of particular aspects of the models under validation. Concepts and related work for these two usages of model coverage criteria within SESAME are described in Sections 2.4 and 2.6.

2.4 Model-driven test selection approaches

In the following, the background and underlying concepts of test selection languages are presented. Then, we present some test selection approaches related to the SESAME approach defined in this thesis. Finally, we summarize what are the requirements fulfilled by the related approaches and the one which are not fulfilled.

2.4.1 Background

As presented in the introduction, exhaustive testing of behavior and data specified in analysis models is likely to be taking an overly large amount of time. That is why it is required to select a subset of the possible behavior and data of the system under test for the purpose of testing. There is several work that has been performed on selecting test based on UML models, we categorize the different kinds of test selection approaches in the literature, as follows:

1. *Test Generation.* These approaches narrow down the test selection activity as the generation of test cases. There is a unique test selection specification that is generic (i.e. not specific to the SUT) and unmodifiable. In this kind of approaches, there is a high risk that the test selection specification is hidden from engineers, and consequently that engineers are not aware of how test cases are produced. The two artifacts of the test selection activity in this kind of approaches are the input model and a set of abstract test cases, i.e. being the output of the test selection activity. The main advantage of these approaches is that they are quickly usable and do not require test expertise. In deed, they may be used in complement with model-driven test selection approaches. Their drawback is that they do not take advantage of human expertise, e.g. lessons learned, best practices, etc.

2. *Test Selection.* Test Selection approaches also include the activity of generating test cases. The main difference is that in these approaches, the test selection is recognized as a separate activity that requires human expertise. Compared to the test generation kind of approaches, an additional artifacts is required that is the specification of the test selection, enabling the possibility to specify a number of test selection specifications. Test selection specifications may be expressed at different modeling abstraction layers:

    - *Generic* test selection specifications. These test selection specifications are expressed at the level of the meta model elements (M2 in Figure 2.2). For instance, the selection of test cases that cover all transitions or all states of the behavior model are two possible generic test selection specifications.
    - *Project-specific* test selection specifications. Project-specific specifications are express at the level of the model elements (M1 in Figure 2.2). They aim at constraining the test generation to exercise some particular elements of the model.

    Test Selection approaches may offer different ways of specifying test selections:

    - *Predefined* model coverage criteria. In that case, the test engineer selects one (or more) criteria from a list of predefined model coverage criteria. The set of selected criteria form the test selection specification. The expressiveness of test selection specifications is restricted to the number of predefined criteria provided in the approach.
    - *User-defined* test selection specifications. The test selection is specified by the user with the help of a *test selection language*.

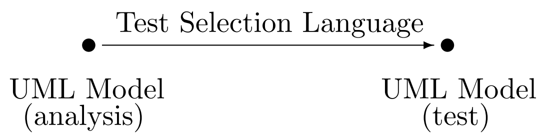


Fig. 2.9: Model-driven Test Selection: test selection via transformations of UML models

We define *Model-driven Test Selection* as Test Selection approaches that provide a language to specify user-defined test selection specifications in a model-driven engineering context. Model-driven Test Selection approaches are integrated within model-driven engineering and, as such, must provide test selection specifications in terms of model transformations; in particular the output of the test selection must be a model. In this thesis, we follow this type of approach, in particular the input and output models from the test selection are UML models, and the test selection language semantics is given in terms of a model transformations, as illustrated in Figure 2.9.

A number of "Test Generation" approaches have been defined in the literature on UML statecharts. These works, for instance by Bogdanov and Liuying [Bog00, LQ99], are mainly based on the work done on the test generation from finite-state machines [Cho78, FvBK$^+$91]. The purpose of the test generation is to have the smallest set of test cases still covering all the potential faults of the system with the minimal human interaction.

Test selection aims at containing the model complexity within reasonable limits. As described earlier, in our context we focus on two aspects of model complexity: data complexity and behavioral complexity. Thus the test selection approaches that we focus on, in the following, are the model-driven test selection approaches offering the selection of at least one of these two aspects.

2.4.2 Related work

Category-partition method and TSL

One of the first works describing the need for test selection was done by Goodenough and Gerhart in 1975 [GG75]. In this work, user-defined test selection specifications were expressed as decision tables. The test selection approach was focused on test data selection and did not address the issue of selection of behavioral aspects of the SUT specifications.

More recently, the Category-Partition Method [OB88] was proposed by Ostrand and Balcer that specifies a systematic approach to the definition of data partitioning. The method comes along with a Test Selection Language (TSL) that allows the systematic specification of categories and partitions in a textual format. The first step of the method is the identification of the attributes of the system that are interesting for testing purposes. The attributes are called categories in the method. Second step is the partition of the categories, which consists in enumerating the partitions for each category. Each partition is characterized by its name. A test specification is the set of identified categories and their related partitions.

In the third, fourth and fifth step, each step is focused on the expression of one type of constraints. The first type of constraint is if-expressions constraints, which are added to the test specification in order to specify incorrect combinations of values. An if-expression constraint is attached to a specific partition, if the constraint is evaluated to "true" then the partition may be selected.

These constraints may possibly use "and/or" Boolean operators. For instance a constraint specifying "The partition P1 of category C1 may be selected only if the partition P1 of category C2 is selected and partition P3 of category C3 is selected." can be expressed in TSL[18]. `[error]` and `[single]` keywords may also be attached to a partition, in order to further restrict the data complexity.

The last step of the method is the generation of abstract test cases, called "Test Frames" in the method, that are correct instances of the constrained test specification. This generation of Test Frames is automated but the semantics of TSL specifications is not described in the method.

Practical model-based testing

Utting and Leagard [UL07] define several model coverage criteria (shortly presented in the following Section 2.3.1). These criteria may be used to express generic test selection specifications[19] on UML models. The input modeling language is UML behavioral state machines for the behavioral specification and class diagrams for data specification of the SUT. The criteria proposed allow the specification of explicit test selections on both data and behavioral aspects. User-defined project-specific test selection specifications may also be described but no test selection language is provided for that purpose, i.e. neither its syntax nor its semantics are defined.

This approach is not integrated in model-driven engineering; the result of the test selection is not a model, it is a sequence of abstract test cases in the form of sequences of SUT methods calls.

UML Testing Profile (UTP)

UTP provides a notation based on UML to specify the Test System as well as some particular execution of the Test System, i.e. the test cases, based on the model of the SUT. The specification of data and behavior aspects of the test cases are expressed on separate models. UTP focuses on the description of project-specific and user-defined test selection specifications. These test selection specifications are described with UTP, so the input model and the test selection language have the same meta model. The test generation activity is not described, so the format of the output model is unknown.

The UML testing profile provides support for data-driven testing, as described in the standard [OMG05] and the book by Baker et al. [BDG$^+$08]. In the Analysis modeling section 2.2.1, the concepts used for data modeling with the UML Testing Profile are shortly presented: data pools and data partitions. *Wildcard* is another concept that may be used, as a complement to data pools and data partitions, to restrict data instantiation, i.e. the selection of data values. Wildcards allow specifying data attributes for which values are not restricted. Unrestricted values are seen as don't care values in this approach. UML Interaction Overview Diagrams are used to specify the test selection instructions related to the data partitions to exercise.

For the test selection of the SUT behavior, the abstract test cases are specified with interaction diagrams (in case of black-box testing) or activity diagrams (in case of white-box testing). There is no test selection language proposed for the selection of the behavior to exercise, the method defines a five steps guideline to derive from a SUT model the test case(s) matching the appropriate coverage criterion.

[18] The actual syntax of TSL does not give more insight on the method, that is why we do not present it here.
[19] they may also be used as metrics for testing, as mentioned by the authors.

SATEL, a test intention language for CO-OPN/2

SATEL [LPB04, LPB05, Luc08] is a language that express test intentions[20] on CO-OPN/2 formal specifications. CO-OPN/2 [Bib97] is a concurrent and object-oriented variant of Petri Nets. CO-OPN/2 specifications comprise data specification described with algebraic data types, and behavior specification specified with Petri nets.

SATEL test intentions aim at expressing constraints to extract data and behavior on the specifications of the (expected) SUT specification. SATEL is a formal language, as its syntax and semantics is defined in terms of formal languages.

The concrete syntax of SATEL test selection specifications are CO-OPN/2 modules. Modules are made of a number of test intentions. Test intentions are expressed as axioms of type "`[condition] => inclusion`"; whose meaning is: if `condition` is satisfied then the behavior described in `inclusion` is included in a test set. The `inclusion` behavior is expressed with temporal logic expressions in HML[21]. The abstract syntax of SATEL is formally defined with the set theory and its definition is integrated with the formal definition of CO-OPN/2. This allows SATEL specifications to make use of existing CO-OPN/2 specifications.

The semantics of SATEL is formally defined in terms of algebra. The semantics described allows the formal definition of the exhaustive test set fulfilling the test intentions specified on the SUT specification. The syntax of the set of generated test cases is given as HML temporal logic expressions.

Test selection with UML, IOLTS and TeLa

The test synthesis[22] approach by Pickin et al. [PJJ+07], described earlier, is a Model-driven Test Selection approach. It is model-driven in the sense that models are used as input and output of the test selection process, but it is not defined within the context of model-driven engineering. In particular, the different activities of the test synthesis process are not specified as model transformations.

Input models are described with the UML notation. The specification of the SUT and of the interface with its environment is made of a class diagram, an object diagram and a behavioral state machine. UML models are given a formal semantics in terms of labeled transition systems (LTS).

Test objectives[23] are expressed with UML sequence diagrams. A formal semantics of test objectives is given in terms of LTS. The formal semantics is implemented with the UML model simulator UMLAUT [HJGP99]. Two kinds of test objectives may be expressed: *accept* or *reject*. An accept test objective describes a scenario that is selected for testing and a reject test objective represents a scenario that is explicitly not selected for testing. Scenarios represent a way to select behavioral aspects of the SUT behavior. Within the scenario, the operation parameter calls may be abstracted by using the "*" wildcard symbol; allowing the selection (or not) of data aspects of the SUT behaviors.

[20] In our terminology, test intentions seem to be similar to test selection specifications. Thus, SATEL may be seen as a test selection language.
[21] HML stands for Hennessy-Milner Logic
[22] The term test synthesis used in this approach stress the fact that the approach is based on formal grounds and automated.
[23] Compatible with our concept of test selection specification.

The formal representation, i.e. LTS, of the test objectives and the input model are processed with the help of the TGV tool [JJ05]. The test synthesis process results in an input-output labeled transition system specification (IOLTS) which represent abstract test cases. These test cases are then translated in a user-oriented syntax that is more readable to engineer. The syntax used is the TeLa [PJH+01, PJ04] language based on UML sequence diagrams.

2.4.3 Conclusion

Among the test selection approaches that we have explored, the category-partition method, UTP, SATEL and Pickin's approach offer a test selection language. Utting's approach does not provide a test selection language, even though a number of criteria are specified that could be used as instructions of a language to be defined. UTP and Pickin's approach both provide a test selection language that is a subset of UML. SATEL's test selection language is based on algebraic specifications and temporal logic expressions.

The semantics of these test selection languages have different degrees of formality. The semantics of TSL, the test selection language of the category-partition method, is not precisely defined. The semantics is given as a tool implementation that translates TSL specifications into test cases. UTP does not define the semantics of its models. SATEL is mathematically defined with algebra. The semantics of the test selection language of the Pickin's approach is given as an implementation within the model transformation tool UMLAUT (precursor of KerMeta).

Input models of TSL are informal representations of the requirements. The input models of UTP, Utting and Picking's approaches are UML models. SATEL takes COOPN specifications as input models. Some work have been performed to provide a UML syntax to COOPN specifications [LPB04].

TSL does not produce an output model but directly a list of test cases in a textual form. Output models of the test selection activity in Pickin's approach are UML models. UTP and Utting's approach do not provide output model as the test selection language do not have semantics defined. SATEL's output model are expressed in terms of temporal logic expressions.

Out of the five approaches presented in this related work section, the approach of Pickin et al. [PJJ+07] is the only one which is a model-driven approach mostly fitting to our context, because it provides a test selection language which has a precise syntax; a semantics defined in terms of model transformations; lastly, input and output from the test selection are (subset of) UML models. The two main differences between Pickin's approach and the SESAME approach, defined in this thesis, are:

1. The test selection language, in Pickin's approach, is based on a general purpose modeling language (i.e. UML). In our approach, we define a domain-specific language, named SERELA, which is defined for the particular purpose of test selection specifications.

2. In the SESAME approach, input and output models of the test selection activity are expressed with the same meta model. Output models are described at a high level of abstraction, and represent analysis models of a view on the SUT for the purpose of test generation. In Pickin's approach, output models are described at a more concrete abstraction layer, and represent scenarios of the behaviors of the SUT to be exercised.

2.5 UML formalization

In the following, we present the background and underlying concepts of formalization of UML models for the purpose of formal analysis. We end this section by summarizing the related and presented the formalization approach selected in the context of the SESAME process.

2.5.1 Background

Formal languages are specification languages that have a formal syntax and semantics given in terms of some kinds of notation grounded on mathematics theory. So-called “formal methods”, i.e. methods based on formal languages, have had commercial success in the condition that formal methods experts were in place. Based on our experiences in the industry, we may say the transfer of formal methods to software engineers have not happened due to the mathematical background required for the definition of formal specifications and handling of formal proofs.

The notion of lightweight formal methods [JW96] rose from the intention to conciliate formal methods and software engineering. Soon after, Heitmeyer among other researchers has given precise recommendations on aspects of formal methods that should be improved for better integration within software development [Hei98], so that software engineers may use them easily. Recommendations of Heitmeyer include: make formal analysis as automatic as possible, integrated the method into the user’s development process, offer a language that software developer find easy to use and easy to understand, etc.

In the industrial context of this thesis, we do indeed agree with the need for introduction of lightweight formal methods in order to take advantage of existing formal techniques in an industrial context at a low cost. In order to generate abstract test cases and precisely evaluate test selections, semantics of constrained models resulting from the test selection activity must be well-defined. In our context, in order to benefit from formal analysis techniques, we formalize the subset of UML models of interest for our process, as described in Section 2.2. In particular, we need to formalize UML class diagrams and UML protocol state machines. Semantics of UML models is described informally in natural language [OMG07b]. Moreover, parts of the UML semantics are left intentionally incomplete, so that UML may be used by a large number of practitioners. Semantics variation points characterize the incomplete aspects of the UML standard. Consequently, it is necessary to decide what semantics to give to UML artifacts having semantic variation points *and* give to UML model elements of interest a semantics that can be systematically analyzed.

2.5.2 Related work

As the UML standard does not fully formalized its semantics, lots of research works have been held in the last ten years on different aspects of the definition of a (formal) semantics for UML. Part of this effort was conducted by the UML Precise Group (PUML) [FELR97, EFLR98] grouping researchers from the software engineering community. UML comprising a large number of diagram types, the research done has focused on some particular kinds of diagrams. Most work has been focusing on class diagrams and state machines.

Two formal languages supported by tools have received particular attention to be used as semantics languages of UML class diagrams and state machines for the purpose of test selection.

We mainly focus this section on these two languages (Z and Alloy), knowing that other formal languages have also been used for the formalization of UML models, e.g. B [LS02], PVS [Are99, Tra00], VDM [Jon86, LdBMB04], algebraic specifications [Pen01], Promela [LMM99a].

Formalization of statecharts

UML state machines are based on the original work of Harel [Har87, Har88] on the definition of the "statecharts" notation. The statecharts notation is based on state machines, as defined by Mealy [Mea55] in the 1950s. Harel et al. [HG97] adapt the statecharts notation to the object-oriented paradigm that will be taken as the basis for the UML's state diagrams standardization.

Semantics of Harel's statecharts have been thoroughly studied. The first version of the notation included an informal description of the statecharts semantics. Then, in 1996 [HN96], the semantics of statecharts as implemented in the STATEMATE CASE tool is formalized, it is an operational semantics defined with the help of algorithms. Damn et al. [DJHP98] and Gnesi et al. [GLM04], among numerous other works, propose their own formalizations of statecharts semantics based on Harel et al. [HN96]. The main drawback of these semantics definition is that they are not defined in terms of formal languages for which tool support is available.

The semantics defined by Lilius et al. [LP99b], in the context of the vUML verification tool, is defined in terms of the Promela language. Promela is the input language of the SPIN model checker [Hol97] and as such enables this semantics definition to be automatically analyzed with a formal tool support. Another semantics definition proposed by Latella et al. [LMM99b] is an operational semantics for a subset of UML 1.1 statecharts. Their approach is to start translating statecharts into extended hierarchical automatons (EHA) then to give the automatons a formal semantics as Promela specifications.

Formalization of UML models with CSP-OZ-DC

Z [ISO02, Spi92] is a formal language based on the set theory. Z is adapted, into Object-Z, to match object-orientation by Smith [Smi00]. Then, from Object-Z, works have been performed by Kim and Carrington [KC99, KC00] to formalize UML class diagrams with this object-oriented formal specification language. Later on, Miao et al. [MLL02] have extended the coverage of formalization of UML artifacts by formalizing class diagrams, sequence diagrams and state chart diagrams in Object-Z.

Recently, another variant of Z, built on Object-Z, named CSP-OZ-DC defined by Hoenicke et al. [HO02, Hoe06], extends Object-Z with the CSP [Hoa78] language (for behavior specification), and a Duration Calculus (DC) [ZH04] to express behavioral constraint. Allowed behavioral sequences of messages are specified in the CSP part of the CSP-OZ-DC classes. In the ObjectZ part of CSP-OZ-DC classes, the initial state of the software and the reactive behavior to the message receptions is specified. In the last part, duration calculus formulae allow the specification of timing constrains on the specified behavior. Duration calculus formulae in CSP-OZ-DC classes describe *undesired* behaviors in form of a linear trace.

CSP-OZ-DC is supported by a tool, named Syspect [Sys], based on the Eclipse [Eclb] development framework, which implements a UML Profile for CSP-OZ-DC defined by Linker [Lin05]. Unfortunately, behavioral and data constraints are specified, respectively, in DC and Z, which do not help the industrial transfer of this approach. Even though, it is allowed by the UML standard [OMG07b] to use DC and Z to specify constraints, in the standard no action language are *prescribed*, the standard only *suggests* using OCL [OMG06]. UML classes are called capsules within the defined UML profile and correspond to Object-Z classes. Each capsule is associated with at most one state machine. A state machine describes the CSP section of the CSP-OZ-DC class.

Formalization of UML models with Alloy

Alloy [Jac06] is a formal language based on relational logic. It has been defined to fulfill the requirements of lightweight formal methods [JW96]. Its syntax is precisely defined with BNF grammar rules and its semantics is given in terms of Boolean formulas. The resulting Boolean formulas are aimed to be fed to a formal tool, a SAT solver, that check satisfiability of propositional logic formulas. SAT solvers have gained maturity over the years and Alloy is (transparently) interfaced with a number of SAT solvers: BerkMin [GN07], MiniSat [NE03], ZChaff [MMZ$^+$01], SAT4J [SAT].

One of the strength of the Alloy language is its tool support. The Alloy Analyzer [JSS00, All] is a *model finder* for a user-defined specification written in Alloy, it provides the set of all possible instances satisfying the specification. The Alloy Analyzer is very well-suited to our needs of test generation and evaluation of models. One of the other strength is that its syntax is rather small, which ease the formalization of the UML artifacts in terms of Alloy expressions.

Recently, a few works have been performed to formalize UML into Alloy. UML class diagrams have been formalized in terms of Alloy [BA05] expressions in the context of the UML2ALLOY [Ana07] tool. The approach is somehow fitted to model-driven engineering as the formalization is based on a meta model of Alloy expressions described within the MagicDraw UML CASE tool. Mostefaoui and Vachon [MV07] formalize an aspect-oriented variant of UML, named Aspect-UML, in terms of Alloy. This approach is defined within model-driven engineering. Lastly, Kelsen et al. formalize a MOF-based language into Alloy [KM08a, KM08b]; their approach may be generalized to other languages defined with MOF, e.g. UML.

2.5.3 Conclusion

Research works have been performed that formalize UML class diagrams in terms of existing formal languages. There are also quite a number of works on the formalization of statecharts and early versions of UML state machines, but no work exist, as far as we know, on the formalization of UML protocol state machines. Thus no work exist also on the formalization of UML protocol state machines and class diagrams altogether.

CSP-OZ-DC provides a language that covers the formal specification of data and reactive behavior, the two main drawbacks, overcome by Alloy, is that firstly the notation is complex and secondly the approach is hardly lightweight due to its tool-support. That is why, in the SESAME approach defined in this thesis, we formalize a subset of UML class diagrams and protocol state machines in terms of Alloy.

2.6 Evaluation of UML models

In this section, we present the underlying concepts and related work on the use of model coverage criteria for the purpose of validations of models.

2.6.1 Background

Two principal techniques are available for the evaluation of (UML) models: *metrics computation* and *model-based verification*.

A typical software engineering technique for the evaluation of models is the computation of metrics on model elements. A *software metric*, as defined by Sommerville [Som04], is any type of measurement that relates to a software system, process or related documentation. In our model-driven engineering context, we focus on software metrics[24] related to the models of software systems. *Metrics computation* is a quantitative technique, i.e. it results in numerical values. Resulting values from metrics computations aim at grasping the complexity of the measured artifacts.

Model-based verification is a formal technique that analyzes models against expected properties. The expected properties may be expressed on static aspects of the model, i.e. on syntactical elements of the model, in that case properties may be for instance well-formedness rules. The expected properties may also be expressed on dynamic aspects of the model, i.e. verification of properties on the semantics of the model. Model checking [CGP99] is the principal formal analysis technique used for model-based verification. Model checking has gained lots of interest, not only in the formal methods community where it was originally defined, but also in the software engineering community thanks to its high degree of automation and numerous tool support, e.g. SPIN [Hol97], SMV [McM99], NuSMV [CCGR99], Java PathFinder [HP00], Bogor [RDH03].

2.6.2 Related work

Metrics for UML models

A large number of metrics for the evaluation of UML models have been defined since its first standardization, a decade ago. Most of these metrics have been defined based on the numerous work on metrics for object-oriented (OO) software, among others work by Chidamber et al. [CK94] and Fenton et al. [FP96].

Genero et al. [GPVCLR04] survey metrics for UML models, some of which are based on OO metrics adapted to UML. Surveyed metrics measure elements from UML use case diagrams, class diagrams and statechart diagrams.

Kim and Boldyreff [KB02] define a set of new metrics tailored to the specificities of UML, based on the elements defined in the UML meta model. This work is focused on metrics related to UML class and use case diagrams elements. Some direct metrics[25] are defined that measure for instance the number of attributes in a class, number of actors in a use-case, etc. Some other more elaborate metrics define, for instance the weighted number of attributes in a class. This

[24] As we only deal with software metrics in the context of this thesis, we will simply refer to them as *metrics* in the following.

[25] Direct metrics are not defined from other metrics.

is performed by assigning weight on attributes depending on their kind of visibility, i.e. high weight for public attributes, and low weight for private attributes.

The metrics presented in the last approaches [GPVCLR04, KB02] are mostly defined informally. Baroni et al. [BA02] contribute to the formalization effort of UML metrics. They define precisely a number of metrics on class diagrams elements with the OCL [OMG06] language.

The previous approaches present general-purpose metrics on UML model elements, these metrics may be used to evaluate UML models for any type of activities of software development processes. Some works have been done to define some metrics on UML models that are specifically useful for particular development phases. For instance, Baudry et al. [BT05] define metrics on UML class diagrams for the particular purpose of evaluating their testability. The metrics suggested count the number and the complexity of class interactions to be covered by test cases.

Recent work by McQuillan and Power [MP07] discusses a number of issues relating to (UML) model metrics. Nine observations are made on current work on model and metrics and possible future directions. One of the observations of the authors is that "differences between metric values are themselves metrics". This observation is particularly pertinent in the context of model-driven engineering. In particular, the authors illustrate it with the computation of the difference of the values of same metrics at the model-level and at the code-level.

Formal verification based on UML models

Lilius and Paltor [LP99a, LP99c] define vUML, a model-based verification approach using (transparently) the SPIN model-checker to verify properties of UML models. The benefit from this approach is that end-users deal only with UML models. The input models are made of UML class diagrams, statechart diagrams and collaboration diagrams. If the model verification is unsuccessful, the output model is a UML sequence diagram describing a sequence that causes the property not to be satisfied. The model verification activity, with vUML, consists in the verification of some predefined properties (e.g. deadlock, queue overflow, etc.) and three types of user-defined properties. User-defined properties are specified within the input model, either by labeling states with the specific stereotypes <<`invalid`>> or <<`progress`>> or by adding invariant constraints on the model.

USE [Ric02, GBR07] is an environment for the validation of UML models. It is composed of an animator for simulating UML class diagrams and an OCL interpreter for constraint checking. The USE environment was defined to check that given "run-time instances" of a model (at the M0 abstraction layer, see Figure 2.2) satisfy the constraint expressed on the user model specification (at the M1 abstraction layer). The USE environment may also be used to verify that given models (at M1 level) satisfies the constraints express on their meta model specification (at M2 level). The validation of a model instance is performed by animation. Originally, model instances were given by end-users as an explicit sequence of commands. Recently, an important improvement has been made on USE that extends it with a snapshot[26] generator [GBR05]. The snapshot generator generates automatically the set of all instances of a model that satisfy user-defined properties in the ad-hoc language ASSL[27] defined together with the snapshot generator. ASSL is a procedural language that constructs snapshots.

[26] A snapshot is an instance of a model in the USE terminology.
[27] standing for "A Snapshot Sequence Language"

2.6.3 Conclusion

As presented in this section, a number of metrics for UML models have been defined in the literature. These metrics solely focus on measuring elements of user models (at the M1 abstraction layer, as illustrated by Figure 2.2), i.e. they measure the *structural* complexity of UML models.

We have explored two lightweight formal approaches handling verification of properties on UML models. On the one hand, vUML provides the verification of properties on the dynamic aspects of the model of the SUT (i.e. statechart diagrams). Class diagrams are also used to define data used in statechart diagrams, but no means of verifying properties on data is provided. vUML provides the verification of general properties and very limited number of user-defined properties. On the other hand, USE only supports UML class diagrams to model the SUT and focuses on the verification of data properties of UML models. A large set of user-defined properties may be expressed thanks to the OCL language. But, USE does not provide support for the verification of behavioral properties.

In the context of testing, the *number* of elements to be exercised has a major impact on the cost of testing, in particular it may potentially increase its execution time. That is why it is crucial to measure the number of elements to exercise, through the computation of metrics. We call this type of metrics *semantical metrics* because they measure elements of the semantics of the model under evaluation.

In the SESAME approach, the formalization of the semantics of (a subset of) UML models enables the formal verification of properties similar to vUML and USE. As we formalize the static (i.e. data) and dynamic (i.e. behavioral) views of (a subset of) UML models. It is possible to verify both kinds of properties, i.e. static and dynamic. Most importantly, we define a set of semantical metrics that provide ways of using the result from the verification of properties to quantify the elements of the semantic model and thus evaluate the size of tests that may be derived from the semantics model. Neither the vUML approach nor the USE approach are defined in a testing context and thus do not allow the computation of metrics on the results of the formal evaluation of properties. In the SESAME approach, the computation of the static and dynamic semantical metrics is performed by the Alloy analyzer [JSS00, All]. We have chosen the Alloy analyzer in order to provide a consistent tool support for both test generation and model validation.

In our context of model-driven test selection, we make use of static and dynamic semantical metrics for the purpose of evaluating the test selection activity. Evaluation of test selections is performed by comparing values of semantical metrics on the input model and on the output model of the test selection under evaluation. For the purpose of evaluating test selections, property verification may be used by verifying properties on the constrained model[28] which are satisfied on the analysis model[29]. In this context, the verification result is interpreted as follows:

- A property that is not satisfied on the constrained model, implies that the test selection activity has discarded the data or behavior described in the property.
- A property that is satisfied on the constrained model, implies that the test selection activity has kept the data or behavior described in the property.

[28] The constrained model is the model that results from the test selection activity; we also simply call it "output model" when the context is unambiguous.

[29] The input model from the test selection activity.

By combining property verification and metrics computation, we provide a framework for the evaluation of test selections within a model-driven engineering environment. The computation of semantical metrics is used for the measurement of the evolution of the number of instances of some model elements between the input model and the output model of the test selection under evaluation.

3. THE SESAME TEST SELECTION PROCESS MODEL

Abstract

This chapter starts with a description of the ATM system that will be used in this thesis to illustrate the different concepts of the SESAME process model. Then, the SESAME test selection process model is defined with the help of the SPEM [OMG08] notation, by describing the different kinds of elements that it comprises. Firstly, the work products are presented. Secondly, who is responsible for these work products (roles of the process model) are introduced. A 5+2 View Model of Test Requirements that reflects various kinds of test requirements stakeholders is defined. Thirdly, the tasks performed, the roles that perform them, and their input and output work products are described. Lastly, to ease the industrial transfer of the process model, a tool chain is presented that supports tasks of the process model.

3.0 The ATM running example

An ATM is a computerized electronic machine that performs basic banking transactions (e.g. cash withdrawals, check deposits, etc.). In order to perform these banking transactions, ATM users must firstly authenticate by inserting a bankcard and then entering via a keyboard the *personal identification number* (PIN) associated with the inserted bankcard (Req.1). When ATM users are finished with their transactions, they may ask the ATM to restore their inserted bankcard (Req.2).

The *automated teller machine* (ATM) system, described in this section, has been chosen as simple as possible to ease the illustration of the concepts described throughout this thesis. The industrial applicability of the SESAME approach is not meant to be assessed by such a simple example. Chapter 7 addresses this issue by describing a detailed application of the SESAME process model within the industrial setting of the IEE company [IEE].

In the following, *ATM* (or also the ATM system) will refer to the particular embedded software of ATMs satisfying the requirements of the running example. The boundary of the ATM system, under study in this thesis, is the software that is embedded in the ATM managing the banking transactions. The environment of the ATM system is all the other (sub-) systems interacting with the ATM. The environment is composed of the *Card Reader Control Unit*, the *Keyboard Control Unit*, the *Cash Delivering Control Unit* and the *Remote Banking Server*. Test Systems of the ATM simulate events coming from the environment of the ATM, i.e. events coming from the four systems previously mentioned. The ATM embedded software must fulfill the following specific requirements, as well as the general ATM requirements described above in the first paragraph of this section:

Req.3 PINs are sequences of 4 digits.

Req.4 After three consecutive failed authentications, the system terminates without restoring the inserted bankcard.

Req.5 The ATM system provides one type of banking transactions: the issuing of cash withdrawals.

Req.6 The ATM system issues banknotes of 20€.

Req.7 ATM users' maximum acceptable request is 1000€.

Req.8 The ATM requires permission from a remote banking server for withdrawals of more than 500€.

3.1 Introduction

A *software process* is an ordered set of tasks contributing partially, or fully, to the development of a software product. During test selection activities of software processes, two physical systems are considered: the *system under test* (SUT) and the *Test System*. The *SUT* is the system that is to be exercised. A *Test System* is a system that simulates the SUT environment. The *SUT environment* is the union of all the systems exchanging (sending and/or receiving) one or more event with the SUT. During the test selection process, it is not required that any of the SUT nor the Test System be implemented yet.

Model-driven engineering places models as first-class entities throughout software development processes. Model-driven testing consists in deriving test cases from a model of the system and test requirements. The model describes the required input and output domain, i.e. the ordered set of all possible legal events that the system may receive and send during execution. The derived test cases are used to determine if the SUT conforms to the model. Depending on the software development phase of interest the testing artifacts, i.e. the SUT and the model, may be of different nature.

The SESAME process model focuses on system-level model-driven testing, during this phase the considered SUT is the whole system seen as a black-box and the considered model is the analysis model (output of the analysis phase of the development life cycle).

This thesis is focused on the system-level software testing activity during which the SUT is the software embedded in the system and the model is a test model. A test model is a view on the SUT for the purpose of testing. It represents the system requirements restricted to a test selection. The testing of the SUT is reduced to checking if the SUT conforms to the test model.

In this chapter, the SESAME software process model is defined that focuses on the test selection phase of the development of software products. A software process is a *SESAME process* if and only if it is an instance of the SESAME process model.

3.2 Work products

A *Work Product* is a piece of information that is produced, modified, or used by a *Task* of the process. In the context of model-driven engineering, a work product is likely to be represented as a model.

The SESAME test selection process model makes use of the following work products presented in the subsections below: *Customer Requirements*, *Analysis Model*, *Formal Analysis Specification*, *Test Selection Model*, *Test Model*, *Formal Test Specification*, *Model Coverage Metrics*, *Test Selection Coverage* and *Abstract Test Cases.* The *Customer Requirements* work product is out-of-scope of this process model, and is shortly introduced with its related work product: the *Analysis Model.*

3.2.1 Analysis model

There may be a number of analysis models produced during in a software process for different purpose. In the SESAME process model, we focus on one of these analysis models, that we name the *Analysis Model* work product. It represents the result from investigating on the customer requirements for testing purpose. The *Customer Requirements* work product is assumed to be existent and their format is not restricted, typically they are available in a contractual document, but may also be minutes from meetings between the customer and the provider. As presented earlier, analysis models comprise two views, a static view and a dynamic view.

The static view is described using the UML2 class diagram [OMG07b] metamodel. In the analysis model, the data definitions are focused on describing what the system must do at a conceptual level. The same applies for event definitions. For instance, an event in an analysis model may represent a set of operation calls in the implementation of the SUT.

The behavioral specification of reactive systems is typically modeled with state machines [HP85, Mea55]. In the SESAME approach, the behavioral specification is modeled with a variant of state machines recently standardized by the Object Management Group (OMG), namely UML2 protocol state machines (PSM). PSMs are particularly appropriate to system-level testing as "they present an external view of the class (in our case the SUT) that is exposed to its client" [OMG07b]. The events and data types used in the PSM correspond to the data types and events specified in the class diagram. Class diagrams together with protocol state machines provide a unified modeling framework for the analysis model.

Figure 3.1 illustrates a possible static view of the ATM example. Data types and event types of the SUT are specified in the static view. Figure 3.2 illustrates a possible dynamic view of the ATM example. In the dynamic view, the allowed behaviors of the SUT as well as the conditions for the SUT execution are specified in the PSM.

System operations are triggered by the Test System of the SUT with the goal of performing one of the system's functionalities. The triggering of the system operation is performed by the sending of an event (possibly with parameters) from the Test System to the SUT. In the following, we will equivalently refer to the system operation and its triggering event.

Figure 3.1 illustrates the class diagram of the ATM running example. Our SUT being the ATM, an analysis component is created and named "ATM". Based on the requirements, The ATM must react to five input signals from the Test System which results in the five system operations: insertCard, authenticate, withdraw, serverPermission and restoreCard. Three types are declared one for each of the parameter types, and one convenience type Digit is declared that is used

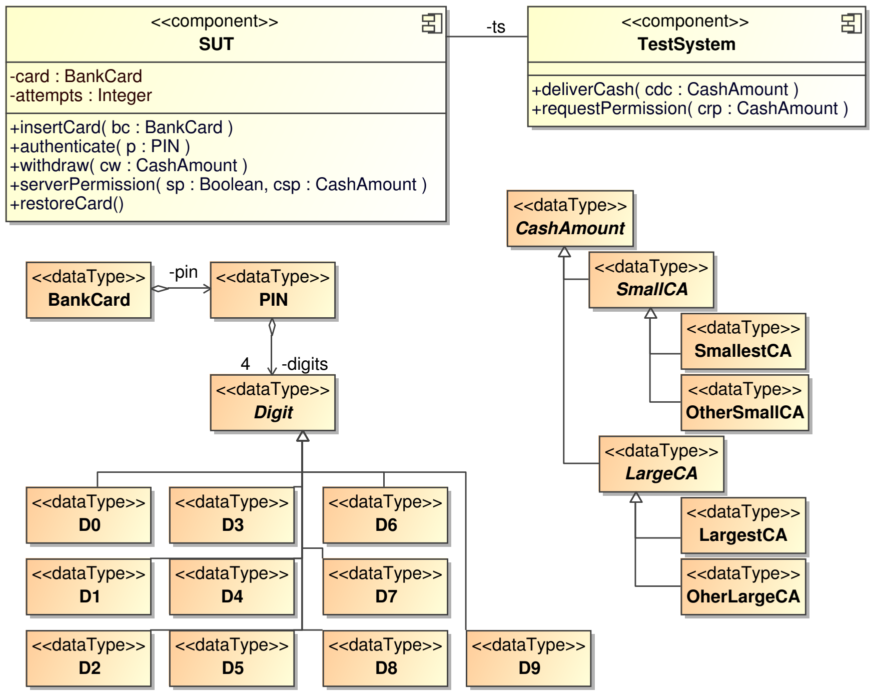
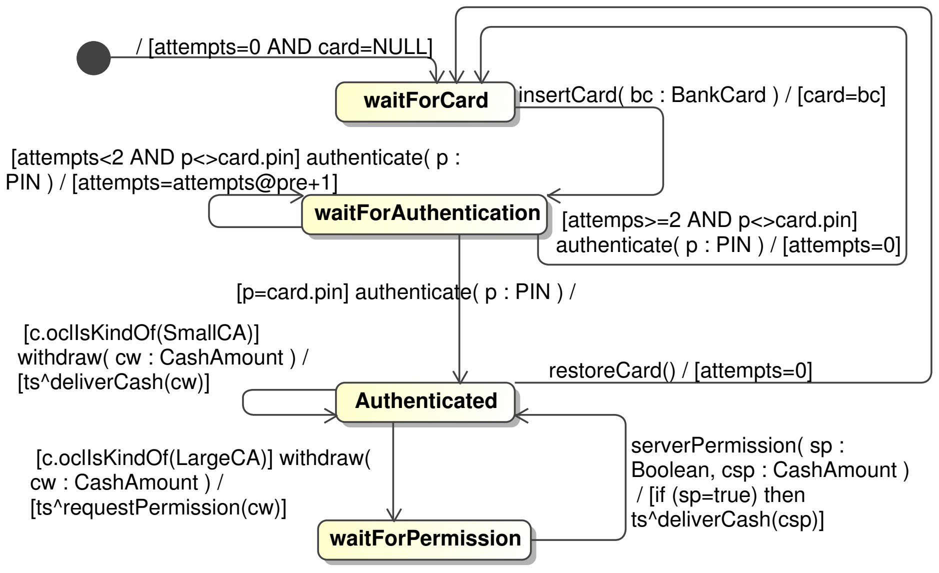


Fig. 3.1: ATM example: static view of the analysis model



/ [attempts=0 AND card=NULL]
waitForCard
insertCard(bc : BankCard) / [card=bc]
[attempts<2 AND p<>card.pin] authenticate(p : PIN) / [attempts=attempts@pre+1]
waitForAuthentication
[attemps>=2 AND p<>card.pin] authenticate(p : PIN) / [attempts=0]
[p=card.pin] authenticate(p : PIN) /
[c.oclIsKindOf(SmallCA)] withdraw(cw : CashAmount) / [ts^deliverCash(cw)]
restoreCard() / [attempts=0]
Authenticated
[c.oclIsKindOf(LargeCA)] withdraw(cw : CashAmount) / [ts^requestPermission(cw)]
serverPermission(sp : Boolean, csp : CashAmount) / [if (sp=true) then ts^deliverCash(csp)]
waitForPermission

Fig. 3.2: ATM example: dynamic view of the analysis model

to complete the definition of the PIN datatype. The requirements that the amount request is maximum 1000€ and that the user may request only amount multiple of 20, as the ATM only provides banknotes of 20€, is specified on the val attribute of the Amount class. A complete description of the FOREST language used to specify the static view is described in the further Chapter 4.

Figure 3.2 illustrates the protocol state machine diagram of the ATM example. Attributes and events of the ATM are directly accessible in the PSM. Attributes and events of the TestSystem component are accessible via the `ts` association between the `SUT` and `TestSystem` components. For instance, the following sequence of operation calls (a) is allowed by the PSM given that the pin number of the card c is 2121; but the sequence (b) is not allowed because authentication has not been performed before requesting the system operation withdraw. A complete description of the FOREST language used to specify the dynamic view is described in the further Chapter 4.

(a) `insertCard(c); authenticate(2121); restoreCard()`

(b) `insertCard(c); withdraw(20); restoreCard()`

Details on how to construct the static and dynamic views of the analysis model are presented in the description of *Task: Analyze Customer Requirements* in Section 3.4.2.

3.2.2 Formal analysis specification

The *Formal Analysis Specification* defines the SUT and a *Test System* that exercises exhaustively (and only) data and behaviors of the SUT specified in the analysis model.

In SESAME, a formal analysis specification is defined with an Alloy specification. The selection of Alloy for the formal analysis specifications is presented in the previous section, 2.5.

```
01 sig insertCard extends Event {
02 bc : BankCard
03 }
04 sig ATM extends SUT {
05 card : BankCard,
06 attempts : Int
07 }
```

Fig. 3.3: ATM example: extract of the static view of the formal analysis specification

The subset of the Alloy expressions that are used to formalize FOREST models in the context of SESAME is presented in the further section, 4.4.

Figure 3.3 shows an extract of the formalization of the Event withdraw (lines 01-03). In this specification line 01 declares the name of the event and that it is an event. Line 02 declares its only parameter named c of Datatype CashAmount. Lines 04-07 define the SUT and its state variables. Lines 05-06 declares the two state variables of the SUT.

Figure 3.4 shows an extract of the formalization of the precondition (lines 01-06) and postcondition (lines 07-10) of the self-transition of the state `WaitForAuthentication`. N.B.: this

```
01 pred preTR4(sut: ATM) {
02   sut.eventReceived in authenticate
03   sut.eventReceived.p != sut.card.pin
04   sut.attempts < Int[3]
05 }
06 pred postTR4(sut, sut': ATM) {
07   sut'.card = sut.card
08   sut'.attempts = sut.attempts + 1
09   no sut.signalSent
10 }
```

Fig. 3.4: ATM example: extract of the dynamic view of the formal analysis specification

transition is numbered and referred to as TR4 in the following. In Alloy, the pre- and postconditions of the UML protocol state machine are defined as predicates. Line 02 specifies that the SUT has received an event authenticate. Line 03 states that the parameter p of the received event authenticate is not equal to the PIN number of the BankCard which was inserted in the SUT earlier. Line 04 states that the number of authentication attempts must be lower than 3. The conjunction of these three lines forms the formal specification of the precondition of transition TR4. The formal specification of the postcondition of the transition TR4 is the conjunction of the three lines 07-09. Postconditions are specified based on the current state of the SUT and its next state. That is why the predicated postTR4 have two parameters. The parameter `sut` refers to the current state of the SUT and the parameter `sut'` refers to the next state of the SUT. Line 07 states that the card inserted in the SUT remains unchanged. Line 08 states that the number of attempts is incremented (because the PIN number given was a wrong number). Line 09 states that no signal should be sent by the SUT to its Test System.

A complete formal analysis model can be found in Appendix C that is defined within the industrial case study of this thesis, see Chapter 7. Its static view is formally defined in Section C.2 and its dynamic view is formally defined in Appendix C.3.

3.2.3 Test selection model

The *Test Selection Model* formally specifies an ordered set of constraints on the current analysis model. The *Test Selection Model* results from the collection of test requirements from multiple stakeholders and their interpretation on the current analysis model. The format of the test requirements is not restricted.

Test Selection Models have a textual syntax that is intended to be used by humans. The abstract syntax of SERELA test selection models is precisely defined with the SERELA meta model (shown in Figure 5.2) which is convenient for automating model transformations.

Figure 3.5 illustrates a possible specification of a test selection on the ATM example using the SERELA language. This test selection specifies that:

1 the initial state for this test selection is the `Authenticated` protocol state.

2 the bank card inserted in the ATM is always the one with a PIN of “0123” and false authentications are not exercised (i.e. attempts = 0).

```
1 initstate Authenticated ;
2 initvar { attempts = 0 and sut.card.pin[0].oclIsKindOf(d0) and
   sut.card.pin[1].oclIsKindOf(d1) and sut.card.pin[2].oclIsKindOf(d2) and
   sut.card.pin[3].oclIsKindOf(d3) } ;
3 finalstate Authenticated ;
4 repeat-event withdraw min 5 max 5 ;
5 repeat-trans tr5 min 1 ;
6 repeat-state waitForPermission min 1 ;
7 scope 10 ;
8 repeat-trans tr8 max 0 ;
```

Fig. 3.5: ATM example: SERELA textual syntax of a test selection model

3 there is a final state in the dynamic view and it may be accessed from the protocol state `Authenticated`

4 for each test case, the SUT must be exercised exactly five times with the `withdraw` event

5 the transition named `tr5` (i.e. the self-transition of the Authenticated protocol state) must be taken at least one time per test case

6 the protocol state `waitForPermission` must be active at least one time for each of the test cases.

7 each test case contains at most 10 event receptions.

8 the protocol transition `tr8` (triggered by the event `restoreCard`) is taken in no test case.

In this chapter, some more details can be found in Section 3.4.3 describing task: *Specify Test Selection* and a complete definition of the SERELA test selection language is presented in Chapter 5.

3.2.4 Test model

A *Test Model* is a view on the SUT for the purpose of testing that comprises information from a given test selection. It represents the system requirements restricted to a given test selection. In SESAME, the Test Model work product explicitly state the test cases selected on the Analysis Model work product using the same modeling notation. Test Models are intermediate models (used for validation of test selections and test generation); they are meant to be read-only.

In the same way than analysis models, test models are composed of a static and a dynamic view modeled using a UML2 class diagram and a protocol state machine. A test model comprises additionally to the analysis model, a set of constraints that are derived from the test selection model. These constraints are expressed in the Object Constraint Language (OCL) [OMG06] on the elements of the class diagram and on elements of the protocol state machine.

The static and dynamic views of the test model, illustrated in Figures 3.6 and 3.7, result from the application of the test selection (specified in Figure 3.5) applied to the analysis model of the ATM (Figures 3.1 and 3.2). We can notice that several test attributes have been added to the ATM analysis component and that these variables are used in the dynamic view to

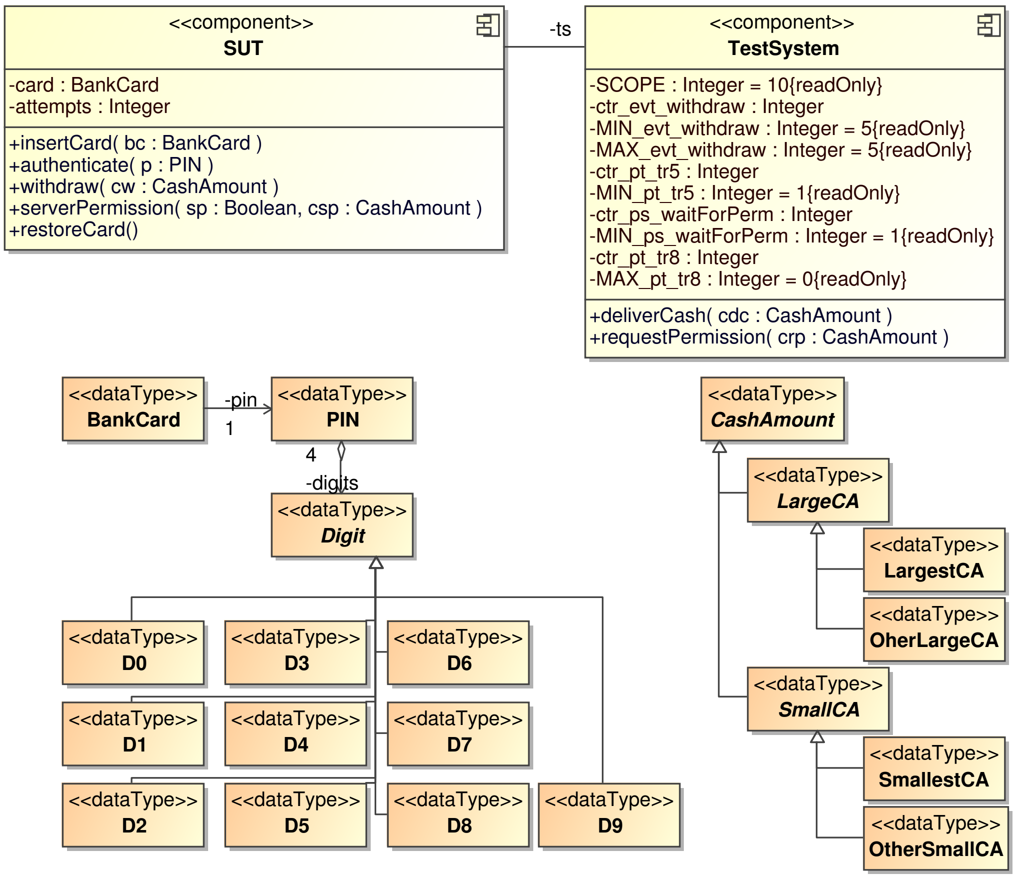


Fig. 3.6: ATM example: static view of the test model

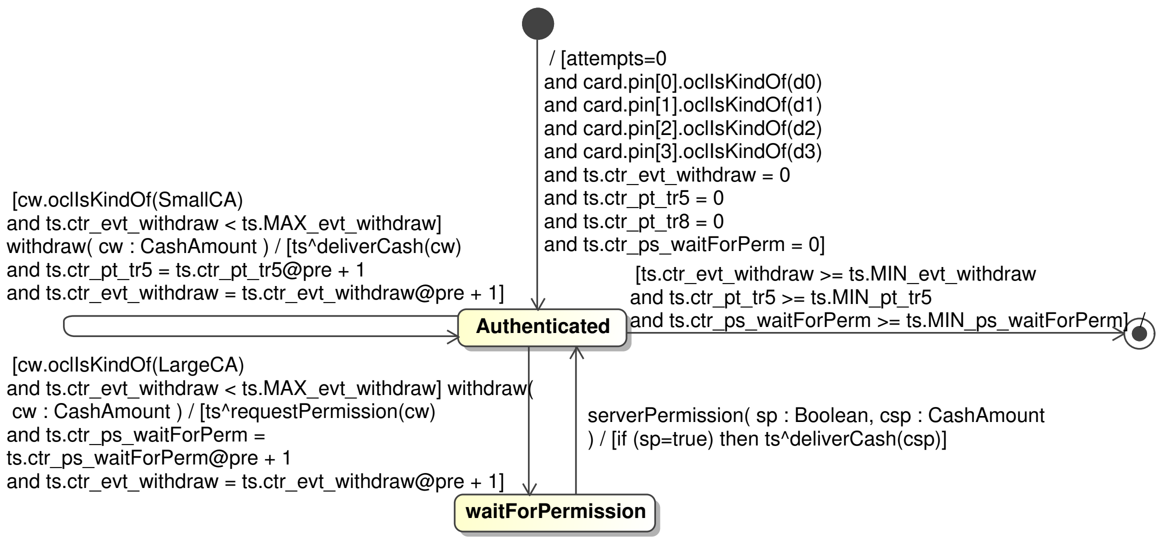

/ [attempts=0
and card.pin[0].oclIsKindOf(d0)
and card.pin[1].oclIsKindOf(d1)
and card.pin[2].oclIsKindOf(d2)
and card.pin[3].oclIsKindOf(d3)
and ts.ctr_evt_withdraw = 0
and ts.ctr_pt_tr5 = 0
and ts.ctr_pt_tr8 = 0
and ts.ctr_ps_waitForPerm = 0]
[cw.oclIsKindOf(SmallCA)
and ts.ctr_evt_withdraw < ts.MAX_evt_withdraw]
withdraw(cw : CashAmount) / [ts^deliverCash(cw)
and ts.ctr_pt_tr5 = ts.ctr_pt_tr5@pre + 1
and ts.ctr_evt_withdraw = ts.ctr_evt_withdraw@pre + 1]
[ts.ctr_evt_withdraw >= ts.MIN_evt_withdraw
and ts.ctr_pt_tr5 >= ts.MIN_pt_tr5
and ts.ctr_ps_waitForPerm >= ts.MIN_ps_waitForPerm]
Authenticated
[cw.oclIsKindOf(LargeCA)
and ts.ctr_evt_withdraw < ts.MAX_evt_withdraw] withdraw(
cw : CashAmount) / [ts^requestPermission(cw)
and ts.ctr_ps_waitForPerm =
ts.ctr_ps_waitForPerm@pre + 1
and ts.ctr_evt_withdraw = ts.ctr_evt_withdraw@pre + 1]
serverPermission(sp : Boolean, csp : CashAmount
) / [if (sp=true) then ts^deliverCash(csp)]
waitForPermission

Fig. 3.7: ATM example: dynamic view of the test model

restrict the execution of the SUT. In particular, the restriction of a maximum of two authentications (line 02 of Figure 5.3) results in the addition of the two variables `ctrOP2` and `MAXop2` to the ATM component. These variables are used in the three transitions triggered by the `authenticate` event (the self transition of the state `waitForAuthentication`, the transition from the state `waitForAuthentication` to `Authenticated` and the transition from the state `waitForAuthentication` to the final state), as described in Section 3.4.3 on the *Task: Specify Test Selection.*

3.2.5 Formal test specification

A *Formal Test Specification* represents the SUT and a *Test System* that exercises exhaustively (and only) data and behaviors of the SUT specified in the test model.

```
01 sig insertCard extends Event {
02   bc : BankCard
03 } {
04   bc.pin[0] = 0
05   bc.pin[1] = 1
06   bc.pin[2] = 2
07   bc.pin[3] = 3
08 }
```

```
09 sig ATM extends SUT {
10   card : BankCard,
11   attempts : Int,
12   ctrOP2 : Int,
13   MAXop2 : Int,
14   ctrTR5 : Int,
15   MINtr5 : Int,
16   MAXtr5 : Int,
17   ctrS4 : Int,
18   MINs4 : Int,
19   MAXs4 : Int
20 }
```

Fig. 3.8: ATM example: extract of the static view of the formal test specification

Figure 3.8 shows an extract of the formalization of the Event withdraw (lines 01-08) and the declaration of the SUT, its state variables (lines 10-11) and its test variables (lines 12-19). This formal representation of the Test Model in Alloy takes into account the test selection specification declared in Figure 3.5. Lines 04-07 are not in the formal analysis specification of Figure 3.3. These lines correspond to the test selection instruction that the card inserted in the ATM for testing will always have a PIN number of “0123” (more precisely, the specification states that the first digit is 1, the second digit is 2, etc.). Lines 12-19 are not in the formal analysis specification of Figure 3.3 either. In deed, these lines correspond to the declaration of the test variables introduced to count the amount of times operation withdraw has been tested (line 12), the self-transition of state Authenticated (transition TR5) has been taken (line 17) and the state waitForPermission (state S4) has been reached. Also the allowed maximum and minimum values of these counters are declared with respect to the test selection specification given in Figure 3.5.

Figure 3.9 shows an extract of the formalization of the pre-/post-conditions of the self-transition (TR4) of state waitForAuthentication. Line 05 and line 11 are not in the formal analysis specification of Figure 3.4. Line 05 corresponds to the test constraint that the maximum amount of withdraw is two (MAXop2 is a constant equals to 2 specified elsewhere in the formal specification). Line 11 corresponds to the incrementation of the counter that keeps track of the number of withdraw event received so far by the SUT from the Test System.

A complete formal semantics of a constrained model can be found in Appendix D that is defined

```
pred preTR4(sut: ATM) {
  sut.eventReceived in authenticate
  sut.eventReceived.p != sut.card.pin
  sut.attempts < Int[3]
  sut.ctrOP2 < MAXop2
}
pred postTR4(sut, sut’: ATM) {
  sut’.card = sut.card
  sut’.attempts = sut.attempts + 1
  no sut.signalSent
  sut’.ctrOP2 = sut.ctrOP2 + 1
}
```

Fig. 3.9: ATM example: extract of the dynamic view of the formal test specification

within the industrial case study of this thesis, see Chapter 7. Its static view is formally defined in Section D.2 and its dynamic view is formally defined in Appendix D.3.

3.2.6 Model coverage metrics

Model coverage metrics are functions that measure some attributes of some elements of a given model. In the SESAME approach, we opt to measure the process of test selection by comparing values of same model coverage metrics computed on the initial analysis model and values computed on the resulting test model from the test selection specification. The *Work Product: Test Selection Coverage* is the result from this comparison. The computation of these metrics is particularly useful, because they provide some hints on the complexity of the selected tests before their execution.

Metrics are used in order to evaluate test selections before the test cases are generated. The SPEM standard [OMG08] defines an Outcome as "*a Work Product that provides a description and definition for non-tangible work products. An outcome describes intangible work products that are a result or state*". In the context of the SESAME Process Model, the *Outcome: Test Selection Satisfaction* describes a result from the informal analysis of the resulting values of the *Work Product: Metrics.* Given informal test requirements and depending on the computed values of the test selection coverage the *Outcome: Test Selection Satisfaction* may be positive or negative.

The three following semantical metrics[1] defined in SESAME are illustrated in Table 3.1. The first and second column present the results of the computation of the three model coverage metrics on the initial analysis model and on the test model (resulting from the test selection specification in Figure 3.5, last column presents the Test Selection Coverage computed from the difference of the values of the initial model and test model.

`CoveredTransitions` metrics measure the number of partially covered protocol transitions[2] of a given FOREST model. `CoveredStates` and `CoveredEvents` metrics are similarly defined on protocol states and event receptions of a given model.

[1] Chapter 6 precisely defines these metrics as well as a number of other metrics in the context of the SESAME process model.

[2] The notion of *coverage*, and in particular of *partial coverage*, is introduced in Chapter 6. For short, a protocol transition is partially covered if at least one of its instance is in the semantics of the model under evaluation.

Metrics	**Initial Model**	**Test Model**	**Test Selection Coverage**
CoveredStates	4	3	75%
CoveredEvents	7	4	57%
CoveredTransitions	8	4	50%

Tab. 3.1: ATM example - Test Selection Coverage

As a first remark, the values of the model coverage metrics are useful indicators for the evaluation of the specification of the Initial analysis model. It is expected, but not necessary, that the number of partially covered events, state and transitions of the protocol state machine equals to the number of structural elements in the model. It is the case with the ATM example, thus we may conclude that there is no dead state, nor unused events, nor dead transitions in the initial analysis model.

The model coverage metrics values on the Test Model inform that three protocol states, four protocol transitions and four event receptions are partially covered. This is indeed, due to the test selection specification that restricted behaviors in the Initial model. For in the Test Selection Model, it is specified that there is a maximum of two events `authenticate` received by the SUT; so there will never be three false authentications, so precondition of the transition from state `waitForAuthentication` to the final state will not be true and the transition will never be taken. This results in one transition not being covered.

Another work product is defined in the context of the Model Coverage Metrics: *Work Product: Metrics Library*. The *Metrics Library* is an existing pool of metrics defined within the company prior to running the SESAME process. The Metrics Library is intended to contain the commonly used metrics of a company or a team. The Metrics Library contains at least the set of structural and semantical metrics defined in Sections 6.2 and 6.3 that define various kinds of measurements on FOREST model elements. The Metrics Library may also be enriched with business-level metrics derived from the SESAME structural and semantical metrics. Some business-level metrics are shortly presented in Section 6.5.

3.2.7 Abstract test cases

An *Abstract Test Case* is a sequence of SUT events reception that brings the SUT from the initial state of its protocol state machine to the final state of the protocol state machine following a legal path as defined in the protocol state machine of the Test Model.

A test case is a sequence of events sent to the SUT with the expected events responses from the SUT.

The actual representation of the Abstract Test Cases is not formally defined in the SESAME process model. The representation should be adapted to the tool-support implementation of the task ”Concretize Test Cases” performed after the task Generate Abstract Test Cases. The task ”Concretize Test Cases” is outside the scope of this thesis.

3.3 Roles

The SESAME process defines a number of roles. These *Role Definitions* are abstractions of concrete human beings who will be part of SESAME processes. When implementing a SESAME process for a particular SUT, the role definitions are associated to humans, with SPEM Role Uses, related to the development of the SUT.

3.3.1 Analyst

The *Analyst* is responsible for the *Analysis Model* work product. Analysts must be able to have a global understanding of customer requirements. Analysts must, in deed, have a good working knowledge of UML in order to model the class diagram and the protocol state machines required in the analysis model.

3.3.2 5+2 View Model on Test Requirements

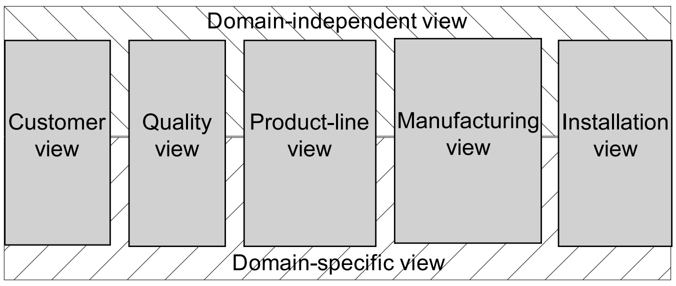


Fig. 3.10: 5+2 View Model on Test Requirements

Test selection activities are based on test requirements coming from a number of stakeholders. In the context of our industrial collaboration with the IEE company, we have identified five views on test requirements specification. In the SESAME approach, we propose to use a 5+2 View Model of Test Requirements, based on these five viewpoints, for the elicitation of which aspects of the SUT must be exercised. The five views defined of our View Model are illustrated in Figure 3.10.

All of the five views aim at specifying test requirements; therefore, they share the same role. The Test Requirement Stakeholder Role Definition, illustrated in Figure 3.11, represents the role of the five views as well as the definition of a role for each of the five views. In the following, we shortly describe the views of the 5+2 View Model and also relate the views with roles of the process:

- *Customer View.* Customer View represents the properties of the SUT that must be exercised from the *Customer*'s point of view. During the elicitation of functional requirements of the system to be developed, the Customer often expresses a set of required tests that the software product must pass.
- *Quality View.* In companies, quality departments have a set of standard technique in order to measure the quality of a software product. In the automotive industry, *Quality Engineers* analyzes potential failures of the system with Failure Modes and Effects Analysis

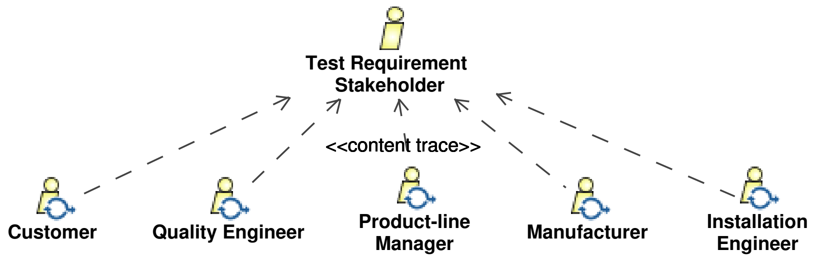


Fig. 3.11: Role: Test Requirements Stakeholders in the 5+2 View Model

(FMEA) [iec85] documents. These failures modes may be used for the specification of some test selections.

- *Product-line View.* In the context of embedded software companies, developed software products share commonalties. It is thus valuable to consider software development within a product-line approach [CN01]. The *Product-line Manager* records previous failure in the field from the same products family and trace these failures to test cases that were not previously selected and that should now be included.
- *Manufacturing View.* Late in the system development process, the *Manufacturer* may express some specific system tests. The manufacturer test requirements must be described during the test selection activities and performed by the company before the actual manufacturing phase.
- *Installation View.* An *Installation Engineer* is responsible for integrating the manufactured product into its physical environment. These engineers have a particular view on the embedded software and are likely to propose unexplored aspects of the SUT.

We also define two transversal views in our View Model which correspond to the "+2" of the View Model. These transversal views are represented in Figure 3.11 by the hatched areas, they overlap the five views previously described:

- *Domain-specific View.* Test requirements from this view are derived from test requirements which are specific to the domain of the SUT; for instance, these test requirements may be based on safety standard, automotive standard or project-specific requirements.
- *Domain-independent View.* Test requirements from this view are derived from existing test selection techniques; for instance, boundary-values selection, data partitioning, etc.

The illustration of the 5+2 View Model with all the different views on the ATM running example would be somehow artificial as there are no real stakeholders involved in the test selection process. We refer the readers to Chapter 7, describing the application of the SESAME approach on an industrial case study, for an illustration of the views on a more realistic SUT.

3.3.3 Test designer

The *Test Designer* is responsible for the *Test Selection Model* and *Test Model* work products. The test designer is in charge of the description of test selections based on the Test Requirements identified with the 5+2 View Model. The Test Designer integrates the test requirements from the different views as illustrated in Figure 3.12. The Test Designer checks the different test requirements for consistency and produces one (or more) Test Selection Model work products from the Test Requirements.

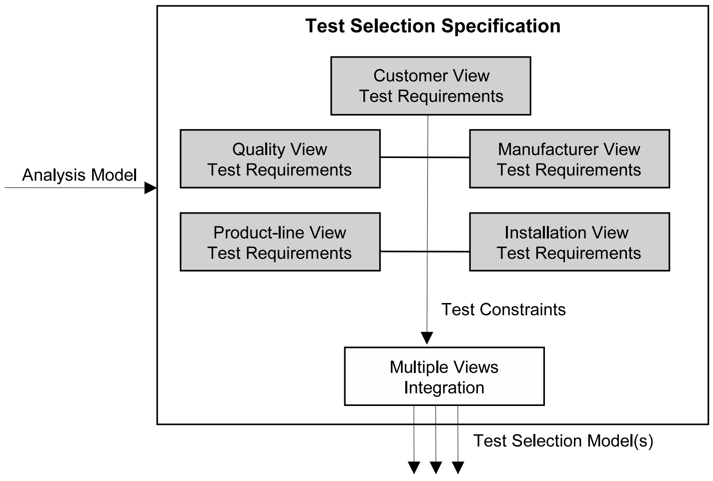

Test Selection Specification
Customer View Test Requirements
Quality View Test Requirements
Manufacturer View Test Requirements
Analysis Model
Product-line View Test Requirements
Installation View Test Requirements
Test Constraints
Multiple Views Integration
Test Selection Model(s)

Fig. 3.12: Multiple views integrations

Test Designers must learn the SERELA test selection language and must have understanding of UML class diagrams and protocol state machines used in FOREST models, in order to read the Test Model artifact resulting from his/her test selection specification.

3.3.4 Tester

The *Tester* is responsible for the work products that result from automated steps: *Formal Analysis Specification*, *Formal Test Specification*, *Test Selection Coverage* and *Abstract Test Cases*. The tester must have skills in tooling issues because the work products he/she is responsible for are produced and fed to tools. He/she must be able to setup and execute the tool-support for the test selection process. In particular, he/she is responsible for installing, configuring and executing the SESAME control center, the model transformation tool, and the formal analysis tool.

3.3.5 Test manager

The *Test Manager* is responsible for the *Test Selection Satisfaction* outcome. The test manager selects the model coverage metrics and test selection coverage metrics to be used for the project and decides on whether the test selection is satisfactory or not, based on the results of the computation of the selected metrics.

3.4 Tasks

3.4.1 The test selection workflow

The SESAME test selection workflow, illustrated by Figure 3.13, is the possible order of execution for the process model's tasks. The process model is composed of four tasks. The first task is to analyze the customer requirements. Then when test requirements are expressed, the test selection specification task is performed. The test selection specification must then be validated; this is performed by an evaluation of the test selection. If the evaluation is satisfactory then the test cases are generated. After test generation further testing tasks outside the scope of SESAME may be performed. If the evaluation is not satisfactory then the task Specify Test Selection is performed an other time until the test selection evaluation returns a satisfactory result. The fork bar between the first two tasks informs that the last three task may be performed for each coherent set of test requirements given as input of the process.

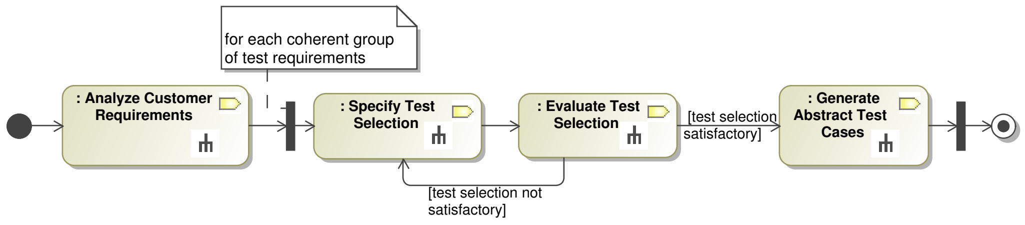

for each coherent group of test requirements
: Analyze Customer Requirements
: Specify Test Selection
: Evaluate Test Selection
[test selection satisfactory]
: Generate Abstract Test Cases
[test selection not satisfactory]

Fig. 3.13: The SESAME test selection workflow

3.4.2 Analyze customer requirements for software test purpose

Prior to this task, customer and provider have agreed on the required behavior and data of the software to be developed which results in the *customer requirements*. Two major weaknesses may arise from the customer requirements. Firstly, they are likely to be insufficient to start designing the system. Secondly, they are usually not expressed in a formal way and thus are difficult to systematically process.

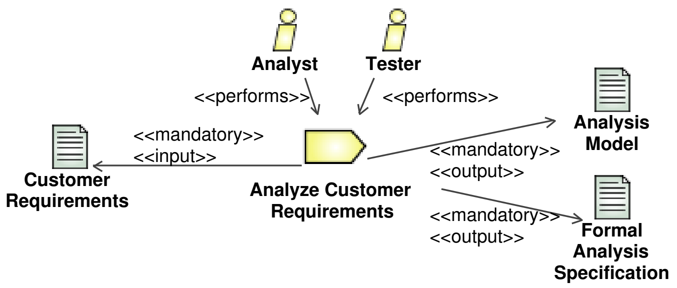


Fig. 3.14: *Task: Analyze Customer Requirements* - related role and work products

The purpose of this task is to describe the SUT, and part of its Test System, as a black-box for the further purpose of test selection and test analysis, i.e. in terms of its interface with its Test System. This task is divided in two steps. The first step is performed by the *Analyst* and describes how the analysis model output work product is created. The second step is performed by the *Tester* and describes how the formal analysis specification output work product is created from the *Analysis Model*.

Step 1 - Define the analysis model

This description is the *Analysis Model* resulting from this task, as shown in Figure 3.14. The Analysis model is iteratively defined following the five steps described below. It consists of: a UML Class Diagram [OMG07b] that describes the static view of the SUT and of the Test System; and a UML Protocol State Machine [OMG07b] that describes the dynamic view of the SUT. More details on, and formalization of, the syntax and semantics of *Analysis Models* can be found in Chapter 4.

Step 1.1 - Identify events exchanged between the SUT and the Test System

As the SUT and the Test System are seen as black-boxes, UML Components are used to represent them in the class diagram. During this step, a class diagram is created that contains two components representing the SUT and the Test System, named respectively `SUT` and `TestSystem`.

Then, the *Analyst* identifies what events may be exchanged between the Test System and the SUT. The result of the analysis of the events exchange may be represented as shown in the Table 3.2. The Event Description column of the table has an informative purpose. When the table is finished, it can be processed. Each type of events that may be received by the SUT from the Test System is translated into a UML operation of the SUT component. Each type of events, that may be received by the Test System from the SUT, is translated into an UML operation of the Test System component.

Event direction	**Event Name**	**Event Description**
SUT ← TestSystem	insertCard	Event sent when the ATM user's bankcard is inserted
SUT ← TestSystem	authenticate	Event sent when the ATM user enters a PIN number
SUT ← TestSystem	withdraw	Event sent when the ATM user request cash withdrawal
SUT ← TestSystem	restoreCard	Event sent when the ATM user request for bankcard restore
SUT ← TestSystem	serverPermission	Event sent when granting, or not, permission to withdraw
TestSystem ← SUT	requestPermission	Event sent when requesting permission to withdraw more than 500 €
TestSystem ← SUT	deliverCash	Event sent to deliver the banknotes to the ATM user

Tab. 3.2: ATM example - list of events exchanged between the SUT and the Test System

Table 3.2 shows the identification of the event that can be exchanged between the SUT and the Test System in the context of the ATM example. The interpretation of the table results in the intermediate class diagram illustrated by Figure 3.15. We can notice that the five events received by the SUT from the Test System (insertCard, authenticate, withdraw, serverPermission and restoreCard) are modeled as five operations in the `ATM` component and that the two events received by the Test System from the SUT (deliverCash and requestPermission) are modeled as two operations in the `TestSystem` component.

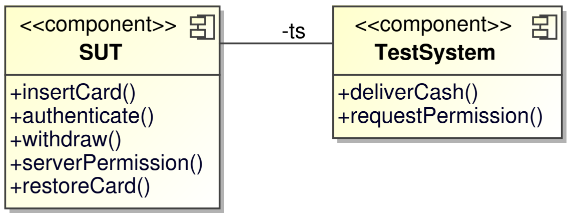


Fig. 3.15: ATM Example: Step 1.1 - Identify incoming and outgoing events

Step 1.2 - Identify allowed order of event receptions of the SUT

When the events are identified, the allowed order of events received by the SUT must be identified by the *Analyst*. In particular, the precedence of events should be exhibited. When the order of event receptions is described informally a protocol state machines is created with an initial state. For each event, that can be received at the initial state, a transition is modeled that starts from the initial state. If after the event is received the same event set is allowed then the transition is a self-transition and returns to the same state than it started from; if not, then a new state is created and the transition ends in this state. Then, for each event that can be received by this new state, a transition is modeled that starts from this state, etc. If an event reception causes the system to stop, then the transition ends in the final state.

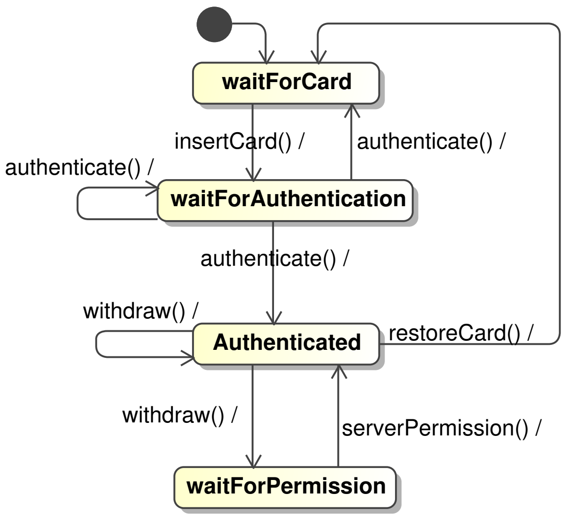


Fig. 3.16: ATM Example: Step 1.2 - Identify allowed order of event receptions of the SUT

In the case, of the ATM, the initial event allowed is the `insertCard` event. When it is received, the `authenticate` and `restoreCard` events are allowed. If `authenticate` is successful the SUT goes to another state for which the events `withdraw` and `restoreCard` are allowed, etc. Figure

3.16 shows the temporary protocol state machine that represents the allowed order of event for the ATM example.

Step 1.3 - Identify data of the exchanged events

During this step, the Analyst focuses on the identification of the data conveyed by each of the events identified in step 1.1. A table showing the different events and describing informally their types may be of interest as an intermediate step to the specification of data in the analysis model. Table 3.3 shows data types identified for the events received by the SUT and the Test System in the context of the ATM example.

Data types of the event parameters may either be a Boolean UML primitive type or user-defined data type. In the SESAME context of test selection, event parameters of other primitive types than Boolean (i.e. Integer, UnlimitedNatural, String) are strongly discouraged, due to their infinite number of values. Abstractions of these primitive types should be used instead. For practical reasons, *state variables* of `Integer` type are tolerated when used as counters having a small range of values. A UML DataType is created for each user-defined data type used in parameters of events of the SUT or Test System components. For instance in the ATM example in Figure 3.17, the four data types `Digit`, `BankCard`, `PIN` and `CashAmount` are user-defined data types and a UML DataType class is created for each of them in the static view.

Event Name	**Event Parameters Data Description**
insertCard	BankCard information simplified to its PIN number (sequence of 4 digits)
authenticate	PIN number (sequence of 4 digits)
withdraw	Amount of money. The operation withdraw behaves differently with small amount of money (concrete values between 20 and 480) and with large amount of money (concrete values between 500 and 1000).
restoreCard	*no data*
serverPermission	Boolean value and the allowed amount of money
requestPermission	Amount of money
deliverCash	Amount of money

Tab. 3.3: ATM example - data conveyed by the incoming/outgoing SUT events

Equivalence partitioning [RC81, GG93] is a mature technique for the selection of test. It is based on the partitioning of data by identifying classes of equivalences, i.e. set of data values for which the SUT is assumed to behave equivalently. Equivalence classes have been used for test selection specification of embedded systems in the automotive industry [CK06]. Based on this technique, we suggest that three tasks may be performed for the precise specification of data, in our test selection context:

1 *Identification of possible test properties.* A test property represents an abstract property of a given data type that should be exercised. A test property is represented as an attribute of the data type. Test properties may introduce additional data types.

2 *Partitioning of the data types.* A partition is an abstract representation of a subset of a given data type. A partition is represented with a Generalization association from the partition data type to the partitioned data type. For instance, data type `CashAmount` is made of two partitions `LargeCA` and `SmallCA`.

3 *Specification of constraints on data types.* The specification of constraints on the data types aim at restricting valid combinations of test properties. Illustrations for these constraints are given in the industrial case study 7.3.

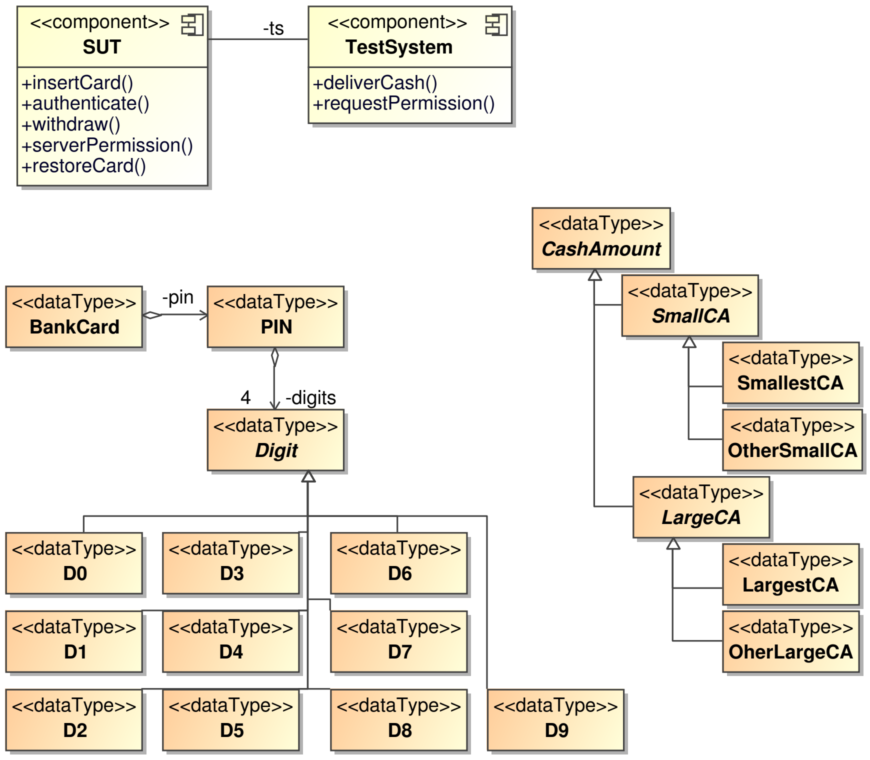


Fig. 3.17: ATM Example: Step 1.3 - Identify data of the exchanged events

Step 1.4 - Characterize the observable state variables of the system

During this step, the Analyst specifies the attributes of the system that are observable from outside the SUT and that characterize the state of the system. State variables are the attributes characterizing the state of the system. When identified, the state variables are expressed as UML Attributes of the SUT component.

For instance, with the ATM, two states variables are identified: the bankcard card that is inserted in the ATM and the current number of false authentications of the user of the ATM. They are specified as the two attributes `card` and `attempts` in the ATM component of the analysis model, see Figure 3.1.

Step 1.5 - Characterize the pre- and post-conditions associated with event receptions

First of all, the Analyst specifies the initial state of the system variable as a post-condition in the transition that starts from the initial state. Then the Analyst specifies the pre-/post-conditions for each transition of the protocol state machine.

The precondition of a transition specifies what must be the values of the state variables of the system and/or the values of the event parameters in order to accept the event reception and process the related system operation. If the values of a state variable, or of a parameter, are indifferent then no values are specified. The postcondition of a transition specifies what must be the values of the state variables of the system after the event and its related operation have been processed. The syntax for labeling of protocol transitions is the following

```
[precondition] event/[postcondition]
```

It may be that the pre- or post-condition miss information to be fully expressed, this may be due to a missing state variable or a missing event parameter. In these cases, the Analyst should reiterate step 1.4, respectively step 1.3.

The completed protocol state machine of the analysis model of the ATM example with the pre-/post-conditions is illustrated by Figure 3.2. For instance, the self-transition of the state waitForAuthentication with its pre- and postcondition expresses that:

- *(event trigger)* when the SUT is in the waitForAuthentication state and the event authenticate is received by the SUT,
- *(precondition)* if the pin value given as a parameter of the event is not equal to the pin of the inserted card and if there were less than three authentication attempts performed,
- *(postcondition)* then the number of false authentications (i.e. self.attempts) is incremented and the SUT remains in the waitForAuthentication state.

Step 2 - Translate the analysis model into a formal analysis specification

In this step, the Tester executes the model transformation that produces the formal analysis specification in Alloy from the analysis model. This model transformation is based on the FOREST meta model (see Figure 4.3); it takes a FOREST model as input and translates it into an Alloy model. Analysis models and test models are both FOREST models. As a consequence, this model transformation used for the analysis model, is the same that is used for the Test model. The *Formal Analysis Specification* is fully derived from the *Analysis Model* via a defined model transformation, implemented in KerMeta. The algorithm of this model transformation is given in Section 4.5.

3.4.3 Specify a test selection

The previous task results in an analysis model of the SUT and its Test System. In general, analysis models represent a large execution set. That is why it is necessary to select parts of this execution set, in order to use a model that represents a tractable execution set for the test generation.

The main contribution of the SESAME process that improves test selection activities is the definition of the SERELA domain-specific language. In this section, we present shortly, the different steps of specifying test selection specifications.

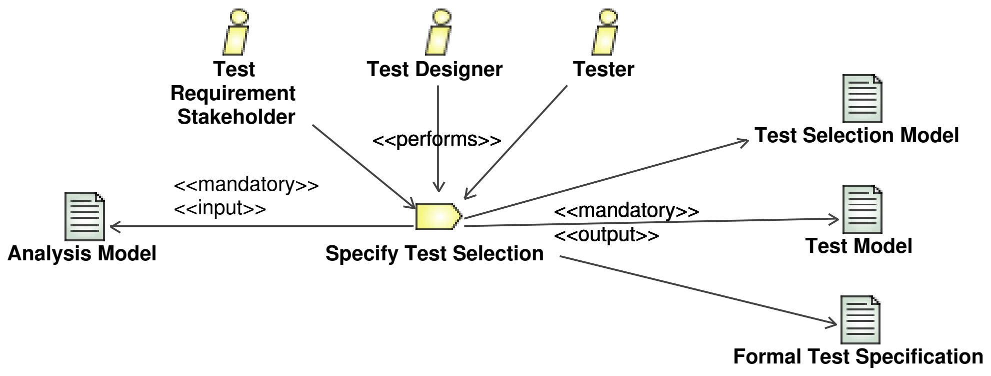


Fig. 3.18: *Task: Specify Test Selection* - related roles and work products

Step 1 - Test selection model specification

Step 1.1 - Collecting test requirements from various stakeholders with the 5+2 View Model

During this step, the Test Designer collects test requirements from the Test Requirement Stakeholders. The Test Designer discusses with the Test Requirement Stakeholders in order to elicit their test requirements. The expressed test requirements are not formalized within our approach, their format is left open. While gathering the test requirements, the Test Designer categorizes them according to the 5+2 View Model. In particular, the Test Designer identifies whether the test requirements is domain-specific or domain-independent; and the Test Designer tags the test requirements with the view, it was collected from (Customer view, or Quality view, etc.).

Step 1.2 - Production of test selection models from the test requirements

After test requirements are collected, the Test Designer analyzes the overall set of test requirements coming from the different views. A straightforward specification of the test selection models from the test requirements is to produce one test selection model for each test requirements. This strategy may lead to an unnecessary large number of test selection models. That is why, the Test Design perform an informal analysis of the requirements that aims at grouping similar test requirements into the same test selection model.

In the following, we illustrate this step with three test requirements specified with a single test selection model. We take the following consistent subset of test requirements collected by the Test Designer in the context of the ATM example:

TR1. The insertion of bank cards and false authentications are not tested.

TR2. Every test case must comprise exactly five withdrawals.

TR3. Each test case comprises at least one withdrawal of a large amount of cash and one of a small amount of cash.

Figure 3.5 illustrates a possible way of specifying these test requirements as a test selection specification with the SERELA language. Test selection instructions 01, 02, 03 and 08 of Figure 3.5

are derived from test requirement TR1. The test selection instructions 04 and 07 are derived from the test requirement TR2. And the test selection instructions 05 and 06 of Figure 3.5 are derived from the test requirement TR3.

Step 2 - Translation into a graphical test selection model

The textual SERELA test selection specification resulting from step 1 is then translated to a graphical test selection model. This step is automated in order to reduce possible manual translations errors. Tools like sintaks [sin] perform this kind of text-to-model transformations.

Step 3 - Construction of a constrained model

During this step the information from the test selection model is interpreted and a constrained model is constructed based on the analysis model and the interpretation of the test selection model. This step is automated and the algorithm for this translation is given in Section 5.3.

Step 4 - Translation of the constrained model into a formal constrained specification

This step translates all the concepts of the constrained model into a formal description of them using the Alloy language. The translation is automated and the algorithm for this translation is given in Section 4.5.

3.4.4 Evaluate the test selection

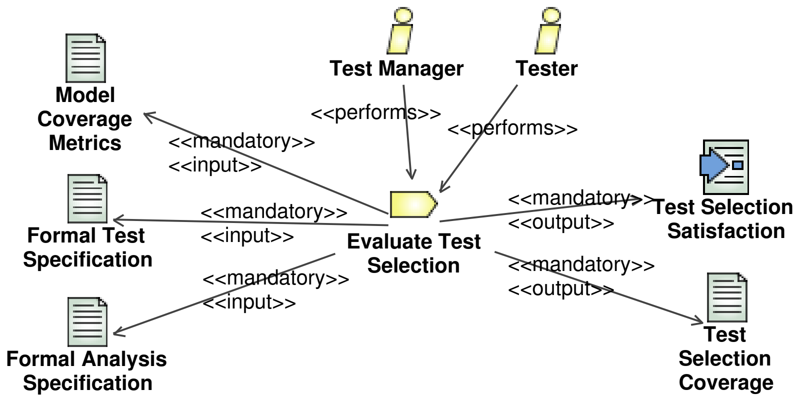


Fig. 3.19: *Task: Evaluate Test Selection* - related roles and work products

The purpose of this task is to evaluate the test selection before the generation of the test cases. In order to evaluate the test selection, the values of some selected metrics are computed. These values are intended to help test managers in deciding if the test selection is satisfactory.

Step 1 - Select existing metrics of interest

The first step of the task: *Evaluate Test Selection* is to look into the available metrics present in the *Metrics Library*. The *Test Manager* looks at the existing metrics present in the Metrics Library and selects the ones that are applicable and useful to the particular test selection evaluation of the current SUT. The *Model Coverage Metrics* work product is the union of these selected metrics.

As already presented in Section 3.2.6, the three metrics state, event and transition coverage from the Metrics Library may be selected for the ATM example.

Step 2 - Define new metrics (optional)

If the Test Manager feels that the *Model Coverage Metrics*, as defined in the previous step 1, will not be sufficient to measure the current test selection then some new metrics may be defined to enrich the *Model Coverage Metrics*. This step may be performed for two reasons, either some project-specific metrics may be required or some general-purpose metrics are not yet present in the Metrics Library. In the latter case, the newly defined metrics are added to the Metrics Library in order to possibly reuse them for test selection of further company projects.

Step 3 - Compute the test selection evaluation

During this step, the Tester implements/configures the general-purpose metrics for the SUT. Then the Tester launches the part of the tool chain that computes the resulting values from computing the metrics on the initial model and test model of the SUT.

3.4.5 Generate abstract test cases from the test selection

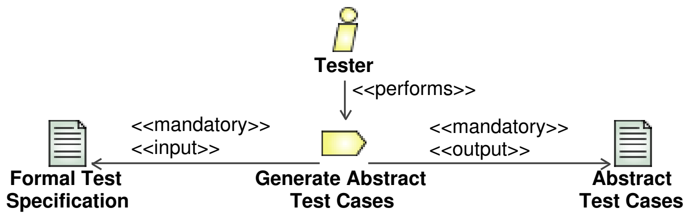


Fig. 3.20: *Task: Generate Abstract Test Cases* - related role and work products

The generation of abstract tests is the interpretation of the test model into a set of sequences of event calls coming with their parameter values. In our process, we promote to perform an exhaustive generation of test cases from the test model; if the exhaustive generation is not possible (e.g. for timing reasons) then there shall be an additional iteration of the test selection process, i.e. the test selection specification must be further constrained (or relaxed) and evaluated until the exhaustive generation from the test model is satisfactory.

3.5 Tool-support for SESAME processes

An implementation of the tool-support has been developed [Rie] for the SESAME processes

The SESAME tool chain interconnects four tools required for the tool-support implementation of SESAME processes:

- A UML-compliant modeling tool to support the task of analyzing the customer requirements.
- A model transformation tool to support the creation of the test model during the test selection specification task.
- a formal analysis tool to support the test selection evaluation task and the abstract test cases generation task.
- the SESAME Control Center, a graphical interface intended for test engineers. This tool is responsible to launch the automated tasks of the process (model transformations, test metrics computation and test generation) and to interpret the output of these tasks.

The following sections present a choice of tools that can be used for the SESAME tool chain.

3.5.1 UML modeling tool

Requirements for the selection of the UML modeling tool

The UML modeling tool of the SESAME tool chain must support the modeling of class diagrams and protocol state machines, as described in Section 3.2.1. In particular, this tool must be able to model: components with attributes and methods, and OCL constraints on their methods; classes with attributes, and OCL constraints on their attributes; apply new stereotypes to components and components' attributes; protocol transitions with labels containing the triggering event and the associated pre-/post-conditions in OCL.

UML modeling tool selected

The UML modeling tool selected as part of the SESAME tool chain is MagicDraw. MagicDraw is a UML2-compliant modeling tool that supports a number of different kinds of UML diagrams as well as some UML profiles (e.g. UML profile for SPEM). In particular, it supports class diagrams and protocol state machines[3] used for the modeling of analysis and test models.

SESAME-specific tool-support implementation

No particular implementation is required to support the SESAME process with the MagicDraw modeling tool.

[3] At the date of the writing of this thesis, MagicDraw is one of the very few UML tools, which support the modeling of the recently standardized protocol state machines. Indeed, it is just a matter of time, for the other UML tools to support this kind of state machines.

3.5.2 Model transformation tool

While model transformations are interesting to structure the software development process, applying model transformations by hand are time-consuming and error-prone. That is why model transformations in the SESAME process are automated. On the one hand, the model transformation tool supports the task "Specify Test Selection" by interpreting the test selection model. Based on this interpretation, the tool performs model transformations that construct the test model. On the other hand, the model transformation tool performs the translation of analysis and test models into formal specifications.

Requirements for the selection of the model transformation tool

Firstly, the model transformation tool must be able to add, modify and remove any kind of elements part of analysis models. Secondly, the models produced with the modeling tool must be usable as input for the model transformation tool.

Model transformation tool selected

The tool selected is KerMeta [MFJ05]. KerMeta [MFJ05] is an open source meta-modeling language developed by the Triskell team at IRISA. It has been designed as an extension to the EMOF 2.0 [ISO05] to be the core of a meta-modeling platform. KerMeta extends EMOF with an action language that allows specifying semantics and behaviors of metamodels. The action language is imperative and object-oriented. It is used to provide an implementation of operations defined in metamodels. As a result the KerMeta language can, not only be used for the definition of metamodels but also for implementing their semantics, constraints and model transformations [MFV$^+$05].

To implement the SESAME model-driven test selection process model proposed in this thesis, KerMeta has been chosen for two reasons. Firstly, a KerMeta program allows implementing model transformations [MFV$^+$05] from any model compliant with a metamodel expressed in EMOF; in particular KerMeta allows the transformations of UML models. Secondly, KerMeta tools are embedded in the Eclipse Modeling Framework (EMF) [Ecla] and consequently support models in the EMF format. As MagicDraw can export its models in EMF format, thus KerMeta can be "chained" with the MagicDraw UML modeling tool.

SESAME-specific tool-support implementation

The implementation required to support SESAME processes with KerMeta is twofold. Firstly, a KerMeta program must be written that automates the semantics of SERELA specifications. This program specifies how each SERELA instructions are interpreted in terms of addition, modification, removal of elements of analysis models. This KerMeta program (see Appendix B) has two parts. One part defines for each of the test selection operation, a model transformation operation to be performed on the analysis model. It corresponds to a generic definition of the semantics of the SERELA test selection constraints. The other part is a main operation that parses the test selection model and calls the associated model transformation operations with parameters specified in the test selection model.

Secondly, another KerMeta program must be written that translates the FOREST models into formal specifications. This KerMeta program (see Appendix A) translates the FOREST concepts into Alloy concepts, as defined by their respective meta models.

3.5.3 Formal analysis tool

The tool-support for the analysis of Alloy formal analysis specification is the Alloy Analyzer [Jac06]. The Alloy Analyzer has already been used successfully for test generation from Alloy models [KMSJ03]. Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking. It is a "model finder": Given an Alloy specification and an Alloy logical formula, it attempts to find a binding of the variables to values that makes the formula true. In the context of test generation, this is particularly interesting, as Alloy Analyzer provides iteratively the exhaustive execution set of the specification. In the context of the support of the computation of the test selection evaluation metrics, if the metric can be expressed as an Alloy logical formula, then Alloy Analyzer provides a result for this metric.

3.5.4 SESAME Control Center

In order to provide a single interface to the *Tester*, the *SESAME Control Center* has been developed. This program communicates and launches the execution of the different tasks associated with the modeling tool, the model transformation tool and the formal analysis tool aforementioned. The SESAME Control Center is implemented in Java as a plug-in to the Eclipse Framework.

4. FOREST: AN ANALYSIS MODELING LANGUAGE FOR BLACK-BOX TESTING

Abstract

This chapter describes the FOREST language and its formalization in terms of a semantics language. The FOREST language concepts are presented. The concrete syntax is described and based on a subset of UML2 elements. Then the semantic language chosen (Alloy) is described. The Alloy concepts and syntax, used for the definition of the FOREST model semantics, are presented. Finally, the semantics of FOREST models is described as a generic algorithm that transform FOREST models into Alloy models.

4.1 Introduction

UML is a general-purpose modeling language; it is defined to fulfill needs from various application domains. This thesis is aimed at industries with a low modeling maturity level. In order to reduce the risk of misuse of UML artifacts, we identify a subset of UML artifacts required in the SESAME methodology and define a graphical modeling language containing this restriccted set of UML elements. Two modeling languages are used within SESAME processes

- a *graphical modeling language* for the specification of the analysis model of the system with behavior and data based on existing UML2 artifacts. This modeling language is intended to end-users, typically software analysts. We name this language FOREST that stands for FORmal STatecharts.
- a *semantic language*. Systems targeted by the SESAME process model are small-size safety-related embedded systems. We perform formal analysis for two purposes: evaluating the test selection and generating test cases. The semantics language is used to precisely specify FOREST models. Formal specifications are derived from FOREST models by performing predefined model transformations.

4.1.1 FOREST model evolution through the process

A FOREST model is modified throughout the progress of a SESAME process. FOREST models may be in three different states, as shown in Figure 4.1:

- *initial state*: this is the initial state of a FOREST model, it represents a model of the customer requirements, as such, in this state the model is also called an "*analysis model*".

- *constrained state*: after the test selection specification with SERELA (see following Chapter 5), the FOREST model is enriched with test selection constraints. The constrained state corresponds to a FOREST model enriched with test selection constraints derived from a SERELA specification. When a constrained FOREST model evaluation results in an unsatisfactory outcome according to the model coverage metrics defined (see following Chapter 6), then it remains in a constrained state. When the test selection evaluation outcome is satisfactory, the FOREST model becomes validated.

- *validated state*: this is the final state of a FOREST model, when the model is in this state it may be used for test generation. When the model is in this state it is named "*test model*". Concretely, the semantics of a test model and of a FOREST model in a constrained state are equivalent; it is the sole outcome from the test selection evaluation that changes the states of a FOREST model from a constrained state to a validated state.

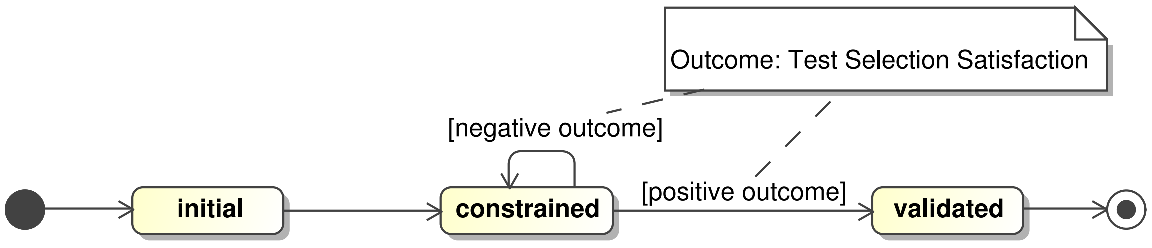


Fig. 4.1: The different states of a FOREST model

4.1.2 FOREST formalization framework

The formalization of models involves the definition of a number of concepts. The FOREST concepts are presented in Section 4.2, FOREST concepts are differentiated between:

- *Structural concepts* define what a FOREST model is made of.

- *Behavioral concepts* define how a FOREST model behaves, they relate to the concepts required to simulate the FOREST model for the purpose of testing.

In a model-driven engineering environment, models comply to their meta models. A meta model is an abstract representation of the concepts of the modeling language. Figure 4.2 illustrates the formalization framework of FOREST models in a model-driven engineering context. In the SESAME approach, the FOREST meta model, shown in Figure 4.3, focuses on the abstract representation of the FOREST structural concepts. It does not include an abstract representation of the FOREST behavioral concepts. The abstract representation of the FOREST behavioral concepts (as well as the FOREST structural concepts) is described in the Alloy meta model, shown in Figure 4.14. Consequently, Alloy models can be simulated as the behavioral concepts are precisely specified within the Alloy models.

As presented in the introduction of this chapter, two types of models are handled in SESAME processes, FOREST models and Alloy models. This results, as shown in Figure 4.2, in a formalization framework containing two types of models, each complying with its meta-model.

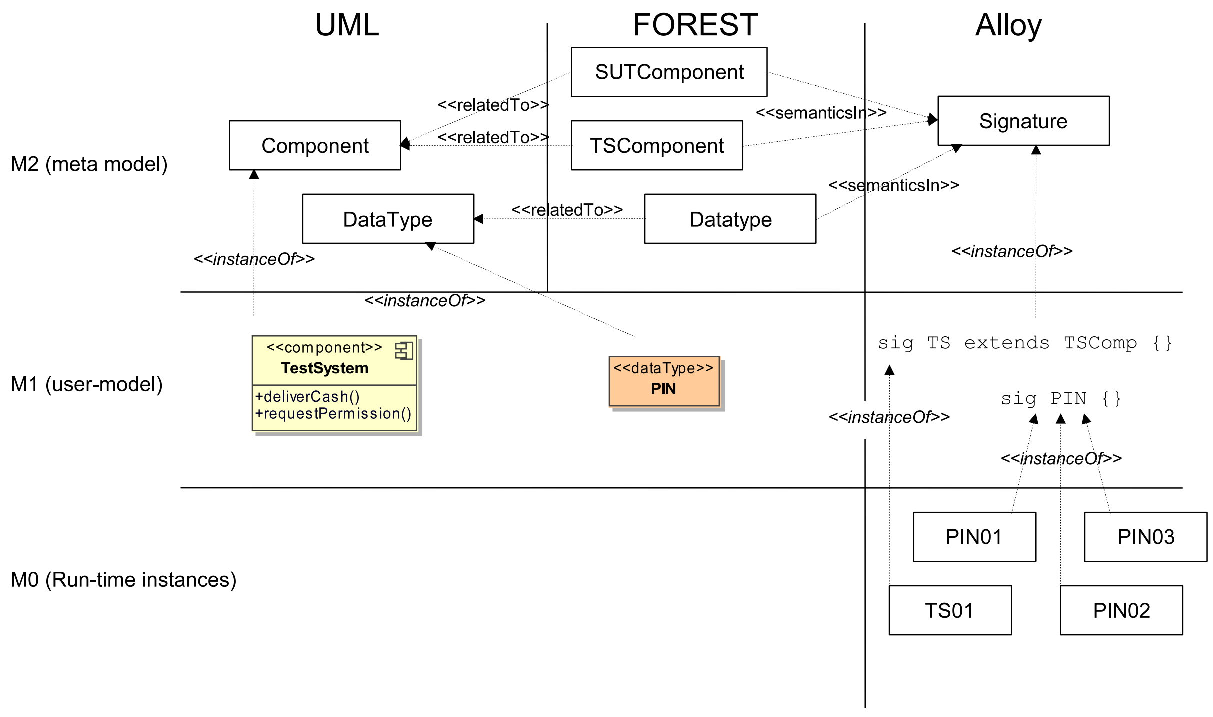


Fig. 4.2: FOREST formalization framework

The FOREST formalization framework also describes the transformation of FOREST models into Alloy models. This is represented by the arrow "has semantics in" in Figure 4.2. The model transformation algorithm is presented in Section 4.5.

Lastly, the FOREST formalization framework comprises a concrete representation of the concepts described in the two meta models. The concrete representation of FOREST models is based on a selection of a subset of UML2 artifacts. It is presented in Section 4.3. The concrete representation of Alloy models is the Alloy language, a short presentation of the Alloy elements used in the SESAME approach is done in Section 4.4.

Semantics of UML is described informally in natural language [OMG07b]. Moreover, UML semantics is left intentionally incomplete. Semantics variation points characterize the incomplete aspects of the UML standard. Consequently, it is necessary to decide what the semantics of elements having semantic variation points is. UML state machines semantics variation points concerns the three following points as identified by Chauvel et al. [CJ05]:

- *Event management*: Select randomly one of the "available" events with respect to the current state and their outgoing transitions' preconditions.
- *Transition management*: If more than one transition can be fired by the selected event (non-determinism) then one transition is chosen randomly
- *Time management*: Synchrony hypothesis: Time is continuous. Events are processed as soon as they are received.

4.2 FOREST concepts

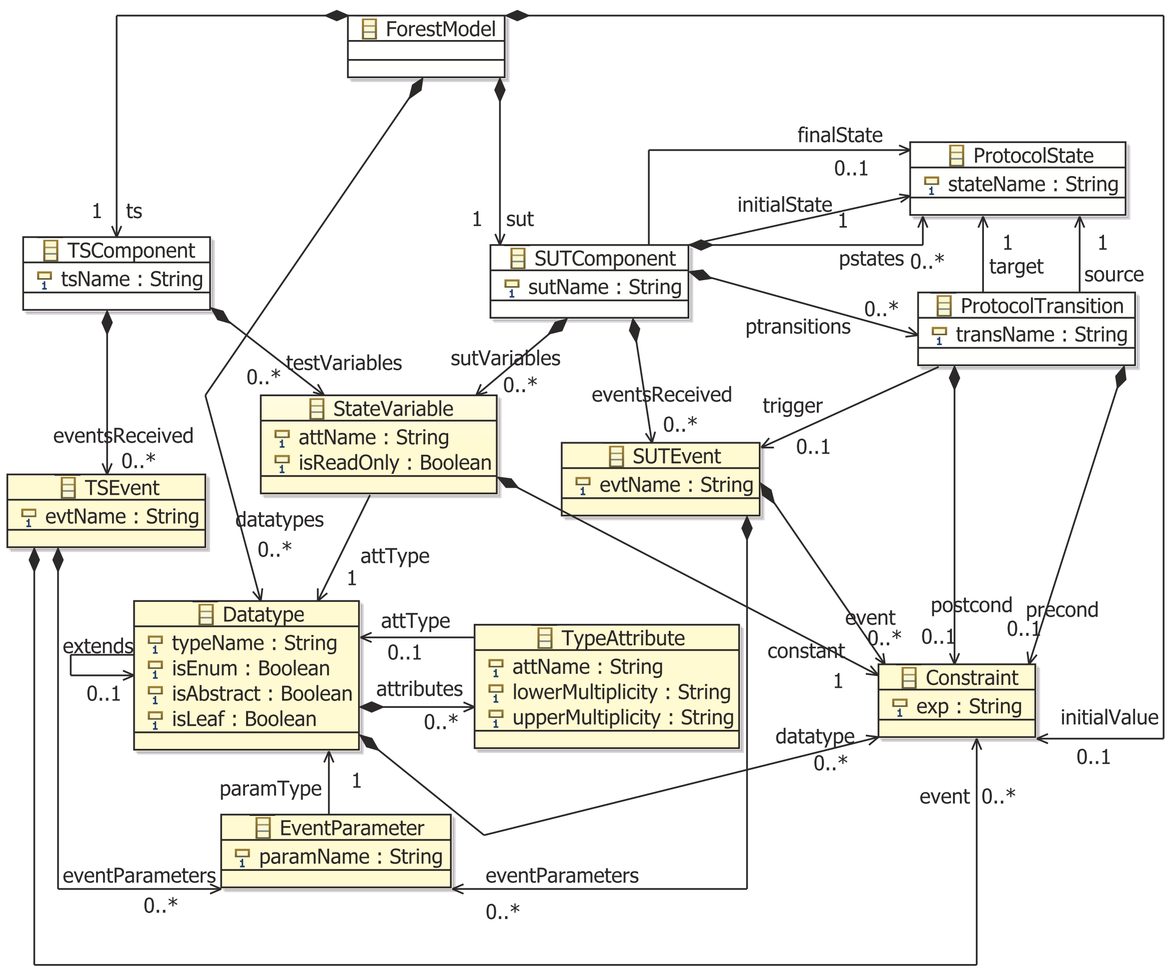


Fig. 4.3: FOREST meta-model

4.2.1 Structural concepts

FOREST models are graphical and their abstract syntax is defined by the meta-model in Figure 4.3. The FOREST meta-model describes the structural concepts, as described in Section 4.3, that can be used for specifying FOREST models.

In this subsection, we describe the FOREST structural concepts informally. Some well-formedness rules (WFR) enrich the description of constraint that models complying with the FOREST meta model must fulfill. The `ForestModel` meta-class is the entry point to read the meta model, as it contains all the meta class representing the FOREST structural concepts.

SUT component

The SUT component is a container that represents the system under test as a black-box. An SUT component is named and contains state variables, test variables and the events that it may receive from the Test System. In a FOREST model a component is modeled as an analysis artifact. The SUT component is the result from analyzing the customer requirements in order to represent the system under test.

WFR FOREST.1 There must be exactly one SUT component per FOREST model.

Test System Component

A *Test System Component* is a container that represents the Test System of the SUT as a black box. A Test System Component is named and comprises the specification of its state variables and of the events that it may receive from the system under test.

WFR FOREST.2 There must be exactly one Test System Component per FOREST model.

State variables

State variables characterize the observable state of a system. A state variable is characterized by its name associated with a data type and an initialization constraint.

State variables may be used to characterize the observable state of the system under test; in that case they are referred to as *SUT variables.* In the meta model they are represented by the association between the `SUTComponent` and `StateVariable` meta classes (i.e. classes of a meta model). State variables may also be used to characterize the observable state of the Test System; in that case they may also be called *test variables*, `testVariables` in the meta model. A test variable is a state variable of the Test System Component that is introduced during the test selection specification process in order to control, and restrict, the allowed behaviors of the system under test.

Values of state variables may be constant over time, in that case they are called *constant state variables* and their attribute is `ReadOnly` is set to true. Default value for the attribute `ReadOnly` of `StateVariable` is false.

WFR FOREST.3 State variable names of the Test System and system under test must be unique.

WFR FOREST.4 Constant state variables are related with exactly one constraint that specifies its constant value.

Event receptions

The Test System Component and the SUT component receive events from each other. Events are usually parameterized. They may contain a number of parameters; each parameter is given a name and a type. An event which does not contain any parameter may also be called a *signal.*

WFR FOREST.5 Two events are different if at least one of their parameters has a different value.

WFR FOREST.6 Event reception names must be unique.

Data type

In FOREST models, the specification of data types results from the analysis of data conveyed by the events and data used to characterize the observable state of the SUT and Test System Components. This data analysis is performed for the purpose of further testing. Data types specified in FOREST models are called *analysis data types* because they abstract away the (concrete) design of the interfaces of the SUT and of the Test System.

In order to actually exercise the SUT with test data, analysis data types must be transformed into concrete data types matching the design of the interfaces of the SUT. For instance, in the ATM example, a cash amount is represented as a large cash amount or a small cash amount. Designing test data types, then consists in assigning some concrete values or range of values to the analysis data types. In the case of the cash amounts of the ATM, this could be performed by specifying that small cash amounts are integer values greater than zero and lower than five hundreds euros; and large cash amounts could be integer values greater than five hundreds and lower than thousand euros. Design of test data is out of the scope of the SESAME approach. Our approach focuses on the *analysis* of test data.

A *data type* is an abstract representation of a set of concrete test values. A *structured data type* is a data type which is constructed from a number of other data types. The structure of a data type is defined through its typed attributes. The multiplicity of attributes may be specified, as a lower bound and an upper bound on the multiplicity; by default both bounds equals to one. As defined in the UML standard [OMG07b], a *leaf data type* is a data that can not be refined, i.e. for which no data type extends it. In the testing context of the SESAME approach, we use leaf data types to represent sets of test values for which the SUT is assumed to behave equivalently (equivalence class). Thus, in our testing context, a leaf data type contains a single value, which stands as the representative value for the other values of the leaf data type. In the SESAME, *enumerations* are seen as abstract data types and their literals are leaf data types of the abstract data types.

WFR FOREST.7 A data type may only extend an abstract data type.

WFR FOREST.8 All types associated with state variables, event parameters, or data type attributes (except for enumerations attributes) must be associated with a DataType of the static view.

WFR FOREST.9 Types of attributes of a data type must be different than their containing data type.

WFR FOREST.10 All data types name must be unique.

Primitive type

A primitive type is a type predefined in the UML meta model. A primitive type does not contain any attribute defining its structure. The two primitive types allowed in FOREST models are `Integer` and `Boolean`. The primitive types of a given FOREST model is the subset of these primitive types used to define the DataTypes of the given model.

WFR FOREST.11 A primitive type does not have any attributes.

Protocol state

A protocol state represents an exposed stable situation of its context classifier [OMG07b]. In our case, the context classifier is the system under test. When the SUT is in a protocol state its observable state, i.e. the values of its state variables, do not change. A protocol state is characterized by its name. The name of a protocol state aims at easing the understanding of the stable situations of the SUT. It is informal and do not influence the behavior of the modeled system.

Two particular protocol states require special care, the initial protocol state and the final protocol state, see the behavioral concepts description in Section 4.2.2 for more details.

WFR FOREST.12 A protocol state name does not comprise space characters.

WFR FOREST.13 Protocol states' names are unique.

WFR FOREST.14 There is exactly one initial protocol state associated with the SUT component.

WFR FOREST.15 The final protocol state, if any, is named "final".

WFR FOREST.16 At least one of the protocol transitions must have its source state being the initial state.

WFR FOREST.17 The initial state and the final state are protocol states of the SUT component, i.e. they are both included in the set of protocol states of the pstates association.

Protocol transition

A protocol transition represents a change in the state of the system under test due to the reception of an event from the Test System. As defined in Figure 4.3, protocol transitions are characterized by one source protocol state, one target protocol state, one trigger (i.e. an event reception), one optional pre-condition constraint, and one optional post-condition constraint. A protocol transition expresses two pieces of information. Firstly, the event specified in the trigger may be received by the SUT from the Test System when the SUT is in the source protocol state and the pre-condition is true. Secondly, when the protocol transition is taken the state of the SUT is modified according to the postcondition and the active protocol state becomes the target state of the protocol transition.

WFR FOREST.18 Variables used in pre- and post-conditions of a protocol transition must be either a parameter of the event triggering the transition or specified as a state variable of the SUT or the Test System Component.

WFR FOREST.19 All protocol transitions must be named and their name must not comprise space characters.

WFR FOREST.20 All protocol transitions' names are unique.

WFR FOREST.21 There must be no outgoing protocol transition from a final state.

WFR FOREST.22 Event receptions specified as triggers of protocol transitions must be specified in the SUT component of the static view. The names of the event, of the event parameters and of event parameters type must be matching.

Constraints

Six different types of constraints may be associated with structural concepts of FOREST models:

- *EventType constraint.* This type of constraint is expressed on an event of either the SUT component or the Test System Component. It restricts the allowed parameters values of the constrained event.
- *DataType constraint.* This type of constraint is expressed on a datatype. It constrains the allowed values of the related attributes of the data type.
- *ConstantVariable constraint.* This type of constraint is expressed on constant state variables. It specifies the constant value of the variable it is associated with.
- *InitialValue constraint.* This type of constraint is expressed on state variables. It specifies the initial value of the variable it is associated with.
- *PreCondition constraint.* This type of constraint is associated with a protocol transition. It represents a condition on the system state that must be true before taking the related protocol transition. It is expressed in terms of state variables of the SUT and the Test System and parameters of the triggering event associated with the protocol transition.
- *PostCondition constraint.* This type of constraint is associated with a protocol transition. It represents how the system states must change when the protocol sate becomes the target state of the protocol transition. It is expressed in terms of variables of the SUT and the Test System and in terms of the next values of the SUT and Test System state variables.

WFR FOREST.23 Events in post-conditions must always be sent to the Test System Component.

4.2.2 Behavioral concepts

In this section, the concepts related to the behavior of the system under test and the Test System are described. The behavioral concepts define how FOREST models are interpreted, thus the behavioral concepts directly impact the way that the test selection evaluation and the test generation tasks behave. The *behavior* of a FOREST model is the set of all possible simulations of the model. A *simulation* of a FOREST model is a sequence of statuses and associated steps of the system under test and its Test System. The *initial status* is the first status of a model simulation. The following statuses are described with respect to their preceding status by performing a semantic *step*. The concepts of *step* and *status* are introduced in the reference paper by Harel and Naamad [HN96] describing the semantics of statecharts. Figure 4.4 illustrates the two concepts of status and steps and their interconnection over time. In our testing context, simulations of FOREST models are used to derive *test cases*.

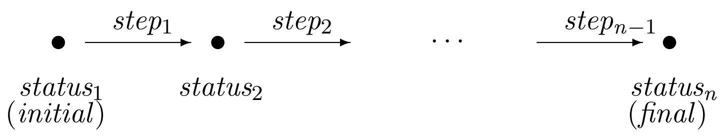


Fig. 4.4: Simulation of a FOREST model: steps and statuses (from [HN96])

Status

In the SESAME context, and given a FOREST model, a *status* is composed of the following set of values, these values are constant for each status of the model:

- A protocol state of the given model. This protocol state represents the *active protocol state* of the model.
- Values of each state variables of the SUT component.
- Values of each state variables of the Test System component.

WFR FOREST.24 The initial status is always associated with the initial protocol state of the model.

WFR FOREST.25 The values of the state variables of the initial status are initialized to their respective constraints.

WFR FOREST.26 The final status is always associated with the final protocol state of the model.

Step

A *step* is the action of changing from a status to a following status. Given a current status, a following status is computed by the execution of a step which consists in performing the following actions:

(1) Select randomly one of the enabled transitions with respect to the current status. An *enabled transition* is a protocol transition that has for source protocol state the protocol state of the current status and for which the precondition is evaluated to true with respect to the values of the current status.

(2) The event of the selected protocol transition is sent from the Test System to the SUT.

(3) The selected protocol transition is taken, which consists in performing the three following actions :

    The values of the state variables of the following status are set according to the post-condition of the selected protocol transition.

    Events are sent to the Test System according to the post-condition of the selected protocol transition.

    The protocol state of the next status is set to the target protocol state of the selected protocol transition.

It is important to stress that given our definition of a step, the Test System is restricted to send only legal events that the SUT can receive as specified in its associated FOREST model. A valid event is an event that can legally be received by the SUT at a certain point in time, as specified in the FOREST model. The parameters of a valid event are of the type specified in the CD of the SUT.

4.3 FOREST concrete syntax

The concrete syntax of FOREST models is based on a restricted set of UML2 [OMG07b] elements. UML has been chosen to ease the use of the SESAME process in industrial context. As previously mentioned in Section 3.2.1, it is particularly well-adapted to model small-size safety-related embedded software.

The concrete syntax gives a graphical representation of the structural concepts described in the previous section, 4.2.1. The behavioral concepts are specified with the semantics language, i.e. Alloy, during the transformation of FOREST model in Alloy models, as described in the following Section 4.5.3.

The concrete syntax of a FOREST model is composed of two parts that represent two complementary views on the system being modeled:

- The *static view* represents data to be handled by the system under test and by the Test System for testing purposes.
- The *dynamic view* represents the allowed set of behaviors of the system under test.

4.3.1 Concrete syntax of the static view

The concrete syntax for the static view of FOREST models is a class diagram which is restricted to certain elements appropriate for the specification of the following FOREST structural concepts: SUT and Test System components, state variables, event receptions and datatypes. In the remaining of this section, we describe these concepts together with their concrete syntax based on UML elements from class diagrams.

SUT and Test System component

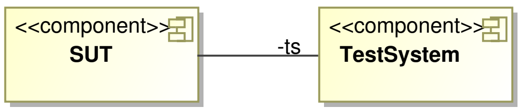
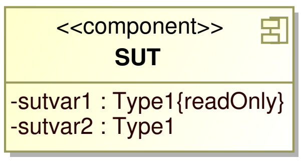


Fig. 4.5: FOREST concrete syntax: the SUT and TestSystem components

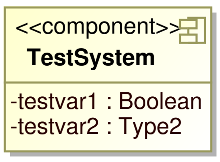
The concrete syntax chosen for the SUT component is a UML `Component` [OMG07b]. UML `Components` "can be used to define systems of arbitrary size and complexity [...] a component is modeled throughout the development life cycle [...]" [OMG07b]. The name for the SUT component is fixed and set to "SUT"; similarly, the name of the Test System component is set to "TestSystem", as shown in Figure 4.5. In addition, an association between the two components is created, so that the state variables and event receptions of the `TestSystem` component may be accessed from within the scope of the `SUT` component.

State variables

The concrete syntax of a state variable is a UML `Property` (i.e. an attribute) of the SUT component. A constant state variable is followed by `{ReadOnly}`, e.g. the `sutvar1` constant



(a) SUT state variables



(b) Test System state variables

Fig. 4.6: FOREST concrete syntax: state variables

state variable in Figure 4.6(a). The syntax is the same for the SUT component and the Test System component as shown in Figure 4.6. A property is a typed element that must be associated with a `DataType` specified in the UML class diagram. A property may be associated with a `ValueSpecification` representing its `defaultValue`. In FOREST models, this defaultValue is interpreted as an initial value of the variable. The syntax of the initialization constraint describing the initial variable value is given in the following *Constraints* subsection.

Event receptions

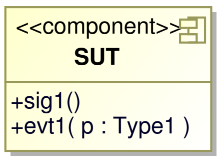


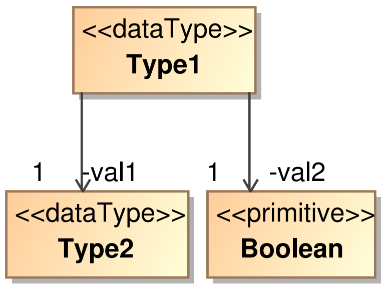
Fig. 4.7: FOREST concrete syntax: event reception

An event reception is specified as a UML `Operation`. The data conveyed by the event is specified as UML `Parameters` of the operation. A signal is specified as an operation without parameter. An event received by the SUT from the Test System is specified as an operation of the SUT component. Similarly, an event received by the Test System from the SUT is specified as an operation in the Test System component. For instance, in Figure 4.7, `evt1` is sent by the Test System and received by the SUT.

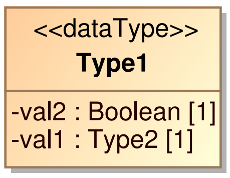
Datatype

The concrete syntax of a data type is a UML `DataType`. It is represented as a classifier with a stereotype <<`dataType`>>. A structured data type contains more than one attribute. Concerning the specification of data type attributes, two different concrete syntax may be used[1]. For instance, a data type `Type1` with two attributes `val1` of type `Type2` and `val2` of type `Boolean`, may be specified with the two different syntax as illustrated by Figures 4.8(a) and 4.8(b). The concrete syntax of abstract data types is the same as for data types except for their names that are written in italics as in Figure 4.8(c). There are no visual differences between the concrete syntax of leaf data types and of a (non-leaf) data type. The difference is within the UML CASE

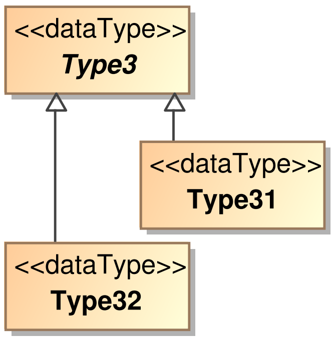
[1] These two syntax have equivalent semantics



(a) Data type attributes



(b) Data type attributes



(c) Abstract and leaf data types

Fig. 4.8: FOREST concrete syntax: data types

tool-support where the attribute isLeaf of the DataType meta-class is set to true. Data types `Type31` and `Type32` in Figure 4.8(c) are leaf data types.

Primitive type

Fig. 4.9: FOREST concrete syntax: primitive type

UML defines a number of primitive types, namely: Boolean, Integer, UnlimitedNatural, and String. In FOREST models, user-defined data types can be derived from the sole `Boolean` primitive types (or from other user-defined data types). The concrete syntax for a primitive type is a UML `Classifier` having the name of the primitive type and a stereotype <<`primitive`>>. The classifiers representing the primitive types do not contain properties or operations. It is mandatory to specify explicitly the `Boolean` primitive type within the static view, though it may be specified for clarity reasons.

4.3.2 Concrete syntax of the dynamic view

In system-level testing, we look at the SUT as a black-box; consequently, we focus on the reactions of the system to external events coming from the Test System. The list of events accepted by the SUT is defined in the static view. The dynamic view of an analysis model describes the set of allowed sequences of event receptions and their associated pre-conditions and post-conditions.

Protocol state

The concrete syntax of a protocol state is a rectangle with rounded corners. The name of the protocol state appears at the middle of the rectangle. For instance, Figure 4.10 contains three

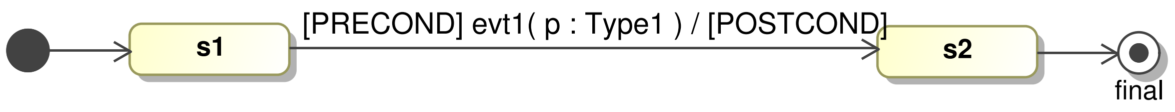


Fig. 4.10: FOREST concrete syntax: protocol state machine

protocol states named `s1`, `s2` and `final`.

The concrete syntax to set a protocol state as the initial state is a small solid filled circle with an arrow starting from this circle and pointing to the protocol state to be marked as initial state. In Figure 4.10, protocol state `s1` is the initial state of the FOREST model. [2]

The concrete syntax of a final state is a circle surrounding a small solid filled circle, as shown in Figure 4.10.

Protocol transition

The concrete syntax of a protocol transition is an arrow starting from a protocol state and ending in a protocol state (possibly the same one). Protocol transitions are labeled; the labels format is a precondition followed by an event and its type parameters and ended by a post condition as illustrated in Figure 4.10 with the protocol transition from state `s1` to state `s2`.

4.3.3 Concrete syntax of constraints

The constraint language used for UML `Constraint` is not enforced within the UML standard: "a constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element." [OMG07b] The OMG has standardized a constraint language that is suitable for expressing UML `Constraint` elements. "A user-defined Constraint is described using a specified language, whose syntax and interpretation is a tool responsibility. One predefined language for writing constraints is OCL." [OMG07b] The concrete syntax chosen for these different constraints makes use of a restricted set of Object Constraint Language (OCL) expressions [OMG06], which we refer to as *OCL-F*. This subset of OCL expressions is described in the following subsection.

OCL-F: OCL for FOREST models

An OCL specification is made of two parts, a context declaration and an expression defined within this context. When using OCL for the specification of constraints on UML models, the explicit definition of the context declaration is omitted; it is implied by the placement of the OCL expression on the UML elements of the UML model. In OCL-F, we follow the same strategy. So in the context of FOREST models, the OCL expression are explicitly specified on elements of the FOREST models and the context declaration of the OCL expression is implied by the element on which the expression is related. A benefit from omitting the context declaration is that the OCL syntax on the FOREST models is more readable due to shorter constraint specification. For

[2] The arrow from the circle to the protocol state is not regarded as a protocol transition.

instance, in Figure 4.12 a DataType constraint `{DT_CONS1}` is expressed on DataType `Type1`. The OCL specification with the explicit context declaration would be specified as follows:

```
context Type1 inv:
DT_CONS1
```

OCL-F concrete syntax is compatible with OCL as it makes use of the same syntactical constructs. Some OCL syntactical constructs are not supported within OCL-F. Types of OCL expressions are numerous as standardized by the OMG [OMG06]. In order to express constraints on FOREST models, we define OCL-F that consists of a subset of OCL expressions types. The supported OCL expression types are the following ones [3]: `AssociationEndCallExpCS`, `VariableDeclarationCS`, `OperationCallExpCS`, `PrimitiveLiteralExpCS`, `NullLiteralExpCS`, `OclMessageExpCS`, `IfExpCS`.

An `AssociationEndCall` expression access an element visible from the current context via an association. Several elements in a model are associated with each other, e.g. an operation is associated with its parameters; a parameter is associated with a type; a component is associated with its attributes, etc. Thus, this type of expression is practical for most constraints specifications. For instance, in the case of `EventType` constraints, the context is an event, i.e. the related classifier is an UML Operation. Thus, it is possible to access values of the parameters of the context operation via an expression `associationEndCallExpCS`. More concretely, in the ATM example, there is an `EventType` constraint specified whose context classifier is the `insertCard` operation. In this expression, the parameter of the context operation named `bc` can be directly used for the constraint specification. This parameter has a type (`BankCard`). Thus, by using the association between the parameter and the type, the attribute `pin` of the type `BankCard` can be constrained to the particular parameter `bc`, by simply stating ”`bc.pin`”. The expression ”`bc.pin`” is of type `AssociationEndCall`.

A `VariableDeclaration` expression is (obviously) an OCL expression that declares a variable. In the context of OCL-F, `VariableDeclaration` expressions are used for the initialization of the state variables values and the specification of the constant values for constant state variables. During the initialization of the state variables, a state variable is assigned to some values of its related primitive type. The state variable may otherwise be assigned to `NULL` (in the grammar, `NullLiteralExpCS`) resulting in the state variable not being related to any value at the initialization.

In OCL, a `PrimitiveLiteralExpCS` is a literal expressing a value of a primitive type. In OCL-F, three primitive types are supported, strings, integers and Booleans, as shown in the EBNF grammar rules in Figure 4.11.

In OCL, the `OperationCallExpCS` is an expression containing side-effect-free operations defined within the classifiers of the UML model. In OCL-F the `OperationCallExpCS` expression is restricted to the use of logical, arithmetic and comparison operators.

Introduction on the description of the constraint types

There are six types of constraints that can be expressed on FOREST models: EventType, DataType, ConstantVariables, InitialValue, PreCondition and PostCondition constraints. On

[3] Names used in this thesis for OCL-F expressions are taken from the definition of the EBNF rules defining OCL [OMG06]

```
OclExpressionCS ::= AssociationEndCallExpCS
                | OperationCallExpCS
                | PrimitiveLiteralExpCS
                | NullLiteralExpCS
                | OclMessageExpCS
                | IfExpCS

AssociationEndCallExpCS ::= ( AssociationEndCallExpCS '.' )? associationEndNameCS
                        ('[' index ']')? '@pre'?

OperationCallExpCS ::= OclfExpressionCS[1] Operator OclfExpressionCS[2]
                       OclfExpressionCS[1] '.oclIsKindOf(' OclfExpressionCS[2] ')'

PrimitiveLiteralExpCS ::= IntegerLiteralExpCS
                      | StringLiteralExpCS
                      | BooleanLiteralExpCS

OclMessageExpCS ::= OclExpressionCS '^' simpleNameCS '(' OclExpressionCS? ')'

IfExpCS ::= 'if' OclExpressionCS[1] 'then' OclExpressionCS[2]
                'else' OclExpressionCS[3] 'endif'

Operator ::= ComparisonOperator
         | ArithmeticOperator
         | LogicalOperator

ComparisonOperator ::= '<'
                   | '>'
                   | '<='
                   | '>='
                   | '='
                   | '<>'

ArithmeticOperator ::= '+'
                   | '-'
                   | '*'
                   | '/'

LogicalOperator ::= 'or'
                | 'and'
```

Fig. 4.11: Definition of OCL-F grammar rules based on existing OCL grammar rules

the one hand, the EventType, DataType and ConstantVariables constraints are invariants, i.e. they always remain true throughout the behavior of the system. As such, we opt to specify these invariant constraints in the static view of the FOREST model, i.e. on class diagrams elements. On the other hand, the InitialValue constraint, PreCondition and PostCondition constraints depend on the behavioral status of the model, i.e. their value may change overtime. These types of constraints are expressed on the dynamic view of the FOREST model, i.e. on protocol state machine elements. Constraints are specified as Boolean expressions. In our context, the six types of expressions are constraints expressions, so all OCL-F constraints are of type Boolean.

In the following subsections, the six constraints types are defined in three parts:

- The *contextual classifier* defines the namespace in which the expression is evaluated. The elements that can be constrained in the OCL expression are identified based on the contextual classifier. The *self instance* is the reference to the object that evaluates the expression. It is always an instance of the contextual classifier. OCL-F does not support the use of `self`, in OCL-F the token `self.` is not permitted. During the translation to Alloy expressions, the context is restored.
- *Specific concrete syntax rules.* Each constraint type has a particular concrete syntax based on the OCL-F grammar. More precisely, some of the expressions in the OCL-F are not allowed to be used for certain types of constraints. For instance, it does not make sense to allow the analyst to specify that a message is sent within an InitialValue constraint, thus the expression `OclMessageExpCS` is not allowed within InitialValue constraints.
- The *placement* is the position where the OCL-F expression is specified in the FOREST model. Each type of constraints have its own placement. Thus from the placement of a constraint expression, it can be deduced its constraints type. The placement of the expression is fundamental information during the translation of OCL-F expressions in Alloy. This translation is described later in Section 4.5.6.

EventType constraint type

Contextual classifier. The contextual classifier of an EventType constraint is the UML Operation that characterizes the constrained event. In consequence, the visible elements that can be accessed within the expression of this type of constraints are the parameters of the constrained operation. For instance this means that in Figure 4.12 the EventType constraints `EVT_CONS1` and `EVT_CONS2` on the event reception `evt1` can only express a constraint on its parameter `p` and elements accessible from the parameter `p`.

Specific concrete syntax rules. An EventType constraint is an `OclExpressionCS` expression of Boolean type. EventType constraints are static constraints and do not express behavioral constraints, that is why the expression `OclMessageExpCS` (i.e. sending of an event) and the '`@pre`' operator (reference to a previous value in time) within `AssociationEndCallExpCS` expression are not allowed.

Placement. EventType constraints are placed on the right-side of the UML `Operation` related to the event reception which they constrain. The constraint expression is surrounded by an opening curly bracket '`{`' and a closing curly bracket '`}`'. These brackets are not considered as part of the constraint expression, they are visual markers for the start and end of the EventType constraints. In the case of a conjunction of constraints for the same event reception, the set of constraints is surrounded by the brackets and each constraint is separated by a comma symbol

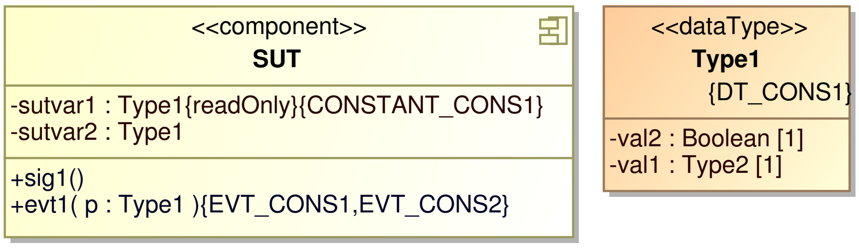


Fig. 4.12: FOREST concrete syntax: constraints on the static view of FOREST models

','. The comma symbol, is also a visual marker which is not part of the expression but exists only to ease the readability of the constraint on the UML/FOREST models.

DataType constraint type

Contextual classifier. The contextual classifier of a DataType constraint is the data type that it constrains. In consequence, the visible elements that can be accessed within the expression of this type of constraints are the attributes of the data type. For instance this means that in Figure 4.12 the DataType constraint `DT_CONS1` on the data type `Type1` can only express a constraint on its attributes `val1` and `val2`.

Specific concrete syntax rules. The same specific concrete syntax rules of the EventType type of constraints apply to the DataType type of constraints.

Placement. The placement of DataType constraints is below the name of the DataType within the upper compartment of the data type classifier. The DataType constraints are also surrounded by curly brackets and separated by commas in case of multiple constraints on the same data type (see comments for the placement of EventType constraints).

ConstantVariable constraint type

Contextual classifier. The contextual classifier of a ConstantVariable constraint is the UML `Attribute` characterizing the constrained state variable. The contextual classifier must be a constant state variable. The visible element that can be accessed within the expression of this type of constraints is the constrained attribute itself.

Specific concrete syntax rules. The same specific concrete syntax rules of the EventType type of constraints apply to the ConstantVariable type of constraints.

Placement. The placement of ConstantVariable constraints is on the most right-side (on the right-side of the `{readOnly}` element) of the UML attribute characterizing the constrained state variable. The ConstantVariable constraints are also surrounded by curly brackets and separated by commas in case of multiple constraints on the same data type (see comments for the placement of EventType constraints).

InitialValue constraint type

Contextual classifier. The contextual classifier of the InitialValue constraint of a FOREST model is the SUT component classifier. In consequence, the visible elements that can be accessed within

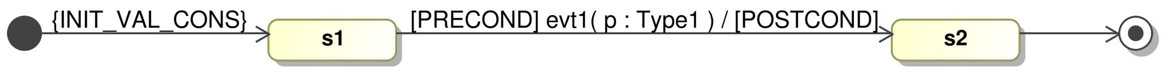


Fig. 4.13: FOREST concrete syntax: constraints on the dynamic view of FOREST models

the expression of this type of constraints are: all the state variables of the SUT component; and through the association between the SUT and the Test System Component, all the state variables of the Test System Component. The constant state variables and the event receptions of the SUT component and of the Test System Component are not visible (i.e. accessible) within an InitialValue constrain. For instance this means that in Figure 4.13 the InitialValue constraint `INIT_VAL_CONS` may express constraints on the state variables `statevar2`, `testvar1` and `testvar2`.

Specific concrete syntax rules. This constraint initializes some of the state variables declared in the static view. As its purpose is the initialization of variables, the only `ComparisonOperator` allowed within the InitialValue constraint is the equal '`=`' operator. Like the three previous constraint types, the use of `OclMessageExpCS` expression and the use of the '`@pre`' operator are not allowed.

Placement. The InitialValue constraint is placed on the arrow that marks the initial protocol state in the dynamic view of FOREST models. The InitialValue constraint is surrounded by square brackets ('`[`' and '`]`').

PreCondition constraint type

Contextual classifier. The contextual classifier of PreCondition constraints is the SUT component. The visible elements that can be accessed within the expression of this type of constraints are: all the state variables (constant and non-constant) from the SUT and the Test System components; as well as the parameter values of the event that triggered the protocol transition related to the PreCondition constraint. For instance this means that in Figure 4.13 the PreCondition constraint `PRECOND` labeling the protocol transition from protocol state `s1` to protocol state `s2` may express constraints on the state variables `statevar1`, `statevar2`, `testvar1`, `testvar2` and on the parameter `p` of the event `evt1`.

Specific concrete syntax rules. The same specific concrete syntax rules of the EventType type of constraints apply to the PreCondition type of constraints.

Placement. PreCondition constraints are parts of protocol transition labels. PreCondition constraints are specified in the first position of the label before the name of the event that triggers the labeled protocol transition, as shown in Figure 4.13. PreCondition constraints are surrounded by square brackets.

PostCondition constraint type

Contextual classifier. The contextual classifier of PostCondition constraints is the SUT component. The visible elements are the same elements than for the PreCondition constraint (see previous subsection).

Specific concrete syntax rules. PostCondition constraints may use all the expressions of the OCL-F expressions as defined in Figure 4.11. The two expressions not allowed in other expressions ('`@pre`' operator and `OclMessageExpCS`) are of particular importance in the PostCondition constraints. PostCondition constraints specify the changes in the status of the SUT and the Test System. These changes are of two kinds. Firstly, a change in data values, this is specified with the help of the '`@pre`' operator which enables specifying a current data value with respect to its previous values, i.e. the values they had before the protocol transition is taken. Secondly, a change in the event exchanged between the SUT and the Test System. The reception of an event is specified within the protocol transition as an UML `Trigger`. But the sending of an event can not be specified with elements from UML protocol state machine. OCL helps filling this gap by offering a way to specify the sending of a message in particular via the expression `OclMessageExpCS`.

Placement. PostCondition constraints are parts of protocol transition labels. PostCondition constraints are specified in the last position of the label after the name of the event that triggers the labeled protocol transition, as shown in Figure 4.13. A slash symbol '`/`' separates the event name and the PostCondition constraint. The PostCondition constraint is surrounded by square brackets.

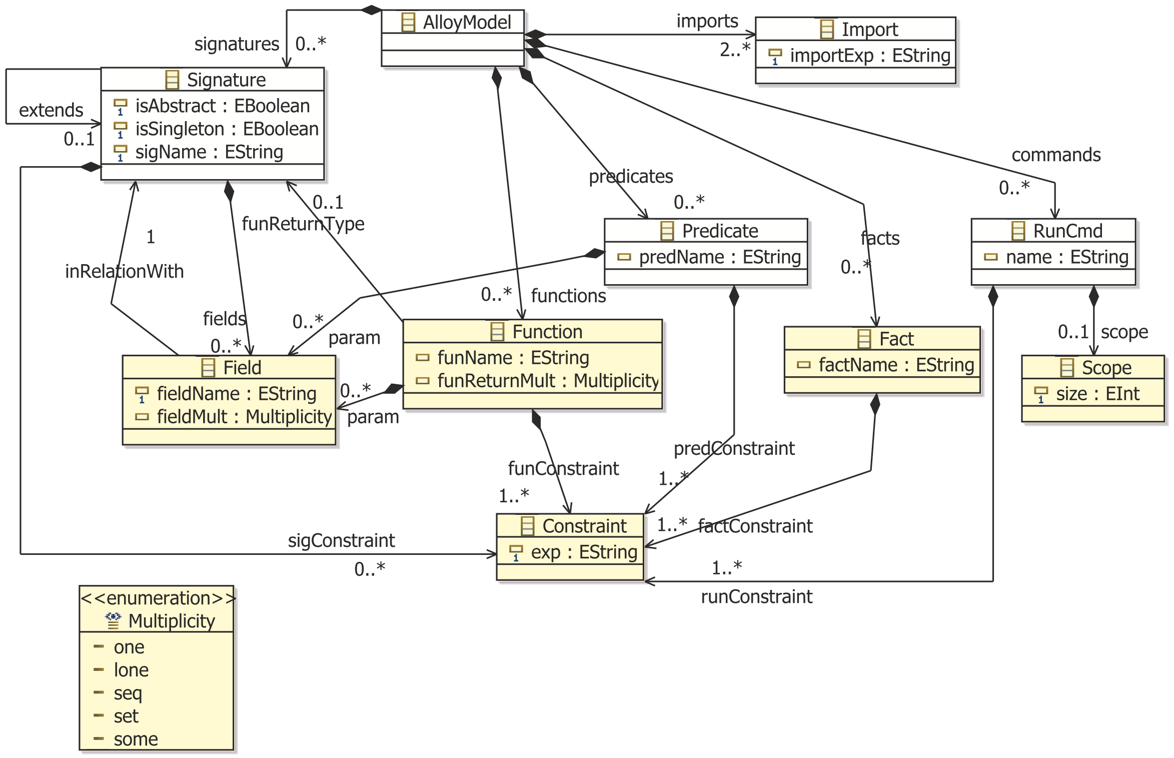


Fig. 4.14: Alloy metamodel in the SESAME approach

4.4 Introduction to Alloy: concepts and syntax

In this section, a short introduction to the Alloy approach is presented. A detailed presentation of the Alloy approach is available in [Jac06]. The Alloy approach is based on three key elements: a logic (fundamental concepts), a language and an analysis.

The logic introduced in the Alloy approach is a relational logic. It combines the quantifiers of first-order logic with the operators of the relational calculus. The fundamental concepts needed to express the semantics of FOREST models are the atoms and the relations. As defined in [Jac06]. An *atom* is a primitive entity that is indivisible, immutable, and uninterpreted. A relation is a structure that relates atoms. It consists of a set of tuples, each tuple being a sequence of atoms. Relations in Alloy may be of any size and of arity greater or equals to one. A relation with no tuples is said to be empty. For instance, unary relations (i.e. with arity one) are sets of atoms; unary relations with a single tuple are scalars.

The abstract syntax of the Alloy language is precisely defined by a BNF grammar in [Jac06]. The Alloy meta-model given in Figure 4.14 is a graphical representation of the subset of the BNF grammar definition used for defining the semantics of FOREST models. It describes precisely the syntactical elements of Alloy model concepts and their inter-relations.

4.4.1 Signatures and their fields

A *signature* is a set of atoms, i.e. it is a unary relation. It can be declared as a subset of another set of atoms with the help of the keyword `extends`. Multiplicity constraints can be placed before the signature declaration to restrict the number of atoms in the declared signature.

For instance, in Figure 4.15, the event reception `restoreCard` signature comprises exactly one atom. As the `restoreCard` event does not convey any data, each event reception of that type is considered to be the same event reception.

```
abstract sig SUTEvent {}
one sig restoreCard extends SUTEvent {}
sig authenticate extends SUTEvent {
  p : one PIN
}
sig PIN {
  val : seq Digit
}
```

Fig. 4.15: Alloy concrete syntax: signature - extract from the ATM example

An *abstract signature* has no atoms except the ones of the signatures extending it. An abstract signature is specified by placing the `abstract` keyword in front of the signature declaration as illustrated for the `SUTEvent` signature in Figure 4.15. In this same figure, the `SUTEvent` abstract signature comprises a set of atoms defined by the union of the singleton set `restoreCard` and the set of atoms from `authenticate`.

A relation in Alloy is declared as a *field* of a signature. In our context, we restrict the relation declarations to binary relations. The syntax for declaring binary relations is as follows `sig S1 {f: mult S2}`. where `f` is a binary relation between atoms of the signature `S1` and atoms of the signature `S2`. In this expression `mult` constrain the multiplicity of the relation `f`. Multiplicities used in our context are as follows:

- *Exactly one.* `sig S1 {f: one S2}` means that the binary relation f relates each atoms of S1 with exactly one atom of S2.
- *Zero or one.* `sig S1 {f: lone S2}` means that the binary relation f relates each atoms of S1 with at most one atom of S2.
- *One or more.* `sig S1 {f: some S2}` means that the binary relation f relates each atoms of S1 with at least one atom of S2.
- *Any.* `seq`: `sig S1 {f: seq S2}` means that the binary relation f relates each atoms of S1 with any number of atoms of S2. In addition, the tuples of the relation `f` are ordered.

For instance, in the ATM example, the two signatures illustrated in Figure 4.15 are derived from the event reception authenticate and the PIN data type. The signature authenticate represents a set of atoms each of these atoms being in relation with exactly one PIN number. It is also specified that the atoms of the authenticate signature form a subset of the SUT events. The signature PIN illustrates an example of a different multiplicity. A PIN signature is a set of atoms (representing pin numbers) each atom being in relation with a sequence (keyword `seq`) of atoms from the Digit signature.

4.4.2 Constraint expressions

Operators

Alloy enables the specification of constraints on the atoms and relations declared in the model. The Alloy set operators that we use in our context are the union ('`+`'), the difference ('`-`'), the subset (keyword `in`), and the equality ('`=`'). These set operators can be applied to any pair of relations of same arity. Given p and q two relations of same arity:

- a tuple is in `p + q` when it is in p or in q (or both)
- a tuple is in `p - q` when it is in p but not in q
- `p in q` is true when every tuple in p is also a tuple in q
- `p = q` is true when p and q have the same tuples

In order to express constraints on the relations, a relational calculus is available that provides a number of operators. In our context, we use only the *join* relational operator, also called *Dot Join* in Alloy textbooks. It is represented by a dot symbol '`.`'. The join operator may be used in a number of contexts depending on the arity of the two relations used in the join. In our context, we use the join operator in the particular case, when p is a unary relation (i.e. a set) and q is a binary relation. In that case, `p.q` is the image of the set `p` under the relation `q`. This use of the join is particularly important to "navigate" between the concepts.

Let's illustrate it with the `authenticate` unary relation and its signature field `p` binary relation, shown in Figure 4.15. For instance, let's suppose the unary relation `authenticate` contains the following tuples (authenticate1), (authenticate2), (authenticate3) and the binary relation `p` contains the following tuples (authenticate1, PIN1), (authenticate2, PIN1), (authenticate3, PIN2). `authenticate.p` results in the unary relation containing the tuples (PIN1) and (PIN2).

Constraint quantifiers

The allowed expressions of these constraints use the following logical operators: the negation, conjunction, disjunction and implication. The respective keywords for these operators are `not`, `and`, `or` and `implies`. Alloy constraints may be quantified. The possible forms of quantification are:

- `all x: e | F` meaning that F holds for every x in e
- `some x:e | F` meaning that F holds for at least one x in e
- `no x:e | F` meaning that there is no x in e for which F holds

The three different types of Alloy quantifications used in SESAME are illustrated in Figure 4.16. The first expression illustrates the "for all" type of quantification. It states that for all statuses atoms in the `Status` signature, the atom resulting from the navigation through the field `ts` and its field `MIN_pt_tr5` is equal to 1. The second expression illustrate the "for some" type of quantification. It states that there is at least one atom (i.e. for some atoms) in the `Status` signature for which the event received is one of the atoms of the signature `serverPermission`. The last expression illustrates the "for no" type of quantification. It states that for no two disjoints atoms in the signature `withdraw` the two atoms have the same value for their respective field `cw`.

```
all status : Status |  status.ts.MIN_pt_tr5 = 1
some status : Status | status.nextStep.eventReceived in serverPermission
no disj evt, evt’: withdraw | evt.cw = evt’.cw
```

Fig. 4.16: Alloy concrete syntax: constraint quantifiers

4.4.3 Facts, predicates and functions

Facts are invariants, they express Boolean expressions evaluated to true. *Predicates* are Boolean expressions that can be called from other expressions within the Alloy model (predicates may evaluate to false). *Functions* are predicates returning a set of atoms; they may also be called from other expressions within the Alloy model. Predicates and functions maybe parameterized (unlike facts). Predicates always return Boolean values. Functions may be evaluated to any signature (or even set of signatures) of the Alloy model.

Let’s illustrate these three concepts with the Figure 4.17 illustrating an extract of a possible Alloy model for the ATM example. The fact `dt_PIN` is an invariant: the fact that all atoms of the set of PIN numbers have four digits must be true all the time. This is not the case for predicates, e.g. the predicate `preT4` describing the precondition constraint of the protocol transition T4, may be sometimes evaluated to true sometimes evaluated to false, depending on the status given as parameter to the predicate. The function `enabledTransitions` is not a Boolean expression, its type is a subset of the set of `PTransition` atoms. This subset is constrained to contain only the `PTransition` atoms satisfying the expression described in the body of the function.

```
fact dt_PIN {
  all dt: PIN | #dt.val = 4
}
pred preT4(status: Status){
  status.nextStep.eventReceived.p = status.ATM.card.pin
  status.TS.ctrOP2 < status.TS.MAXop2
}
fun enabledTransitions(status:Status): set PTransition {{
  t : PTransition | {
    (t.source = status.curPState)
    and (status.nextStep.eventReceived = t.trigger)
    and precond[status, t]
  }
}}
```

Fig. 4.17: Alloy concrete syntax: fact, predicate and functions - extract from the ATM example

4.4.4 Commands

Commands are not used for giving the semantics of FOREST models; they may be used to instruct the execution of the formal analysis of the model. In particular, in SESAME processes, Alloy commands are used for test selection evaluation, and test generation.

4.5 FOREST semantics

The algorithm below defines the translation from FOREST models into Alloy models as a model transformation. The first sections 4.5.1, 4.5.2 and 4.5.3, describe invariant parts of the Alloy model. These sections describe model transformations, which are independent from the FOREST model, to translate and provide a constant result. The next sections 4.5.4, 4.5.5 and 4.5.6 describe model transformations which are dependent from the input FOREST model.

Given a FOREST model m_f in M_{forest}, a model m_a in M_{alloy} is created by the steps desdibed in the following sections. With M_{forest} representing the set of all models conforming to the FOREST meta-model given by Figure 4.3 and M_{alloy} the set of all models conforming to the Alloy meta-model given by Figure 4.14.

4.5.1 Import declarations

The import declaration of Alloy specifications of FOREST models are composed of two lines, as illustrated in Figure 4.18. First line imports the Boolean library, in order to be able to use Boolean variables within the Alloy specification. Second line declares a linear ordering over the atoms in `STATUS`. This linear ordering specifies one simulation of the FOREST model (see the behavioral concepts definition in section 4.2.2). The `STATUS` signature is declared in the following sub-section 4.5.3.

```
open util/boolean
open util/ordering[STATUS] as statusSeq
```

Fig. 4.18: Alloy import declaration of FOREST models

4.5.2 FOREST structural concepts

The concepts of the FOREST meta-model are specified in Alloy as abstract signatures. The translation of the concepts of the FOREST meta-model is independent from the FOREST model under translation.

Abstract signatures are well-suited because they define a set of elements that must be instantiated. The SUT component is specified as `SUT_COMP` abstract signature and the Test System component as the `TS_COMP` abstract signature. The events received by the SUT and by the Test System are specified as two abstract signatures: `SUTEvent` and `TSEvent`.

The protocol state is specified as an abstract signature `PState`. The final state is specified in the meta model independently from the presence or not of a final state in the static view of the FOREST model in order to define the behavioral concepts of FOREST models. The semantics of a FOREST model without final state would still be preserved even if the final state is defined in the Alloy model, because there would not be any protocol transitions reaching this final state and thus the final state would be as non-existent.

A protocol transition, as specified in lines 07-11 of Figure 4.19, is in relation with zero or one triggering event `SUTEvent` and two protocol states `PState`, one state being the source of the transition, and the other one the target of the transition. A constraint, in line 12, is specified on the signature `PTransition` which allows that a transition does not have any triggering event only if its target state is the final state.

The other FOREST structural concepts (`StateVariable`, `DataType`, `TypeAttribute`, `EventParameter` and `Constraint`) are directly specified with existing Alloy constructs.

```
abstract sig SUTComp{}
abstract sig TSComp{}
abstract sig SUTEvent{}
abstract sig TSEvent{}
abstract sig PState{}
one sig ps_final extends PState{}
abstract sig PTransition{
  source : one PState,
  target : one PState,
  trigger : lone SUTEvent
} {
  no trigger implies target = ps_final
}
```

Fig. 4.19: Metamodel of FOREST structural concepts

4.5.3 FOREST behavioral concepts

The FOREST behavioral concepts described in Section 4.2.2 are translated in Alloy with the two signatures `Status` and `Step`, the fact `step`, the function `enabledTransitions` and the predicate `UnchangedStatus`, as shown in Figure 4.20. A particular attention is given to the status(es) for which the current protocol state is the final state.

In Figure 4.20 the lines 01-06 specify the status behavioral concept. The status concept is specified as a `STATUS` abstract signature which contains four fields. The first three fields correspond to respectively, the active protocol state `curPState`, the values of the SUT state variables `sut` and the values of the Test System state variables `ts`. In order to ease the specification of some constraints a status references the information related to the next step `nextStep`.

A step contains three pieces of information. Firstly, a step contains the selected protocol transition `selectedPTransition` that will be taken to reach the next `STATUS`. Secondly, it contains the event that is received by the SUT `eventReceived` from the Test System. The multiplicities of `selectedPTransition` and `eventReceived` are zero to one (keyword lone). The multiplicity zero is allowed if and only if the current protocol state is the final state, as defined in lines 07 and 08 of Figure 4.20, i.e. for any status where the current state is not the final state there must be an event received and a selected transition. Thirdly, it *may* contain the sending of an event `eventSent` by the SUT to the Test System, the keyword `lone` states that zero or one event is sent.

The fact `step` states how the ordering of STATUS is established by relating any `STATUS` (except the last one) in the sequence of statuses with its next status. Line 16 states that for every `STATUS` except the last status, a status named `status'` is the next status of `status` and that the following must be true:

- If the current active protocol state is not the final state then the three following statements must be true

```
sig Status {
  curPState : one PState,
  sut : one SUTComp,
  ts : one TSComp,
  nextStep: one Step
{
  (curPState = ps_final) iff ( no nextStep.selectedPTransition )
  (curPState = ps_final) iff ( no nextStep.eventReceived )
}
sig Step {
  selectedPTransition : lone PTransition,
  eventReceived : lone SUTEvent,
  eventSent : lone TSEvent
}
fact step {
  all status : Status - statusSeq/last | let status' = statusSeq/next[status] {
    (status.curPState != ps_final) implies {
      status.nextStep.selectedPTransition in enabledTransitions[status]
      status'.curPState = status.nextStep.selectedPTransition.target
      postcond[status, status']
    }
    (status.curPState = ps_final) implies UnchangedStatus[status, status']
  }
}
pred UnchangedStatus(status, status': Status) {
  status'.curPState = status.curPState
  status'.sut = status.sut
  status'.ts = status.ts
  status'.nextStep = status.nextStep
}
fun enabledTransitions(status:Status): set PTransition {
  {
    t : PTransition | {
      (t.source = status.curPState)
      and (status.nextStep.eventReceived = t.trigger)
      and precond[status, t]
    }
  }
}
fact noUnusedTransitions{
  all t : PTransition | some status : Status | status.nextStep.selectedPTransition = t
}
fact noUnusedEvents{
  all evt : SUTEvent | some status : Status | status.nextStep.eventReceived = evt
  all evt : TSEvent | some status : Status | status.nextStep.eventSent = evt
}
```

Fig. 4.20: Metamodel of FOREST behavioral concepts

(1) the chosen protocol transition is selected randomly in the set of enabled transitions computed by the function `enabledTranstions`. (line 18)

(2) the protocol state of the next status `status'` is the target protocol state of the chosen protocol transition. (line 19)

(3) the post-condition associated with the selected protocol transition holds (line 20), i.e. the predicate `postcond` is true given the status and its next status as parameters. This predicate is specified in details in the following subsection 4.5.6 which describes how the different types of model constraints are expressed in Alloy.

- If the current active protocol state is not the final state then the predicate `UnchangedStatus` (described below) must be true.

The predicate `UnchangedStatus` takes two statuses as parameters, a status and its successor status. It constrains the successor `status'` to have the same attributes values than its predecessor `status`. The predicate is defined in the lines 25 to 30 of Figure 4.20.

The set of enabled transitions for a given `status` is specified in the function `enabledTransitions` (lines 31-39). The body of the function is surrounded by additional curly brackets `{}` (line 32 and line 38) in order to construct a set, as the result type of the function is a set. The function returns a set of protocol transitions `PTransition`. The protocol transitions part of the resulting set must satisfy the three following properties:

(1) the source state of the protocol transition must be the current protocol state of `status`, (line 34)

(2) the trigger of the protocol transition must be the event that is going to be received by the SUT at the next step (line 35)

(3) and the pre-condition constraint associated with the protocol transition must be true for the current `status` (line 36). The pre-condition is specified in the predicate `precond` described in more details in the following subsection 4.5.6

Two additional constraints are specified on the protocol transitions and protocol states. Fact `noUnusedTransitions` specifies that the set `PTransition` contain only transitions that are chosen in at least one `STATUS`. Fact `noUnusedEvents` specifies that the set `SUTEvent` contain only events received by the SUT in at least on STATUS and that the set `TSEvent` contain only events received by the Test System in at least on STATUS.

4.5.4 Model: static view

The FOREST structural concepts are formalized as abstract signatures. The elements of a FOREST model are instances of the FOREST structural concepts, as such in Alloy they are specified as signatures extending the abstract signatures declared in the previous sections. Unlike the three previous sections whose results are independent from the input FOREST model, the result from the translation steps, described in the remaining subsections, depend on the FOREST input model.

```
sig ATM extends SUTComp {
  card : lone BankCard,
  attempts : one Int
}
```

Fig. 4.21: ATM example: Alloy specification of the SUT state variables

```
fact distinctEvents {
  no disj evt, evt': restoreCard |
  no disj evt, evt': withdraw | evt.cw = evt'.cw
  no disj evt, evt': insertCard | evt.bc = evt'.bc
  no disj evt, evt': authenticate | evt.p = evt'.p
  no disj evt, evt': serverPermission | evt.sp = evt'.sp and evt.csp = evt'.csp
  no disj evt, evt': deliverCash | evt.cdc = evt'.cdc
  no disj evt, evt': requestPermission | evt.crp = evt'.crp
}
```

Fig. 4.22: ATM example: Alloy specification of the WFR FOREST.5 for events

SUT and Test System components and their state variables

The `SUTComponent` and the `TSComponent` of the m_f FOREST model are specified as signatures extending respectively `SUTComp` and `TSComp`. The specified signatures are given the names of the `SUTComponent` and of the `TSComponent` in m_f.

The state variables associated with the SUT component and with the Test System component of the m_f FOREST model are translated as fields of their respective signatures. Each state variable, is specified by stating the name of the `StateVariable` followed by a colon (:), followed by its `Dataype` name, and ended by a comma (,). There are three special cases to care of. Firstly, the integer data type is named `Int` in Alloy (instead of `Integer` in UML), thus when an integer state variable is translated into Alloy its type is set to `Int`. Secondly, for all non-integer state variables an additional keyword `lone` is inserted between the semi-colon and the datatype name. This is because all integers are initialized in FOREST models, and thus it does not make sense to have an integer with no value. Thirdly, the last state variable specified does not end with a comma, this is due to the concrete syntax of field declaration in Alloy.

Figure 4.21 illustrates the resulting Alloy specification of the state variables from the SUT component of the ATM example. The two state variables `card` and `attempts` are declared as fields of the `ATM` signature.

Events receptions

The events received by the SUT (`SUTEvent` in the FOREST meta-model) are specified in the Alloy model m_a as signatures extending the `SUTEvent` abstract signature. Similarly, the events received by the Test System (`TSEvent` in the FOREST meta model) are specified in the Alloy model m_a as signatures extending the `TSEvent` abstract signature. Each parameter of a given event is specified as a field of its related signature. For instance, the `withdraw` event reception of the `SUTComponent` is specified in Alloy as shown in Figure 4.23.

```
sig withdraw extends SUTEvent {
  cw : one CashAmount
}
```

Fig. 4.23: ATM example: Alloy specification of the event withdraw

```
01 sig BankCard {
02   pin : one PIN
03 }
04 sig PIN {
05   val: seq Digit
06 }
07 abstract sig CashAmount{}
08 one sig LargestCA extends LargeCA{}
09 abstract sig Enum{}
10 one sig EnumLiteral1, EnumLiteral2 extends Enum{}
```

Fig. 4.24: ATM example: Alloy specification of the BankCard and PIN datatypes

In addition to the specification of the event receptions and their related parameters, an invariant constraint must be expressed on the Alloy model that expresses the "WFR FOREST.5", as defined in Section 4.2.1. The WFR states that there are no two disjoint events with the same parameter values, in Alloy it is written as in Figure 4.22.

Datatype

Four kinds of data types may be specified in FOREST models: structured data types, abstract data types, leaf data types and enumerations. Each of this kind of data types is transformed in Alloy in a specific way.

The data types specified in the FOREST model m_f are transformed in m_a as signatures. For each data type dt in m_f:

- For all kinds of data types: a signature *sig* is created in m_a and named like the data type dt. If dt is associated with a Datatype $dt2$ through an `extends` association, then *sig* extends the signature named $dt2$, as for instance in line 08 of Figure 4.24 where the signature `LargestCA` extends the signature `LargeCA`. Depending on the attributes of dt inherited from the Datatype meta class, namely: `isLeaf`, `isAbstract`, `isEnum`, the declaration of the signature *sig* will be specified as follows:
  - If the attribute `isAbstract` of s_f is true then *sig* is an abstract signature, as illustrated in line 07 of Figure 4.24.
  - If the attribute `isLeaf` of s_f is true then *sig* is a singleton signature, as illustrated in line 08 of Figure 4.24.
  - If the attribute `isEnum` of s_f is true then *sig* is an abstract signature, as illustrated in line 09 of Figure 4.24. Then, as many signatures are created in m_a as the number of literals in the enumeration dt. Each signature is given the name of the enumeration literal and extends the signature *sig*, as in line 10 of Figure 4.24.

- The attributes, if any, of the datatype *dt* are translated into fields of the signature *sig* with their associated data type name. For instance, in the ATM example, the `BankCard` structured data type is composed of one attribute `pin` of type `PIN`. The `BankCard` data type is translated as shown in lines 01 to 03 of Figure 4.24. Structured datatypes may have various kinds of multiplicities. By default the multiplicity is set to exactly one, as for instance with the `pin` attribute of the `BankCard` data type. In the following cases the multiplicity of the field of the signature *sig* is different than one:

  (1) If the lower bound value of *dt* is zero and the upper bound value of *dt* is one then the multiplicity `lone` is associated with the field to be created with signature *sig*. The lower bound and upper bound values of *dt* are defined respectively in its attributes `lowerMultiplicity` and `upperMultiplicity`.

  (2) If the lower bound of *dt* is one and the upper bound of *dt* is greater than one then the multiplicity `some` is associated with the field to be created with signature *sig*. The constraint that this sequence contains exactly four digits is described with the model constraints in the following section 4.5.6. The exact upper bound multiplicity value may be defined as a Datatype constraint. Datatype constraints are described in the following section 4.5.6.

  (3) If the lower bound of *dt* and the upper bound of *dt* are both different than one and do not satisfy the two previous cases then the multiplicity `seq` is associated with the field to be created with signature *sig*. The exact multiplicity values (upper and lower bounds) may be defined as a Datatype constraint.

4.5.5 Model: dynamic view

Protocol states

A protocol state of a FOREST model is translated as a singleton signature extending the protocol state `PState` abstract signature. For each protocol state *ps* in m_f (except for the state named "final", if any, already defined with the meta model concepts) corresponds such a signature in m_a. All signatures names in an Alloy model must be unique. In order to reduce the likeliness of conflicts in signature names, all the names of the protocol state signatures are the names of *ps* prefixed by the string "`ps_`". The FOREST model of the ATM example contains four protocol states declared as five singleton signatures, as illustrated by Figure 4.25.

```
one sig ps_waitForCard extends PState{}
one sig ps_waitForAuthentication extends PState{}
one sig ps_Authenticated extends PState{}
one sig ps_waitForPerm extends PState{}
```

Fig. 4.25: ATM example: declaration of the protocol states

Protocol transitions

A protocol transition of a FOREST model is translated as a signature extending the protocol transition `PTransition` abstract signature. For each protocol transition in m_f corresponds a signature in m_a. Names of the protocol transition signatures are names of the `PTransition` from the m_f model prefixed by the string "`pt_`". The three fields `source`, `trigger` and `target` of the abstract signature `PTransition` are set in the constraint of the transition signature.

In the ATM example, there are eight protocol transitions specified in the protocol state machine. Figure 4.26 illustrates the specification of protocol transition `pt_tr1` whose source protocol state is `ps_waitForCard` and target protocol state is `ps_waitForAuthentication`. This protocol transition is triggered by the event reception `insertCard`.

```
sig pt_tr1 extends PTransition{}{
  source = ps_waitForCard
  trigger in insertCard
  target = ps_waitForAuthentication
}
```

Fig. 4.26: ATM example: declaration of a protocol transition

4.5.6 Constraints on the Model

Constraints in the input FOREST model m_f are expressed using OCL-F expressions as defined previously in Section 4.3.3. Concrete syntax of the restricted set of OCL expressions (referred to as `Constraints` in the FOREST meta model shown in Figure 4.3) and Alloy expressions used in SESAME processes share commonalties. Still, some differences remain. Thus OCL expressions must be interpreted and rewritten into Alloy-compliant expressions to produce valid Alloy models. The context of the OCL expression is implied by the UML element it is associated with. One of the main difficulty in translating OCL expressions in Alloy is to explicitly state the context of the expressions.

General translations rules of OCL-F expressions into Alloy expressions

In this subsection, we describe a number of translation rules which must be performed on all OCL-F expressions that must be translated into Alloy. These translation rules are independent from the FOREST model m_f given as input of the semantic model transformation. They are mainly due to differences in the concrete syntax of the OCL and the Alloy languages. For each OCL-F expression in m_f the five following general translations rules must be performed before the model transformations in the following subsections are performed.

(1) The logical operator *is-different-than* is written "<>" in OCL-F and must be replaced by its equivalent "! =" in Alloy. So for any constraint `c` in m_f, the related constraint in m_a will not contain "<>" but "! =" instead.

(2) Another type of OCL expression that must be taken special care of is when a variable `var` is compared to the particular `NULL` value, e.g. `var = NULL`. The NULL value is not supported in Alloy, we suggest translating the assignment of a variable to a NULL value in OCL-F into the statement that the variable has no value in Alloy. In Alloy, this is specified by using the keyword `no` followed by the name of the signature, e.g. `var = NULL` in m_f is translated into `no var` in m_a.

(3) The Boolean values `var = true` and `var = false` in OCL-F must be translated in their equivalent `var.isTrue` and `var.isFalse` in Alloy.

(4) OCL-F expressions of the type `IfExpCS`, as described in 4.11, are translated in Alloy with the `implies` and `else` keywords. For instance in the ATM example, the OCL-F expression `If (sp=true) then ts^deliverCash(cw)` should be translated into `(sp.isTrue) implies ts^deliverCash(cw)`; note that in that case two general translations rules have been applied, number (3) and (4).

(5) An OCL-F expression using the `oclIsKindOf` operator are translated using the `in` keyword in Alloy. For instance, in the ATM example, `cw.oclIsKindOf(SmallCA)` is translated in the Alloy expression `cw in SmallCA`.

EventType constraints

Given an event `evt` in m_f, a constraint of event `evt` is constructed by creating a fact in m_a. The fact is expressed as the conjunction of all the constraints associated with the event. In m_a, all event parameter names are prefixed with "`evt.`". Each constraint is declared on a single line separated by an "`and`" logical operator. This conjunction of expressions is surrounded by starting "`{`" and ending curly brackets "`}`" and prefixed by `all evt: evtType` with evtType being the type of the event being constrained. For instance, as illustrated in Figure 4.27, the constraint associated with the `insertCard` event in m_f is transformed in m_a as a fact starting with `all evt: insertCard` and with the parameters name `bc` prefixed with `evt`.

```
fact {
  all evt: insertCard | {
    evt.bc.pin.digits[0] = D1
    and evt.bc.pin.digits[1] = D2
    and evt.bc.pin.digits[2] = D3
    and evt.bc.pin.digits[3] = D4
  }
}
```

Fig. 4.27: ATM example: event parameter constraint on the insertCard event

DataType constraints

In a similar way than the event constraints, all constraints associated with a data type are translated into a fact that represents the conjunction of these constraints. This conjunction of constraint is specified as holding for all values of the data type.

As mentioned previously about the translation of data types in section 4.5.4, there is a special case of multiplicities, when the lower bound and/or the upper bound are different than zero or one. These kinds of multiplicity values are specified with a DataType constraint. In Alloy, multiplicities are specified with the symbol `#`. This symbol constrains the number of atoms of a signature. In our case, we constrain the number of atoms of the field of the data type signature. For instance, in the ATM example as illustrated in Figure 4.28, the field `digits` of the datatype `PIN` is restricted to have four atoms, so that only sequences of four digits are allowed.

```
fact {
  all dt: PIN | {
    #dt.digits = 4
  }
}
```

Fig. 4.28: ATM example: datatype constraint on the PIN datatype

Constant variables constraints

The specification of a constant value of state variables is done by providing a fact that states that for all statuses of a model simulation, the constant state variable will be equals to a certain value. For instance, `MAX_evt_authenticate` is a constant state variable in the ATM example; Figure 4.29 shows how the constant value of this state variable is specified.

```
fact constant_MAX_evt_authenticate {
  all status : Status | status.ts.MAX_evt_authenticate = 2
}
```

Fig. 4.29: ATM example: Constant constraint on state variables

Initial status

The initial status specifies the constraints on the first `STATUS` of the sequence of statuses `statusSeq`. In m_a it results in the creation of one fact `initialStep` specifying these constraints. The fact is split in two parts. Firstly, the initial protocol state of the protocol state machine is set. This is done by adding the expression `statusSeq/first.curPState =`" followed by the protocol state name of the initial protocol state. Secondly, the initialization constraints of the non-constant state variables of the SUT and Test System component signatures are specified. For each state variable `InitialValue` constraint, the state variable name is prefixed by "`statusSeq/first.`" in order to state that this variable is set to this at the first status of the sequence of statuses.

In the ATM example the initial state is `ps_waitForCard`. Line 02 in Figure 4.30 illustrates how the initial protocol state is specified. Lines 03-07 show how the state variables are initialized.

```
01 fact initialStatus{
02   statusSeq/first.curPState = ps_waitForCard
03   no statusSeq/first.sut.card
04   statusSeq/first.sut.attempts = 0
05   statusSeq/first.ts.ctr_evt_authenticate = 0
06   statusSeq/first.ts.ctr_pt_tr5 = 0
07   statusSeq/first.ts.ctr_ps_waitForPermission = 0
08 }
```

Fig. 4.30: ATM example: initialization of the first status

Final status

The final status is the last status of the sequence of status `statusSeq`. There are two constraints for the final status which are expressed as an Alloy fact. The `finalStatus` fact is specified as shown in Figure 4.31. This fact is independent from the FOREST model m_f. The first constraint is that the current protocol state in the final status must be the final state. The second constraint is that there is no event sent to the Test System during the final status.

```
fact finalStatus {
  statusSeq/last.curPState = ps_final
  no statusSeq/last.nextStep.eventSent
}
```

Fig. 4.31: Description of the final status

PreCondition Constraints

PreCondition Constraints constrain the allowed values of state variables of a given `STATUS` and the parameters of the event triggering the protocol transition associated with the PreCondition constraints. In m_f the context of the state variables and the event parameters is implied by the location of the constraint on the model. In m_a the implied contexts must be restored by explicitly stating them.

For each `PTransition pt` in m_f, a predicate `pred` is created in m_a. The name of `pred` is the name of the protocol transition prefixed by "`pre_`". This predicate contains the description of the PreCondition constraint `pcc` associated with the protocol transition `pt`. As for every constraint, the PreCondition constraint `pcc` is translated given the general rules given in Section 4.5.6. Then each parameter name in `pcc` is prefixed by "`status.nextStep.eventReceived.`", each SUT state variable name in `pcc` is prefixed by "`status.sut.`" and each Test System state variable name in `pcc` is prefixed by "`status.ts.`".

In the ATM example, the protocol transition `pt_tr3` (i.e. self transition of the protocol state `ps_waitForAuthentication`) is associated with one PreCondition which is a conjunction of three comparison expressions. Let's illustrate the translation by using the first expression of the precondition which states that the event parameter `p` must be different than the `pin` attribute of the SUT state variable named `card`. In m_f this precondition is specified as `p <> card.pin` in m_a, as shown in Figure 4.32, the `<>` is replaced by `!=` and the event parameter `p` is prefixed by `status.nextStep.eventReceived.` and the `card` state variable is prefixed by `status.sut.`.

```
pred pre_pt_tr3(status: Status){
  status.nextStep.eventReceived.p != status.sut.card.pin
  and status.sut.attempts < 2
  and status.ts.ctr_evt_authenticate < status.ts.MAX_evt_authenticate
}
```

Fig. 4.32: ATM example: PreCondition constraint of transition tr3

When all the preconditions predicates are specified in m_a, an additional predicate simply named `precond` is defined that relates the protocol transitions with their pre-conditions predicates.

The `precond` predicate is true if the predicate associated with the protocol transition given as parameter of `precond` predicate is true. The `precond` predicate is used in the fact `step` when selecting the protocol transition to take for the step to the next status.

Figure 4.33 shows an extract of the `precond` predicate, as it is obvious to extrapolate with the other omitted protocol transitions. It specifies that `precond` predicate is evaluated to predicate `pre_pt_tr1` if the parameter protocol transition `t` is in the set of `pt_tr1` protocol transitions.

```
pred precond(status: Status, t: PTransition) {
  (t in pt_tr1) implies pre_pt_tr1[status]
  (t in pt_tr2) implies pre_pt_tr2[status]
  [etc. for the other protocol transitions]
}
```

Fig. 4.33: ATM example: extract of the Alloy declaration of the mapping between pre-conditions and their related protocol transitions

PostCondition constraints

PostCondition constraints are expressed in terms of the values of the sut and test state variables of a given status; the values of the state variables of its next status; the values of the parameters of the event triggering the protocol transition associated with the post-condition; and finally, the event sent to the Test System, if any. In the input FOREST model m_f, the context of these four types of terms is implied by the location of the constraint on the model. In m_a the implied contexts must be restored by explicitly stating them.

Expression of the values of the state variables of a given status and its next status. The expression "`var@pre`", used in OCL post-conditions, refers to the value of a variable `var` before the post-condition holds, i.e. before the protocol transition associated with the post-condition is taken. When using a variable name alone (without the "`@pre`" operator) the values of this variable refer to the values after the protocol transition is taken. Unlike, the OCL post-condition expression, we have opted to describe the behavior of the system in terms of its current status and its next status. Concretely, the transformation is performed in two parts, every state variable in the post-condition followed by the "`@pre`" operator is considered to be a value of a state variable at the given status; thus, this state variable is prefixed by `status.` in the Alloy model m_a. Every state variables that is not followed by the "`@pre`" operator is considered to be a value of a state variable at the next status, thus this state variable is prefixed by `status'.` in the Alloy model m_a. For instance, in the FOREST test model of the ATM example, one of the post condition is that the state variable `attempts` is incremented. In the FOREST model, it is specified by "`attempts = attempts@pre+1`" in Alloy, see Figure 4.34, it is translated as "`status'.attempts = status.attempts+1`".

Translation of the sending of an event in m_f into m_a. In m_f the sending of an event is written using the "`^`" operator, before the operator, the name of the component target of the event is specified and after the event and its parameter values are given (e.g. `COMPONENT_NAME^event (val1,val2)`). In m_a, the sending of an event is specified in the particular state variable named `eventSent` of abstract type `TSEvent`. When an event is sent to the Test System the eventSent state variable is constrained to be of the type of the sent event. For instance, in the ATM example, the `requestPermission` must be sent when the protocol transition `pt_tr6` is taken, in m_a this will be translated in "`status.eventSent in requestPermission`". Now that the

```
01 pred post_pt_tr6(status, status': Status) {
02  status.nextStep.eventSent in requestPermission
03  status.nextStep.eventSent.crp = status.nextStep.eventReceived.cw
04  status'.ts.ctr_ps_waitForPermission = status.ts.ctr_ps_waitForPermission + 1
05  status'.sut.card = status.sut.card
06  status'.sut.attempts = status.sut.attempts
07  status'.ts.ctr_evt_authenticate = status.ts.ctr_evt_authenticate
08  status'.ts.ctr_pt_tr5 = status.ts.ctr_pt_tr5
09 }
```

Fig. 4.34: ATM example: PostCondition constraint of transition tr3

```
pred postcond(status, status': Status) {
  let t = status.nextStep.selectedPTransition | {
    (t in pt_tr1) implies post_pt_tr1[status, status']
    (t in pt_tr2) implies post_pt_tr2[status, status']
    [etc. for the other protocol transitions]
  }
}
```

Fig. 4.35: ATM example: Alloy declaration of the mapping between postconditions and their related protocol transitions

type of the event is specified the values associated with the parameters must be specified. This is performed by specifying the attributes of the state variable `status.nextStep.eventSent`. For instance, the event `requestPermission` is sent with its parameter value being the value of the parameter `cw` of the event received `withdraw`. This is transformed in m_a by stating "`status.nextStep.eventSent.crp = status.nextStep.eventReceived.cw`". As in the previous type of constraints, the event parameter names are prefixed with "`status.nextStep .eventReceived.`" in order to explicitly state the context of the parameter variable.

Unconstrained state variables. State variables which are not constrained in the post condition in m_f must be explicitly constrained in m_a. An unconstrained state variable in a post condition in m_f implicitly means that the value of this state variable remains unchanged. Thus the model transformation must identify which state variables have not been constrained in the post-condition and then for each of them state that their value in the next status is equals to their value in the current status. In the ATM example, the post condition of the protocol transition `tr6` constrains the state variable `ctr_ps_waitForPerm`, so the other state variables remain implicitly unchanged but in Alloy this must be explicitly stated. So for the state variables `card`, `attempts`, `ctr_evt_authenticate` and `ctr_pt_tr5` the constraints in lines 05 to 08 of Figure 4.34 are added to the post condition of the transition tr6.

In a similar way than the `precond` predicate, the `postcond` predicate relates the protocol transitions with their corresponding post-condition predicate. Figure 4.35 shows an extract of the `postcond` predicate.

4.6 Implementation of FOREST semantics with KerMeta

The SESAME approach is supported by a tool chain that automates a number of tasks of the process. The construction of the semantic models associated to FOREST models is one of these automated tasks supported by the SESAME tool chain. Semantics of FOREST models is implemented as a generic model transformation using the KerMeta meta modeling environment[4]. In this section, we illustrate the implementation of this model transformation with the semantics of FOREST data types and constraints on data types.

4.6.1 Semantics implementation of FOREST data types in KerMeta

In the following, we present the implementation of the semantics of the data types of FOREST model as presented in Section 4.5.4. This particular aspect of the semantics of FOREST models is implemented as the operation `static_view_datatypes`, shown in Figure 4.36, extracted from the KerMeta program given in Appendix A.

The operation `static_view_datatypes` has two parameters: the FOREST model `mf`, and the semantic model `ma`. The types of these two parameters (`ForestModel` and `AlloyModel`) are defined by the respective meta models shown in Figures 4.3 and 4.14.

The first step of the translation of data types, implemented in lines from 02 to 11, handles the syntactical differences of the integer and boolean data types. In OCL-F, an integer data type is written “`Integer`” and in Alloy it is written “`Int`”. Similarly for boolean data types, written “`Boolean`” in OCL-F and “`Bool`” in Alloy.

The second step of the translation of data types, implemented in lines from 12 to 25, takes care of constructing the signatures in `ma`[5] derived from the data types of the FOREST model `mf`. Each data type of the FOREST model is parsed using the `each` KerMeta operation (line 12). Enumerations are treated separately, attributes of enumerations are seen as leaves data types (unlike, other kinds of data types). A signature is created for each enumeration (line 14) and for each attribute of each enumeration (line 15). Other kinds of data types are treated in lines 19-24. A special case is defined for signatures that do not extend any other signature (line 21). Otherwise, the typical case is defined in line 23.

The last step of this translation, implemented in lines from 26 to 39, takes care of constructing the fields of the newly constructed signatures in model `ma` derived from the attributes and their multiplicity in model `mf`. Lines 26-27 parse each attribute of each data type of the model `mf`. For each attribute, its multiplicity is set depending on its upper and lower bounds. If an attribute has a lower bound of 0 and an upper bound of 1, then the Alloy multiplicity keyword `lone` is assigned (line 29-31). For an attribute that has a lower bound of 1 and an upper bound of 1, the Alloy multiplicity keyword `one` is assigned (line 32-34). For an attribute that has a lower bound of 1 and an upper bound strictly greater than 1, the Alloy multiplicity keyword `some` is assigned (line 35-37).[6] Lastly, the attribute is created in the model ma (line 38).

[4] The reasons that led us selecting KerMeta as the model transformation tool for the SESAME approach are presented in Section 3.5.

[5] The createSig operation is not illustrated in Figure 4.36 but given in Appendix A.8. The createSig have five parameters: the model in which the signature must be created, the name of the signature to create, two boolean values that inform if the signature is abstract, respectively if it is a leaf data type, lastly the name of the signature it extends (if none, void is given).

[6] For this kind of multiplicity, an additional constraint (in terms of an Alloy fact) is created in the model ma, that defines more precisely the upper bound.

```
operation static_view_datatypes(mf: ForestModel, ma: AlloyModel) is do
 mf.datatypes.each{ fDT |
   if fDT.typeName.equals("Integer") then
     fDT.typeName := "Int"
   end
 }
 mf.datatypes.each{ fDT |
   if fDT.typeName.equals("Boolean") then
     fDT.typeName := "Bool"
   end
 }
 mf.datatypes.each{ fDT |
   if fDT.isEnum then
     createSig(ma, fDT.typeName, true, false, void)
     fDT.attributes.each{ att |
       createSig(ma, att.attName, false, true, fDT.typeName)
     }
   else
     var sig : ALLOY::Signature init void
     if fDT.extends.isVoid then
       sig := createSig(ma, fDT.typeName, fDT.isAbstract,
                        fDT.isLeaf, void)
     else
       sig := createSig(ma, fDT.typeName, fDT.isAbstract, fDT.isLeaf,
         fDT.extends.typeName)
     end
 end }
 mf.datatypes.each{ fDT |
   fDT.attributes.each{ att |
     var mult : ALLOY::Multiplicity init ALLOY::Multiplicity.~seq
     if att.lowerMultiplicity.equals("0")
        and att.upperMultiplicity.equals("1") then
       mult := ALLOY::Multiplicity.lone
     end
     if att.lowerMultiplicity.equals("1")
        and att.upperMultiplicity.equals("1") then
       mult := ALLOY::Multiplicity.one
     end
     if att.lowerMultiplicity.equals("1")
        and not att.upperMultiplicity.equals("0")
          and not att.upperMultiplicity.equals("1") then
       mult := ALLOY::Multiplicity.some
     end
     addNewFieldToSig(ma, getSigInModel(ma, fDT.typeName), att.attName,
        mult, att.attType.typeName)
 } }
end
```

Fig. 4.36: Semantics implementation of FOREST data types in KerMeta

5. SERELA: A TEST SELECTION LANGUAGE FOR THE SESAME APPROACH

Abstract

This chapter describes the language supported within SESAME processes to specify test selection on FOREST models. This language, named SERELA, offers a number of instructions allowing Test Designers to reduce the number and length of test cases derived from a given FOREST model. This chapter starts by introducing the test selection approach within SESAME processes. Then the concepts and syntax of the SERELA language are described. Finally, the semantics of SERELA is defined in terms of transformations of FOREST models.

5.1 Introduction

5.1.1 FOREST model complexity

FOREST models may be in three different states (initial, constrained and validated) as illustrated in Figure 4.1. Chapter 4 describes the FOREST language which allows Analysts to produce an initial description of the system under test and of its Test System. These initial FOREST models are precisely defined through their semantic models. The simulation of the semantics is likely to contain an overly large amount of test cases due to the complexity of the initial FOREST models. The nature of this complexity is of two kinds:

- *Static (data) complexity.* The number of possible values of data types in a FOREST model may be too large to be exercised exhaustively. For instance, in the ATM example, the Digit data type is made of ten thousand possible values (i.e. PIN numbers from 0000 to 9999). The computation cost may be too high to exercise all these values. Thus in order to be able to exercise the ATM system in a reasonable time, the data complexity driven by this data type may be reduced.
- *Dynamic (behavior) complexity.* The number of legal sequences of event receptions specified in the dynamic view of FOREST models might be infinite or very large. As for example, the initial FOREST model of the ATM has an infinite amount of legal behaviors. This is due to the self-transitions and the circular paths present in the protocol state machine. Thus in order to be able to exercise the ATM system, the behavior complexity must be reduced to a finite and reasonable number.

SERELA described in the following sections of this chapter allows Test Designers to deal with these two kinds of complexity.

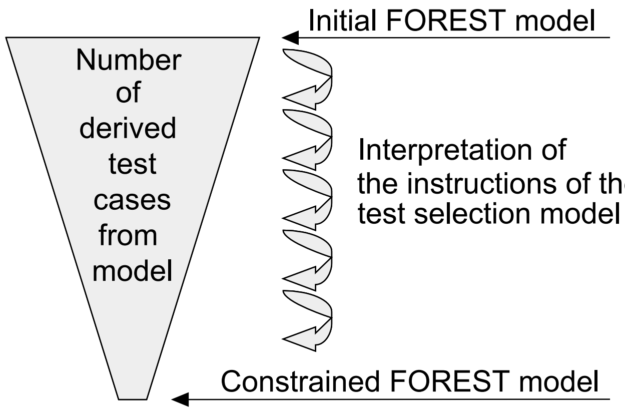

Initial FOREST model
Number of derived test cases from model
Interpretation of the instructions of the test selection model
Constrained FOREST model

Fig. 5.1: SESAME test selection approach

5.1.2 SERELA: a test selection domain-specific language

"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [vDKV00].

SERELA is a domain-specific language that is focused on the particular problem of describing the test selection of a SUT. SERELA stands for ***SE****SAME test* ***RE****duction* ***LA****nguage*, as it provides a number of different kinds of instructions aiming at reducing data and behavior complexity. SERELA test selection instructions aim at selecting important aspects of a given FOREST model that the Test Designer wishes to cover during testing and at ignoring aspects which will not be taken into account for testing.

5.1.3 SESAME test selection approach

The semantics of each kind of test selection instructions is given as a generic FOREST model transformation. A SERELA *test selection model* is made of a set of test instructions and is defined for a given test purpose. A *test purpose* is a coherent subset of the test requirements of the SUT. Thus, in order to exercise the SUT with all its test requirements, it may be necessary to specify a number of test selection models. Figure 5.1 illustrates the test selection approach followed in SESAME. A SERELA test selection model is defined with respect to a given input FOREST model. An initial FOREST model is taken as input of the test selection process and then for each instruction in a test selection model the initial FOREST model is transformed accordingly. The sequence of multiple arrows in Figure 5.1 illustrates that there will be a number of model transformations performed (one for each instruction) in order to produce the FOREST model resulting from the test selection. The model resulting from the test selection is called a constrained model.

In this chapter, SERELA concepts, syntax, and semantics are illustrated with three test selection models specified in the context of the ATM example. These test selection models, shown in Figures 5.3, 5.4 and 5.5, are derived from the following test purposes (TP):

TP1. Exercise the SUT with cash withdrawals of at least one small amount and one large amount.

TP2. Exercise the SUT with consecutive false authentications.

TP3. Exercise the SUT with the boundary values of digits for the authenticate events and boundary values of cash amounts for the withdraw events.

5.2 SERELA concepts and syntax

In a model-driven engineering environment, concepts of a language are abstractly represented by a meta model. In the context of the definition of SERELA, the concepts of SERELA are the different instruction types that SERELA test selection models may contain. The abstract representations of the SERELA concepts are defined in the SERELA meta model given in Figure 5.2. The SERELA meta model express that a test selection model contains a set of test selection instructions. This set of instructions[1] contains at least one instruction of type `SetScope`. There are eleven types of instructions grouped in two categories static instructions and dynamic instructions.

The concrete syntax of SERELA models is textual. EBNF [ISO96] is used (as for OCL-F expressions in FOREST models) to define the grammar of the concrete syntax. The EBNF grammar rules can be found in Figure 5.6.

In this section, the different SERELA concepts are presented together with their concrete syntax and an illustration with the ATM example is provided for each instruction type.

5.2.1 Static instructions

Static instructions aim at reducing the data complexity of FOREST models. The data complexity is directly depending on the size of the input and output domain of the SUT. The *input domain* of a component is all the possible combination of parameters values of the events received by this component. The *output domain* of a component is all the possible combination of parameters values of the events sent by this component. In the context of SESAME, there are two input and two output domains from the two SUT and Test System components. The input domain of the SUT is the output domain of the Test System, as all the events received by the Test System are sent by the SUT. Vice versa, the input domain of the Test System is the output domain of the SUT. SERELA provides three expression types for the reduction of the data complexity: `SelectEvent`, `SelectTrans` and `SelectData`.

SelectEvent instruction type

In the context of testing all the possible events and their respective conveyed data that can be sent by the Test System to exercise the system under test, the Test Designer faces a problem of data complexity. In deed, event receptions convey a certain amount of possible data. All the possible combination of events with their parameter values may be overly large to be exercised.

For instance, in the ATM example, the possible combination of events of type `insertCard`, is dependent on its parameter, in that case the parameter `PIN` composed of four digits, which results in 10000 possible `insertCard` events. It may not be relevant to exercise the SUT with the insertion of all the possible cards.

SERELA provides a type of test selection instructions, which address this issue. This instruction type allows Test Designers specifying a constraint on an event reception type. This constraint applies to all events of the targeted event type.

The syntax of this instruction type is formally specified as the BNF `SelectEvent` grammar rule shown in Figure 5.6. This instruction type accepts exactly two parameters: the name of the

[1] "Test selection instructions" are also simply referred to as "SERELA instructions" or just "instructions".

event type to constrain and the constraint applicable on the event type. The constraint is an OCL-F expression of type EventType constraint, see sub-section 4.3.3 describing the OCL-F language and in particular the EventType constraints.

For instance, in the test purpose TP2 that consists in exercising the SUT with false authentications. The test selection model could make use of this instruction type by setting the card inserted to a specific card (line 01 in Figure 5.4) and constraining the possible parameter values for the authentication to be different than the PIN number of the inserted card (line 04 in Figure 5.4).

SelectTrans instruction type

This type of instructions aims at restricting the possible values conveyed by events received from the Test System during a particular protocol transition. These instructions allow Test Designers to be more specific in the selection of event parameters values than the previous `SelectEvent` instruction type. The previous instruction type applies to all event receptions of an event type, this instruction type applies to the event receptions of an event type during a given protocol transition. Indeed, these two types of instructions are equivalent when targeting an event type triggering a single protocol transition of a FOREST model. This instruction type is defined to be used for event receptions appearing on multiple protocol transitions in a FOREST model but for which its parameters must be constrained to one of these protocol transitions.

In order to constrain the parameter values of an event type for a given protocol transition, SERELA provides an instruction type that is described by the BNF `SelectTrans` grammar rule in Figure 5.6. `SelectTrans` instructions accept two parameters. Firstly, the protocol transition name which contains the event reception and secondly the expression that constrain the parameter values of the event reception. The expression is expressed as an OCL-F expression of PreCondition constraint type (see sub-section 4.3.3).

For instance, in the test purpose TP3 that consists in exercising some boundary values of the input domain of the SUT in particular digits and amounts withdrawn. Let's illustrate this instruction type to select the largest amount value only when the transition tr6 is taken (line 02 in Figure 5.5), i.e. only when the withdraw event is received with a large amount, and select the smallest amount value only when the transition tr5 is taken (line 01 in Figure 5.5).

SelectData instruction type

Data types in analysis FOREST models may contain too many values to be tested exhaustively. An efficient way to reduce the data complexity is to reduce the number of values of the data types. SERELA provides the `SelectData` instruction type which expresses values of a given data type that must be exercised on the SUT. The selected values are the ones satisfying the constraint given as a parameter of the instruction of type `SelectData`.

This instruction type is described by the BNF `SelectData` grammar rule in Figure 5.6. `SelectData` instructions accept two parameters. Firstly, the data type name which refers to the data type to be constrained and secondly the expression that constrains the data type. The expression is specified as an OCL-F expression of DataType constraint type (see sub-section 4.3.3).

For instance, in the test purpose TP3 that consists in exercising some boundary values of the input domain of the SUT in particular digits and amounts withdrawn. Let's illustrate this instruction type to select the particular '0' and '9' digits specified in the FOREST model as the

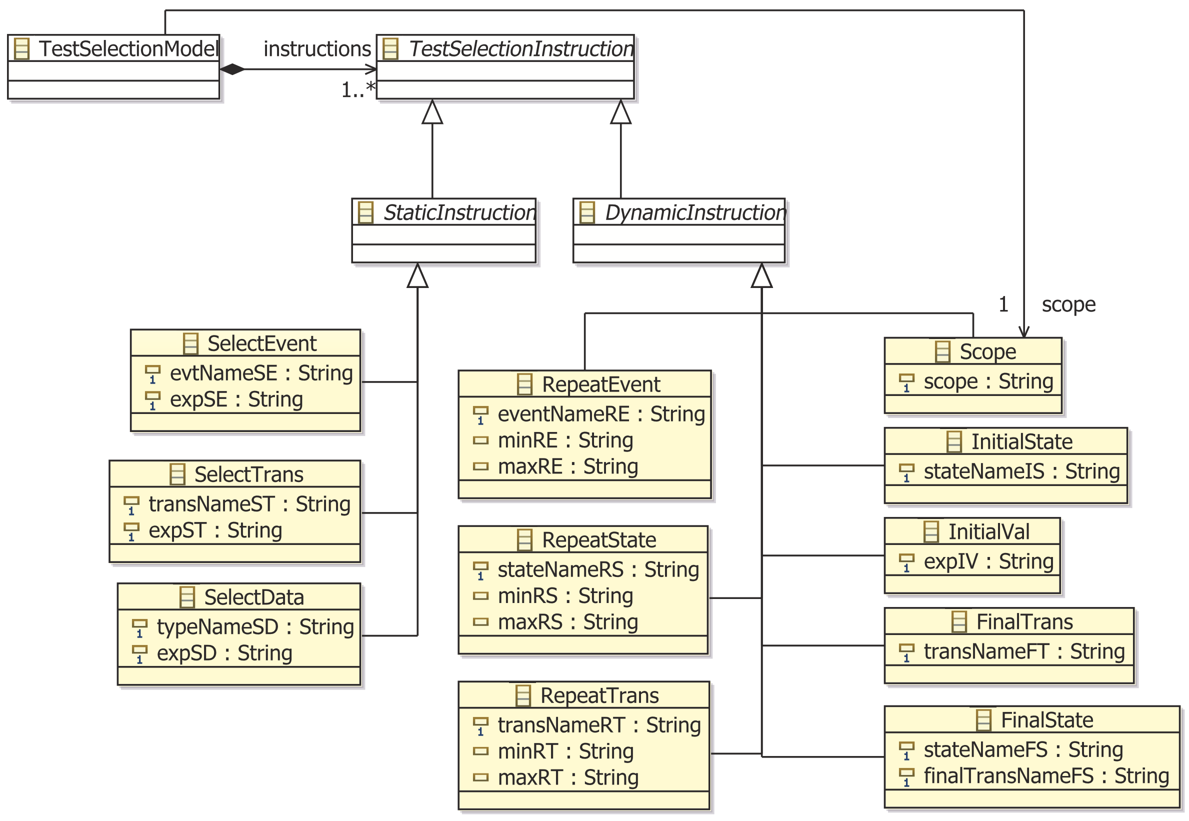


Fig. 5.2: SERELA meta model

d0 and d9 abstract data types. This selection may be expressed with a SelectData instruction, as shown in the third line in Figure 5.5 which states that the data type PIN is only made of the subtypes d0 and d9 for the purpose of this test selection.

5.2.2 Dynamic instructions

Dynamic instructions aim at reducing the behavior complexity of FOREST models. The behavior complexity is due to the possibly large length of execution traces specified in FOREST models; and to the combinatorial exposition resulting from the number of legal sequences defined in the dynamic view of FOREST models. SERELA provides eight expression types for the reduction of the behavior complexity: RepeatEvent, RepeatTrans, RepeatState, Scope, FinalState, FinalTrans, InitialState and InitialVal.

RepeatEvent instruction type

In the dynamic view of a FOREST model, it is likely that an event type may be received a certain amount of time. This amount of time may be overly large, or even infinite. As for instance, the withdraw event in the ATM example, may be received an infinite amount of time. Indeed, in order to produce test cases with a finite number of steps. There should be a (reasonably) finite amount of event receptions for each event type. This SERELA instruction type allows to set a

```
01 initstate Authenticated ;
02 initvar { attempts = 0 and sut.card.pin[0].oclIsKindOf(d0) and
   sut.card.pin[1].oclIsKindOf(d1) and sut.card.pin[2].oclIsKindOf(d2) and
   sut.card.pin[3].oclIsKindOf(d3) } ;
03 finalstate Authenticated ;
04 repeat-event withdraw min 5 max 5 ;
05 repeat-trans tr5 min 1 ;
06 repeat-state waitForPermission min 1 ;
07 scope 10 ;
08 repeat-trans tr8 max 0 ;
```

Fig. 5.3: ATM example: a SERELA test selection model

```
01 select-event insertCard { bc.pin[0].oclIsKindOf(d1) and
   bc.pin[1].oclIsKindOf(d2) and bc.pin[2].oclIsKindOf(d3) and
   bc.pin[3].oclIsKindOf(d4) } ;
02 finaltrans tr2 ;
03 scope 4 ;
04 select-event authenticate { !bc.pin[0].oclIsKindOf(d1) or
   !bc.pin[1].oclIsKindOf(d2) or !bc.pin[2].oclIsKindOf(d3) or
   !bc.pin[3].oclIsKindOf(d4) } ;
```

Fig. 5.4: ATM example: a second SERELA test selection model

maximum bound on the number of receptions of a given event type.

The syntax of this instruction type is formally specified as the BNF `RepeatEvent` grammar rule shown in Figure 5.6. This instruction type accepts exactly two parameters. Firstly, the name of the event type for which the number of repetition is constrained. Secondly, a natural number preceded by the `min` keyword or the `max` keyword or both keywords depending on if a minimal and/or a maximal bound is/are specified.

For instance, in the test purpose TP1, the Test Designer may choose to bound the number of withdraw event receptions to be exactly five times. This results in the specification of a RepeatEvent instruction which comprises a minimum bound of 5 and a maximum bound of 5, as illustrated in line 04 of Figure 5.3.

RepeatState instruction type

Similarly to the previous `RepeatEvent` instruction type, SERELA provides an instruction type that set a maximal and/or a minimal bound on the number of times a given protocol state of a FOREST model may be active during a simulation.

The syntax of this instruction type is formally specified as the BNF `RepeatState` grammar rule shown in Figure 5.6. This instruction type accepts exactly two parameters. Firstly, the name of the protocol state for which the number of repetition is constrained. Secondly, a natural number preceded by the `min` or the `max` keyword or both keywords depending on if a minimal and/or a maximal bound is/are specified.

For instance, in the test purpose TP1, the SUT must be exercised at least once with a big amount.

```
01 select-trans tr5 { c.oclIsKindOf(smallestCA) }
02 select-trans tr6 { c.oclIsKindOf(largestCA) }
03 select-data PIN { digits.oclIsKindOf(d0) or digits.oclIsKindOf(d9) }
```

Fig. 5.5: ATM example: a third SERELA test selection model

The Test Designer may choose to specify this by stating that the protocol state `waitForPerm` must be active at least once during a simulation, as described in line 07 of Figure 5.3. If the state is active at least once then the withdraw event is received with a large amount at least once, as this is the only way to reach the protocol state.

RepeatTrans instruction type

Similarly to the previous `RepeatEvent` and `RepeatState` instruction types, SERELA provides an instruction type that set a maximal and/or a minimal bound on the number of times a given protocol transition of a FOREST model may be taken during a simulation.

The syntax of this instruction type is formally specified as the BNF `RepeatTrans` grammar rule shown in Figure 5.6. This instruction type accepts exactly two parameters. Firstly, the name of the protocol transition for which the number of repetition is constrained. Secondly, a natural number preceded by the `min` and/or the `max` keyword or both keywords depending on if a minimal and/or a maximal bound is/are specified.

For instance, in the test purpose TP1, the SUT must be exercised at least once with a small amount. The Test Designer may choose to specify this by stating that the protocol self-transition `tr5` (from and to state `Authenticated`) must be taken at least once during a simulation, as described in line 05 of Figure 5.3. If the transition `tr5` is taken at least once then the withdraw event is received with a small amount at least once, because of the precondition of this transition stating that the transition is enabled only if the withdraw event is received with a small amount.

Scope instruction type

A test case is a simulation of a FOREST model as described in Section 4.2. A simulation of a FOREST model may be composed of a number of statuses and related steps connecting a status to its next status. The scope is the maximum number of statuses of a FOREST model simulation. In order to perform formal analysis with the Alloy analyzer, part of the SESAME tool-support, it is necessary to specify a bound on the number of steps that will be performed during the test case, i.e. during the simulation of the FOREST model.

SERELA provides an instruction type for specifying an upper bound on the number of statuses and steps in a test case. The syntax of this instruction type is formally specified as the BNF `Scope` grammar rule shown in Figure 5.6. This instruction type accepts exactly one parameter, which is a string that is in the form of a natural number. This number represents the maximum number of statuses allowed in a test case.

The determination of the upper bound is not an easy task and may be empirically experimented to find a trade-off between a low scope number and a high scope number. On the one hand, if the scope number is too small, some behaviors of the SUT will not be exercised. For instance, if the scope is set to the number four (as specified in line 03 of Figure 5.4), the protocol transition from the state `waitForPerm` to the state `Authenticated` will not be tested, because there must

```
SERELAExp     ::= SERELAExp ';' SERELAExp
              | SelectEvent
              | SelectTrans
              | SelectData
              | RepeatEvent
              | RepeatState
              | RepeatTrans
              | Scope
              | InitialState
              | InitialVal
              | FinalState
              | FinalTrans

SelectEvent   ::= 'select-event' EVT-NAME '{' OCLExpressionCS '}'

SelectTrans   ::= 'select-trans' TR-NAME '{' OCLExpressionCS '}'

SelectData    ::= 'select-data' DT-NAME '{' OCLExpressionCS '}'

RepeatEvent   ::= 'repeat-event' EVT-NAME MinMax

RepeatState   ::= 'repeat-state' S-NAME MinMax

RepeatTrans   ::= 'repeat-trans' TR-NAME MinMax

MinMax        ::= 'min' NAT
              |   'max' NAT
              |   'min' NAT 'max' NAT

Scope         ::= 'scope' NAT

InitialState ::= 'initstate' S-NAME

InitialVal    ::= 'initval' '{' OCLExpressionCS '}'

FinalState    ::= 'finalstate' S-NAME FT-NAME

FinalTrans    ::= 'finaltrans' TR-NAME

DT-NAME       ::= <string>
TR-NAME       ::= <string>
FT-NAME       ::= <string>
S-NAME        ::= <string>
EVT-NAME      ::= <string>

NAT           ::= <natural-integer>
```

Fig. 5.6: Definition of SERELA grammar rules

be a minimum of five steps (taking five protocol transitions) to reach this transition. On the other hand, if the number is too high, there will be a combinatorial explosion resulting in the formal analysis computation taking a very long time. For instance, the generation of a single test case for a test model of the ATM example with a scope number of 10 took around twenty seconds with the Alloy analyzer.

InitialState and InitialVal instruction types

By default, the initial status of test cases is the initial status as modeled in the FOREST model, i.e. the first active state is the initial protocol state and the state variables of the SUT and Test System components are initialized with the InitialValue constraint of the FOREST model. In most cases, some of the first steps of the SUT simulation are not necessary to be performed for each test case. In these cases, the SUT simulation will start with a different initial status: an active protocol state different than the initial protocol state of the initial FOREST model and/or state variables initialized differently from the InitialValue of the initial FOREST model. SERELA provides two instruction types to specify the initial testing status: `InitialState` that specifies the new initial protocol state and `InitialVal` that specifies the new initialization values of the state variables. For instance, in the ATM example, the first steps involving the insertion of a card and the authentication may not be tested (either because they are assumed to be working correctly or that they are going to be tested separately from the rest of the behavior).

The syntax of this instruction type is formally specified as the two BNF `InitialState` and `InitialVal` grammar rules shown in Figure 5.6. The first rule enables the specification of the protocol state active during the initial status. This expression accepts one parameter which is the name of the protocol state. The second expression type accepts one parameter which is a constraint on a state variable. The constraint is an OCL-F expression of type InitialValue constraint, see sub-section 4.3.3 describing the OCL-F language and in particular the InitialValue constraints.

For instance, the test purpose TP1 focuses on the testing of the withdrawal of money, ignoring the aspects of the authentication and the card insertion. The Test Designer may specify the initial state as shown in the line 01 of Figure 5.3. Then the state variables must be initialized according to the protocol states which were skipped. Let's say for this example, that the withdraw events are tested assuming that a card with PIN 0000 is inserted and that the first authentication is successful. Thus the InitialValue constraint of the FOREST model is set to initialize the `attempts` state variable to zero and the `card` state variables are set to a card with PIN 0000 as shown in the line 02 of Figure 5.3.

FinalState and FinalTrans instruction types

Reactive embedded systems are typically never-ending systems, i.e. they start when power is provided and obviously must stop when power is down. In that type of system, it is likely that the dynamic view of the initial FOREST models describing these systems do not comprise any final protocol state. In the context of testing a never-ending system, a termination must be stated for the purpose of testing the system. In order to specify the termination, SERELA provides an instruction type that adds to FOREST models a final protocol state.

In the case of terminating systems, this type of instructions may also be used and would specify additional alternative ways of terminating the system behavior.

Two SERELA instructions allow to express how the SUT may terminate, formally specified as the two BNF `FinalState` and `FinalTrans` grammar rules shown in Figure 5.6. `FinalState` instructions accept two parameters, the first one referring to a protocol state name of the FOREST model and the second one to the name of the transition that is going to be created to link the protocol state to the final state. `FinalTrans` instructions accept one parameter, which is the name of a given protocol transition which will be redirected to the final state.

For instance, in the test purpose TP1, the SUT must be exercised with different withdraw amounts. The Test Designer may choose to reduce behavior of the SUT to those terminating in the `Authenticated`, as specified in line 03 of Figure 5.3.

5.3 SERELA semantics

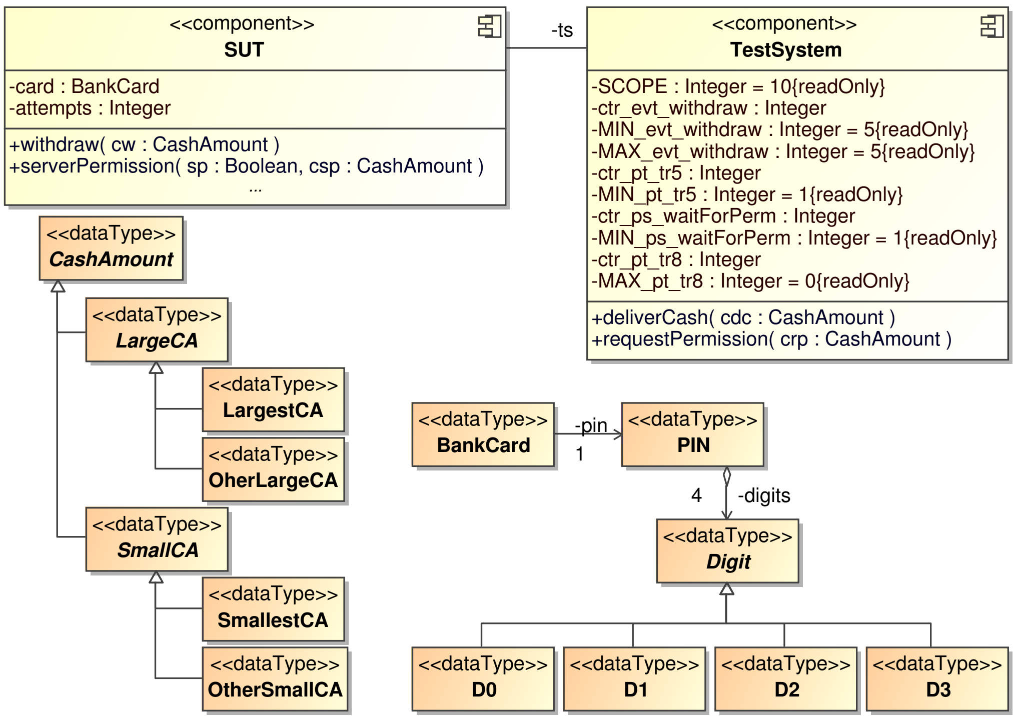
The algorithm, given in the following subsections, defines the transformation of a FOREST model into a constrained FOREST model. Thanks to the FOREST semantics defined in Chapter 4. The constrained FOREST model may further be transformed in the SESAME semantics language (i.e. Alloy) in order to generate test cases that take into account the test selection model initially expressed in SERELA.

Given an input FOREST model m_i in M_{forest} and a test selection model m_s in M_{serela}, the resulting constrained FOREST model m_o in M_{forest} is produced by interpreting each instruction from the test selection model m_s on the output model m_o. M_{forest} is the set of all possible models conforming to the FOREST meta model, given by Figure 4.3. M_{serela} is the set of all possible models conforming to the SERELA meta model, given by Figure 5.2. Each type of test selection instruction expressed in a SERELA model corresponds to a model transformation. For each test selection instruction *tsi* of the test selection model m_s, a model transformation is performed. The performance of this set of model transformations computes the semantics of the test selection model m_s expressed on the FOREST model m_i.

A SERELA test selection model is a *conjunction of properties* that specifies a model transformation. Each test selection instruction of a test selection model is seen as a property that must be satisfied by the model resulting from the transformation. Some of the test selection instructions are correlated. For instance, the instructions of types `FinalTrans` and `FinalState` are related to the ones of type `RepeatEvent`, `RepeatState` and `RepeatTrans`. On the one hand, the three last instruction types enrich the preconditions of all the protocol transitions going to the final state; on the other hand, the two first instruction types add new protocol transitions going to the final state. It is important to be aware of the correlation of the test selection instructions during the implementation of their semantics.

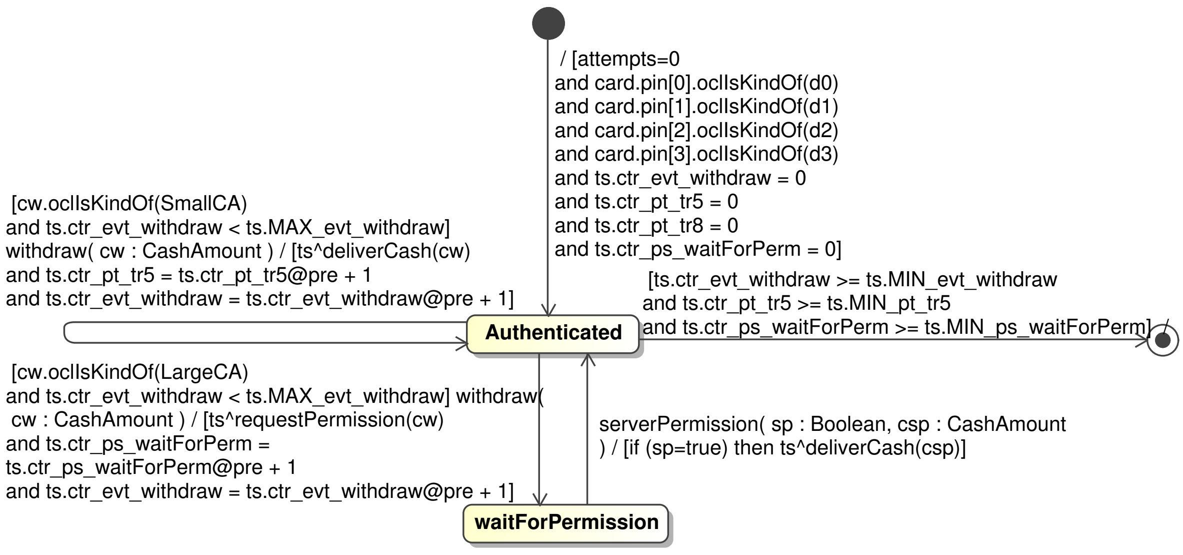
The very first step of the model transformation, before the processing of SERELA instructions of the test selection model m_s, is to copy all the elements in the input model m_i to the output model m_o, i.e. the input model is duplicated into the output model. Then for each instruction *tsi* of type `TestSelectionInstruction` in the test selection model m_s, the corresponding model transformation is performed depending of its type. The following subsections describe the semantics of each instruction type in two parts. The first part describes what the model transformation does. The second part describes how the model transformation may be performed by describing an informal algorithm based on the meta-classes (and their related associations) from the SERELA and the FOREST meta models.

The very last step of the model transformation, after the processing of all the SERELA instructions of the test selection model m_s, is to prune all the elements of the static and dynamic views for which none of their instance may be exercised. In particular, all protocol states that are not reachable from the initial protocol state are pruned. For instance, in the Test model of Figure 5.8, protocol states `waitForCard` and `waitForAuthentication` are not reachable due to additional constraints on the preconditions of protocol transitions introduced by the test selection specification. All protocol transitions that are not reachable from the initial protocol state are also pruned. Concerning the static view, the data types, for which no instance is exercised, are pruned and the event reception, for which not instance is exercised, are also pruned. For instance, in the Figure 5.8, event reception `insertCard`, `authenticate` and `restoreCard` are removed from the static view of the Test Model. This processing is performed from the results of the semantical metrics based on instance coverage presented in the next chapter, 6.



<<component>>
SUT
-card : BankCard
-attempts : Integer
+withdraw(cw : CashAmount)
+serverPermission(sp : Boolean, csp : CashAmount)
...
-ts
<<component>>
TestSystem
-SCOPE : Integer = 10{readOnly}
-ctr_evt_withdraw : Integer
-MIN_evt_withdraw : Integer = 5{readOnly}
-MAX_evt_withdraw : Integer = 5{readOnly}
-ctr_pt_tr5 : Integer
-MIN_pt_tr5 : Integer = 1{readOnly}
-ctr_ps_waitForPerm : Integer
-MIN_ps_waitForPerm : Integer = 1{readOnly}
-ctr_pt_tr8 : Integer
-MAX_pt_tr8 : Integer = 0{readOnly}
+deliverCash(cdc : CashAmount)
+requestPermission(crp : CashAmount)
<<dataType>> CashAmount
<<dataType>> LargeCA
<<dataType>> LargestCA
<<dataType>> OherLargeCA
<<dataType>> SmallCA
<<dataType>> SmallestCA
<<dataType>> OtherSmallCA
<<dataType>> BankCard
-pin
1
<<dataType>> PIN
4
-digits
<<dataType>> Digit
<<dataType>> D0
<<dataType>> D1
<<dataType>> D2
<<dataType>> D3

Fig. 5.7: ATM example: a constrained FOREST model - static view



/ [attempts=0
and card.pin[0].oclIsKindOf(d0)
and card.pin[1].oclIsKindOf(d1)
and card.pin[2].oclIsKindOf(d2)
and card.pin[3].oclIsKindOf(d3)
and ts.ctr_evt_withdraw = 0
and ts.ctr_pt_tr5 = 0
and ts.ctr_pt_tr8 = 0
and ts.ctr_ps_waitForPerm = 0]
[cw.oclIsKindOf(SmallCA)
and ts.ctr_evt_withdraw < ts.MAX_evt_withdraw]
withdraw(cw : CashAmount) / [ts^deliverCash(cw)
and ts.ctr_pt_tr5 = ts.ctr_pt_tr5@pre + 1
and ts.ctr_evt_withdraw = ts.ctr_evt_withdraw@pre + 1]
[ts.ctr_evt_withdraw >= ts.MIN_evt_withdraw
and ts.ctr_pt_tr5 >= ts.MIN_pt_tr5
and ts.ctr_ps_waitForPerm >= ts.MIN_ps_waitForPerm] /
Authenticated
[cw.oclIsKindOf(LargeCA)
and ts.ctr_evt_withdraw < ts.MAX_evt_withdraw] withdraw(
cw : CashAmount) / [ts^requestPermission(cw)
and ts.ctr_ps_waitForPerm =
ts.ctr_ps_waitForPerm@pre + 1
and ts.ctr_evt_withdraw = ts.ctr_evt_withdraw@pre + 1]
serverPermission(sp : Boolean, csp : CashAmount
) / [if (sp=true) then ts^deliverCash(csp)]
waitForPermission

Fig. 5.8: ATM example: a constrained FOREST model - dynamic view

5.3.1 Semantics of SelectEvent instructions

Description

The model transformation associated with instructions of type `SelectEvent` consists in two steps. In step (1), the event targeted by the instruction must be found in the FOREST model with the help of its name and type. In step (2), the constraint of the instruction is created in the FOREST model and associated with the targeted event.

Algorithm

If the current test selection instruction *tsi* is of type `SelectEvent` then the following model transformation is performed on the model m_o:

(1) There exists an event *evt* in model m_o such that the two following conditions (a) and (b) are true. If there is no event *evt* found then the SERELA instruction *tsi* is ill-formed. If there are more than one events found then the FOREST model m_o, and subsequently m_i, is ill-formed because they do not fulfill the rule "WFR FOREST.6" described in Section 4.2.1.

   (a) *evt* is an instance of the meta-class `SUTEvent` or *evt* is an instance of the meta-class `TSEvent`

   (b) the attribute `eventName` of transition *evt* is equal to the attribute `eventNameSE` of instruction *tsi*

(2) A new `Constraint` c is created in m_o and linked to the event *evt* through its `event` association. The attribute `exp` of c is assigned the value from the attribute `expSE` of *tsi*.

5.3.2 Semantics of SelectTrans instructions

Description

The model transformation associated with instructions of type `SelectTrans` consists in two steps. In step (1), the protocol transition targeted by the instruction must be found in the FOREST model with the help of its name and type. In step (2), the constraint expression of the instruction is appended, as a conjunction, to the precondition of the targeted protocol transition.

Algorithm

If the current test selection instruction *tsi* is of type `SelectTrans` then the following model transformation is performed on the model m_o, in two steps:

(1) There exist a protocol transition *pt* in model m_o such that the two following conditions (a) and (b) are true. If there is no transition *pt* fulfilling the condition then the SERELA instruction *tsi* is ill-formed. If there are more than one transitions fulfilling the condition then the FOREST model m_o, and subsequently m_i, are ill-formed because they do not fulfill the rule "WFR FOREST.18" described in Section 4.2.1.

   (a) *pt* is an instance of the meta-class `ProtocolTransition`

   (b) the attribute `transName` of transition *pt* is equal to the attribute `transNameST` of instruction *tsi*

(2) The precondition of *pt* is appended with the string resulting from the concatenation of the strings " `AND` " and the string `expST` from *tsi*.

5.3.3 Semantics of SelectData instructions

Description

The model transformation associated with instructions of type `SelectData` consists in two steps. In step (1), the data type targeted by the instruction must be found in the FOREST model with the help of its name and type. In step (2), a new constraint is created in the output FOREST model and associated with the targeted data type, the new constraint's expression being the parameter expression of the instruction.

Algorithm

If the current test selection instruction *tsi* is of type `SelectData` then the following model transformation is performed on the model m_o:

(1) There exists exactly one data type *dt* in model m_o such that the two following conditions (a) and (b) are true. If there is no data type *dt* fulfilling the conditions then the SERELA instruction *tsi* is ill-formed. If there are more than one data types *dt* fulfilling the conditions then the FOREST model m_o, and subsequently m_i, are ill-formed because they do not fulfill the rule "WFR FOREST.7" described in Section 4.2.1.

   (a) *dt* is an instance of the meta-class `Datatype`

   (b) the attribute `typeName` of transition *dt* is equal to the attribute `typeNameSD` of instruction *tsi*

(2) A new `Constraint` *c* is created in m_o and linked to the data type *dt* through its `datatype` association. The attribute `exp` of *c* is assigned the value from the attribute `expSD` of *tsi*.

5.3.4 Semantics of RepeatEvent instructions

Description

The model transformation associated with instructions of type `RepeatEvent` consists in seven steps. In step (1): the event targeted by the instruction must be found in the FOREST model with the help of its name and type. In steps (2), (3), (4): a counter is created to count the number of times the event has been received by the SUT since the start of a simulation.

The purpose of the counter is to be incremented when an event of the type given in the instruction is received. The counter incrementation is enforced, in step (5), by enriching all the post-conditions of the protocol transitions triggered by the event targeted by the instruction.

If a minimum bound is set in the expression, a simulation of the SUT may terminate only after the event has been received a number of time greater than the minimum bound. This is described in step (6) by enriching all the preconditions of the protocol transitions going to the final state with a constraint stating that the counter must be greater than the minimum bound.

If a maximal bound is set in the expression, an event may be received only if the counter value is not higher than the maximum allowed by the test selection instruction. This is described in step (7) by enriching all the preconditions of the protocol transitions triggered by the event with a constraint stating that the transition is enabled if the counter is strictly lower than the maximum bound.

Algorithm

If the current test selection instruction *tsi* is of type `RepeatEvent` then the following model transformation is performed on the model m_o:

(1) There exists an event *evt* in model m_o such that the two following conditions (a) and (b) are true. If there is no event *evt* found then the SERELA instruction *tsi* is ill-formed. If there are more than one events fulfilling the conditions then the FOREST model m_o, and subsequently m_i, is ill-formed because they do not fulfill the rule "WFR FOREST.6" described in Section 4.2.1.

  (a) *evt* is an instance of the meta-class `SUTEvent`

  (b) the attribute `eventName` of transition *evt* is equal to the attribute `eventNameRE` of instruction *tsi*

(2) A new `StateVariable` sv_{ctr} is created in m_o and linked to the Test System component through its `testVariables` association. The attribute `attName` of sv_{ctr} is assigned the string resulting from the concatenation of the string `ctr_evt_` and the string value of the attribute `eventNameRE` of *tsi*. For instance, for the `authenticate` event in the ATM example, sv_{ctr} would be named `ctr_evt_authenticate`. sv_{ctr} is associated with the data type dt_{int} whose `typeName` attribute is "Integer" through its association `attType`.

(3) If there is no InitialValue `Constraint` *c* in m_o then *c* is created in m_o and the attribute `exp` of *c* is set to the name of sv_{ctr} appended with the string " `= 0` ".

(4) If there is an InitialValue `Constraint` *c* in m_o then the attribute `exp` of *c* is appended with the string resulting from the concatenation of " `AND` ", name of sv_{ctr} and " `= 0` ". For instance, with the event name `authenticate`, this would result in appending the string `AND ctr_evt_authenticate = 0`

(5) For all transitions *pt* for which the triggering event name of *pt* is equals to the attribute `eventNameRE` of *tsi*. The post-condition of *pt* is appended with the string resulting from the concatenation of the strings " `AND` ", the name of sv_{ctr}, the string `=` , the name of sv_{ctr}, and the string "`@pre + 1`".

(6) If the `min` attribute of *tsi* is set to a value:

  (a) A new `StateVariable` sv_{min} is created in m_o and linked to the Test System component through its `testVariables` association. The attribute `attName` of sv_{min} is assigned the string value resulting from the concatenation of the string `MIN_evt_` and the string value of the attribute eventNameRE of *tsi*. For instance, for the authenticate event in the ATM example, sv_{min} would be named MIN_evt_authenticate. sv_{min} is associated with the data type dt_{int} whose typeName attribute is "Integer" through its association attType. The attribute isReadOnly of sv_{min} is assigned to the Boolean value "true".

(b) A new Constraint c is created in m_o and linked to the state variable sv_{min} through its constant association. The attribute exp of c is assigned the value from the attribute minRE of *tsi*.

(c) For all transitions $pt2$ for which the target state is the final state. The precondition of $pt2$ is appended with the string resulting from the concatenation of the strings " AND ", the name of sv_{ctr}, the string >=, and the name of sv_{min}.

(7) If the max attribute of *tsi* is set to a value:

(a) A new StateVariable sv_{max} is created in m_o and linked to the Test System component through its testVariables association. The attribute attName of sv_{max} is assigned the string resulting from the concatenation of the string MAX_evt_ and the string value of the attribute eventNameRE of *tsi*. For instance, for the authenticate event in the ATM example, sv_{max} would be named MAX_evt_authenticate. sv_{max} is associated with the data type dt_{int} whose typeName attribute is "Integer" through its association attType. The attribute isReadOnly of sv_{max} is assigned to the Boolean value "true".

(b) A new Constraint $c2$ is created in m_o and linked to the state variable sv_{max} through its constant association. The attribute exp of $c2$ is assigned the value from the attribute maxRE of *tsi*.

(c) For all transitions $pt3$ for which the triggering event name of $pt3$ is equals to the attribute eventNameRE of *tsi*. The precondition of $pt3$ is appended with the string resulting from the concatenation of the strings " AND ", the name of sv_{ctr}, the string <, and the name of sv_{max}.

5.3.5 Semantics of RepeatState instructions

Description

The model transformation associated with instructions of type RepeatState consists in seven steps. These steps are similar to the model transformation of instructions of type RepeatEvent described previously so that only the main differences are described in the following.

The created counter variable counts the number of times the given state has been active since the start of a simulation of the FOREST model. Consequently, the counter is incremented when a protocol transition is taken that leads to the state. In order to control the maximum number of time the state can be active, all the protocol transitions going to that state have their preconditions constrained so that the counter value must be lower than the maximum bound of the instruction. The handling of the minimum bound is the same than for the RepeatEvent instructions.

The constrained model illustrated by Figures 5.7 and 5.8 result from the application of the test selection model in Figure 5.3. We can notice that a counter variable ctr_ps_waitForPerm has been created as a state variable in the Test System component. That this variable counts the number of times the state `waitForPerm` is active by incrementing the variable in the postcondition of the protocol transition going from state `Authenticated` to state `waitForPerm`. The minimum of one time activation of the state `waitForPerm` is enforced by a constraint on the precondition of the protocol transition going to the final state.

Algorithm

If the current test selection instruction *tsi* is of type `RepeatState` then the following model transformation is performed on the model m_o:

(1) There exists a protocol state *ps* in model m_o such that the two following conditions (a) and (b) are true. If there is no state *ps* found then the SERELA instruction *tsi* is ill-formed. If there are more than one states fulfilling the conditions then the FOREST model m_o, and subsequently m_i, is ill-formed because they do not fulfill the rule "WFR FOREST.13" described in Section 4.2.1.

  (a) *ps* is an instance of the meta-class `ProtocolState`

  (b) the attribute `stateName` of state *ps* is equal to the attribute `stateNameRS` of instruction *tsi*

(2) A new `StateVariable` sv_{ctr} is created in m_o and linked to the Test System component through its `testVariables` association. The attribute `attName` of sv_{ctr} is assigned the string value resulting from the concatenation of the string `ctr_ps_` and the string value of the attribute `stateNameRS` of *tsi*. For instance, for the `waitForCard` state in the ATM example, sv_{ctr} would be named `ctr_ps_waitForCard`. sv_{ctr} is associated with the data type dt_{int} whose `typeName` attribute is "Integer" through its association `attType`.

(3) If there is no InitialValue `Constraint` *c* in m_o then *c* is created in m_o and the attribute `exp` of *c* is set to the name of sv_{ctr} appended with the string " `= 0` ".

(4) If there is an InitialValue `Constraint` *c* in m_o then the attribute `exp` of *c* is appended with the string resulting from the concatenation of " `AND` ", name of sv_{ctr} and " `= 0` ". For instance, with the state name `waitForCard`, this would result in appending the string `AND ctr_ps_waitForCard = 0`

(5) For all transitions *pt*3 for which the target state name equals to the attribute `stateNameRS` of *tsi*. The post-condition of *pt*3 is appended with the string resulting from the concatenation of the strings " `AND` ", the name of sv_{ctr}, the string `=` , the name of sv_{ctr}, and the string "`@pre + 1`".

(6) If the `min` attribute of *tsi* is set to a value:

  (a) A new `StateVariable` sv_{min} is created in m_o and linked to the Test System component through its `testVariables` association. The attribute `attName` of sv_{min} is assigned the string value resulting from the concatenation of the string `MIN_ps_` and the string value of the attribute `stateNameRS` of *tsi*. For instance, for the `waitForCard` state in the ATM example, sv_{min} would be named `MIN_ps_waitForCard`. sv_{min} is associated with the data type dt_{int} whose `typeName` attribute is "Integer" through its association `attType`. The attribute `isReadOnly` of sv_{min} is assigned to the Boolean value "true".

(b) A new `Constraint` c is created in m_o and linked to the state variable sv_{min} through its `constant` association. The attribute `exp` of c is assigned the value from the attribute `minRS` of *tsi*.

(c) For all transitions *pt* for which the target state is the final state. The precondition of *pt* is appended with the string resulting from the concatenation of the strings " `AND` ", the name of sv_{ctr}, the string >=, and the name of sv_{min}.

(7) If the `max` attribute of *tsi* is set to a value:

(a) A new `StateVariable` sv_{max} is created in m_o and linked to the Test System component through its `testVariables` association. The attribute `attName` of sv_{max} is assigned the string value resulting from the concatenation of the string `MAX_ps_` and the string value of the attribute `stateNameRS` of *tsi*. For instance, for the `waitForCard` state in the ATM example, sv_{max} would be named `MAX_ps_waitForCard`. sv_{max} is associated with the data type dt_{int} whose `typeName` attribute is "Integer" through its association `attType`. The attribute `isReadOnly` of sv_{max} is assigned to the Boolean value "true".

(b) A new `Constraint` $c2$ is created in m_o and linked to the state variable sv_{max} through its `constant` association. The attribute `exp` of $c2$ is assigned the value from the attribute `maxRS` of *tsi*.

(c) For all transitions $pt2$ for which the target state name equals to the attribute `stateNameRS` of *tsi*. The precondition of $pt2$ is appended with the string resulting from the concatenation of the strings " `AND` ", the name of sv_{ctr}, the string <, and the name of sv_{max}.

5.3.6 Semantics of RepeatTrans instructions

Description

The model transformation associated with instructions of type `RepeatTrans` consists in seven steps. These steps are similar to the model transformation of instructions of type `RepeatEvent` described previously, only the main differences are described in the following.

The created counter variable counts the number of times the parameter transition of the instruction has been taken since the start of a simulation of the FOREST model. Consequently, the counter is incremented when the given protocol transition is taken, via its post condition. In order to control the maximum number of times the transition can be taken, the protocol transition's precondition is constrained so that the counter value must be lower than the maximum bound of the instruction. The handling of the minimum bound is the same than for the `RepeatEvent` instructions.

Algorithm

If the current test selection instruction *tsi* is of type `RepeatTrans` then the following model transformation is performed on the model m_o:

(1) There exists a transition *pt* in model m_o such that the two following conditions (a) and (b) are true. If there is no transition *pt* found then the SERELA instruction *tsi* is ill-formed. If there are more than one transitions fulfilling the conditions then the FOREST

model m_o, and subsequently m_i, is ill-formed because they do not fulfill the rule "WFR FOREST.20" described in Section 4.2.1.

(a) *pt* is an instance of the meta-class `ProtocolTransition`

(b) the attribute `transName` of transition *pt* is equal to the attribute `transNameRT` of instruction *tsi*

(2) A new `StateVariable` *sv* is created in m_o and linked to the Test System component through its `testVariables` association. The attribute `attName` of *sv* is assigned the string value resulting from the concatenation of the string `ctr_pt_` and the string value of the attribute `transNameRT` of *tsi*. For instance, for the `tr5` transition in the ATM example, *sv* would be named `ctr_pt_tr5`. *sv* is associated with the data type dt_{int} whose `typeName` attribute is "Integer" through its association `attType`.

(3) If there is no InitialValue `Constraint` *c* in m_o then *c* is created in m_o and the attribute `exp` of *c* is set to the name of *sv* appended with the string " ` = 0 ` ".

(4) If there is an InitialValue `Constraint` *c* in m_o then the attribute `exp` of *c* is appended with the string resulting from the concatenation of " ` AND ` ", name of *sv* and " ` = 0 ` ". For instance, with the transition named `tr5`, this would result in appending the string ` AND ctr_pt_tr5 = 0`

(5) For all transitions *pt*3 named like the attribute `transNameRT` of *tsi*. The post-condition of *pt*3 is appended with the string resulting from the concatenation of the strings " ` AND ` ", the name of sv_{ctr}, the string ` = `, the name of sv_{ctr}, and the string "`@pre + 1`".

(6) If the `min` attribute of *tsi* is set to a value:

(a) A new `StateVariable` *sv*2 is created in m_o and linked to the Test System component through its `testVariables` association. The attribute `attName` of *sv*2 is assigned the string value resulting from the concatenation of the string `MIN_pt_` and the string value of the attribute `transNameRT` of *tsi*. For instance, for the `tr5` transition in the ATM example, *sv*2 would be named `MIN_pt_tr5`. *sv*2 is associated with the data type dt_{int} whose `typeName` attribute is "Integer" through its association `attType`. The attribute `isReadOnly` of *sv*2 is assigned to the Boolean value "true".

(b) A new `Constraint` *c* is created in m_o and linked to the state variable *sv*2 through its `constant` association. The attribute `exp` of *c* is assigned the value from the attribute `minRT` of *tsi*.

(c) For all transitions *pt* for which the target state is the final state. The precondition of *pt* is appended with the string resulting from the concatenation of the strings " ` AND ` ", the name of sv_{ctr}, the string >=, and the name of sv_{min}.

(7) If the `max` attribute of *tsi* is set to a value:

(a) A new `StateVariable` *sv*3 is created in m_o and linked to the Test System component through its `testVariables` association. The attribute `attName` of *sv*3 is assigned the string value resulting from the concatenation of the string `MAX_pt_` and the string value of the attribute `transNameRT` of *tsi*. For instance, for the `tr5` transition in the ATM example, *sv*3 would be named `MAX_pt_tr5`. *sv*3 is associated with the data type dt_{int} whose `typeName` attribute is "Integer" through its association `attType`. The attribute `isReadOnly` of *sv*3 is assigned to the Boolean value "true".

(b) A new `Constraint` $c2$ is created in m_o and linked to the state variable $sv3$ through its `constant` association. The attribute `exp` of $c2$ is assigned the value from the attribute `maxRT` of *tsi*.

(c) For all transitions $pt2$ named like the attribute `transNameRT` of *tsi*. The precondition of $pt2$ is appended with the string resulting from the concatenation of the strings " `AND` ", the name of sv_{ctr}, the string <, and the name of sv_{max}.

5.3.7 Semantics of Scope instructions

Description

The model transformation associated with instructions of type `Scope` consists in two steps. In steps (1) and (2): a new constant state variable representing the simulation scope (number of maximum steps of a simulation of the FOREST model) is created in the Test System component and set to the value of the scope instruction parameter.

Algorithm

If the current test selection instruction *tsi* is of type `Scope` then the following model transformation is performed on the model m_o:

(1) a new state variable *sv* is created in m_o. *sv* is associated with the Test System component through its `testVariables` association. The attribute `varName` of *sv* is set to "scope". The attribute `isReadOnly` of the variable *sv* is set to true.

(2) a new `Constraint` c is created in m_o. The constraint c is associated with the variable *sv* through its `constant` association. The attribute `exp` of constraint c is set to the attribute `scope` of *tsi*.

5.3.8 Semantics of InitialState instructions

Description

The model transformation associated with instructions of type `InitialState` consists in two steps. In step (1): the protocol state targeted by the instruction must be found in the FOREST model with the help of its name and type. In step (2), the protocol state is set to be the new initial state of the dynamic view of the FOREST model.

Algorithm

If the current test selection instruction *tsi* is of type `InitialState` then the following model transformation is performed on the model m_o:

(1) There exists a protocol state *ps* in model m_o such that the two following conditions (a) and (b) are true. If there is no state *ps* found then the SERELA instruction *tsi* is ill-formed. If there are more than one states fulfilling the conditions then the FOREST model m_o, and subsequently m_i, is ill-formed because they do not fulfill the rule "WFR FOREST.13" described in Section 4.2.1.

   (a) *ps* is an instance of the meta-class `ProtocolState`

   (b) the attribute `stateName` of state *ps* is equal to the attribute `stateNameIS` of instruction *tsi*

(2) the `initialState` association of the SUT component with a `ProtocolState` is modified and is now set to the protocol state *ps*.

5.3.9 Semantics of InitialVal instructions

Description

The model transformation associated with instructions of type `InitialVal` consists in two steps. In step (1) an InitialValue constraint is created, if necessary. In step (2): the InitialValue constrain of the FOREST model is set to the expression given as parameter of the test selection instruction. As a consequence, if there was already an InitialValue set in the FOREST model, it is reset by this instruction.

Algorithm

If the current test selection instruction *tsi* is of type `InitialVal` then the following model transformation is performed on the model m_o:

(1) If there is no InitialValue constraint in m_o then a new `Constraint` c is created in m_o and linked to the model through the `InitialValue` association.

(2) the attribute `exp` of the initial value of the model m_o is set to the attribute `expIV` of *tsi*.

5.3.10 Semantics of FinalState instructions

Description

The model transformation associated with instructions of type `FinalState` consists in three steps. In step (1): the protocol state targeted by the instruction must be found in the FOREST model with the help of its name and type. The final state is created if necessary in step (2). In step (3) a new protocol transition[2] is created that goes from the protocol state parameter of the instruction to the final state. The name of this new transition is given by the second parameter of the instruction.

[2] As mentioned in the introduction of this section, some instructions are correlated. FinalState is one of them. It is related to RepeatEvent, RepeatState and RepeatTrans test selection instructions. This is because, the three last instruction types modify protocol transitions going to the final state; and the FinalState instruction creates a new protocol transitions going to the final state. During the implementation of the semantics, special care is given to this correlation so that the meaning of the SERELA test selection is preserved. In this case, the FinalState instructions must be interpreted after RepeatEvent, RepeatState and RepeatTrans test selection instructions, so that the new protocol transition introduced by the FinalState instruction is created first, then the preconditions of all transitions including the newly created protocol transitions are modified.

Algorithm

If the current test selection instruction *tsi* is of type `FinalState` then the following model transformation is performed on the model m_o:

(1) There exists a protocol state *ps* in model m_o such that the two following conditions (a) and (b) are true. If there is no state *ps* found then the SERELA instruction *tsi* is ill-formed. If there are more than one states fulfilling the conditions then the FOREST model m_o, and subsequently m_i, is ill-formed because they do not fulfill the rule "WFR FOREST.13" described in Section 4.2.1.

  (a) *ps* is an instance of the meta-class `ProtocolState`

  (b) the attribute `stateName` of state *ps* is equal to the attribute `stateNameFS` of instruction *tsi*

(2) if there is none of the protocol states in `pstates` associated with the SUT component named "final", a new protocol state `ProtocolState` $ps2$ is created in m_o and linked to the Test System component through its `pstates` association. The attribute `stateName` of $ps2$ is assigned to the string "final".

(3) A new protocol transition *pt* is created in m_o and linked to the SUT component through its association `ptransitions`. The attribute `transName` of transition *pt* is set to the string value of the attribute `finalTransNameFS` of *tsi*. The transition *pt* is associated with the final protocol state through its `target` association. The transition *pt* is associated with the protocol state *ps* through its `source` association. For instance, in the ATM example, if the instruction type `FinalState` is processed given the `Authenticated` protocol state as a parameter, a new protocol transition from the state `Authenticated` to the final state is created.

5.3.11 Semantics of FinalTrans instructions

Description

The model transformation associated with instructions of type `FinalTrans` consists in three steps. In step (1): the protocol transition targeted by the instruction must be found in the FOREST model with the help of its name and type. The final state is created if necessary in step (2). The association is redirected to the final state[3] in step (3).

Algorithm

If the current test selection instruction *tsi* is of type `FinalTrans` then the following model transformation is performed on the model m_o:

(1) There exists a protocol transition *pt* in model m_o such that the two following conditions (a) and (b) are true. If there is no transition *pt* found then the SERELA instruction *tsi* is ill-formed. If there are more than one transitions fulfilling the conditions then the

[3] Similarly than for the FinalState instruction. The redirection of the protocol transition to the final state result in a correlation of the FinalTrans instructions and the RepeatEvent, RepeatState and RepeatTrans instructions.

FOREST model m_o, and subsequently m_i, is ill-formed because they do not fulfill the rule "WFR FOREST.20" described in Section 4.2.1.

(a) *pt* is an instance of the meta-class `ProtocolTransition`

(b) the attribute `transName` of transition *pt* is equal to the attribute `transNameFT` of instruction *tsi*

(2) if there is none of the protocol states in `pstates` associated with the SUT component named "final", a new protocol state `ProtocolState` *ps* is created in m_o and linked to the Test System component through its `pstates` association. The attribute `stateName` of *ps* is assigned to the string "final".

(3) the `target` association of transition *pt* is modified and is now associated with the final protocol state.

5.4 Implementation of SERELA semantics with KerMeta

The SERELA semantics is implemented as a KerMeta model transformation[4]. The implementation of the SERELA semantics is composed of a main operation, given in Appendix B.1, which processes each test selection instruction of the input test selection model and calls the operation associated to the kind of test selection instruction being processed. Eleven operations are defined, one for each of the eleven kind of SERELA instructions. In this section, we present two of these operations that implement the semantics of the SERELA instructions `SelectEvent` and `FinalTrans`.

5.4.1 Implementation of the semantics of the SelectEvent instruction

The implementation of the semantics of the SelectEvent instruction, given in Figure 5.9 by the operation `tiSelectEvent`, is based on the SelectEvent semantics definition, given in Section 5.3.1. The operation `tiSelectEvent` is composed of one step (implemented as the operation `addEventConstraint`), that is applied twice; once, for each event receptions of the SUT (line 02) and then for each event receptions of the Test System (line 03). Lines 02-03 correspond to the part 1.a of the semantics definition in Section 5.3.1.

The addEventConstraint performs the addition of the constraint only if the current event is the one to which the constraint must be added to (line 06 of Figure 5.9) that corresponds to part 1.b in Section 5.3.1. Then, finally if the names of the events match, then a new constraint is created in model `ma` and added to the current event (lines 07-09 of Figure 5.9) that corresponds to part 2) in Section 5.3.1.

5.4.2 Implementation of the semantics of the FinalTrans instruction

The implementation of the semantics of the FinalTrans instruction, given in Figure 5.10 by the operation `tiFinalTrans`, is based on the FinalTrans semantics definition, given in Section 5.3.11.

The first step of the semantics (step 1.a and 1.b in Section 5.3.11) is to find the protocol transition in `ma` that matches the name given as parameter of the operation `tiFinalTrans`. This first step is implemented in lines 02-04 of Figure 5.10.

The second step (i.e. step 2 in Section 5.3.11) is to create a final state if there is no final state in the model `ma`, implemented in lines 07-10; the newly created final state is assigned to the variable `finalPS`. If a final state is present in model `ma`, line 11-12, then it is selected and assigned to the variable `finalPS`.

The last step (i.e. step 3 in Section 5.3.11) is to set the target state of the protocol transition given as parameter of the FinalTrans instruction to the final state `finalPS`, straightforwardly implemented in line 14 of Figure 5.10.

[4] In a similar way than the implementation of the FOREST semantics previously described in Section 4.6

```
operation tiSelectEvent(tsi: SelectEvent, model : ForestModel) is do
 model.sut.eventsReceived.each{ evt | addEventConstraint(tsi,evt)}
 model.ts.eventsReceived.each{  evt | addEventConstraint(tsi,evt)}
end

operation addEventConstraint(tsi: SelectEvent, evt : Event) is do
   if (evt.evtName.equals(tsi.evtNameSE)) then
     var c : Constraint init Constraint.new
     c.exp := tsi.expSE
     evt.event.add(c)
   end
 end
```

Fig. 5.9: Semantics implementation of SERELA SelectEvent instruction in KerMeta

```
operation tiFinalTrans(tsi: FinalTrans, model : ForestModel) is do
 model.sut.ptransitions.each{pt |
 do
   if (pt.transName.equals(tsi.transNameFT)) then
     var finalPS : ProtocolState
     //if no final state in the PSM then we create one
     if (model.sut.pstates.select{
          ps2 | ps2.stateName.equals("final")}.one.isVoid) then
       finalPS := ProtocolState.new
       finalPS.stateName := "final"
       model.sut.pstates.add(finalPS)
     else
       finalPS := model.sut.pstates.select{
                   ps2 | ps2.stateName.equals("final")}.one
     end
     pt.target := finalPS
   end
 end
 }
end
```

Fig. 5.10: Semantics implementation of SERELA FinalTrans instruction in KerMeta

6. METRICS FOR THE EVALUATION OF THE TEST SELECTION ACTIVITIES

Abstract

Constrained models are FOREST models which reflect a particular test selection. Constrained models must be validated by the Test Manager before being used for test generation. In this chapter, we present a number of metrics allowing the evaluation of constrained models. We start by defining two different types of metrics: structural and semantical metrics. Then we give the semantics for these metrics. The semantics for structural metrics is given in terms of KerMeta instructions. The semantics of the semantical metrics is given in terms of Alloy specifications. Lastly, we present how the metrics may be used at a business-level. And how these business-level metrics may be used in three different levels of re-usability: general-purpose, domain-specific and project-specific metrics.

6.1 Introduction

In the previous chapter, we have presented SERELA, a test selection language allowing the expression of test selection constraints on FOREST models. A SERELA test selection specifications may be incorrect with respect to some test selection objectives from the Test Manager. In this chapter, we present a number of *model coverage metrics* [1] that aim at expressing the Test Manager's objectives, and validating the SERELA test selection specification[2] with respect to the objectives. The validation of the test selection is an important activity in testing. By introducing the validation of the test selection, the SESAME approach allows providing an early feedback on the test case design before the test cases before their concrete execution on the system under test. In the context of embedded systems, this early feedback is particularly valuable as the test execution may be taking lots of time, thus knowing about the test selection before its execution may gain considerable time.

A *metric* is a specification of something to be measured. Metrics are means to analyze the complexity of things. In the context of the SESAME approach, FOREST models are complex in the sense that they comprise a multitude of model elements and that each model element may have a number of instances. We categorize the notion of complexity in two complementary notions: *structural complexity* that is the complexity resulting from the different elements of a model; the *semantical complexity* that is the complexity resulting from the multitude of instances of the model elements. In this chapter, we provide metrics that define what to measure within

[1] *Model coverage metrics* are also simply referred to as *metrics* in the remaining of this thesis.

[2] The validation of SERELA test selection specifications imply the validation of the test model resulting from the test selection specification. A validated test model is ready for test generation.

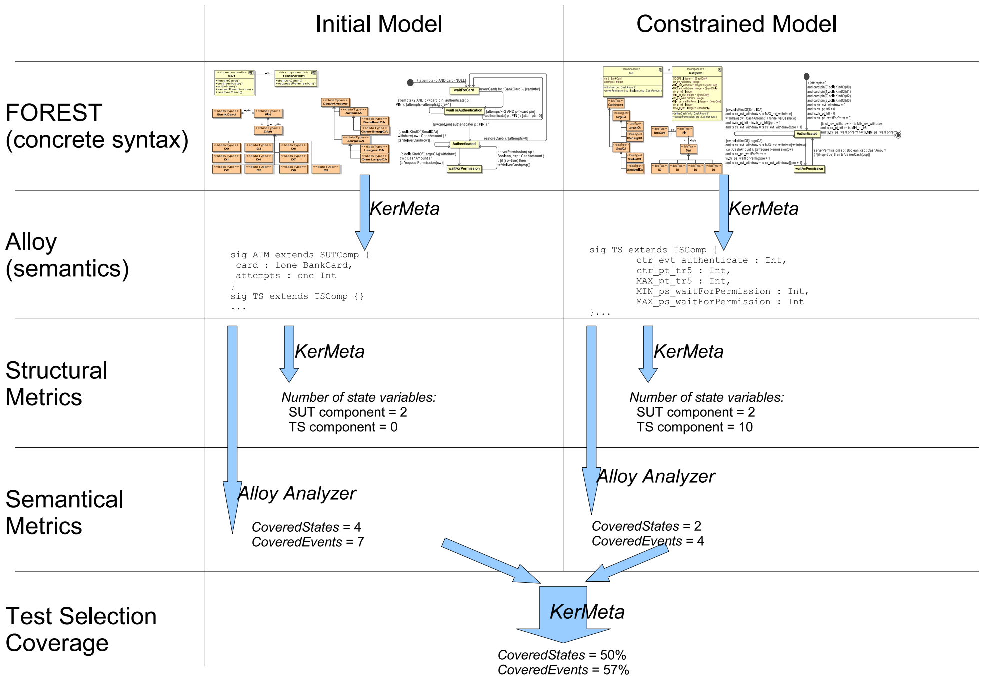


Fig. 6.1: SESAME test selection evaluation approach

FOREST models, in order to deal with the different kinds of models' complexity. *Structural metrics* and *semantical metrics* analyze respectively the structural and semantical complexity of FOREST models.

In the SESAME approach, the structural and semantical metrics are used for two purposes. Firstly, as presented above, the metrics values are used to analyze the complexity of a given FOREST model.

Secondly, the metrics are used for the evaluation of the test selection. The evaluation is based on analyzing the test selection coverage for each computed metrics. The *test selection coverage* consists in comparing the values of the metrics before the test selection, i.e. on the initial model, and after the test selection, i.e. on the constrained model.

6.2 Structural metrics

Structural metrics define measurements on the number of elements of FOREST models. The semantics of each structural metrics is defined in the following Section 6.4.1.

6.2.1 Metrics based on the number of FOREST model elements

There could be as many structural metrics as there are kinds of FOREST model elements. In our context, as described in the introduction, we focus on evaluating the progress of the test selection process. Consequently, we are only interested in measurements involving kinds of FOREST model elements that may be modified with SERELA test selection instructions. Some of the SERELA instructions create new elements in FOREST models, but none of them remove elements. Consequently, the two values of a structural metric for a model and a related constrained model may only result in an increase of the value.

The kinds of FOREST models elements that may be added by the interpretation of SERELA instructions are:

- *State variables.* More particularly, the four SERELA instructions `RepeatEvent`, `RepeatState`, `RepeatTrans` and `Scope` create new state variables in the TestSystem Component.
- *Protocol states.* the two SERELA instructions `FinalTrans` and `FinalState` may create a final protocol state;
- *Protocol transitions.* the SERELA instruction `FinalState` creates a new protocol transition.

We suggest that metrics measuring the total number of these three kinds of elements may be of help to evaluate how a given test selection impacts the initial input model. Three metrics are defined `TotalTestSystemStateVariables`, `TotalStates` and `TotalTransitions` which respectively count the total number of state variables of the Test System, the total number of protocol states and the total number of protocol transitions specified in a given FOREST model.

Table 6.1 illustrates these three metrics on the FOREST initial model of the ATM (shown in Figures 3.1 and 3.2) and its constrained model resulting from the SERELA test selection of Figure 5.3. In this example, the test selection have added 10 state variables to the Test System component, one protocol state and one protocol transition to the dynamic view of the model.

	Metrics	**Evaluated Model**		**Test Selection Coverage**
		Initial	**Constrained**	
Structural Metrics				
1	TotalTestSystemStateVariables	0	10	
2	TotalStates	4	5	
3	TotalTransitions	8	9	
Semantical metrics: number of instances				
6	TotalTestCases	100 000+	100 000+	N/A
7	TotalInstancesOfDataType(CashAmount)	4	4	100%
8	TotalInstancesOfEvent(insertCard)	10.000	10.000	100%
Semantical metrics: partial coverage of FOREST model elements				
9	CoveredLeafDataTypes	14	8	57%
10	CoveredLeafDataTypeOf(CashAmount)	4	4	100%
11	CoveredLeafDataTypeOf(Digit)	10	4	40%
12	CoveredStates	4	3	75%
13	CoveredState(WaitForCard)	true	false	0%
	...	...	...	...
17	CoveredState(Authenticated)	true	true	100%
18	CoveredEvents	7	4	57%
19	CoveredEvent(insertCard)	true	false	0%
	...	...	...	...
25	CoveredEvent(withdraw)	true	true	100%
26	CoveredTransitions	8	4	50%
27	CoveredTransition(tr1)	true	false	0%
	...	...	...	...
35	CoveredTransition(tr5)	true	true	100%
36	CoveredStatePath(Authenticated, WaitForPermission, Authenticated)	true	true	100%
37	CoveredTransPath(tr3, tr4, tr5)	true	false	0%
Semantical metrics: partial coverage of specific (set of) instances				
38	CoveredValueOfStructuredDatatype(PIN, digits[0].oclIsKindOf(D0) and digits[1].oclIsKindOf(D0))	true	false	0%
39	CoveredValueOfEvent(withdraw, cw.oclIsKindOf(LargestCA))	true	true	100%
40	CoveredValueOfTrans(tr1, bc.pin.digits[0].oclIsKindOf(D0))	true	true	100%

Tab. 6.1: ATM example - Illustration of some of the structural and semantical metrics on the sample test selection described in Figure 5.3

6.3 Semantical metrics

We suggest using two kinds of metrics to measure the semantic complexity of FOREST models:

- semantical metrics defining measurements based on the number of instances of the model or its model elements.
- semantical metrics defining measurements based on the coverage of model elements instances.

The semantics of each semantical metrics is defined in the following Sections 6.4.2 and 6.4.3.

6.3.1 Metrics based on the number of instances of FOREST model elements

Number of test cases

The metric `TotalTestCases` describes the overall semantic complexity of a given FOREST model. It is defined as the amount of all the possible test cases of a FOREST model, i.e. the number of all the simulations of the model starting from the initial protocol state and ending in the final protocol state. This metric may be used, for instance, when Test Managers wishes to set a maximum amount of test cases to exercise the SUT. The metric `TotalTestCases` inform about the number of test cases and thus tell Test Managers if the test selection has been sufficient to keep the number of test cases within the maximum they have set.

This metric is not sufficiently fine-grained to evaluate all the aspects of the test selection process. We provide additional semantical metrics defining measurements on possible instances of FOREST models elements. Similarly than for structural metrics, semantical metrics are defined which measure the total number of possible instances of each element type of interest. The three following metrics are defined in the following: `TotalInstancesOfDataType`, `TotalInstancesOfEvent` and `TotalInstancesOfTrans`. [3]

Number of instances of a given complex data type

The metrics `TotalInstancesOfDataType()` are defined as the number of instances of a given complex data type. The measurement is performed based on the specification of the date type in the static view of the related FOREST model, regardless of whether any data of this type is actually used or not during a model simulation. This metric allows the Test Manager to evaluate if the data types are sufficiently constrained. The value for this type of metrics is directly linked to the SERELA instructions of type `SelectData` which constrain data types.

Abstract data types and enumerations can not be instantiated and leaf data types have a unique instance. Thus, it is not relevant to have metrics counting the number of instances for these three other kinds of data types.

[3] Each protocol state model element has a unique instance. Thus a metric counting the "number of instances of a given protocol state" is not relevant as it would always result in the number of instances being one.

Number of instances of a given event

The metrics `TotalInstancesOfEvent()` are defined as the number of instances of a given event type. The measurement is performed based on the specification of the event type in the static view of the related FOREST model, regardless of whether any events of this type are actually used or not during a model simulation. This metric allows the Test Manager to evaluate if the event types are sufficiently constrained. The value for this type of metrics is directly linked to the SERELA instructions of type `SelectEvent` which constrain event types.

6.3.2 Metrics based on the coverage of instances of FOREST model elements

In the context of testing, the measurement based on coverage of the model element instances is crucial information to grasp the complexity of the potential test cases of a given model. We say that a given instance of a FOREST model element is covered when the instance is part of at least one simulation of the FOREST model. The notion of a model element instance being part of a model simulation is described in the following. The definitions depend on the type of the model element. We define two types of semantical metrics based on coverage of model element instances:

- *Partial coverage metrics.* We say that a model element is partially covered, if there is at least one instance of that model element that is part of at least one simulation of the FOREST model.
- *Exhaustive instance coverage metrics*[4]. We say that a model element is exhaustively covered, if and only if all the instances of that model element are part of at least one simulation of the FOREST model. As a consequence, if exhaustive coverage is fulfilled then partial coverage is also fulfilled.

Data type instance coverage metrics

Before describing the metrics based on the measurement of the coverage of data types, we define the notion of instance coverage for data type model elements. FOREST allows specifying different kinds of data types: structured, abstract, leaf, enumeration. Instance coverage of a data type is defined differently for each kind of data type. Given a FOREST model:

- a leaf data type is *covered*[5] if and only if there exists one simulation during which there is at least one event reception which uses the data type either as a parameter type or as a part of a structured parameter type.
- an abstract data type is *covered* if and only if all its direct subtypes (i.e. data types extending it) are covered. An abstract data type is *partially covered* if at least one of its direct subtypes is partially covered but not all of them are.
- a structured data type is *covered* if and only if all its instances is covered. It is partially covered if at least one of its instances is covered.

[4] In order to make the remaining of this chapter more readable, we imply *exhaustive coverage* when referring to *coverage*. *Partial coverage* will always be mentioned explicitly.

[5] Leaf data types always have a single instance, thus there is no need to differentiate between partial coverage or exhaustive coverage.

- an enumeration data type is *covered* if and only if all its enumeration literals (i.e. the leaf data types extending the enumeration) are covered.

Abstract data type instance coverage metrics

The set of metrics `CoveredLeafDatatypeOf` evaluate the coverage of abstract data types, by measuring the number of covered leaf data types of a given abstract data type out of the number of specified leaf data types for the given abstract data type. This metric compares the measurement of some information of the semantics complexity (instance coverage of leaf data types) with the structural metric (`TotalPartitionOf`). For instance in Table 6.1, a metric of this type is computed given the abstract data types `CashAmount` on the one hand, and `Digit` on the other hand. It results in that the data type `CashAmount` is covered as all its leaf data types are covered (4 out of 4) and that the data type Digit is partially covered as not all its leaf data types are covered.

Enumerations are abstract data types; consequently, instance coverage metrics for abstract data types are also applicable to enumeration data types.

Leaf data type instance coverage metrics

The metric `CoveredLeafDatatypes` gives an overview of the coverage of the leaf data type regardless of their relation to an abstract data type. It is defined as the overall number of covered leaf data types out of the number of leaf data types of a given model.

The test selection coverage informs about the effect of a given test selection on the initial model. The test selection coverage of the metrics `CoveredLeafDatatypeOf` and `CoveredLeafDatatypes` is the proportion, expressed as a percentage, of the value of the metric for the constrained model compared with the value of the initial model. For instance, in Table 6.1, The value of the metric `CoveredLeafDatatypes` for the initial model is 100% (i.e. all the leaf data types are covered in the initial model) and it is 57% for the constrained model (because only 8 leaf data types are covered out of 14 in total). Thus the test selection coverage is 57 out of 100 which results in 57%. This example is a particular case for which the initial model has 100% instance coverage that is why the constrained model metric value equals the test selection coverage.

Structured data type instance coverage metrics

The metric `CoveredValueOfStructuredDatatype` is defined as the partial coverage of the given structured data type. The instance must fulfill the OCL-F constraint given as a parameter of the metric. For instance, Table 6.1 illustrates a metric of this type giving as parameter (1) the structured data type `PIN` and (2) the constraint that the instance must be with the first and second digits equals to 0.

The metric `CoveredValueOfStructuredDatatype` has a Boolean result, so the previous definition of test selection coverage does not apply to this metric. For Boolean metrics values, the test selection coverage is 100% if and only if to the value of the metric for the constrained model evaluates to true otherwise it is 0%.

State variable instance coverage metrics

Given a FOREST model and an OCL-F constraint on a state variable of the model, a state variable is *partially covered* if and only if there is at least one status of some simulation of the given model for which the state variable satisfies the given constraint.

The metrics `CoveredValueOfStateVar()` is defined as the partial coverage of the state variable(s) targeted by the OCL-F parameter of the metric.

Protocol state instance coverage metrics

Given a FOREST model, a protocol state is covered if and only if it is an active state of some status of some simulation of the given model.

The metric `CoveredStates` is defined as the amount of covered protocol states in the model. A set of finer-grained metrics `CoveredState()` are defined as the coverage of a given protocol state.

Event reception instance coverage metrics

An event type may have a number of instances depending on the data type of its parameters, thus we define the coverage of an event type and the coverage of its instances.

Given a FOREST model, an event type is partially covered if at least one of its instances is received (by the SUT or by the Test System) in some step of some simulation of the given FOREST model.

The metric `CoveredEvents` is defined as the amount of partially covered event types in the model. A set of finer-grained metrics `CoveredEvent()` are defined as the coverage of a given event type.

The metrics `CoveredValueOfEvent()` are defined as the coverage of an instance of a given event type. The instance must fulfill the OCL-F constraint given as a parameter of the metric. For instance, Table 6.1 illustrates a metric of this type giving as parameter (1) the event type `withdraw` and (2) the constraint that the instance must be with its parameter `c` of type large amount.

Protocol transition instance coverage metrics

Given a FOREST model, a protocol transition is covered if and only if it may be taken in some step of some simulation of the given model. The protocol transition coverage is the proportion of covered protocol transitions compared to the amount of protocol transitions present in the model.

The metric `CoveredTransitions` is defined as the amount of partially covered protocol transitions in the model. A set of finer-grained metrics `CoveredTrans()` are defined as the coverage of a given protocol transition.

The metrics `CoveredValueOfTrans()` are defined as the coverage of an instance of a given protocol transition. The instance must fulfill the OCL-F constraint given as a parameter of the metric.

Model simulation instance coverage metrics

The `TotalTestCases` metrics, described in the former section, measures the number of all possible simulations of a given FOREST model. The *model simulation coverage metrics* define finer-grained metrics that aim at checking if certain simulations satisfying certain constrains are covered, i.e. are one of the possible simulations of the given model or not. We define two model simulation coverage metrics: `CoveredStatePath` and `CoveredTransitionPath`.

The metric `CoveredStatePath` is parameterized by a number of protocol states expressing a path. The metric is defined as a Boolean value, which evaluates to true when at least one model simulation contains a sequence of the parameter protocol states.

Similarly, the metric `CoveredTransitionPath` is parameterized by a number of protocol transitions expressing a path. The metric is defined as a Boolean value, which evaluates to true when at least one model simulation contains a sequence of the parameter protocol transitions.

6.4 Metrics semantics

6.4.1 Semantics of the structural metrics

The semantics of structural metrics is defined based on the FOREST meta-model (given in Figure 4.3 of Chapter 4). The semantics of the structural metrics have been straightforwardly implemented in KerMeta as shown in Figure 6.2.

Semantics of TotalTestSystemStateVariables metric

The semantic of the metric `TotalTestSystemStateVariables` is the number of instances of the meta-class `StateVariable` associated with the `TSComponent` meta-class through the association `testVariables`.

Semantics of TotalStates metric

The semantic of the metric `TotalStates` is the number of instances of the meta-class `ProtocolState` associated with the `SUTComponent` meta-class through the association `pstates`.

Semantics of TotalTransitions metric

The semantic of the metric `TotalTransitions` is the number of instances of the meta-class `ProtocolTransition` associated with the `SUTComponent` meta-class through the association `ptransitions`.

```
operation metricsTotalTestSystemStateVariables(model : ForestModel) : Integer is do
  result := model.ts.testVariables.size()
end

operation metricsTotalStates(model : ForestModel) : Integer is do
  result := model.sut.pstates.size()
end

operation metricsTotalTransitions(model : ForestModel) : Integer is do
  result := model.sut.ptransitions.size()
end
```

Fig. 6.2: ATM: semantics implementation in KerMeta of the structural metrics

6.4.2 Semantics of the metrics based on the number of instances of FOREST model elements

In order to compute the number of instances of particular FOREST model elements, these particular model elements are isolated in an Alloy model. Then all the instances of this Alloy model are generated and counted.

Semantics of TotalTestCases metric

FOREST model semantics is given in terms of Alloy models. The TotalTestCases metric is the number of possible instances of a FOREST model. Thus the Alloy command is simply an evaluation of the FOREST model with the scope given in the state variable SCOPE of the TestSystem component. For instance, for a scope of 10, the command is simply an Alloy command `run` with the given scope specified as follows: `run {} for 10`.

Semantics of TotalInstancesOfDataType metric

In order to define the total number of instances of a data type, the definition of the data type must be isolated in a separate Alloy model to be evaluated. This Alloy model comprises the signatures declaration of the data type as well as the signatures required to fully define the data type.

For instance, the resulting Alloy model for the evaluation of the metric `TotalInstancesOfDataType()` with the data type `BankCard` as parameter is specified as shown in Figure 6.3.

Semantics of TotalInstancesOfEvent metric

In order to define the total number of instances of an event type, the definition of the event type must be isolated in a separate Alloy model to be evaluated. This Alloy model must comprise the signatures declaration of the event type as well as the signatures required to fully define the event type.

For instance, the resulting Alloy model for the evaluation of the metric `TotalInstancesOfEvent()` with the event type `insertCard` as parameter is specified as shown in Figure 6.4.

```
one sig BankCard  {
  pin: one PIN
}
sig PIN {
  digits: seq Digit
}
abstract sig Digit {}
one sig D0, D1, D2, D3, D4, D5, D6, D7, D8, D9 extends Digit{}

fact {
  all pin: PIN | #pin.digits = 4
}
fact distinctDatatypes {
  no disj dt, dt': PIN | dt.digits = dt'.digits
}
fact noUnusedData {
  all dt: PIN | some dtp : BankCard | dtp.pin = dt
}
```

Fig. 6.3: ATM: semantics in Alloy of the metric TotalInstancesOfDataType(BankCard)

```
one sig insertCard {
  bc: one BankCard
}
sig BankCard  {
  pin: one PIN
}
sig PIN {
  digits: seq Digit
}
abstract sig Digit {}
one sig D0, D1, D2, D3, D4, D5, D6, D7, D8, D9 extends Digit{}

fact {
  all pin: PIN | #pin.digits = 4
}
fact distinctDatatypes {
  no disj dt, dt': PIN | dt.digits = dt'.digits
}
fact noUnusedData {
  all dt: PIN | some dtp : BankCard | dtp.pin = dt
  all dt: BankCard | some evt : insertCard | evt.bc = dt
}
```

Fig. 6.4: ATM: semantics in Alloy of the metric TotalInstancesOfEvent(insertCard)

6.4.3 Semantics of the metrics based on the coverage of instances of the FOREST model elements

The metrics based on the coverage of instances of FOREST model elements involve the simulation of the FOREST model in order to evaluate whether the instance is covered within at least one simulation of the model. That is why this kind of metrics, unlike the previous kinds of metrics, is expressed within the initial and constrained FOREST models.

Semantics of CoveredLeafDataTypes and of CoveredLeafDataTypeOf metric

The semantics of `CoveredLeafDataTypes` is given as a set of Alloy predicates. This set comprises as many predicates as there are leaf data types in the FOREST model under evaluation. For each leaf data type a predicate is created in the Alloy model that specifies the coverage of the leaf data type depending on its use within the model under evaluation. In this predicate, it is checked whether there is any simulation of the FOREST model for which a state variable or a sent or received event contains the leaf data type value.

For instance in the ATM example, the `LargestCA` leaf data type is used as a parameter of the events `withdraw`, `serverPermission`, `deliverCash` and `requestPermission`. This leaf data type is not used in any state variable data type. The two first event types are events received by the SUT and the two last ones are events sent to the Test System. The coverage of the `LargestCA` leaf data type is true if one of the four events uses the data type in some (at least one) status of the FOREST model. This is specified in Alloy as shown in Figure 6.5.

The semantics of the metric `CoveredLeafDataTypeOf` is given as a set of Alloy predicates. The set of Alloy predicates to consider is the subset of the set of Alloy predicates, specifying the coverage of leaf data types, restricted to the ones for which the abstract data type is a super type.

For instance, the metric `CoveredLeafDataTypeOf(CashAmount)` is defined with a set of four Alloy predicates (`coveredLargestCA`, `coveredOtherLargeCA`, `coveredSmallestCA` and `covered-OtherSmallCA`). Each predicate inform about the coverage of one of the leaf data type of the `CashAmount` abstract data type. After the evaluation of each predicate, the overall coverage of the abstract data type, i.e. the conjunction of all predicate Boolean evaluation, may be computed.

Semantics of CoveredValueOfStructuredDataType metric

The semantics of a `CoveredValueOfStructuredDataType` metric is given as an Alloy predicate. The predicate specification depends on its two parameters and it is composed of two parts.

The first part of the predicate takes care of selecting the FOREST model statuses for which either the parameters of an event reception or a state variable is of the type (or its type is composed of the type) of the metric's first parameter.

Figure 6.6 shows the semantics of the `CoveredValueOfStructuredDataType` metric given the `PIN` structured data type and the OCL-F expression "`digits[0].oclIsKindOf(D0) and digits-[1].oclIsKindOf(D0)`" as its two parameters. Lines 03 to 07 illustrate the first part of the predicate; they select the possible event receptions or state variables that may be of the `PIN` data type. In our example, the two event receptions, `insertCard` and `authenticate`, comprise an attribute of type `PIN`. Also, the state variable `card` from the SUT component comprises an attribute of type `PIN`.

The second part of the predicate restricts the FOREST model statuses to the ones for which the event receptions and state variables satisfy the OCL-F expression given as the metric's second parameter.

Lines 08 and 09 of Figure 6.6 illustrate the second part of the predicate. The OCL-F expression (of DataType kind) is translated in a valid Alloy expression, in the same way than described in the previous section 4.5.6.

Semantics of CoveredValueOfStateVar metric

A `CoveredValueOfStateVar()` metric is implemented as an Alloy predicate. The content of the Alloy predicate is the OCL-F constraint prefixed with " `some status : Status |` ". In addition the state variables of the OCL-F constraint are prefixed with "`status.`".

For instance a metric of that type that evaluates if the state variable `attempts` may be exercised with the value 3, written `CoveredValueOfStateVar(sut.attempts = 3)` has the following semantics in Alloy: `pred { some status : Status | status.sut.attempts = 3}`

Semantics of CoveredStates metric

The `CoveredStates` metrics is a set of Alloy predicates. This set comprises as many predicates as there are protocol states in the FOREST model under evaluation. For each protocol state a predicate is created in the Alloy model that specifies the coverage of the protocol state. Each predicate checks whether there is some status of some simulation of the FOREST model for which the protocol state under evaluation can be an active protocol state of the status.

For instance, let's evaluate the `CoveredStates` metrics on the constrained model of the ATM example resulting from the test selection given in Figure 5.3. This model is composed of five protocol states, thus five predicates are created, one for each protocol states. The predicate evaluating the coverage of the protocol state `waitForCard` is given in Figure 6.7.

Semantics of CoveredEvents metric

The `CoveredEvents` metrics is a set of Alloy predicates. This set comprises as many predicates as there are event types in the FOREST model under evaluation. For each event type a predicate is created in the Alloy model that specifies the partial coverage of the event type, i.e. the predicate evaluates if there is at least one covered instance of the event type. Each predicate checks whether there is some status of some simulation of the FOREST model for which the event received (or sent) is of type of the event type under evaluation.

Let's take, for instance, the evaluation of the `CoveredEvents` metrics on the constrained model of the ATM example resulting from the test selection given in Figure 5.3. This model is composed of five protocol states, thus five predicates are created, one for each protocol states. The predicate evaluating the coverage of the protocol state `insertCard` is given in Figure 6.8.

```
pred coveredLargestCA {
  some status : Status |
    (status.nextStep.eventReceived in withdraw
      and status.nextStep.eventReceived.cw = LargestCA)
    or (status.nextStep.eventReceived in serverPermission
      and status.nextStep.eventReceived.csp = LargestCA)
    or (status.nextStep.eventSent in deliverCash
      and status.nextStep.eventSent.cdc = LargestCA)
    or (status.nextStep.eventSent in requestPermission
      and status.nextStep.eventSent.crp = LargestCA)
}
```

Fig. 6.5: ATM example: an Alloy predicate evaluating the coverage of the LargestCA leaf data type

```
pred coveredStructuredDT{
  some status : Status | some dt: PIN |
   ((status.nextStep.eventReceived in insertCard
    and status.nextStep.eventReceived.bc.pin = dt)
   or (status.nextStep.eventReceived in authenticate
        and status.nextStep.eventReceived.p = dt)
   or (status.sut.card.pin = dt))
   and dt.digits[0] in D0
   and dt.digits[1] in D0
}
```

Fig. 6.6: ATM example: an Alloy predicate evaluating the coverage of some particular instances of the PIN structured data type

```
pred coveredStateWaitForCard {
  some status : Status | status.curPState = ps_waitForCard
}
```

Fig. 6.7: ATM example: Alloy predicate evaluating the coverage of the waitForCard protocol state

```
pred coveredEventInsertCard {
  some status : Status | status.nextStep.eventReceived in insertCard
}
```

Fig. 6.8: ATM example: Alloy predicate evaluating the coverage of the insertCard event type

Semantics of CoveredValueOfEvent metric

The semantics of a `CoveredValueOfEvent` metric is given as an Alloy predicate. The predicate specification depends on its two parameters and it is composed of two parts.

The first part of the predicate takes care of selecting the FOREST model statuses for which an event is received (or sent) that is of the type of the metric's first parameter.

Figure 6.9 shows the semantics of the `CoveredValueOfEvent` metric given the `withdraw` event type and the OCL-F expression "`cw.oclIsKindOf(LargestCA)`" as its two parameters. Line 02 illustrates the first part of the predicate, it restricts the set of statuses to be the ones for which an event of type `withdraw` is received.

The second part of the predicate restricts the FOREST model statuses to the ones for which the event receptions satisfy the OCL-F expression given as the metric's second parameter.

Line 4 of Figure 6.9 illustrates the second part of the predicate. The OCL-F expression (of EventType kind) is translated in a valid Alloy expression, in the same way than described in the previous section 4.5.6.

```
01 pred coveredEventWithdrawInstance {
02   some status : Status | some evt: withdraw |
03     status.nextStep.eventReceived = evt
04     and evt.cw in LargestCA
05 }
```

Fig. 6.9: ATM example: an Alloy predicate evaluating the coverage of some particular instances of the withdraw event type

Semantics of CoveredTransitions metric

The `CoveredTransitions` metrics is a set of Alloy predicates. This set comprises as many predicates as there are protocol transitions in the FOREST model under evaluation. For each protocol transition a predicate is created in the Alloy model that specifies the partial coverage of the protocol transition, i.e. the predicate evaluates if there is at least one covered instance of the protocol transition. Each predicate checks whether there is some status of some simulation of the FOREST model for which the protocol transition is selected.

Let's take, for instance, the evaluation of the `CoveredTransitions` metrics on the constrained model of the ATM example resulting from the test selection given in Figure 5.3. This model is composed of nine protocol transitions, thus nine predicates are created, one for each protocol transitions. The predicate evaluating the coverage of the protocol transition `tr1` (from state `waitForCard` to state `waitForPIN`) is given in Figure 6.10.

```
pred coveredTransitionTR1 {
 some status: Status | status.nextStep.selectedPTransition in pt_tr1
}
```

Fig. 6.10: ATM example: Alloy predicate evaluating the coverage of the tr1 protocol transition

Semantics of CoveredValueOfTrans metric

The semantics of a `CoveredValueOfTrans` metric is given as an Alloy predicate. The predicate specification depends on its two parameters and it is composed of two parts.

The first part of the predicate takes care of selecting the FOREST model statuses for which a protocol transition is selected that is of the type of the one given as parameter of the metric.

```
01 pred coveredTransTR1Instance {
02   some status: Status | some pt: pt_tr1 |
03     status.nextStep.selectedPTransition = pt
04     and pt.trigger.bc.pin.digits[0] = D0
05 }
```

Fig. 6.11: ATM example: an Alloy predicate evaluating the coverage of some particular instances of the tr1 protocol transition

Figure 6.11 shows the semantics of the `CoveredValueOfTrans` metric given the `tr1` protocol transition and the OCL-F expression "`bc.pin.digits[0].oclIsKindOf(D0)`" as its two parameters. Line 03 illustrates the first part of the predicate, it restricts the set of statuses to be the ones for which a protocol transition of type `tr1` is selected.

The second part of the predicate restricts the FOREST model statuses to the ones for which the event receptions and state variables satisfy the OCL-F expression given as the metric's second parameter.

Line 04 of Figure 6.11 illustrates the second part of the predicate. The OCL-F expression (of PreCondition kind) is translated in a valid Alloy expression, in the same way than described in the previous section 4.5.6.

Semantics of CoveredStatePath metric

The semantic of `CoveredStatePath` metrics is given as Alloy predicates. For each protocol state given as a parameter of the metric, a status is looked for that has the protocol state as active state. As the metrics parameters are ordered, the protocol state of the first parameter must be an active state before the protocol state of the second parameter is an active state, etc. Note that the protocol states may not be consecutively active, i.e. there may be some other protocol states active between the states given as parameters.

Figure 6.12 illustrates the metric `CoveredStatePath(Authenticated, WaitForPermission, Authenticated)`. This metric evaluates whether there is (or not) at least one simulation for which the protocol state `Authenticated` is active then the protocol state `WaitForPermission` is active then the protocol state `Authenticated` is active again.

```
pred testatbleStatePath1 {
  some disj status1, status2, status3 : Status |
    status1.curPState = ps_Authenticated
    and status2.curPState = ps_waitForPermission
    and status2 in statusSeq/nexts[status1]
    and status3.curPState = ps_Authenticated
    and status3 in statusSeq/nexts[status2]
}
```

Fig. 6.12: ATM example: an Alloy predicate evaluating the coverage of a particular sequence of protocol states

Semantics of CoveredTransitionPath metric

The semantic of `CoveredTransitionPath` metrics is given as Alloy predicates. For each protocol transition given as a parameter of the metric, a status is looked for that take the protocol transition. As the metrics parameters are ordered, the protocol transition of the first parameter must be taken before the protocol transition of the second parameter, etc. Note that the protocol transitions may not be consecutively taken, i.e. there may be some other protocol transitions taken between the transitions given as parameters.

```
pred testatbleTransPath1{
  some disj status1, status2, status3 : Status |
    status1.nextStep.selectedPTransition = pt_tr3
    and status2.nextStep.selectedPTransition = pt_tr4
    and status2 in statusSeq/nexts[status1]
    and status3.nextStep.selectedPTransition = pt_tr5
    and status3 in statusSeq/nexts[status2]
}
```

Fig. 6.13: ATM example: an Alloy predicate evaluating the coverage of a particular sequence of protocol transitions

Figure 6.13 illustrates the metric `CoveredTransitionPath(tr3, tr4, tr5)`. This metric evaluates whether there is (or not) at least one simulation for which the protocol transition `tr3` is taken then the protocol transition `tr4` is taken and then the protocol transition `tr5` is taken.

6.5 Specifying reusable business-level metrics

In the previous sections of this chapter, we have presented structural and semantical metrics. These different metrics are defined such that they may be used for various kinds of purposes. Structural and semantical metrics are closely related to the concepts of FOREST models. In order to ease their use, we define another kind of metrics, which we name business-level metrics.

Business-level metrics are defined informally, at a more abstract level than the structural and semantical metrics, using the business terminology of the company where they are used. Business-level metrics are based on structural and semantical metrics defined previously. As business-level metrics are defined at an abstract level, they may be used in several different contexts. We define three reusability levels for business-level metrics:

- they may be used for the evaluation of any kinds of system under test. In that case, we say that its reusability level is high, and we name these metrics general-purpose metrics.
- the metrics are evaluating features of specific kinds of systems. We say that its reusability is medium, and we name these metrics domain-specific metrics.
- the metrics are evaluating features which are specific to system under test. The reusability is low and the metrics are said to be project-specific metrics.

In this section, we present some sample general-purpose and domain-specific business-level metrics. The presented metrics may be further adapted to the special needs of a given company. Some project-specific metrics are defined within the context of the industrial case study of this thesis (see further Section 7.5).

6.5.1 General-purpose metrics

General-purpose metrics are intended to the evaluation of the test selection of any kind of systems. Below are some questions that a Test Manager may ask himself. Each of these questions may be expressed as a general-purpose business-level metric.

(1) Are there priority behaviors that we are not testing with these reductions?

(2) Is each requirements still covered with the constrained model?

(3) Have all categories of values been included in the constrained model?

(4) Are all error messages covered in the constrained model?

(5) Has the test selection reduced the amount of test cases?

Priority behavior coverage

Priority behaviors are the minimal set of behaviors of the system that must be tested. *Priority behavior coverage* metrics are defined with the help of the semantical metrics `CoveredStatePath` and `CoveredTransitionPath`. In order to define *Priority behavior coverage* metrics for a given system under test, the priority behaviors must be identified and expressed as a constraint on the ordering of the transition to be taken or on the ordering of the states to be active.

For instance, in the ATM example, one of the priority behaviors to be tested is to be able to withdraw a large amount of cash. This behavior is represented by taking the two transitions `tr6` (from state `Authenticated` to state `waitForPerm`) and `tr7` (from state `waitForPerm` to state `Authenticated`). The evaluation of whether this behavior is covered or not may be expressed with the parameterized metric `CoveredTransitionPath(tr6, tr7)`.

Requirements coverage

Different kinds of metrics may be used to express the coverage of a requirement depending on its type. If it is a simple behavior requirement then the metrics `CoveredState()`, `CoveredEvent()`, `CoveredValueOfEvent()`, `CoveredTransition()` or `CoveredValueOfTransition()` may be used to express the requirements coverage. If it is a behavior requirement involving the coverage of a path then the metrics `CoveredStatePath` and `CoveredTransitionPath` may be used. If it is a data requirement then the metrics `CoveredLeafDataTypeOf()` or `CoveredValueOfStructured-Datatype()` may be used.

For instance, in the ATM example, one of the behavioral requirements is that when there are three consecutive false authentications the system returns to its initial state. The return of the system to its initial state after three consecutive false authentications is specified in the dynamic view of the ATM FOREST model by the transition `tr2` (from state `waitForAuthentication` to the initial state `waitForCard`). The metric `CoveredTransition(tr2)` is particularly fitted to evaluate if the transition `tr2` is covered or not.

Categories of values coverage

The categories of values are expressed by leaf data types in the static view of FOREST models. In order to check the coverage of all the categories of values, the metric `CoveredLeafDataTypes()` may be used to inform about how many leaf data types are covered. Then, the coverage of the categories of values is simply performed by checking if the value of this metric equals the total number of leaf data types specified in the static view.

Error messages coverage

In order to check for the coverage of the error messages in a constrained model, the error messages must be identified out of the event receptions of the SUT and Test System components. And then for each of the event receptions representing an error message, the metric `CoveredEvent` is used to check for the coverage of the error message.

Reduction of the amount of test cases

The metrics `TotalTestCases` computes the number of possible instances of a FOREST model, in order to answer the question if the test selection has reduced the number of test cases, the values of the `TotalTestCases` is simply compared for the initial FOREST model and the constrained FOREST model resulting from the test selection.

6.5.2 Domain-specific metrics

Domain-specific metrics are intended to the evaluation of the test selection of systems of a particular domain. The SESAME approach targets small-sized safety-related embedded systems. These systems may be part of different application domains (automotive, avionic, railway, medical, etc.).

Let's take for instance, the automotive domain. The automotive domain comes with a set of standards that must be followed (e.g. MISRA guidelines [MIS00] and AUTOSAR [AUT08]). Some of these standards have requirements on the testing of some features of systems. In our approach, we suggest to evaluate the coverage of the test requirements of these standards.

AUTOSAR: sleep mode accessibility

One of the many AUTOSAR requirements on network management [AUT08] of automotive embedded systems is that the system must always be able to switch to a sleep mode. Depending on how the sleep mode has been specified in the FOREST model, the coverage of this requirement may be expressed differently. In the case that the sleep mode is specified as a protocol state, the coverage of this AUTOSAR requirements is simply that the sleep mode protocol state must be accessible from any other protocol state.

This could be expressed with a set of `CoveredStatePath()` metrics. `CoveredStatePath( state1, sleep_mode_state)`, `CoveredStatePath( state2, sleep_mode_state)`, `CoveredStatePath( state3, sleep_mode_state)`, and so on for the other protocol states of the dynamic view.

7. INDUSTRIAL CASE STUDY

Abstract

In this chapter, we apply the SESAME process model to a case study based on a real industrial project developed at the I.E.E. company. We start by giving a short presentation of the company and the industrial project. Then, we apply the four main tasks of the SESAME process model to the industrial case study. The work products defined for the purpose of this case study, and created by the process activities, give concrete illustrations of a possible application of the process on an industrial project.

7.1 Industrial project introduction

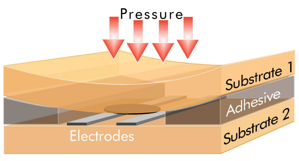
7.1.1 Overview of IEE company

IEE is an international company leading in the development of innovative sensor-based embedded systems. The company has more than 1200 employees worldwide, spread over sites in Europe, North America and Asia. Headquarters and main R&D activities of the company are located in Luxembourg.

The original market of the company was solely focused on the provision of products for the automotive industry. Most of its customers are car manufacturers or Tier 1/Tier 2 suppliers. Recently, the company has broaden its targeted market and sold sensors for products related to the health industry, multimedia consumer applications (e.g. mobile phones, mp3 players, audio equipments), etc. The four key technologies developed at IEE are illustrated in Figure 7.1:

- *Force Sensing Resistance.* This technology uses variable resistance to measure pressure applied to a matrix of sensor cells. Products developed from this technology are: Occupant Classification related products, seat-belt reminder, pedestrian protection, and multimedia consumer products.
- *Electromagnetic Sensing.* This technology use resonators and antennas to identify child seat characteristics. Products developed from this technology are Child Seat Presence and Orientation Detection (rear-facing or forward-facing child seats).
- *Electric Field Sensing.* This technology evaluates change in electrical field to detect seat occupancy.
- *3D-MLI Sensor*[1]. The most recent technology measures time-of-flight of modulated infra-red light to provide topographic vision. This technology particularly requires the development of embedded software and is of particular interest in the context of our study.

[1] MLI stands for Modulated Light Intensity.



(a) Force Sensing Resistance

(b) Electromagnetic Sensing

(c) Electric Field Sensing

(d) 3D-MLI Sensor

Fig. 7.1: IEE's four key technologies. *(Figures and Photos: IEE)*

7.1.2 Description of the VOCS industrial project

As illustrated in Figure 7.2, seat occupancy may be diverse, there may be some bags on the seat, a rear-facing child seat, a woman, a man, a child on a forward facing child seat, etc. The Visual Occupant Classification System (VOCS) developed at IEE classifies the front passenger seat occupancy into the four categories: empty, child seat, 5thpercentile female[2], adult[3].

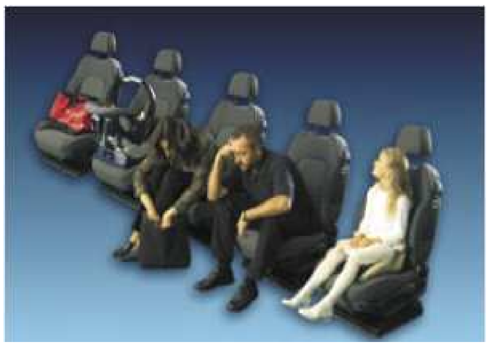


Fig. 7.2: Occupant classification. *(Photo: IEE)*

The purpose of the VOCS is to classify the seat occupancy so that other Electronic Control Units (ECUs) within the car are aware of the nature of the occupant sitting in the front passenger seat. In particular, this product was developed for the Airbag Control Unit to take into account the occupant category for compliance with the safety standard [NHT08]. The VOCS product came as a solution to the three following issues. Firstly (and most importantly), the deployment of an airbag while a baby is in a rear-facing child seat is likely to result in the death of the baby. In this case, the airbag of the front passenger seat must be disabled. Secondly, the new generation of airbags introduced different strengths of deployment. In this context, the occupant classification gives valuable information so that the airbag may be deployed at full strength in the case of an adult and reduced strength may be requested in the case of a 5th%ile female. Lastly, in case of a "light" car crash[4] with no one seated in the front passenger seat, the deployment of the airbag is unnecessary and should be avoided[5].

[2] A fifth percentile female cover the five percent of the female population which are the smallest and lightest. It corresponds to a small amount of the population that is particularly sensitive to car crash. The characteristics of this portion of the population are: weight of 45 to 60 kg and height of 135 to 155 cm (any person with a weight or an height higher than these range is not considered to be part of 5thpercentile female).

[3] This category covers adults who are not part of the 5thpercentile female category.

[4] If the car may still be repaired.

[5] The action of putting back an airbag in its compartment involves consequent additional cost

7.2 Input work product

7.2.1 Customer requirements

The Customer Requirements is a work product that is outside of the SESAME process model but required as input to the Analyze Customer Requirements task, illustrated in the next section, 7.3. The format of customer requirements is not restricted by the SESAME process. In the context of the VOCS industrial project, some use-cases were defined by IEE on the basis of discussions with the customer and the customer specifications received. These can be found in Appendix F.

Use-cases received as input work product from the company are defined at the system design level, i.e. a concrete definition of the vehicle interface. In our analysis context, we have raised the level of abstraction to the analysis level so that the use-cases and the FOREST models are at the same abstraction level. The four analysis use-cases are described in the following.

Primary Use Case - Classification of occupant

Trigger: Power is applied

Primary actor: Airbag Control Unit

Goal: Determine whether airbag should be enabled or not based on occupant classification

1.1 System is initialized by the reception of the `ignitionOn` event

1.2 Active mode[6] is effective by receiving signal `wakeUp`

1.3 Image is requested with the `imageReq` event to the DARSS component[7].

1.4 Image is conveyed with the reception of the event `image` from the DARSS component.

1.5 Occupant is classified[8].

1.6 When the event `timeoutSendClass` is received, the classification result is sent via the event `classification`.

Extension Use Case - Change in classification of occupant

Trigger: Occupant is classified and is different from last classification result.

Primary actor: Airbag Control Unit

Goal: Determine whether airbag should be enabled or not based on occupant classification

2.1 The system immediately[9] sends classification result via the event `classification`.

[6] Modeled by the protocol state Active in the dynamic view of Figure 7.10.

[7] This component is described in the further Section 7.3

[8] This step describes an internal activity of the SUT that is why it does not appear in the dynamic view of the initial model in Figure 7.10. This is because our models focus on the description of the exchanges of events between the SUT and its environment.

[9] Unlike in 1.6, the system does not wait for the timeOutSendClass event reception.

Extension Use Case - Storage of data in event of crash

Precondition: System is in active mode

Trigger: Crash message sent from chassis module

Primary actor: Airbag Control Unit

Goal: Store record of data that was used in the decision to fire or not fire the airbag

3.1 Reception of the `crash` event from the Airbag Control Unit.

3.2 Last classification result and last received image are stored in NVM through the sending of event `storeClassification`.

Extension Use Case - Retrieval of data after a crash

Trigger: Diagnostic tool is connected by a technician

Primary actor: Service-bay technician

Goal: Retrieval of data that was used in the decision to fire or not fire the airbag

4.1 The system receives an `authenticate` event from the service-bay technician to access diagnostic mode that match the authentication data of the system.

4.2 The system receives a `diagnosticReq` event from the service-bay technician to retrieve crash data.

4.3 The diagnostic data is sent to the technician through the event `diagnostic`.

7.3 Analyze customer requirements

7.3.1 Defining the analysis model

The definition of the analysis model is done based on the customer requirements, in our context, given in terms of use-cases. In the following, we apply the five steps of the SESAME approach for the definition of an initial analysis model of the SUT and its Test System, for the purpose of selecting tests from this model.

Identification of the events exchanged between VOCS and its test environment

The environment of the SUT is composed of different physical elements, these elements are outside of the boundary of the SUT, and communicate with the SUT through the sending of events:

- *Driver.* The driver of the car impacts the behavior of the VOCS by use of the key.
- *Airbag Control Unit* (ACU). This component is in charge of the deployment of the airbag. The seat occupancy classification is sent to this component.
- *DARSS*[10] is an Application-Specific Integrated Circuit (ASIC) that is responsible for retrieving the image from the 3D-MLI sensor and performing some basic data processing to produce a raw image for the SUT, i.e. the VOCS embedded software.
- *Service-bay technician* (SB-TECH). This represents a person, typically working in a garage, who requests diagnostics data from the SUT.
- *Timer.* The timer sends events to the VOCS at particular time intervals.
- *Non-volatile memory* (NVM). This component is responsible for the storage of some important data, mainly it stores classification results.

The events that are received by the VOCS from the elements aforementioned and the events that are sent from the VOCS to its environment are summarized in Table 7.1. In the SESAME approach, the set of physical elements out of the boundary of the SUT are grouped in a single concept, the Test System. The Test System is represented as the `TestSystem` UML component.[11]

Based on Table 7.1, we create a UML class diagram, illustrated in Figure 7.3 that describes a first sketch of the static view of the initial model of the SUT. Two UML Components `SUT` and `TestSystem` are created, representing respectively the system under test and its related Test System. An Operation is added to the `SUT` component when the event is received by the VOCS from one of the physical element of its environment; similarly, an UML Operation is added to the `TestSystem` component when an event is sent by the VOCS to any of the physical element part of its environment. In the end, ten operations are created in the `SUT` component and four operations in the `TestSystem` component.

[10] The Darss is originally a part of a peninsula at the South of the Baltic sea. Pick as a name because it starts D A (Digital ASIC) and this geographic region was used by IEE to name other devices.

[11] In the remaining of this chapter the Test System concept and its representation, the TestSystem component are used interchangeably.

Event direction	Event Name	Event Description
SUT $\Leftarrow$ DRIVER	ignitionOn / ignitionOff	event sent when the driver turns its key in the car.
SUT $\Leftarrow$ ACU	wakeUp / sleep	event received when the ACU requires classification information to be computed. These two events are an abstraction of a presence or absence of a periodic signal.
SUT $\Leftarrow$ DARSS	image	Event received from the DARSS external component.
SUT $\Leftarrow$ SB-TECH	authenticate	Event received when the service-bay technician (SB-TECH) wishes to authenticate.
SUT $\Leftarrow$ SB-TECH	diagnosticReq	Event received when the SB technician requests diagnostic data from the SUT.
SUT $\Leftarrow$ ACU	crash	Event received when a crash occurs.
SUT $\Leftarrow$ TIMER	timeoutImageReq	Event received when it is time to send a request for an updated image to the DARSS component.
SUT $\Leftarrow$ TIMER	timeoutSendClass	Event received when it is time to send the classification to the ACU component.
SUT $\Rightarrow$ NVM	storeClassification	Event sent to the non-volatile memory when a crash occurs.
SUT $\Rightarrow$ DARSS	imageReq	Event sent periodically to the DARSS component to request an updated image from the camera.
SUT $\Rightarrow$ ACU	classification	Event sent to the airbag control unit (ACU) when an image has been classified.
SUT $\Rightarrow$ SB-TECH	diagnostic	Event sent when SB technician request diagnostic data and is authenticated properly.

Tab. 7.1: VOCS case study - list of events exchanged between the SUT and the Test System components

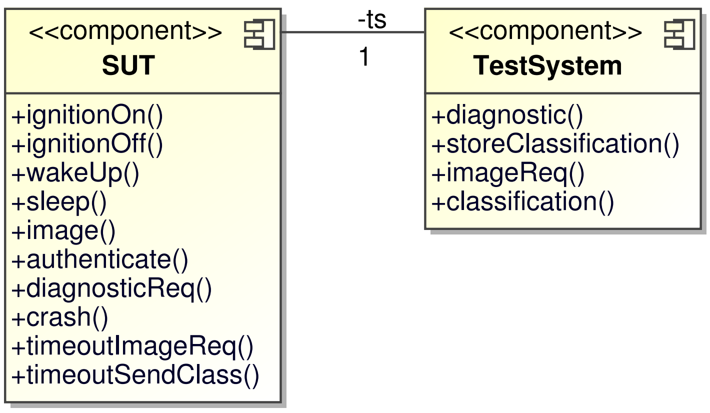


Fig. 7.3: VOCS industrial case study: first sketch of the FOREST analysis model static view - SUT and Test System components

Identification of the allowed order of events reception

The order of event is extracted from the use-cases given in the previous section, 7.2. At the initial state the system is in an `EnergySaving` mode, the driver must turns his key in the ignition switch before any event receptions have an effect on the SUT. When the `ignitionOn` event is received the system is in a `Sleep` mode that we represent in the dynamic view by a protocol state. In this mode, no events are interpreted before the `wakeUp` signal is received from the ACU. When this signal is received the timeout signals and `image` reception event, as well as `crash` signals may be received. Finally, the diagnostic request event may only be received after a successful authentication.

Based on this (informal) analysis of the use-cases, a UML protocol state machine is created that reflects this order of event reception. This state machine is illustrated in Figure 7.4.

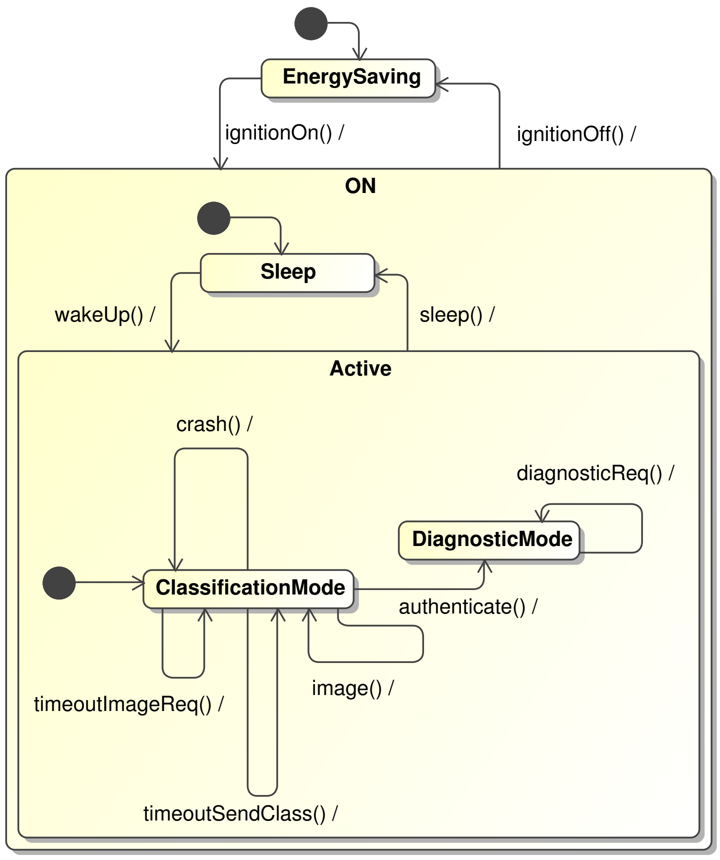


Fig. 7.4: VOCS industrial case study: FOREST analysis model dynamic view

Identification of the data conveyed by the exchanged events

The data conveyed by the events exchanged between the SUT and the Test System are identified and briefly described in Table 7.2. The static view is updated with this data and parameters are added to the operations related to the events conveying data. For instance, the `image` operation now has an `image` parameter of type `ImageCT`. Figure 7.5 shows the `SUT` and `TestSystem` components of the static view updated with data conveyed within events.

The next step is to define precise data types for the purpose of testing. In this step, three tasks may be performed related to the specification of data types, as defined in Section 3.4.2:

(1) Identification of possible test attributes for each data type.

(2) Partitioning of the data types into equivalence classes.

(3) Specification of constraints on data types to ensure only valid combinations of values.

Five data types are defined for the event parameters in this case study: `ClassifiedImage`, `Class`, `Image`, `Authentication` and `Diagnostic`. A possible definition of these data types, for the purpose of testing them, is defined in the UML class diagram of Figure 7.7. It can be noticed that not all the data types are defined at the same level of precision. The more precisely the data types are defined, then the more they will be exercised in detail. In the context of this project, we particularly focus on the testing of the `Image` data type, neglecting the testing of the `Diagnostic` and the `Authentication` data types. That is why the `Diagnostic` data type is just described as an abstract data type, with no attributes defined.

We partition the `Authentication` data type in two partitions, `AuthValid` and `AuthInvalid`. This is specified with the introduction of two new data types (one for each partition) that extends the `Authentication` data type, as illustrated in Figure 7.7.

Five test attributes have been identified for the `Image` data type: the background of the image, the seat in the image, the object of interest (abbreviated OOI) in the image, a number of parasitic objects (abbreviated PO) that may influence the classification of the OOI, and lastly the focus of the image. Each of these test attributes result in the creation of an attribute in the `Image` data type with the related multiplicity. Each of these test attributes are themselves characterized by a data type.

Focusing on the definition of the OOI data type, shown in Figure 7.8[12]. Three test attributes are identified for the OOI. These attributes characterize the object of interest in terms of attributes on its legs, torso, and head. In the class diagram of Figure 7.8 the test attributes are modeled as directed associations, which has the same meaning as adding an attribute to the OOI data type. Each of these attributes are further refined. For instance, a leg has three test attributes (the size of the leg, its position, and angle). Figure 7.6 shows some values of the OOI data type depending on the values of its leg and torso attributes.

Lastly, constraints are specified that restrict valid combination of values. These constraints are used in the case of data types with test attributes. For instance, the `ClassifiedImage` data type has two test attributes, `Class` and `Image`. One of the constraints applied to this data type is that if the classification is empty then no object of interest will be exercised in the image.

[12] A number of other class diagrams have been defined that define the other test attributes of the Image data type. For sake of completeness, these class diagrams can be found in the Appendix G of this thesis. These class diagrams are constructed in the same manner as the class diagram describing the OOI data type.

Event Name	Event Parameters Data Description
ignitionOn	This event is a signal, no data is conveyed.
ignitionOff	This event is a signal, no data is conveyed.
wakeUp	This event is a signal, no data is conveyed.
sleep	This event is a signal, no data is conveyed.
image	An image of the front seat occupant is conveyed by this event.
authenticate	The authenticate request from a service-bay technician convey authentication data.
diagnosticReq	The event is a signal, no data is conveyed.
crash	The event is a signal, no data is conveyed.
timeoutImageReq	The event is a signal, no data is conveyed.
timeoutSendClass	The event is a signal, no data is conveyed.
storeClassification	This event conveys the last classification result and the last image received which was used to compute the classification.
imageReq	The event is a signal, no data is conveyed.
classification	This event conveys the current classification result.
diagnostic	Diagnostic data is conveyed with this event, but the nature of diagnostic data is not precisely defined, for the purpose of this case study.

Tab. 7.2: VOCS case study - data conveyed by the incoming/outgoing SUT events

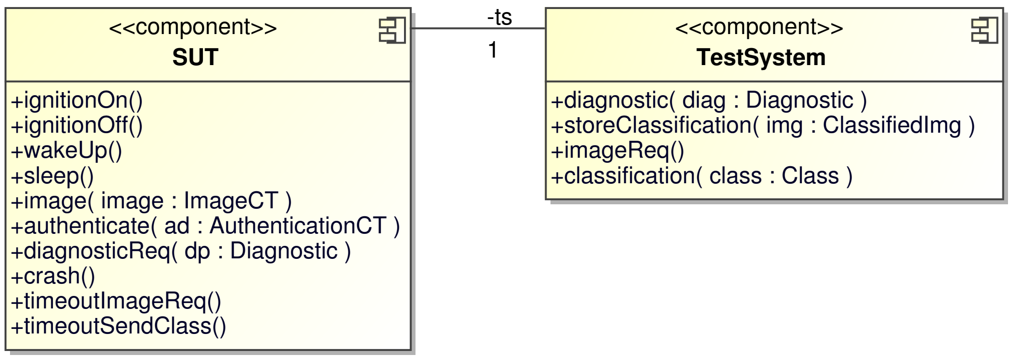
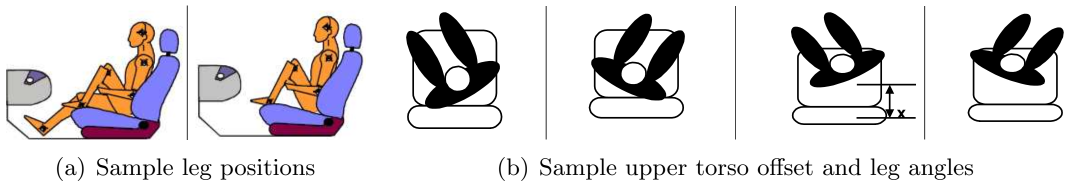

<<component>>
SUT
+ignitionOn()
+ignitionOff()
+wakeUp()
+sleep()
+image(image : ImageCT)
+authenticate(ad : AuthenticationCT)
+diagnosticReq(dp : Diagnostic)
+crash()
+timeoutImageReq()
+timeoutSendClass()
-ts
1
<<component>>
TestSystem
+diagnostic(diag : Diagnostic)
+storeClassification(img : ClassifiedImg)
+imageReq()
+classification(class : Class)

Fig. 7.5: VOCS industrial case study: FOREST analysis model static view - SUT and Test System components with data



(a) Sample leg positions (b) Sample upper torso offset and leg angles

Fig. 7.6: Sample possible front passenger postures to be exercised.

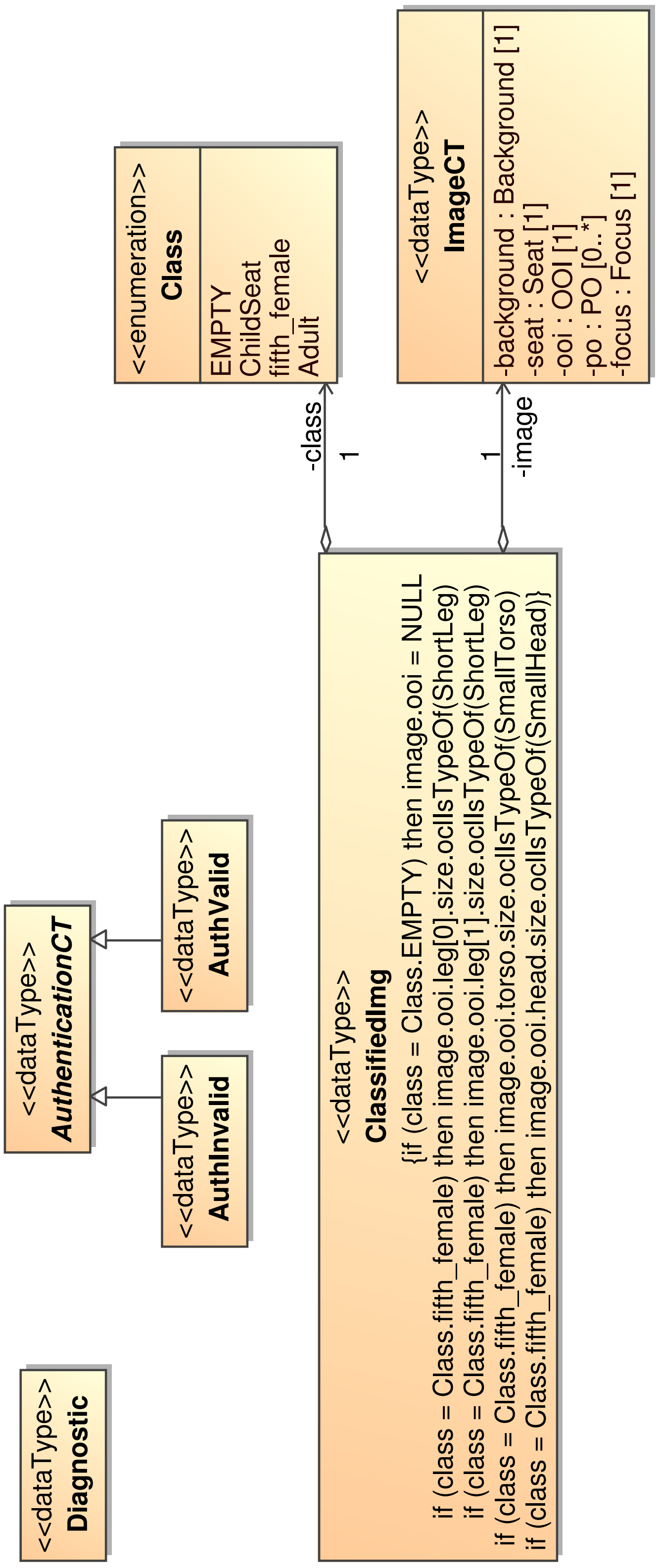

<<dataType>>
Diagnostic
<<dataType>>
AuthenticationCT
<<dataType>>
AuthInvalid
<<dataType>>
AuthValid
<<enumeration>>
Class
EMPTY
ChildSeat
fifth_female
Adult
-class
1
<<dataType>>
ClassifiedImg
{if (class = Class.EMPTY) then image.ooi = NULL
if (class = Class.fifth_female) then image.ooi.leg[0].size.oclIsTypeOf(ShortLeg)
if (class = Class.fifth_female) then image.ooi.leg[1].size.oclIsTypeOf(ShortLeg)
if (class = Class.fifth_female) then image.ooi.torso.size.oclIsTypeOf(SmallTorso)
if (class = Class.fifth_female) then image.ooi.head.size.oclIsTypeOf(SmallHead)}
1
-image
<<dataType>>
ImageCT
-background : Background [1]
-seat : Seat [1]
-ooi : OOI [1]
-po : PO [0..*]
-focus : Focus [1]

Fig. 7.7: VOCS industrial case study: definition of the data types of the event parameters

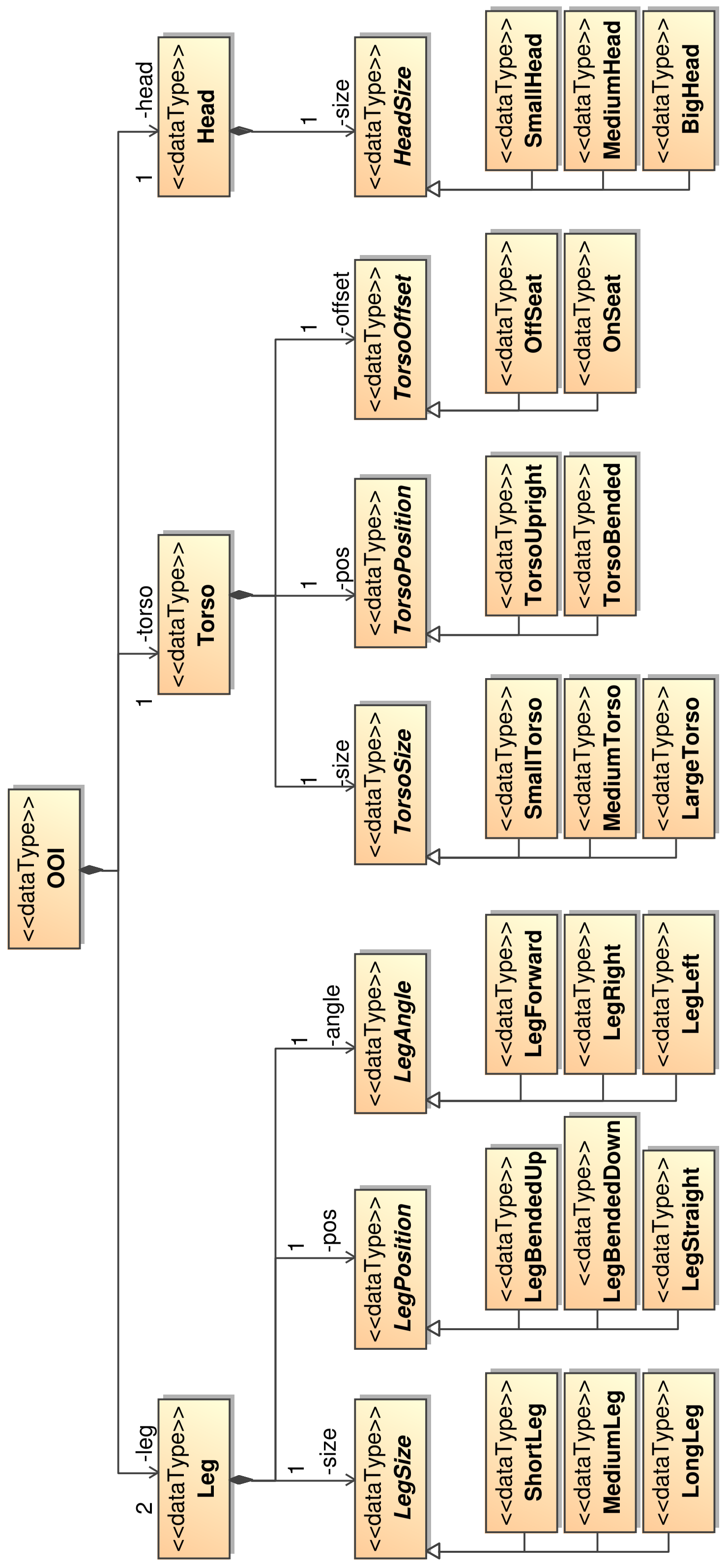


Fig. 7.8: VOCS industrial case study: OOI data type definition

Characterization of the observable state of the system under test

The observable state of the system under test is characterized by the last classified image together with its classification result, the second last classified image together with its classification result, and its authentication data. These three state variables represent information of the SUT that may be accessed from its environment for any purpose. They are modeled in the SUT component as UML attributes (`lastClass`, `secondLastClass`, and `auth`) of the component, as shown in Figure 7.9. This last piece of information completes the static view of the initial model of the SUT.

Characterization of the pre- and post-conditions associated with the event receptions

After the completion of the data specification, the behavioral specification of the SUT may be completed. During this step, pre- and post-conditions are specified on the protocol transitions of the dynamic view of the SUT.

Pre-conditions are interpreted as guard conditions and restrict taking a transition based on the evaluation of its pre-condition. In this case study, we have identified on pre-condition, which is related to the reception of the authentication event. The protocol transition that is triggered by the `authenticate` event reception may only be taken if the parameter `ad` (parameter of the `authenticate` event reception) of the event trigger equals to the authentication data (`auth`) of the SUT. This is shown in Figure 7.10 by prefixing the label of the protocol transition with `[ad=sut.auth]`.

Post-conditions specify how the SUT state variables are modified after the reception of an event and if an event must be sent to the Test System in reaction to the event reception. For instance, the protocol transition that represent the reception of the event of type `crash` reacts to this event reception by the sending of the event `storeClassification` to the Test System having as parameter value the last computed classification, i.e. the value of the state variable `lastClass`. This is specified by appending at the end of the label of targeted protocol transition the expression: `[ts^storeClassification(sut.lastClass)]`.

The addition of pre- and post-conditions specified on the protocol transitions of the protocol state machine completes the dynamic view of the initial model of the SUT.

7.3.2 Translating the UML analysis model into an Alloy specification

This is an automated task supported by the KerMeta environment. Test engineer must launch the execution of the “FOREST Semantics” model transformation implemented as a KerMeta model transformation program from within the Eclipse environment. The result of the model transformation is an Alloy file, formally describing the static and dynamic view of the industrial case study. An Alloy file describing the semantics of the FOREST initial model of the VOCS industrial case study can be found in Appendix C.

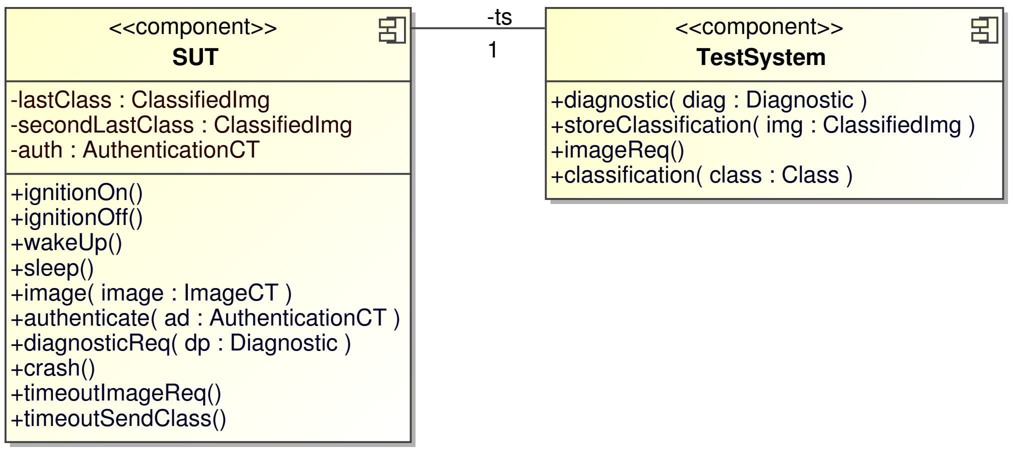

<<component>>
SUT
-lastClass : ClassifiedImg
-secondLastClass : ClassifiedImg
-auth : AuthenticationCT
+ignitionOn()
+ignitionOff()
+wakeUp()
+sleep()
+image(image : ImageCT)
+authenticate(ad : AuthenticationCT)
+diagnosticReq(dp : Diagnostic)
+crash()
+timeoutImageReq()
+timeoutSendClass()
-ts
1
<<component>>
TestSystem
+diagnostic(diag : Diagnostic)
+storeClassification(img : ClassifiedImg)
+imageReq()
+classification(class : Class)

Fig. 7.9: VOCS industrial case study: FOREST analysis model static view - SUT and Test System components with data and state variables

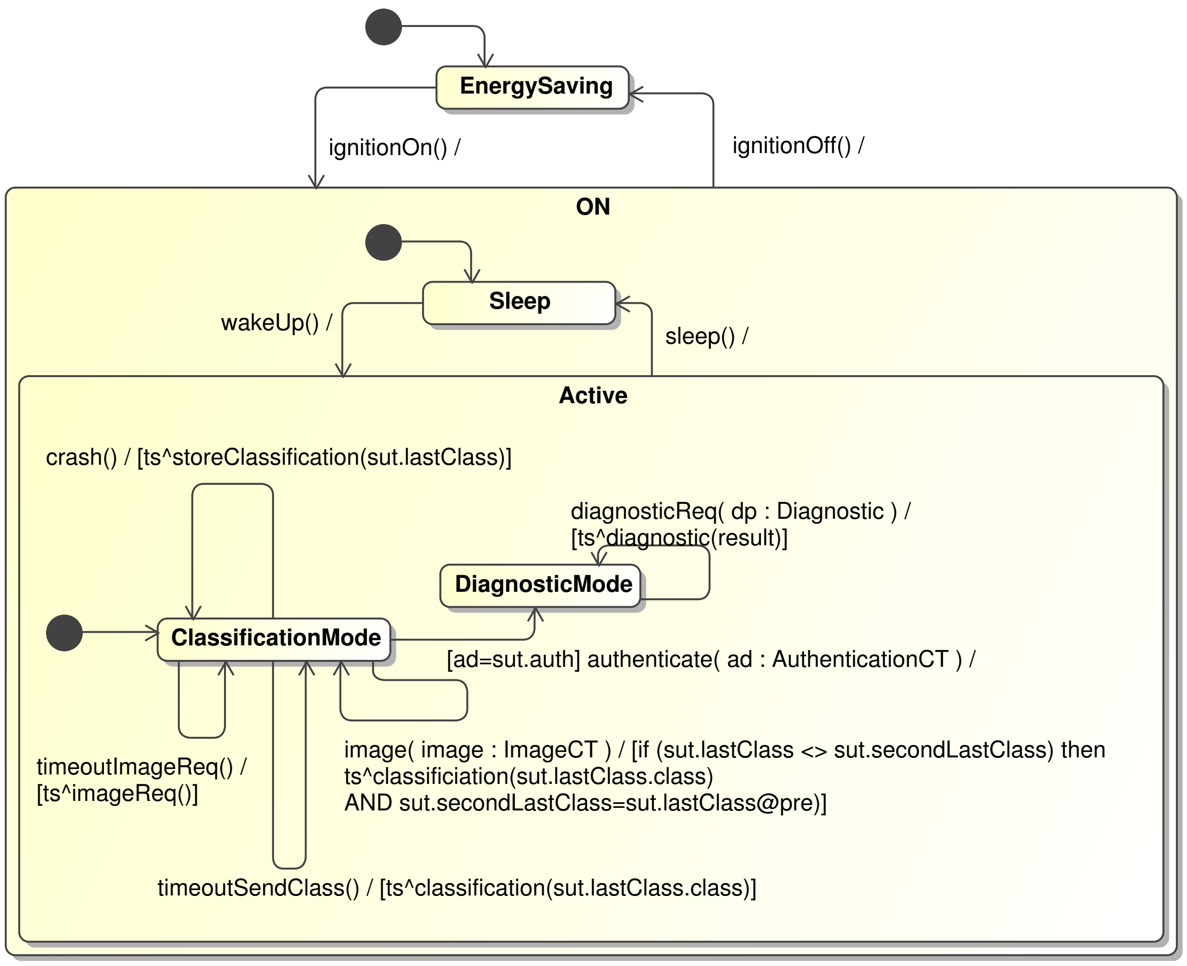

EnergySaving
ignitionOn() /
ignitionOff() /
ON
Sleep
wakeUp() /
sleep() /
Active
crash() / [ts^storeClassification(sut.lastClass)]
diagnosticReq(dp : Diagnostic) /
[ts^diagnostic(result)]
DiagnosticMode
ClassificationMode
[ad=sut.auth] authenticate(ad : AuthenticationCT) /
image(image : ImageCT) / [if (sut.lastClass <> sut.secondLastClass) then
ts^classificiation(sut.lastClass.class)
AND sut.secondLastClass=sut.lastClass@pre)]
timeoutImageReq() /
[ts^imageReq()]
timeoutSendClass() / [ts^classification(sut.lastClass.class)]

Fig. 7.10: VOCS industrial case study: FOREST analysis model dynamic view

7.4 Specify test selection

In this industrial case study, we focus on the test selection of data complexity, and more precisely on the selection of the `ImageCT` data type. As an introduction to the Specify Test Selection task, we calculate informally the number of all the possible instances of the `ImageCT` data type, in order to grasp the data complexity[13] of the VOCS system.

For this purpose, we present an intuitive way of computing the number of instances of data types. On the one hand, the complexity of an abstract data type is the addition of the number of its leaf data types. For instance `Background` data type is of complexity 3, as it is an abstract data type which is related to three leaf data types. On the other hand, the complexity of a structured data type is the multiplication of the complexity of its attributes. For instance, the complexity of the Leg structured data type is the multiplication of the complexities of `LegSize`, `LegPosition` and `LegAngle` each of them having a complexity of 3, so the complexity of Leg is $3 \times 3 \times 3 = 27$. Based on these two considerations, we compute the complexity of the `ImageCT` data type, as shown in Table 7.3.

Data type	**Number of possible instances to be exercised**
`Background`	3
`Seat`	$(3 \times 3) \times (3 \times 3) = 81$
`OOI`	$(3 \times 3 \times 3)^2 \times (3 \times 2 \times 2)^1 \times 3 = 26,244$
`PO`	$(3 \times 3 \times 2) + (2 \times 3) + 1 = 25$
`Focus`	2
`ImageCT`	$3 \times 81 \times 26,244 \times 25^{nb_{po}} \times 2 = 12,754,584 \times 25^{nb_{po}}$
`ImageCT` for one parasitic object	$51,018,336$
`ImageCT` for two parasitic object	$1,275,458,400$ (around a billion possible instances!).

Tab. 7.3: Complexity of some data types of the VOCS initial analysis model

Assuming that one test is performed per second, around fifteen thousand days are required to exercise all the possible value combinations, which is hardly tractable. We conclude that there is indeed a need for test selection of the `ImageCT` data type, in order to reduce the number of instances to be exercised.

[13] In the particular case of the VOCS project as defined in this chapter, this approximation is quite reliable, since the other data types AuthenticationCT and Diagnostic are not specified in details. This results in the other data type having a low number of instances and thus do not influence significantly the computation presented here.

7.4.1 Collecting test requirements with the 5+2 View Model

This section presents the first step of the specification of test selections which consists of the collection of test requirements based on the different views of the 5+2 View Model defined within the SESAME process in Section 3.3. The test requirements elaborated in the following are based on discussions with IEE and reflect real-life concerns of the company.

Customer View

The Customer View comprises test requirements that are expressed by the Customer. These test requirements must usually be successfully performed before the delivery of the final product.

In the industrial context of VOCS, the test requirements expressed by the Customer are given in the form of a set of figures that represent different positions of the test passenger to be exercised on the SUT. Twenty of these postures are illustrated by Figure 7.11[14]. These postures vary in the position of the legs, of the upper torso, etc. For each of the twenty postures, a combination of values from the static view of the VOCS initial model should be selected. For instance, the test requirements resulting from the posture 1 of Figure 7.11 is an object of interest (OOI) with both legs in position bended down and angle forward; and with its upper torso in up right position and with no offset (i.e. touching the seat backrest).

Quality View

The quality view is the view on test requirements from the quality department (or quality engineer) of a company. In the automotive industry, potential failures are analyzed and specified in FMEA[15] document. An FMEA document analyzes parts of the system design for potential failures. Parts of the system design are linked to their related requirements. Failures models[16], like FMEA documents, are good basis for test selection. Indeed, if a potential failure is identified then it is important to select the conditions that lead to the potential failure, so that these conditions are exercised.

A possible failure mode of VOCS is that the NVM component is suddenly out-of-order. If this happens, the system should be working in degraded mode. This degraded mode must allow a proper functioning of all the receptions and sending of events to all components of the SUT environment, excluded the NVM component. In order to exercise, this failure mode, the NVM component may be disconnected from the SUT, and the SUT should be exercised with events coming from other (i.e. non-NVM) components. More precisely, the event(s) related to the NVM component (i.e. the `storeClassification` event as shown in Table 7.1) should not be exercised.

Product-line View

This view comprises test requirements expressed by the product-line manager. At IEE, the product-line manager[17] is in charge of the products that provide the same functionalities. In the

[14] This figure is a direct extract from a German OEM specification. Postures are given names in German; these names do not have an impact on the understanding of this industrial case study and may safely be ignored by the reader.
[15] Failure Mode and Effect Analysis
[16] Also called sometimes “fault models”.
[17] called Program Manager in the company.

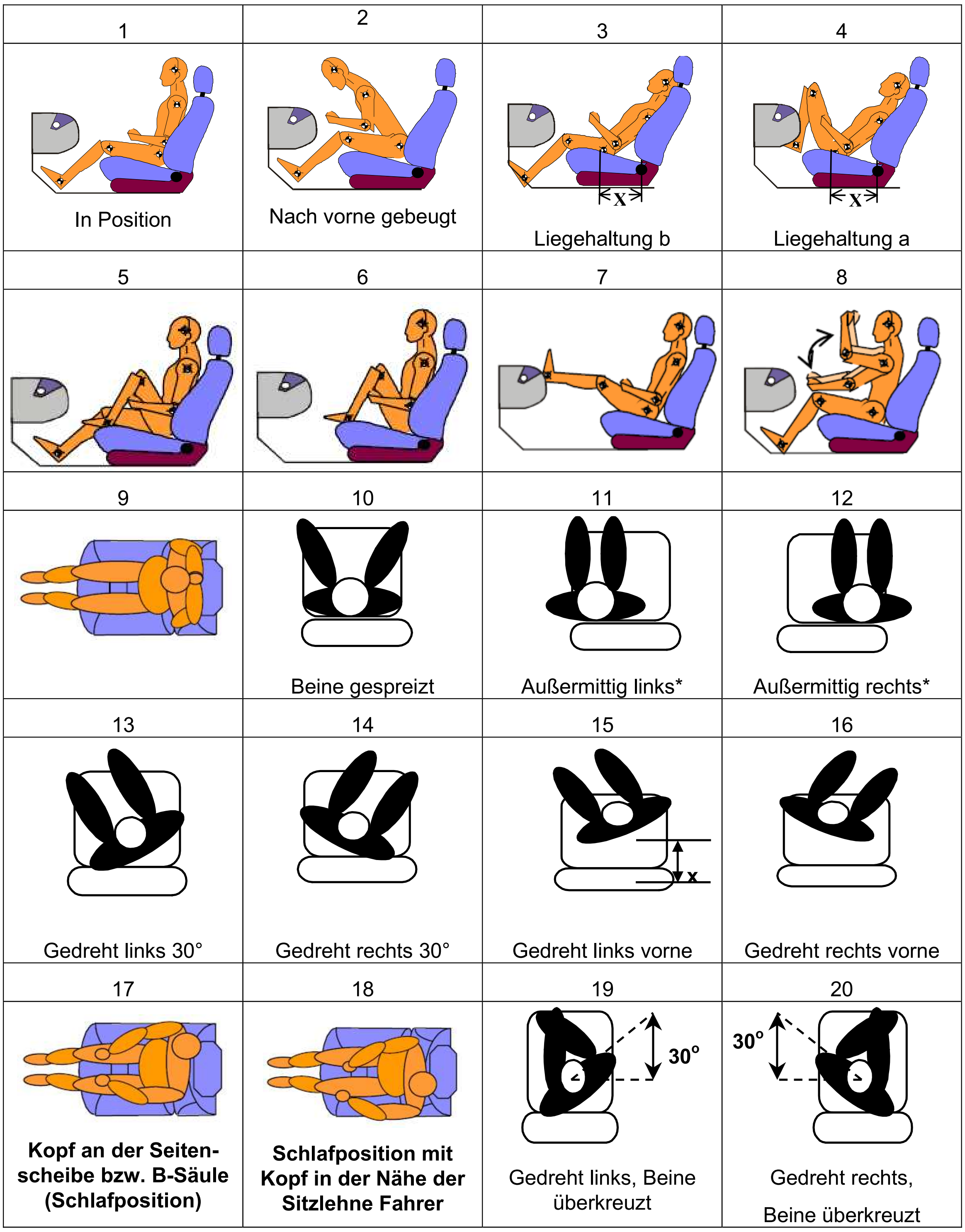


Fig. 7.11: VOCS industrial case study: customer test requirements

case of this project, a product-line manager is in charge of the various variants of VOCS systems. VOCS variants differ with respect to their related customer. These differences will result, in the case of this project, in taking into account particular positions of the front passenger, and particular car environment (e.g. seat size, sun shield shape, etc.). The knowledge of the Product-line manager is aware of a global set of postures which is taken from the union of the postures expressed by the different customers, as such he is capable of selecting test that are outside of the range of the customer test requirements but that still may be valuable to be exercised.

In a product-line context, several variants sharing common functionalities are produced over time. This enables the product-line manager to be aware of previous failures in the field of other variants than the current variant (i.e. the SUT). This knowledge of previous failures is indeed very valuable for test selection, and test selection should also take into account this particular failure cases. Unfortunately, in our context, as the VOCS is the first developed variant of the product-line no previous variants were in the field and thus no failures in the field could have been recorded.

Installation View

The installation engineer is particularly concerned with the physical environment in which the system is going to be used. The physical environment of the SUT may be in some cases more restricted to the one it was designed for. In consequence, the installation engineer may express some information related to the test selection of the SUT.

In the context of the VOCS, the physical environment of the system is the front inside part of cars. This includes: the front passenger seat, a part of the dash-board, the passenger's sun-shield. In automotive, manufacturers sometimes produce limited editions of a car model. These limited editions may provide specific indoor design, like a unique color for the seat and a unique type of material covering the seat. Other limited editions, may be sport models, in that case, it is common that backrest of the seats may not be modified (i.e. always upright), etc. It is possible to take advantage of these restrictions in order to express test selections that reflect these particular properties. This could result, with the example of the sport edition, in restricting the data type `Seat` to a single possible seat being in an upright position and with medium material remission[18] for the backrest and the sitting part of the seat.

Manufacturing View

Manufacturing is the final part of the system development life cycle before the product is delivered to the customer. In our context of embedded software, the hardware platform is assembled in an automated assembly line and the embedded software is uploaded to its hardware platform. Finally, end-of-line testing is performed that consists in performing some tests on both the hardware and the embedded software aspects of the assembled products. The execution time of end-of-line testing is critical and must be as low as possible. A large end-of-line testing execution time would result in less products being manufactured.

In this context, the tests that are performed are very basic test that must not take much execution time (in the order of a few seconds maximum) so that the assembly line is not disturbed by the end-of-line testing. Hence, the test selection must particularly restrict the data and

[18] In spectroscopy, it is the light which is scattered back from a material, as opposed to that which is "transmitted" through the material.

behavior to be exercised. Typical end-of-line tests for embedded software, is the test of each communication channel with one event. In the VOCS context, as shown in Table 7.1, there are five communication channels, as many as components of the SUT environment.

7.4.2 Production of test selection models from the test requirements

A number of test requirements have been identified in the previous section coming from different stakeholders. This following step consists in analyzing the informal test requirements for the purpose of precisely specifying them with the SERELA test selection language. The specified SERELA test selection specifications are then automatically translated into test selection models in the EMF [Ecla] format. In the following subsection, we illustrate the production of test selection models from test requirements via the presentation of one SERELA test selection specification derived from one of the several test selection requirements presented in Section 7.4.1.

Customer view: posture 1

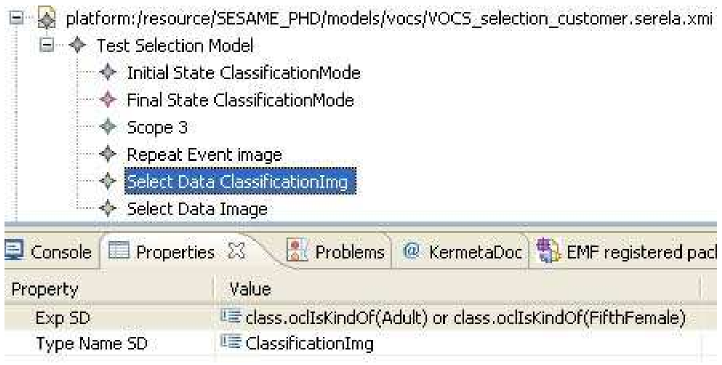
The customer primarily focuses on exercising the reception of images that satisfy given types of posture, as presented in the Customer View of the previous section. In the following, we present the specification of a SERELA specification, given in Figure 7.12, derived from the Posture 1 given in Figure 7.11:

- *Lines 01 to 04 of Figure 7.12.* In the context of selecting tests to exercise Posture 1, the focus is on exercising the static view of the SUT, rather than its dynamic view. In particular, with respect to the dynamic view, `image` event reception is solely exercised. a first selection regarding the behavior of the SUT is to focus on the `ClassificationMode` protocol state and on the `image` event reception. Consequently in line 01, the protocol state is set to be the initial state of the behavior; this means that the behaviors happening before this protocol state is not exercised. From the protocol state `ClassificationMode`, it is also specified that the final protocol state may be reached.
- *Lines 05 to 07 of Figure 7.12.* Posture 1 represents a human being sited in a nominal position; as such this test selection does not comprise empty seats nor child seats classifications but only exercise `Adult` or `FifthFemale` classes of object of interest. In SERELA, this may be expressed with instruction `select-data` by restricting the attribute `class` of data type `ClassifiedImg` to be either `Adult` or `FifthFemale`.
- *Lines 08 to 18 of Figure 7.12.* The characteristics of Posture 1 are analyzed in order to precisely define them as a test selection instruction. The `select-data` instruction is used to restrict the Image data type exercised to the ones that have back seat positioned upright (09) and nominal horizontal offset (10), both legs of the passenger are bended down and positioned forward (11-14), passenger's torso is upright (15) and touches the back seat (16). Lastly, there is no parasitic object in the images to exercise (17). The other attributes of the Image not expressed within the parameter of the select-data instruction are not restricted. For instance, the background illumination of the image is not restricted, thus it may be exercised with its three values (`Bright`, `NormalIllumination` and `Dark`).

When the SERELA test selection specification is defined, as given for instance in Figure 7.12, a text-to-model transformation interprets the textural specification and produce an equivalent test selection model that complies with the SERELA meta model given in the previous figure, 5.2. This transformation is implemented with Sintaks [sin], a graphical way of specifying text-to-model and model-to-text transformations. A test selection model that is equivalent to the SERELA specification is illustrated by the screen shot of the Eclipse environment in Figure 7.13.

```
initial-state ClassificiationMode
final-state ClassificationMode
repeat-event image min 1
scope 3
select-data ClassifiedImg {
 class.oclIsKindOf(Adult) or class.oclIsKindOf(FifthFemale)
}
select-data Image {
 seat.back.pos.oclIsKindOf(BRUpright)
 and seat.seat.offset.oclIsKindOf(SeatInBetween)
 and ooi.leg[0].pos.oclIsKindOf(LegBendedDown)
 and ooi.leg[0].angle.oclIsKindOf(LegForward)
 and ooi.leg[1].pos.oclIsKindOf(LegBendedDown)
 and ooi.leg[1].angle.oclIsKindOf(LegForward)
 and ooi.torso.pos.oclIsKindOf(TorsoUpright)
 and ooi.torso.offset.oclIsKindOf(OnSeat)
 and po = NULL
}
```

Fig. 7.12: Test selection specification derived from the posture 1 of the Customer View



```
platform:/resource/SESAME_PHD/models/vocs/VOCS_selection_customer.serela.xmi
  Test Selection Model
    Initial State ClassificationMode
    Final State ClassificationMode
    Scope 3
    Repeat Event image
    Select Data ClassificationImg
    Select Data Image
```

Property	Value
Exp SD	class.oclIsKindOf(Adult) or class.oclIsKindOf(FifthFemale)
Type Name SD	ClassificationImg

Fig. 7.13: EMF representation of the SERELA specification in Figure 7.12 (Screenshot from the Eclipse Environment)

7.4.3 Construction of a constrained model

In the previous subsection, a test selection model is produced that is derived from a SERELA test selection specification. In this subsection, the test selection model is applied on the static view (Figures 7.7, 7.8, 7.9, G.1, G.2 and G.3) and the dynamic view (Figure 7.10) of the VOCS initial model. Concretely, the application of the test selection model on the VOCS initial model is the automated execution of a generic model transformation written as a KerMeta program, given in Appendix B. The result of the model transformation is a constrained model that complies with the FOREST meta-model.

Within the SESAME tool chain, the resulting Constrained Model is in the format of an EMF model[19]. In the following, we present the resulting constrained model with its equivalent UML notation. In addition, in order to stress the modification of the initial model, the UML elements that are not reachable after test selection are not displayed[20]. The static and dynamic views of the constrained model, resulting from the application of the test selection model specified in Section 7.4.2, are presented in the following:

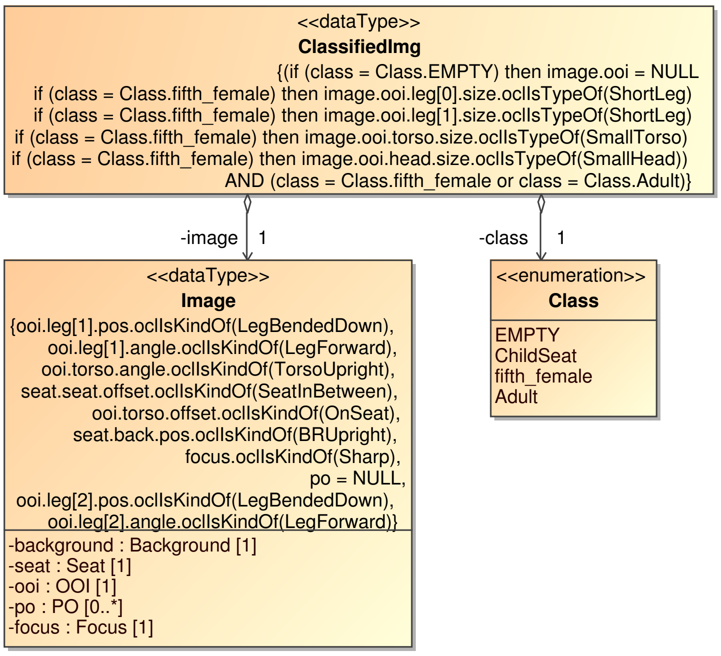
- Figure 7.15 illustrates the dynamic view of the constrained model. It comprises the reachable protocol states and protocol transitions of the constrained model. In deed, the dynamic view is rather restricted as the test selection expressed in the previous section focuses on exercising data aspects of the SUT. Thanks to the initial-state instructions, the protocol state `ClassificationMode` is the initial state of the dynamic view and only the transition triggered by the `image` event reception is reachable. This is due to the `repeat-event` instruction enforcing to exercise the `image` event at least one time combined with the `scope` instruction restricting the SUT behavior to 3 statuses[21].

- Figures 7.14, 7.16 and 7.17 illustrate parts of the static view of the constrained model. In Figure 7.14 constraints have been added to the data types `ClassifiedImg` and `Image` according to the two `select-data` instructions of the SERELA test selection specification of Figure 7.12. Figures 7.16 and 7.17 display the reachable leaf data types according to the SERELA specification. For instance, the backrest position is restricted to the upright position, i.e. there is only one leaf data type (`BRUpright`) for the abstract data type `BackrestPosition`. Similarly, the test also focuses solely on object of interests with torso in upright position and touching the back seat.

[19] It is represented in a tree-like notation, in a similar fashion than Figure 7.13

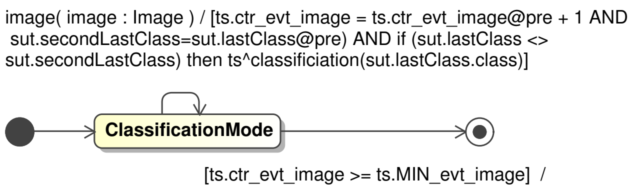
[20] The UML elements that are not displayed in the UML diagrams still remain part of the constrained model. For instance, the protocol state DiagnosticMode of the dynamic view is an element of the constrained model even though it is not visible in Figure 7.15

[21] These 3 statuses comprise the initial status and the final status, thus there is only one status (the second status) during which an event may be received that exercises the SUT.



<<dataType>>
ClassifiedImg
{(if (class = Class.EMPTY) then image.ooi = NULL
if (class = Class.fifth_female) then image.ooi.leg[0].size.oclIsTypeOf(ShortLeg)
if (class = Class.fifth_female) then image.ooi.leg[1].size.oclIsTypeOf(ShortLeg)
if (class = Class.fifth_female) then image.ooi.torso.size.oclIsTypeOf(SmallTorso)
if (class = Class.fifth_female) then image.ooi.head.size.oclIsTypeOf(SmallHead))
AND (class = Class.fifth_female or class = Class.Adult)}
-image 1
-class 1
<<dataType>>
Image
{ooi.leg[1].pos.oclIsKindOf(LegBendedDown),
ooi.leg[1].angle.oclIsKindOf(LegForward),
ooi.torso.angle.oclIsKindOf(TorsoUpright),
seat.seat.offset.oclIsKindOf(SeatInBetween),
ooi.torso.offset.oclIsKindOf(OnSeat),
seat.back.pos.oclIsKindOf(BRUpright),
focus.oclIsKindOf(Sharp),
po = NULL,
ooi.leg[2].pos.oclIsKindOf(LegBendedDown),
ooi.leg[2].angle.oclIsKindOf(LegForward)}
-background : Background [1]
-seat : Seat [1]
-ooi : OOI [1]
-po : PO [0..*]
-focus : Focus [1]
<<enumeration>>
Class
EMPTY
ChildSeat
fifth_female
Adult

Fig. 7.14: VOCS industrial case study: FOREST constrained model static view



image(image : Image) / [ts.ctr_evt_image = ts.ctr_evt_image@pre + 1 AND
sut.secondLastClass=sut.lastClass@pre) AND if (sut.lastClass <>
sut.secondLastClass) then ts^classificiation(sut.lastClass.class)]
ClassificationMode
[ts.ctr_evt_image >= ts.MIN_evt_image] /

Fig. 7.15: VOCS industrial case study: FOREST constrained model dynamic view

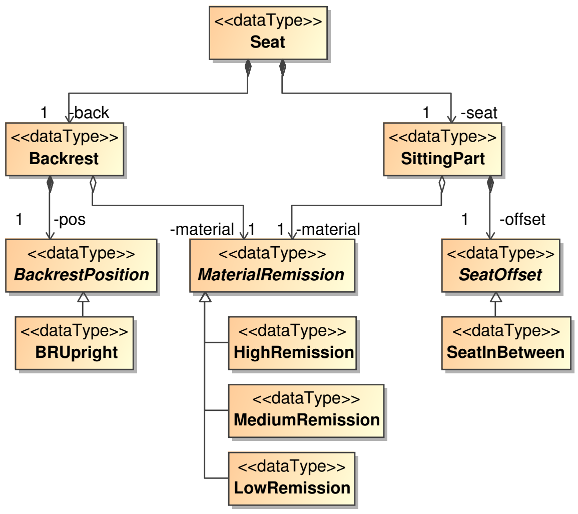


Fig. 7.16: VOCS industrial case study: Seat constrained data type definition

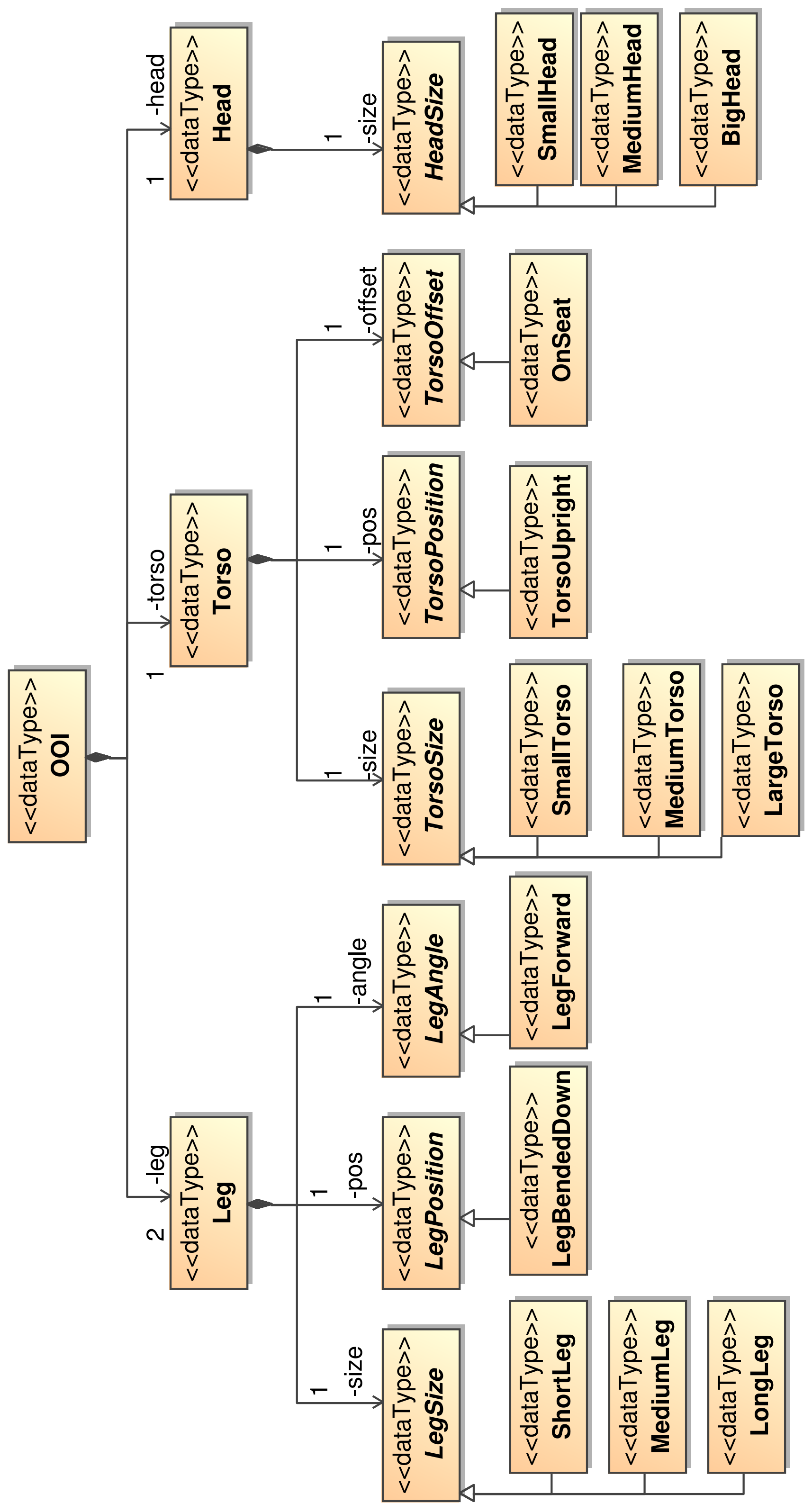


Fig. 7.17: VOCS industrial case study: OOI constrained data type definition

7.5 Evaluate test selection

In Chapter 6, a number of structural and semantical metrics are suggested that evaluate initial and constrained models for the purpose of evaluating the underlying test selections.

During the Evaluate Test Selection task, the metrics of interest for the particular VOCS project are identified with respect to the company's test requirements. Then SESAME metrics are selected that express the metrics of interest in the context of the VOCS project. Lastly, we present the values of the selected metrics for the initial model (described in Section 7.3) and the constrained model (described in Section 7.4).

7.5.1 Metrics for the VOCS project

The five following types of metrics have been identified as a result from discussions with the collaborating company:

(1) *state and event coverage.* For a test selection to be valid, the resulting constrained model must enable to exercise each (protocol) state and each event reception/sending at least once. These two types of coverage are defined within the SESAME approach (`CoveredState` and `CoveredEvent` metrics), and thus may be used as is. Four `CoveredState` metrics are specified that measure the coverage of each protocol state and fourteen `CoveredEvent` metrics are specified that measure the coverage of each event of the VOCS static view.

(2) *use-case coverage.* The constrained model resulting from a valid test selection must enable to exercise the four scenarios expressed by the four use-cases given in Section 7.2. The scenarios are expressed as a sequence of exchanges of events. `CoveredTransPath` metrics are used to measure if the scenarios may be exercised with the constrained model.

(3) *output data domain coverage.* Output data domain comprises the different data values that are conveyed by events sent from the SUT to its environment. In particular, the classification values (data type `Class`) conveyed by the sending of the `classificiation` event is of high priority in the context of the VOCS project. This metric measures the coverage of each classification type. For this purpose, we use the `CoveredValueOfStructuredDatatype` metric to define four metrics, one for each classification type, as shown in Figure 7.18.

(4) *VOCS-specific.* The VOCS project aims at classifying the front passenger seat occupancy into four different categories. A set of predefined passengers' postures are defined by the customer. In order to fulfill the customer's test requirements, it is important to check that the test selections do not discard some postures specified by the customer. More precisely, for each typical posture defined by the customer, a metrics may be defined that evaluates if the posture may still be exercised with the constrained model.

Use-case coverage

The test selection covers the primary use-case (UC1) if the behavior of the constrained model comprises at least the sequence of event receptions[22]: `ignitionOn` (tr1); `wakeUp` (tr5); `timeOutImageReq` (tr9); `image` (tr11); and `timeOutSendClass` (tr10).

[22] In brackets are given the related protocol transitions.

The test selection covers the extension use-case “Change in classification of occupant” (UC2) if there is at least one status of the behavior of the constrained model for which the state variable of the last classification (`sut.lastClass`) has a different value than the state variable of the second last classification (`sut.secondLastClass`). Unlike the three other use-cases, UC2 is based on the values of the SUT state variables, that is why we make use of the `CoveredValueOfStateVar` metric instead of the `CoveredTransPath` metrics, as shown in Table 7.4.

The test selection covers the extension use-case “Storage of data in event of a crash” (UC3) if the behavior of the constrained model comprises at least the `crash` event reception (triggering the protocol transition `tr8`).

The test selection covers the extension use-case “Retrieval of data after a crash” (UC4) if the behavior of the constrained model comprises at least the sequence of: `crash` event reception (tr8); `authenticate` (tr12); and `diagnosticReq` (tr13).

VOCS-specific metrics

A metric may be defined for each posture in Figure 7.11. The metrics is defined to evaluate if it is possible to exercise the SUT with the image event reception satisfying the given constraint. Figures 7.19 and 7.20 illustrate how these metrics may be defined using the `CoveredValueOfEvent` metrics for postures 1 and 2.

7.5.2 Compute selected metrics

In this subsection, a table is given in a similar fashion than Table 6.1. This table includes values for the VOCS-specific metrics defined in the previous sub-section.

Firstly, it can be noticed that the only covered state is the protocol state `ClassificationMode`, which results in the test selection coverage being of 25%.

Secondly, concerning event coverage, the image and the classification events may be exercised with the test selection; all the other events are not reachable and thus can not be exercised.

Thirdly, the use-case UC2 is covered by this test selection. Indeed, the classification state variable is assigned to the classification (either adult or fifth-female) related to the reception of the `image` event, which in turn result in a change of occupant classification.

Fourthly, the classification coverage is of fifty percent because the test selection focuses on the adult and fifth female object of interest discarding empty seat and child seat classifications.

Lastly, only the posture 1 is covered by the test selection. Indeed this was the objective of the test selection while defining it.

7.5.3 Conclusion

The (sole) test selection, derived from posture 1, is not a valid test selection as the metrics set by the company are not fully satisfied. This is not surprising as this test selection is quite restrictive and focuses on exercising the data aspects of the SUT neglecting the behavioral aspects.

In order to produce a valid test selection, additional test selection must be specified that cover other aspects of the SUT. In the end, the set of the various test selections should altogether fulfill the different metrics of interest for the VOCS project.

```
CoveredValueOfStructuredDatatype( ClassifiedImg, class.oclIsKindOf(Empty))
CoveredValueOfStructuredDatatype( ClassifiedImg, class.oclIsKindOf(ChildSeat))
CoveredValueOfStructuredDatatype( ClassifiedImg, class.oclIsKindOf(FifthFemale))
CoveredValueOfStructuredDatatype( ClassifiedImg, class.oclIsKindOf(Adult))
```

Fig. 7.18: Metrics for the output data domain coverage of the Class data type

```
CoveredValueOfEvent(image,
  seat.back.pos.oclIsKindOf(BRUpright)
  and seat.seat.offset.oclIsKindOf(SeatInBetween)
  and ooi.leg[0].pos.oclIsKindOf(LegBendedDown)
  and ooi.leg[0].angle.oclIsKindOf(LegForward)
  and ooi.leg[1].pos.oclIsKindOf(LegBendedDown)
  and ooi.leg[1].angle.oclIsKindOf(LegForward)
  and ooi.torso.pos.oclIsKindOf(TorsoUpright)
  and ooi.torso.offset.oclIsKindOf(OnSeat)
  and po = NULL
)
```

Fig. 7.19: Metrics measuring coverage of posture 1

```
CoveredValueOfEvent(image,
  seat.back.pos.oclIsKindOf(BRUpright)
  and seat.seat.offset.oclIsKindOf(SeatInBetween)
  and ooi.leg[0].pos.oclIsKindOf(LegBendedDown)
  and ooi.leg[0].angle.oclIsKindOf(LegForward)
  and ooi.leg[1].pos.oclIsKindOf(LegBendedDown)
  and ooi.leg[1].angle.oclIsKindOf(LegForward)
  and ooi.torso.pos.oclIsKindOf(TorsoBended)
  and ooi.torso.offset.oclIsKindOf(OnSeat)
  and po = NULL
)
```

Fig. 7.20: Metrics measuring coverage of posture 2

Description	**Metrics**	**Evaluated Model**		**Test Selection Coverage**
		Initial	**Constrained**	
State coverage				
	CoveredState(EnergySaving)	true	false	0%
	CoveredState(Sleep)	true	false	0%
	CoveredState(ClassificationMode)	true	true	100%
	CoveredState(DiagnosticMode)	true	false	0%
	CoveredStates	4	1	25%
Event coverage				
	CoveredEvent(image)	true	true	100%
	CoveredEvent(classification)	true	true	100%
	CoveredEvent(ignitionOn)	true	false	0%
	...	...	...	...
	CoveredEvent(imageReq)	true	false	0%
	CoveredEvents	14	2	14%
Use-case coverage				
UC1	CoveredTransPath(tr1, tr5, tr9, tr11, tr10)	true	false	0%
UC2	CoveredValueOfStateVar(sut.lastClass <> sut.secondLastClass)	true	true	100%
UC3	CoveredTransPath(tr8)	true	false	0%
UC4	CoveredTransPath(tr8, tr12, tr13)	true	false	0%
		4	1	25%
Output classification data coverage				
EMPTY	See line 01 of Figure 7.18	true	false	0%
CS	See line 02 of Figure 7.18	true	false	0%
5th Female	See line 03 of Figure 7.18	true	true	100%
ADULT	See line 04 of Figure 7.18	true	true	100%
		4	2	50%
VOCS-specific metrics				
Posture 1	See Figure 7.19	true	true	100%
Posture 2	See Figure 7.20	true	false	0%
		20	1	5%

Tab. 7.4: VOCS: metrics on the sample test selection described in Figure 5.3

7.6 Generate abstract test cases

FOREST models are given semantics in terms of Alloy models[23]. This semantics relation between FOREST models and Alloy models is implemented as a model transformation in a KerMeta program. The Tester must execute the KerMeta program that takes care of transforming the given FOREST model (for which the Tester wishes to generate abstract test cases from) and results in an Alloy model. For instance, taking the constrained model of the previous section (7.4) as input, the KerMeta model transformation produces the related semantic model in Alloy.

In the context of the SESAME approach, the main advantage of Alloy models is that they may be formally analyzed. The formal analysis is performed by the Alloy Analyzer tool that supports this task automatically. The formal analysis is automated by the Alloy Analyzer. The Alloy Analyzer takes as input an Alloy model and produces sequentially the set of all possible instances of the input model. Instances are shown graphically within the Alloy Analyzer as in Figure 7.21. Instances may also be read via a Java API in order to automate their interpretation. The latter representation (i.e. via API) is used within the SESAME tool-support to generate abstract test cases automatically.

Figure 7.21 is a graphical representation of an instance of the constrained model previously described. The graphical notation used by the Alloy Analyzer is composed of rectangles and arrows. On the one hand, rectangles represent instances of Alloy signatures; the name of the instance is given on the first inner line of the rectangle. For instance, in Figure 7.21, there are two instances of signature `TS`, namely `TS0` and `TS1`. Instances of signatures are given the name of the related signature suffixed with a number to distinguish the different instances of the same signature[24]. On the other hand, instances of Alloy relations may be represented in two (equivalent) ways[25]. Firstly, they may be represented as arrows between the rectangles (i.e. association-like style). For instance, signatures `Step2` and `image` are related to each other via the binary relation `eventReceived` [26]. Secondly, they may be represented within the signatures' boxes (i.e. attribute-like style). For instance `curPstate` relation relates the two signature instances `Status1` and `ps_CM` [27].

The different statuses of the instance are represented as a sequence of `Status` signatures[28] `Status0` is the initial status, `Status1` is the status following the initial status and `Status2` is the final status. The initial status is in relation with the instance of the Test System `TS1` which is itself in relation with the state variables set to their respective initial values. The event received in the initial status is the image event with an image that has the characteristics as described in the test selection specification (backrest upright, leg forward and bended down, etc.) and in this instance the other unconstrained properties of the image are the backrest and the sitting part have low remission values, size of head legs, and torso are small and the background has a normal illumination. In the following status, i.e. `Status1`, the `sut` state variable `lastClass` is set to a `ClassifiedImg` that is of type `FifthFemale`, and for which its image property is the image received by the `image` event reception. In this status, the selected transition is the

[23] The semantics of FOREST models is described in Section 4.5

[24] It is important to note that the numbering of the signatures instances are assigned randomly by the Alloy Analyzer, except for signature where a linear order is defined.

[25] In Figure 7.21, we have used both representations for the purpose of making the figure fit nicely in one page!

[26] This may be informally read as the image event is received during Step2

[27] This may be informally read as the current protocol state of the Status1 is the protocol state ps_CM, i.e. the Classification Mode

[28] As described in the previous section, 4.5.3, the sequence is defined in Alloy as a linear order, consequently the numbering of the Status signatures of an instance are following the sequencing of the statuses. First status being Status0, next one being Status1, etc.

transition `pt_tr14`, whose target protocol state is the final state. Thus the final status, i.e. `Status2` has for current protocol state the state `ps_final`.

As a concluding remark on the abstract test case generation, the generation of all Alloy instances (i.e. test cases) with the Alloy Analyzer have been performed in a total execution time of 258 seconds and resulted in a total number of instances of 4 428.

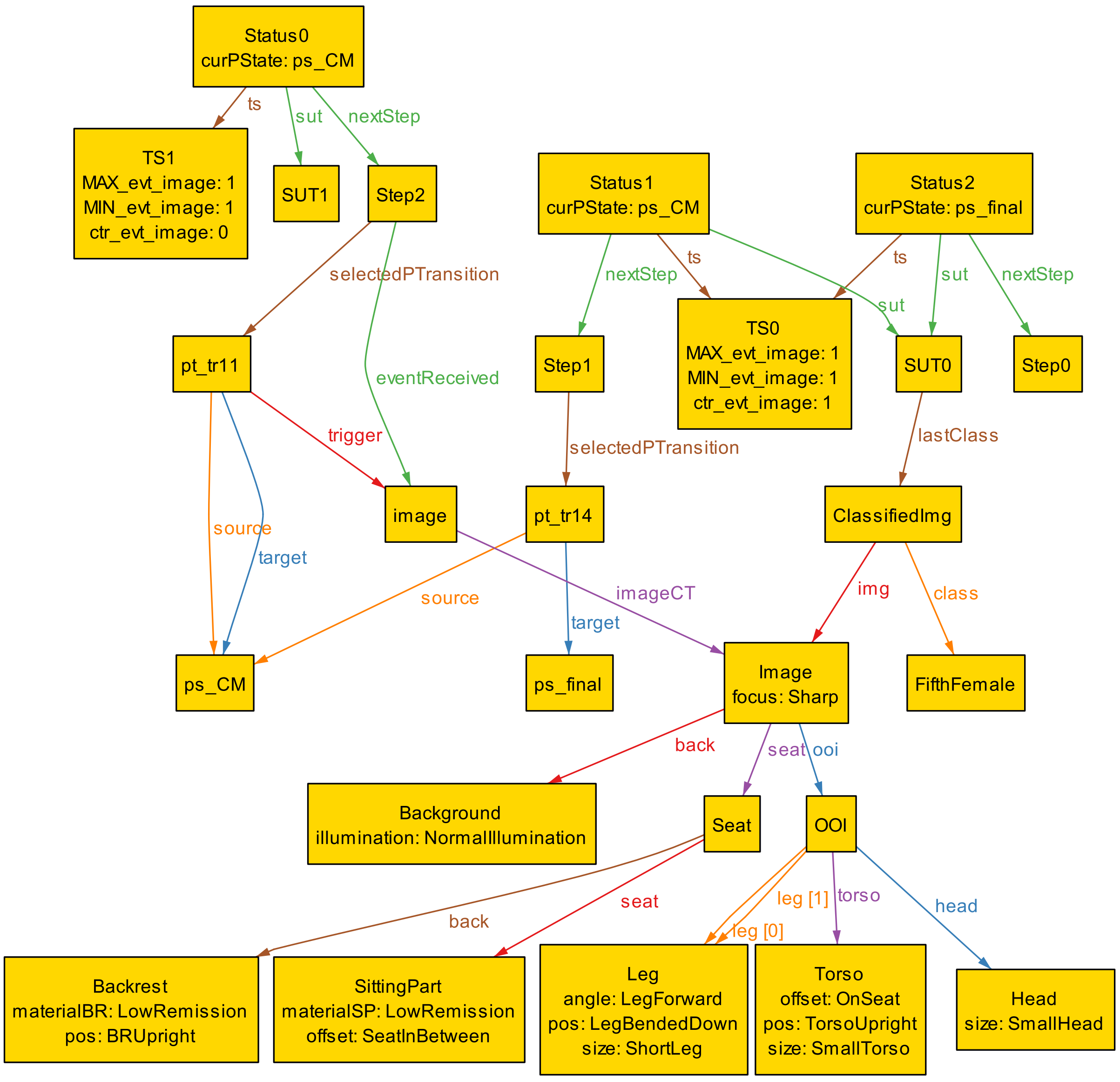


Fig. 7.21: VOCS: an Alloy instance of a constrained model

7.7 Conclusion on the industrial case study

This chapter establishes a proof of concepts of the SESAME approach taking into account industrial concerns. In this chapter, the different tasks of the SESAME approach have been illustrated with the VOCS industrial case study.

As a first task, we defined abstract (analysis-level) use-cases from the existing implementation-level use-cases that were defined within the project. From these use-cases, we specified the static and dynamic views of the initial FOREST model of the system under test. The static view takes the form of a number of class diagrams that define the events exchanged between the VOCS software and its environment, as well as the data that is conveyed within the exchanged events. The dynamic view informs about the reactive behavior of the SUT to the events sent from the Test System. The FOREST model notation, defined within the SESAME approach, fit well to the specification of the data and behavior of the industrial case study.

The following task illustrated in this chapter, is the test selection specification task. As a preamble to this task, we show that the number of possible tests cases for this project is too large to be exhaustively tested, even though the SUT is a small-size software. Then, we describe the different concerns of the multiple stakeholders, as defined in the 5+2 View Model on Test Selection of the SESAME approach. We pursue by providing a test selection specification of one of the test requirements of one of the multiple test stakeholders. The test selection is described with the SERELA domain-specific test selection language defined within the SESAME approach. We illustrate what is the concrete representation of the SERELA specification within the SESAME tool chain (EMF model in a tree format). Lastly, we show the result of the application of the test selection specification on the initial FOREST model. This last step is illustrated by UML diagrams that show the OCL-F constraints added to the initial model's static and dynamic views.

The test selection evaluation objectives of the company are presented (state, event coverage, use-case coverage, etc.). For each of these objectives, metrics are defined based on the structural and semantical metrics of the SESAME approach. A table summarizes the computation of these metrics on the initial and constrained model; the test selection coverage is also computed that shows how much of the metrics is covered after the test selection.

Last task illustrated in this chapter, is the generation of abstract test cases from the constrained model with the help of the Alloy Analyzer. An instance of the constrained model that results from the formal analysis of the Alloy Analyzer is presented.

This chapter is a first step toward a wide-spread adoption of the SESAME approach in the IEE company, perspectives for the wide-spread adoption of the approach are presented shortly in the further Chapter 9.

8. CONCLUSION

In this thesis, we have defined the SESAME model-driven test selection approach. The SESAME approach is composed of the following aspects:

- The SESAME process model defines an ordered set of activities related to test selection, the work products required as input and output of these activities, as well as the human roles involved for each activity. The process is described in Chapter 3 with the SPEM notation specifically aimed at defining processes.
- The three following specification languages are used in the context of SESAME processes:
  - FOREST, a graphical modeling language for the specification of the analysis model of the system with state machines and data based on existing UML artifacts [OMG07b]. This modeling language is intended to end-users, typically software analysts. This language is presented in Chapter 4.
  - The FOREST language is formally defined in terms of the existing Alloy formal specification language. The idea is to model the system with FOREST and translates FOREST models into Alloy [Jac06] specifications. These formal Alloy specifications are then used for formal analysis of the FOREST models in order to evaluate the test selection activity and generate abstract test cases. The semantics of FOREST is presented in Section 4.5 of Chapter 4.
  - SERELA, a domain-specific language defined for the sole specification of test selections. This test selection language intended to be used by end-users, typically test engineers or software analysts. This language is defined in Chapter 5.
- A set of metrics are defined that help determining whether the results of the test selection procedure are of acceptable quality. The metrics that measure FOREST models are presented in Chapter 6. Their purpose is twofold. Firstly, the introduced metrics may be used to measure attributes of the input and output models of the test selection activity in order to evaluate test selections. Secondly, the introduced metrics may be used for the purpose of evaluating if a constrained model matches the test requirements and thus if it is ready for test generation.

The main contribution of the SESAME approach is to define a test selection process that is fully integrated in the model-driven engineering: it is grounded on the coherent and systematic usage of *model transformations* for the evolution of models and *meta models* for the precise specification of handled models.

A novelty that is proposed in the SESAME approach is that the test selection activity produces a test model that is represented with the same meta model and at the same abstraction layer than the analysis model of the SUT, which clarifies the impact of the test selection specifications with respect to the behavior and data that will be exercised.

Another contribution is the usage of a domain-specific language for the purpose of test selection specification, this enforces engineers to explicit the test selection that is made on models and ease the understanding of its specification as the test selection specification is defined using test selection terminology.

Lastly, we define metrics that are targeted at the measurement of the *semantics* of the analysis models and test models. In contrast with most other testing approaches that focus on the computation of structural metrics, i.e. measurements of static elements of models.

As a summary, the main advantages of using the SESAME approach are:

(1) the specification of the system (analysis model) is the main artifact to decide for further test activities

(2) a domain specific language is used for test selection specification

(3) the test model specifies the tests that will be effectively performed with the same notation as the analysis model

(4) metrics are available for the evaluation of the test selection

(5) the overall process is supported by open-source tools (KerMeta and Alloy).

9. PERSPECTIVES

Abstract

In this chapter, we sketch the possible extensions of the SESAME approach. We present the possible improvements of the approach in four parts: possible improvements of the SESAME process model, of the SERELA test selection language, of the model coverage metrics, and of the test generation activity.

9.1 SESAME process extension

In the following subsections, we shortly present the activities that should be investigated for the SESAME process to cover testing activities taking place after test selection. Figure 9.1 illustrates these activities (dashed-boxes) and their inter-relations over time.

Test Concretization

Abstract test cases which are generated from the SESAME Test Generation activity can not be executed on the SUT as they are: they must be tailored to the specific design of the system under test. Test concretization is an activity that associates concrete values to the abstract test cases and that creates a wrapper to execute the input sequence of the test cases with the SUT. A concrete test case is composed of:

(1) Preamble (optional): A set of instructions, message calls, etc. that set the system to the initial state appropriate to the test cases.

(2) Test Body: The instruction(s), message call(s), etc. that correspond(s) to the test set input. E.g. CAN messages, C function calls, etc.

(3) Test Identification (optional): Call some observation operations to help the oracle.

(4) Postamble (optional): A set of instructions, message calls, or else ... that set the system back in its original state, i.e. its state before executing preamble.

Test Execution

During the test execution, each concrete test case is executed on the system under test. This activity is usually automated through the implementation of scripts that take care of calling the SUT operations related to the concrete test cases. The Observable Behavior, i.e. the

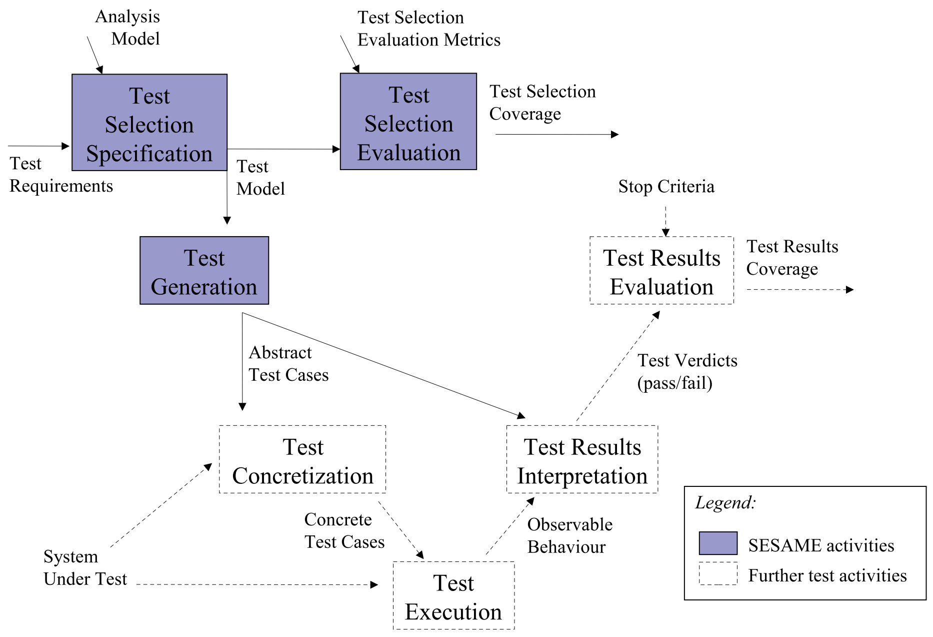


Fig. 9.1: Further work on test activities

output of the system under test, must be memorized for each test case execution for further interpretation. In the case of an embedded system, the system usually communicates with its environment through a data bus, e.g. CAN bus in the automotive industry, thus the observable behavior is gathered by "listening" to the bus for messages sent by the SUT. For instance, the Vector company [Vec] provides two tools (CANalyzer and CANoe) adapted to the listening of CAN buses communication activity, a hardware component is physically connected to the bus and reports to a PC the observed messages on the bus.

Test Results Interpretation

During this activity, the memorized Observable Behavioral of the previous activity is interpreted so that its contained information may be compared to the expected result. The abstraction level of the observed behavior must be raised to a higher level fitting to the analysis models specifications. A particular test component, the oracle, is responsible to decide if the Observable Behavior matches the expected results. In our context, the oracle may be derived from the dynamic view of the test model, in particular from the post-conditions of the protocol transitions. The decision from the oracle results in the test case being in one of the three following state:

- Test case has *passed*, if the observed behavior matches the expected output.
- Test case has *failed*, if the observed behavior does not match the expected output.
- Test case execution result is *inconclusive*, if not enough information is known to decide if it has passed or failed.

Test Results Evaluation

This activity consists in evaluating the executed tests by measuring the number of passed, failed, inconclusive test cases. During this activity, the succeeded and failed test cases are evaluated against a Stop Criteria, which describes when the evaluation of the test cases is satisfactory. Depending on the resulting evaluation of the Stop Criteria, engineers need to decide:

- To stop testing, when the Stop Criteria is satisfied because engineers have gained enough confidence in the system, or
- To go on testing, because the Stop Criteria was not satisfied thus some errors were founds in the SUT, and faults need to be located and corrected. Then the test set will be executed again, to check that the errors are not present anymore and to check that no additional errors were introduced.

9.2 Test selection language

In the state of the art chapter, we have shortly presented the different kinds of model coverage criteria identified by Utting and Legeard [UL07]. The SERELA specification language, as presented in this thesis, is a first version of a domain-specific test selection language that provides test selection instructions for the selection of data and behavioral aspects of the system under test. Indeed, there is plenty of room for extending the SERELA language to include additional instructions of different types. For instance, SERELA instructions could be defined that intented to exercise particular faults of the SUT model. This would require extending FOREST models with explicit fault specification. Some work on the direction of explicit fault specification within UML models exist [BG08] that could be adapted to the SESAME approach. Another possible perspective, for new SERELA instructions could be to directly select particular requirements to be exercised. This would require systematic descriptions of the requirements that may be selected for test in terms of data or behavioral aspect of FOREST models.

9.3 Model coverage metrics

Previous work by Baudry and Le Traon [BT05] has identified structural metrics on UML class diagrams that measure the design testability. This paper exhibits the impact of depth of inheritance over the testability of the model. In the context of FOREST models, we have also observed that the inheritance complexity of the static view of FOREST models has a strong impact on the computational cost. This is due to the introduction of additional variables in the formal representation of the model in Alloy, resulting in an increased test space to be explored by the SAT solver. In this context, it would be interesting to investigate on the metrics that may be used to reveal the structural complexity aspects of FOREST models which have an impact on the computational cost of the test generation.

9.4 Test generation

In the current implementation of the test generation process with the Alloy Analyzer, the exploration of all possible behaviors of the SUT constrained to the test selection specifications is performed randomly. As the computation of all the instances is costly, it may save valuable time to find a way to guide the instantiation process so that instances of particular importance are produced first. A possible perspective for the test generation process is to define priorities on the order of test case generation. An intuitive way of doing that is to produce test cases in two steps. A first step would be to constrain the (already constrained) test model so that particular aspects of the test model are explored first, i.e. priority test cases are generated. For instance, a priority aspect could be that it is important to exercise each protocol state of the dynamic view of the test model. Then, when priority aspects have been explored, the second step would be to constrain the test model so that the previous explored aspects are not taken into account in the further generation of test cases. These two steps, could be performed when there are two levels of prioritization (important or not important). It would be interesting to investigate on the applicability of this intuitive two-prioritization levels and also to on the possible generalization to n-prioritization levels.

9.5 Industrial perspective

One of the most important perspectives in the context of this industrial PhD thesis, is indeed the industrial perspective of the SESAME approach. SESAME has been developed in the context of a collaboration between academia and industry. More precisely the approach has been defined as an improvement of the company's current practice concerning its software development process. Throughout the PhD project, the approach has been defined iteratively taking into account valuable feedbacks from the manager responsible for the embedded software team. Moreover, the industrial case study presented in Chapter 7 has been performed during work meetings together with the embedded software team manager from the company.

The SESAME approach represents an important improvement for IEE's software development process, in particular for its requirements analysis and test selection phase. The process fulfills the process improvements requirements set by the company.

Now that a first version of the process is defined the industrial transfer of the approach may be performed. The two main steps for industrialization of the process are: firstly, go through a complete evaluation on a medium-size pilot project at the company with embedded software engineers; secondly, widespread adoption on all new projects of the company.

9.5.1 Improving tool-support for an easier industrial transfer

Current tool-support of the SESAME approach is composed of KerMeta programs that define the semantics of the different languages and Alloy Analyzer that perform formal analysis of the models. KerMeta development environment is integrated in the Eclipse development framework [Eclb]. The current state of implementation allows engineers to launch the KerMeta programs that interpret SERELA specifications, compute metrics, and generate test cases. This execution within the development environment of KerMeta (i.e. in Eclipse, this is supported by a KerMeta Perspective). An interesting improvement concerning the tool-support is to implement a SESAME Control Center, with its own Perspective within the Eclipse framework. This would enable to produce test models from SERELA specifications, evaluate metrics on test selections and generate test cases without a direct execution with KerMeta. Another improvement would be to have syntax highlighting facility for the specification of SERELA specifications. Lastly, a simple user-interface could be valuable to select the metrics available so that engineers may choose what metrics should be computed. All these would indeed facilitate the industrial transfer, as engineers would not need to get to know the KerMeta environment and would perform SESAME-related automated activities transparently from its implementation.

BIBLIOGRAPHY

[ABJK06] Freddy Allilaire, Jean Bezivin, Frederic Jouault, and Ivan Kurtev. Atl: Eclipse support for model transformation. In *eTX*, 2006.

[Abr96] J. R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[AFGC03] Anneliese Andrews, Robert France, Sudipto Ghosh, and Gerald Craig. Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.

[AK02] Colin Atkinson and Thomas Kühne. The role of metamodeling in MDA. In *Workshop in Software Model Engineering (WISME@UML'2002)*, 2002.

[AK03] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[AK06] Colin Atkinson and Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[All] Alloy. Alloy analyzer 4. online. http://alloy.mit.edu/alloy4/.

[Ana07] Kyriakos Anastasakis. UML2ALLOY reference manual v0.26. draft, The University of Birmingham, 2007.

[Are99] Demissie Aredo. Formalizing UML class diagrams in PVS. In *OOPSALA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations*, 1999.

[AUT08] AUTOSAR. Requirements on network management. Technical Report v2.0.4, Automotive Open System Architecture, 2008.

[BA02] A. L. Baroni and F. B. Abreu. Formalizing object-oriented design metrics upon the UML meta-model. In *Brazilian Symposium on Software Engineering*, 2002.

[BA05] Behzad Bordbar and Kyriakos Anastasakis. UML2ALLOY: A tool for lightweight modelling of discrete event systems. In *IADIS*, 2005.

[Bar97] Stéphane Barbey. *Test Selection for Specification-Based Testing of Object-Oriented Software Based on Formal Specifications*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Département d'Informatique, 1015 Lausanne, Switzerland, 1997.

[BDG+08] Paul Baker, Zhen Ru Dai, Jens Grabowski, Oystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-driven Testing: Using the UML Testing Profile*. Springer, 2008.

[Ber00] A. Bertolino. Knowledge area description of software testing - SWEBOK. Technical report, Joint IEEE-ACM Software Eng. Coordinating Committee, 2000. http://www.swebok.org.

[Bez04] Jean Bezivin. In search of a basic principle for model-driven engineering. *UPGRADE - European Journal for the Informatics Professional*, 5(2):21–24, April 2004.

[BG08] Andrey Berlizev and Nicolas Guelfi. Fault-tolerance requirements analysis using deviations in the CORRECT development process. In *Methods Models and Tools for Fault Tolerance*. Springer Verlag, 2008.

[Bib97] Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, 1997.

[Bin99] Robert Binder. *Testing object-oriented systems: models, patterns, and tools*. object technology series. Addison-Wesley Professional, Reading, Massachusetts, USA, 1999.

[BJK$^+$05] M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner. *Model-based Testing of Reactive Systems - Advanced Lectures*, volume LNCS 3472. Springer, 2005.

[Bla05] Paul E. Black. abstract data type. Dictionary of Algorithms and Data Structures [online], February 2005. U.S. National Institute of Standards and Technology (http://www.nist.gov/).

[Bog00] Kirill Bogdanov. *Automated Testing of Harel's Statecharts*. PhD thesis, University of Sheffielf, 2000.

[Boo94] Grady Booch. *Object-oriented analysis and design with applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, second edition, 1994.

[BS05] Bruno Bouyssounouse and Joseph Sifakis. *Embedded Systems Design, the ARTIST Roadmap for Research and Development*, volume LNCS 3436. Springer, 2005.

[BT05] B. Baudry and Y. Le Traon. Measuring design testability of a UML class diagram. *Information and Software Technology*, pages 1–21, 2005.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Spring days*, 2003.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[CJ05] Franck Chauvel and Jean-Marc Jézéquel. Code generation from UML models with semantic variation points. In *MoDELS : model driven engineering languages and systems*, volume 3713 of *LNCS*, pages 54–68. Springer, 2005.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CK06] Mirko Conrad and Alexander Krupp. An extension of the Classification-Tree Method for embedded systems for the description of events. *Electronic Notes in Theoretical Computer Science*, 164(4):3–11, October 2006.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison Wesley, Reading, MA, USA,, 2001.

[Dai06] Z. R. Dai. *An Approach to Model-Driven Testing - Functional and Real-time Testing with UML 2.0, UTP and TTCN-3.* PhD thesis, Technical University of Berlin, 2006.

[DH89] D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(7):798–807, 1989.

[DJHP98] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 186–238. Springer-Verlag, 1998.

[Ecla] Eclipse. Eclipse modeling framework. online. http://www.eclipse.org/modeling/emf.

[Eclb] Eclipse. Eclipse.org. online. http://www.eclipse.org.

[EFLR98] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Developing the UML as a formal modelling notation. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 297–307, 1998.

[ETS03] ETSI. The testing and test control notation (TTCN-3) version 3. Multipart standard 201 873: Methods for Testing and Speification (MTS), European Telecommunications Standards Institute, 2003.

[FELR97] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. In *OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81, 1997.

[FKN$^+$92] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L. Finkelstein, and Michael Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

[Fle06] Franck Fleurey. *Language et methode pour une ingenierie des modeles fiables.* PhD thesis, University of Rennes, 2006.

[FP96] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* International Thompson Computer Press, 1996.

[FvBK$^+$91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[GBR05] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling*, 4(4):386–398, 2005.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, 1975.

[GG93] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[GLM04] Stefania Gnesi, Diego Latella, and Mieke Massink. Formal test-case generation for UML statecharts. In *ICECCS*, pages 75–84, 2004.

[GLR$^+$02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 90–105, London, UK, 2002. Springer-Verlag.

[GN07] Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[GPVCLR04] Marcela Genero, Mario Piattini-Velthuis, José-Antonio Cruz-Lemus, and Luis Reynoso. Metrics for UML models. *UPGRADE - European Journal for the Informatics Professional*, 5(2):43–48, April 2004.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Har88] David Harel. On visual formalism. *Communications of the ACM*, 31(5):514–530, 1988.

[Hei98] Constance Heitmeyer. On the need for practical formal methods. In *5th Intern. Symposium Formal Techniques in RealTime and Real-Time Fault-Tolerant Systems (FTRTFT'98)*, pages 18–26. Springer Verlag, 1998.

[Hes06] Anders Hessel. *Model-based Test Case Selection and Generation for Real-time Systems*. It licentiate thesis, UPPSALA University, 2006.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.

[HJGP99] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: An extendible UML transformation framework. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 275, Washington, DC, USA, 1999. IEEE Computer Society.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[HO02] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A combination of specification techniques for processes, data and time. *Nordic Journal of Computing*, 9(4):301–334, 2002. appeared March 2003.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hoe06] Jochen Hoenicke. *Combination of Processes, Data, and Time.* PhD thesis, Universität Oldenburg, 2006.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[iec85] Analysis techniques for system reliability - procedure for failure mode and effects analysis (FMEA). std IEC 60812 ed. 1.0 b, 1985.

[IEE] IEE SA. http://www.iee.lu.

[ISO95] ISO/IEC. Information technology - software life cycle processes. International standard 12207:1995(E), ISO/IEC, August 1995.

[ISO96] ISO/IEC. Extended BNF. international standard 14977:1996(E), ISO/IEC, December 1996.

[ISO02] ISO. Z formal specification notation: Syntax, type system and semantics. international standard 13568:2002, ISO, 2002.

[ISO04] ISO/IEC. Information technology - process assessment (SPICE). international standard 15504:2004, ISO/IEC, 2004.

[ISO05] ISO/IEC. Meta object facility (MOF). international standard 19502:2005, ISO/IEC, 2005.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, Cambridge, MA, USA, March 2006. ISBN 0-262-10114-9.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Professional, 1992.

[JJ05] Claude Jard and Thierry Jeron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[Jon86] C. B. Jones. *Systematic software development using VDM.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1986.

[JSS00] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In *22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[JUn] JUnit. online. http://www.junit.org.

[JW96] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *Computer*, 29(4):21–22, 1996.

[KB02] H. Kim and C. Boldyreff. Developing software metrics applicable to UML models. In *6h ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002)*, 2002.

[KC99] Soon-Kyeong Kim and David Carrington. Formalizing the UML class diagram using Object-Z. In Robert France and Bernhard Rumpe, editors, *UML'99 - the Unified Modeling Language*, volume 1723 of *LNCS*, pages 83–98. Springer-Verlag, 1999.

[KC00] Soon-Kyeong Kim and David Carrington. A formal mapping between UML models and Object-Z specifications. In *ZB 2000: Formal Specification and Development in Z and B*, volume 1878 of *LNCS*, pages 2–21. Springer-Verlag, 2000.

[Ken02] Stuart Kent. Model driven engineering. In *IFM*, pages 286–298, 2002.

[KHDC99] Y.G. Kim, H.S. Hong, D.H.Bae, and S.D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings Software*, 146(4):187–192, 1999.

[KM08a] Pierre Kelsen and Qin Ma. A formal definition of the EP language. Laboratory for Advanced Software Systems technical report TR-LASSY-08-03, University of Luxembourg, May 2008.

[KM08b] Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In *ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008)*, volume 5301 of *LNCS*, pages 690–704. Springer, 2008.

[KMSJ03] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *SAT*, 2003.

[Kot99] Gerald Kotonya. Practical experience with viewpoint-oriented requirements specification. *Requirements Engineering*, 4(3):115–133, 1999.

[Kru95] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[KS96] G. Kotonya and I. Sommerville. Requirements engineering with viewpoints. *Software Engineering Journal*, 11(1):5–11, 1996.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Professional, April 2003. ISBN 0-321-19442-X.

[LdBMB04] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering tobias combinatorial test suites. In *FASE*, volume 2984 of *LNCS*, pages 281–294. Springer, 2004.

[Lei] Leirios - Smart testing. online. http://www.leirios.com.

[LHSC03] M. R. Lyu, Z. Huang, S. K. S. Sze, and X. Cai. An empirical study on testing and fault tolerance for software reliability engineering. In *Proc. of 14th Int. Symposium on Software Reliability Engineering (Nov. 2003)*, pages 119–130, 2003.

[Lin05] S. Linker. Ein UML profil für CSP-OZ-DC. technical report, Universität Oldenburg, 2005.

[LMM99a] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[LMM99b] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 465. Kluwer, B.V., 1999.

[LP99a] Johan Lilius and Ivan P Paltor. The production cell: An exercise in the formal verification of a UML model. Technical report, Turku Centre for Computer Science, 1999.

[LP99b] Johan Lilius and Iván Porres Paltor. The semantics of UML state machines. TUCS Technical Report No. 273, Turku Centre for Computer Science, June 1999.

[LP99c] Johan Lilius and Iván Porres Paltor. vUML: a tool for verifying UML models. TUCS Technical Report No. 272, Turku Centre for Computer Science, May 1999.

[LPB04] Levi Lucio, Luis Pedro, and Didier Buchs. A methodology and a framework for model-based testing. In *Rapid Integration of Software Engineering Techniques (RISE)*, volume 3475 of *LNCS*, pages 57–70. Springer Verlag, 2004.

[LPB05] Levi Lucio, Luis Pedro, and Didier Buchs. A test language for CO-OPN specifications. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 195–201, Washington, DC, USA, 2005. IEEE Computer Society.

[LQ99] Liuying Li and ZhiChang Qi. Test selection from UML statecharts. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 273–279. IEEE Computer Society, 1999.

[LS02] Hung Ledang and Jeanine Souquieres. Contributions for modelling UML statecharts in B. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 109–127. Springer-Verlag, 2002.

[Luc08] Levi Lucio. *SATEL: A Test Intention Language for Object-Oriented Specifications of Reactive Systems*. PhD thesis, University of Geneva, 2008.

[Mag] MagicDraw UML. online. http://www.magicdraw.com/.

[Mat08] Aditya P. Mathur. *Foundations of Software Testing*. Pearson Education, 2008.

[McM99] K. L. McMillan. Getting started with smv. Technical report, Cadence Berkeley Labs, Berkeley, CA, USA, March 1999.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *The 8th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences)*, 2005.

[MFV$^+$05] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jezequel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, 2005.

[MIS00] MISRA. Development guidelines for vehicle based software. ISO/TR 15497:2000, The Motor Industry Research Association, 2000.

[MJ02] Stephen J. Mellor and Marc J.Balcer. *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley Professional, 2002.

[MLL02] Huaikou Miao, Ling Liu, and Li Li. Formalizing UML models with Object-Z. In *Formal Methods and Software Engineering*, volume 2495 of *LNCS*, pages 523–534. Springer, 2002.

[MMZ$^+$01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC 2001)*, June 2001.

[Mor90] L. J. Morell. A theory of fault-based testing. *IEEE Trans. on Soft. Eng.*, 8(16):844–857, August 1990.

[MP05] J. A. McQuillan and J. F. Power. A survey of UML-based coverage criteria for software testing. Technical Report NUIM-CS-TR-2005-08, Department of Computer Science, NUI Maynooth, 2005.

[MP07] Jacqueline A. McQuillan and James F. Power. On the application of software metrics to UML models. In *Models in Software Engineering workshop at MoDELS 2006*, volume 4364 of *LNCS*, pages 217–226. Springer, 2007.

[MS01] John D. McGregor and David A. Sykes. *A practical guide to testing object-oriented software.* object technology series. Addison-Wesley Professional, Upper Saddle River, NJ, USA, 2001.

[MV07] Farida Mostefaoui and Julie Vachon. Verification of Aspect-UML models using Alloy. In *AOSD*, 2007.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, second edition, 1979.

[NE03] Niklas Sörensson Niklas Een. An extensible SAT-solver. In *SAT*, 2003.

[NHT08] National Highway Traffic Safety Administration NHTSA. Occupant crash protection. Standard No. 208, Federal Motor Vehicle Safety Standards and Regulations (FMVSS), August 2008.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Second International Conference on the Unified Modeling Language (UML99)*, volume 1723 of *LNCS*, pages 416–429. Springer, January 1999.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[Obj] Objecteering. online. http://www.objecteering.com/.

[OMG03a] OMG. MDA guide version 1.0.1. Document omg/2003-06-01, Object Management Group, June 2003.

[OMG03b] OMG. XML Metadata Interchange (XMI). Full Specification formal/03-05-02, Object Management Group, May 2003.

[OMG05] OMG. UML Testing Profile (UTP). Full Specification formal/05-07-07, Object Management Group, July 2005.

[OMG06] OMG. Object Constraint Language (OCL). Full Specification formal/06-05-01, Object Management Group, May 2006.

[OMG07a] OMG. Unified Modeling Language: Infrastructure (UML), v. 2.1.2. Full Specification formal/07-11-04, Object Management Group, 2007.

[OMG07b] OMG. Unified Modeling Language: Superstructure (UML), v. 2.1.2. Full Specification formal/07-11-02, Object Management Group, 2007.

[OMG08] OMG. Software & systems Process Engineering Meta-model (SPEM), v 2.0. Full Specification formal/08-04-01, Object Management Group, 2008.

[Pen01] Liang Peng. Formalization of UML using algebraic specifications. master thesis, Universiteit Brussel / Ecole des Mines de Nantes, 2001.

[PJ04] Simin Pickin and Jean-Marc Jézéquel. Using UML sequence diagrams as the basis for a formal test description language. In *Integrated Formal Methods*, volume 2999 of *LNCS*, pages 481–500. Springer Verlag, 2004.

[PJH$^+$01] Simon Pickin, Claude Jard, Thierry Heuillard, Jean-Marc Jézéquel, and Philippe Desfray. A UML-integrated test description language for component testing. In *Workshop of the pUML-Group held together with the UML2001 on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, pages 208–223, 2001.

[PJJ$^+$07] Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel, and Yves Le Traon. Test synthesis from UML models of distributed software. *IEEE Transactions on Software Engineering*, 4(3):252–268, April 2007.

[PP04] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. In Mauro Pezze, editor, *TACoS0́4: Test and Analysis of Component-based Systems*, 2004.

[PTV96] Amit Paradkar, K. C. Tai, and M. A. Vouk. Automatic test generation for predicates. In *The seventh international symposium on software reliability engineering (ISSRE'96)*, pages 66–75, 1996.

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[RC81] D. Richardson and L. Clarke. A partition analysis method to increase program reliability. In *5th International Conference on Software Engineering*, 1981.

[RDH03] Robby, , Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *ESEC/FSE'03.* ACM, September 2003.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints.* PhD thesis, Universität Bremen, 2002.

[Rie] Benoît Ries. The SESAME project website. online. http://lassy.uni.lu/projects/SESAME.

[ROT89] D. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 86–96. ACM Press, 1989.

[SAT] SAT4J. Bringing the power of SAT technology to the java platform. online. http://www.sat4j.org/.

[Sch06] Douglas C. Schmidt. Model-driven engineering. *Computer*, 39(2):25–31, February 2006.

[Sei03] Ed Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.

[sin] Sintaks web site. online. www.kermeta.org/sintaks.

[Smi00] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publisher, 2000.

[Som04] Ian Sommerville. *Software Engineering.* Addison-Wesley, seventh edition, 2004.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 298–312, London, UK, 2002. Springer-Verlag.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.

[Sys] Syspect. SYstem SPECification Tool. online. http://syspect.informatik.uni-oldenburg.de.

[TR03] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+ : defining and using domain-specific modeling languages and code generators. In *OOPSLA'03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93, New York, NY, USA, 2003. ACM Press.

[Tra00] I. Traoré. An outline of PVS semantics for UML statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108, 2000.

[Tri05] Leonard L. Tripp. Software engineering body of knowledge. Technical report, IEEE, 2005.

[UL07] M. Utting and B. Legeard. *Practical Model-based Testing.* Morgan Kaufmann, San Francisco, 2007.

[UPL06] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical Report ISSN 1170-487X, University of Waikato, April 2006.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, October 2000.

[Vec] Vector. Software + services for automotive engineering. online. http://www.vector.com.

[W3C99] W3C. XSL Transformation (XSLT) 1.0. Technical recommendation REC-xslt-19991116, World Wide Web Consortium, 1999. http://www.w3.org/TR/xslt.

[W3C08] W3C. Extensible Markup Language (XML) 1.0. Technical recommendation REC-xml-20081126, World Wide Web Consortium, 2008. http://www.w3.org/TR/xml/.

[WC80] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Trans. on Soft. Eng.*, SE-6:247–257, May 1980.

[ZDSD05] Justyna Zander, Zhen Ru Dai, Ina Schieferdecker, and George Din. From U2TP models to executable tests with TTCN-3: An approach to model driven testing. In F. Khendek and R. Dssouli, editors, *TestCom*, volume 3502 of *LNCS*, pages 289–303. Springer-Verlag, 2005.

[ZH04] C. Zhou and M.R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems.* Springer Verlag, 2004.

APPENDIX

A. KERMETA IMPLEMENTATION OF FOREST SEMANTICS

A.1 Main operation

A.1.1 Header of the KerMeta program

```
@mainClass "FOREST_semantics::Main"
@mainOperation "main"
package FOREST_semantics;
require kermeta
//EMF Profiles
require "platform:/resource/SESAME_PHD/metamodels_emf/Ecore.profile.ecore"
//SESAME Profiles
require "platform:/resource/SESAME_PHD/metamodels_sesame/FOREST.ecore"
require "platform:/resource/SESAME_PHD/metamodels_sesame/ALLOY.ecore"
using kermeta::persistence
using kermeta::standard
using FOREST
using ALLOY
```

A.1.2 Opening of the input files

```
class Main {
  operation main() : Void is do
    // Input class
    var mf : ForestModel
    var ma: AlloyModel init AlloyModel.new
    // Create the repository, then the resource
    var repositoryIn : EMFRepository init EMFRepository.new
    var resourceIn : EMFResource
    //VOCS Initial Model
    resourceIn ?= repositoryIn.createResource("../models/vocs/InitialModel.xmi",
     "http://FOREST")
    resourceIn.load
    // Load the model (we get the instance)
    mf ?= resourceIn.instances.one
```

A.1.3 Calling the model transformations

```
    import_declaration(ma)
    metamodel_structural(ma)
    metamodel_behavioral(ma)
    static_view(mf, ma)
    dynamic_view(mf, ma)
    constraints(mf, ma)
```

A.1.4 Saving the output file

```
    var repositoryOut : EMFRepository init EMFRepository.new
    //VOCS initial semantic model
    var resourceOut : Resource init repositoryOut.createResource
      ("../models/vocs/AlloyInitialModel.xmi" , "http://ALLOY")
    resourceOut.instances.add(ma)
    resourceOut.save
  end
```

A.2 Import declarations

```
operation import_declaration(ma: AlloyModel) is do
  var importBool : ALLOY::Import init ALLOY::Import.new
  var importOrder : ALLOY::Import init ALLOY::Import.new
  importBool.importExp := "util/boolean"
  importOrder.importExp := "util/ordering[Status] as statusSeq"
  ma.imports.add(importBool)
  ma.imports.add(importOrder)
end
```

A.3 Metamodel: structural concepts

```
operation metamodel_structural(ma: AlloyModel) is do
  createSig(ma, "SUTComp", true, false, void)
  createSig(ma, "TSComp", true, false, void)
  createSig(ma, "SUTEvent", true, false, void)
  createSig(ma, "TSEvent", true, false, void)
  createSig(ma, "PState", true, false, void)
  createSig(ma, "ps_final", false, true, "PState")

  var sigPTrans: ALLOY::Signature init createSig(ma, "PTransition",
      true, false, void)
  addNewFieldToSig(ma, sigPTrans, "source", ALLOY::Multiplicity.one, "PState")
  addNewFieldToSig(ma, sigPTrans, "target", ALLOY::Multiplicity.one, "PState")
  addNewFieldToSig(ma, sigPTrans, "trigger", ALLOY::Multiplicity.lone, "SUTEvent")
  createConstraintToSig(sigPTrans, "no trigger implies target = ps_final")
end
```

A.4 Metamodel: behavioral concepts

```
operation metamodel_behavioral(ma: AlloyModel) is do
  var sigStatus: ALLOY::Signature init createSig(ma, "Status", true, false, void)
  var sigStep: ALLOY::Signature init createSig(ma, "Step", true, false, void)
  addNewFieldToSig(ma, sigStatus, "curPState", ALLOY::Multiplicity.one, "PState")
  addNewFieldToSig(ma, sigStatus, "sut", ALLOY::Multiplicity.one, "SUTComp")
  addNewFieldToSig(ma, sigStatus, "ts", ALLOY::Multiplicity.one, "TSComp")
  addNewFieldToSig(ma, sigStatus, "nextStep", ALLOY::Multiplicity.one, "Step")
  sigStatus.sigConstraint.add(createConstraint("(curPState = ps_final) iff
      ( no nextStep.selectedPTransition )"))
  sigStatus.sigConstraint.add(createConstraint("(curPState = ps_final) iff
      ( no nextStep.eventReceived )"))
  addNewFieldToSig(ma, sigStep, "selectedPTransition",
     ALLOY::Multiplicity.lone, "PTransition")
  addNewFieldToSig(ma, sigStep, "eventReceived", ALLOY::Multiplicity.lone,
     "SUTEvent")
  addNewFieldToSig(ma, sigStep, "eventSent", ALLOY::Multiplicity.lone, "TSEvent")
```

```
    var factStepCons : String init "all status : Status - statusSeq/last
          | let status' = statusSeq/next[status] {"
    factStepCons.append("(status.curPState != ps_final) implies (")
    factStepCons.append("status.nextStep.selectedPTransition in
       enabledTransitions[status] and ")
    factStepCons.append("status'.curPState = status.nextStep.selectedPTransition
       .target and ")
    factStepCons.append("postcond[status,status'] ) and ")
    factStepCons.append("(status.curPState = ps_final) implies
       UnchangedStatus[status, status'] }")
    createFact(ma, "step", createConstraint(factStepCons))

    var factTransCons : String init "all t: PTransition |
       some status : Status | status.nextStep.curPTransition = t"
    createFact(ma, "noUnusedTransitions", createConstraint(factTransCons))

    var factEventsCons : String init "all evt : SUTEvent |
        some status : Status | status.nextStep.eventReceived = evt"
    var factNoUnusedEvents: ALLOY::Fact init createFact(ma, "noUnusedEvents",
        createConstraint(factEventsCons))
    factNoUnusedEvents.factConstraint.add(createConstraint("all evt : TSEvent |
        some status : Status | status.nextStep.eventSent = evt"))

    var predUnchangedStatus: ALLOY::Predicate init ALLOY::Predicate.new
    predUnchangedStatus.predName := "UnchangedStatus"
    predUnchangedStatus.predArgs.add(createArgument(ma, "status",
             ALLOY::Multiplicity.one,"Status"))
    predUnchangedStatus.predArgs.add(createArgument(ma, "status'",
             ALLOY::Multiplicity.one,"Status"))
    predUnchangedStatus.predConstraint.add(createConstraint("status'.curPState =
             status.curPState"))
    predUnchangedStatus.predConstraint.add(createConstraint("status'.sut = status.sut"))
    predUnchangedStatus.predConstraint.add(createConstraint("status'.ts = status.ts"))
    predUnchangedStatus.predConstraint.add(createConstraint("status'.nextStep
            = status.nextStep"))

    var funEnabledTrans: ALLOY::Function init ALLOY::Function.new
    funEnabledTrans.funName := "enabledTransitions"
    funEnabledTrans.funArgs.add(createArgument
        (ma, "status", ALLOY::Multiplicity.one, "Status"))
    funEnabledTrans.funReturnType := getSigInModel(ma, "PTransition")
    funEnabledTrans.funReturnMult := ALLOY::Multiplicity.~set
    funEnabledTrans.funConstraint.add(
       createConstraint("{ t: PTransition | { (t.source = status.curPState)
           and (status.nextStep.eventReceived = t.trigger)
           and precond[status, t] } }"))
    ma.functions.add(funEnabledTrans)
  end
```

A.5 Model: static view

```
operation static_view(mf : ForestModel, ma: AlloyModel) is do
  static_view_datatypes(mf, ma)
  static_view_sut_ts_components(mf, ma)
  static_view_event_receptions(mf, ma)
end
```

A.5.1 SUT and TS components

```
operation static_view_sut_ts_components(mf: ForestModel, ma: AlloyModel)
     is do
  var sigSut : ALLOY::Signature init createSig(ma, mf.sut.sutName,
      false, false, "SUTComp")
  var sigTs : ALLOY::Signature init createSig(ma, mf.ts.tsName, false,
      false, "TSComp")

  mf.sut.sutVariables.each{sv |
    addNewFieldToSig(ma, sigSut, sv.attName, ALLOY::Multiplicity.lone,
         sv.attType.typeName)
  }
  mf.ts.testVariables.each{sv |
    addNewFieldToSig(ma, sigTs, sv.attName, ALLOY::Multiplicity.lone,
         sv.attType.typeName)
  }
end
```

A.5.2 Event receptions

```
operation static_view_event_receptions(mf : ForestModel, ma: AlloyModel) is do
  var factDistinctEvents : ALLOY::Fact init ALLOY::Fact.new
  factDistinctEvents.factName := "distinctEvents"

  mf.sut.eventsReceived.each{ fEVT |
    var sig : ALLOY::Signature init createSig(ma, fEVT.evtName, false,
         fEVT.eventParameters.size.equals(0), "SUTEvent")
    var cons : ALLOY::Constraint init ALLOY::Constraint.new
    cons.consExp := "no disj evt, evt' : " + fEVT.evtName + " | "
    fEVT.eventParameters.each{ param |
      cons.consExp.append("evt." + param.paramName + " = evt'."
           + param.paramName + " and true")
      addNewFieldToSig(ma, sig, param.paramName, ALLOY::Multiplicity.one,
           param.paramType.typeName)
    }
    factDistinctEvents.factConstraint.add(cons)
  }
  mf.ts.eventsReceived.each{ fEVT |
    var sig : ALLOY::Signature init createSig(ma, fEVT.evtName, false,
          fEVT.eventParameters.size.equals(0), "TSEvent")
    var cons : ALLOY::Constraint init ALLOY::Constraint.new
    cons.consExp := "no disj evt, evt' : " + fEVT.evtName + " | "
    fEVT.eventParameters.each{ param |
      cons.consExp.append("evt." + param.paramName + " = evt'."
          + param.paramName + " and true")
      addNewFieldToSig(ma, sig, param.paramName, ALLOY::Multiplicity.one,
          param.paramType.typeName)
    }
    factDistinctEvents.factConstraint.add(cons)
  }
  ma.facts.add(factDistinctEvents)
end
```

A.5.3 Data types

```
  operation static_view_datatypes(mf: ForestModel, ma: AlloyModel) is do
    mf.datatypes.each{fDT |
      if fDT.typeName.equals("Integer") then
        fDT.typeName := "Int"
      end
    }
    mf.datatypes.each{fDT |
      if fDT.typeName.equals("Boolean") then
        fDT.typeName := "Bool"
      end
    }
    mf.datatypes.each{fDT |
      if fDT.isEnum then
        createSig(ma, fDT.typeName, true, false, void)
        fDT.attributes.each{ att |
          createSig(ma, att.attName, false, true, fDT.typeName)
        }
      else
        var sig : ALLOY::Signature init void
        if fDT.extends.isVoid then
          sig := createSig(ma, fDT.typeName, fDT.isAbstract, fDT.isLeaf, void)
        else
          sig := createSig(ma, fDT.typeName, fDT.isAbstract, fDT.isLeaf,
              fDT.extends.typeName)
        end
      end
    }
    mf.datatypes.each{fDT |
      fDT.attributes.each{ att |
        var mult : ALLOY::Multiplicity init ALLOY::Multiplicity.~seq
        if att.lowerMultiplicity.equals("0") and att.upperMultiplicity.equals("1") then
          mult := ALLOY::Multiplicity.lone
        end
        if att.lowerMultiplicity.equals("1") and att.upperMultiplicity.equals("1") then
          mult := ALLOY::Multiplicity.one
        end
        if att.lowerMultiplicity.equals("1") and not att.upperMultiplicity.equals("0")
               and not att.upperMultiplicity.equals("1") then
          mult := ALLOY::Multiplicity.some
        end
        addNewFieldToSig(ma, getSigInModel(ma, fDT.typeName), att.attName,
             mult, att.attType.typeName)
      }
    }
  end
```

A.6 Model: dynamic view

```
operation dynamic_view(mf : ForestModel, ma: AlloyModel) is do
  dynamic_view_protocol_states(mf, ma)
  dynamic_view_protocol_transitions(mf, ma)
end
```

A.6.1 Protocol states

```
operation dynamic_view_protocol_states(mf : ForestModel, ma: AlloyModel) is do
  mf.sut.pstates.each{ ps |
    createSig(ma, "ps_" + ps.stateName, false, true, "PState")
  }
end
```

A.6.2 Protocol transitions

```
operation dynamic_view_protocol_transitions(mf : ForestModel, ma: AlloyModel) is do
  mf.sut.ptransitions.each{ pt |
    var sig : ALLOY::Signature init createSig(ma, "pt_" + pt.transName, false,
         getSigInModel(ma,pt.trigger.evtName).isSingleton, "PTransition")
    sig.sigConstraint.add(createConstraint("source = " +
         getSigInModel(ma, "ps_" + pt.source.stateName).sigName))
    sig.sigConstraint.add(createConstraint("trigger in " +
         getSigInModel(ma, pt.trigger.evtName).sigName))
    sig.sigConstraint.add(createConstraint("target = " +
         getSigInModel(ma, "ps_" + pt.target.stateName).sigName))
  }
end
```

A.7 Constraints on the model

```
operation constraints(mf : ForestModel, ma: AlloyModel) is do
```

```
  event_type_constraints(mf, ma)
  data_type_constraints(mf, ma)
  constant_variables_constraints(mf, ma)
  initial_status_constraints(mf, ma)
  final_status_constraints(mf, ma)
  precondition_constraints(mf, ma)
  postcondition_constraints(mf, ma)
end
```

A.7.1 Event type constraints

```
operation event_type_constraints(mf: ForestModel, ma: AlloyModel) is do
  mf.sut.eventsReceived.each{ evt |
    if not evt.event.size().equals(0) then
      var fact : ALLOY::Fact init ALLOY::Fact.new
      evt.event.each{ evtCons |
        evtCons.exp := commonRules(evtCons.exp)
        fact.factConstraint.add(createConstraint("all evt : " + evt.evtName
             + " | " + explicitSUTEventtypeConstraint(evtCons.exp, evt)))
      }
      ma.facts.add(fact)
    end
  }
  mf.ts.eventsReceived.each{ evt |
    if not evt.event.size().equals(0) then
      var fact : ALLOY::Fact init ALLOY::Fact.new
      evt.event.each{ evtCons |
        evtCons.exp := commonRules(evtCons.exp)
        fact.factConstraint.add(createConstraint("all evt : " + evt.evtName
              + " | " + explicitTSEventtypeConstraint(evtCons.exp, evt)))
      }
      ma.facts.add(fact)
    end
  }
end
operation explicitSUTEventtypeConstraint(oclf: String, event : FOREST::SUTEvent)
         : String is do
  oclf := commonRules(oclf)
  oclf := explicitContextSUTEventParameters(oclf, "evt.", event)
  result := oclf
end
operation explicitTSEventtypeConstraint(oclf: String, event : FOREST::TSEvent)
          : String is do
  oclf := commonRules(oclf)
  oclf := explicitContextTSEventParameters(oclf, "evt.", event)
  result := oclf
end
operation explicitContextSUTEventParameters(oclf: String, prefix : String,
     evt : FOREST::SUTEvent) : String is do
  evt.eventParameters.each{param |
    if oclf.contains(param.paramName) then
      oclf := oclf.replace(param.paramName.toString, prefix
          + param.paramName.toString)
    end
  }
  result := oclf
end
operation explicitContextTSEventParameters(oclf: String, prefix : String,
         evt : FOREST::TSEvent) : String is do
  evt.eventParameters.each{param |
    if oclf.contains(param.paramName) then
      oclf := oclf.replace(param.paramName.toString, prefix
          + param.paramName.toString)
    end
  }
  result := oclf
end
```

A.7.2 Data type constraints

```
operation data_type_constraints(mf : ForestModel, ma: AlloyModel) is do
  mf.datatypes.each{ dt |
    dt.attributes.each{ att |
      if (not att.lowerMultiplicity.equals("0"))
        and (not att.lowerMultiplicity.equals("1")) then
        if att.lowerMultiplicity.equals(att.upperMultiplicity) then
          createFact(ma, void, createConstraint("all dt : " + dt.typeName
             + " | #dt." + att.attName + " = " + att.lowerMultiplicity))
        else
          createFact(ma, void, createConstraint("all dt : " + dt.typeName
             + " | #dt." + att.attName + " >= " + att.lowerMultiplicity))
          createFact(ma, void, createConstraint("all dt : " + dt.typeName
             + " | #dt." + att.attName + " <= " + att.upperMultiplicity))
        end
      end
    }
  }
  mf.datatypes.each{ dt |
    if not dt.datatype.size().equals(0) then
      var fact : ALLOY::Fact init ALLOY::Fact.new
      dt.datatype.each{ evtCons |
        fact.factConstraint.add(createConstraint("all dt : " + dt.typeName
            + " | " + explicitDatatypeConstraint(evtCons.exp, mf, dt)))
      }
      ma.facts.add(fact)
    end
  }
end
```

```
operation explicitDatatypeConstraint(oclf: String, mf : ForestModel,
              dt : FOREST::Datatype) : String is do
  oclf := commonRules(oclf)
  oclf := explicitContextDatatypeAttr(oclf, "dt.", dt)
  result := oclf
end
operation explicitContextDatatypeAttr(oclf: String, prefix : String,
          dt : FOREST::Datatype) : String is do
  dt.attributes.each{att|
    if oclf.contains(att.attName) then
      oclf := oclf.replace(att.attName.toString, prefix + att.attName.toString)
    end
  }
  result := oclf
end
```

A.7.3 Constant variables constraints

```
operation constant_variables_constraints(mf : ForestModel, ma: AlloyModel) is do
  mf.sut.sutVariables.each{ sv |
    if sv.isReadOnly then
      createFact(ma, void, createConstraint("all status : Status | status.sut."
              + sv.attName + " = " + sv.constant.exp))
    end
  }
  mf.ts.testVariables.each{ sv |
    if sv.isReadOnly then
      createFact(ma, void, createConstraint("all status : Status | status.ts."
          + sv.attName + " = " + sv.constant.exp))
    end
  }
end
```

A.7.4 Initial status constraints

```
operation initial_status_constraints(mf: ForestModel, ma: AlloyModel) is do
  var fact : ALLOY::Fact init createFact(ma, "initialStatus",
      createConstraint("status.statusSeq/first.curPState = ps_"
          + mf.sut.initialState.stateName))

  if not mf.initialValue.isVoid then
    fact.factConstraint.add(createConstraint(explicitInitialValueConstraint(mf)))
  end
end
operation explicitInitialValueConstraint(mf : ForestModel) : String is do
  var oclf : String init mf.initialValue.exp
  oclf := commonRules(oclf)
  oclf := explicitContextSUTStateVar(oclf, "statusSeq/first.sut.", mf)
  oclf := explicitContextTSStateVar(oclf, "statusSeq/first.ts.", mf)
  result := oclf
end
operation explicitContextSUTStateVar(oclf: String, prefix : String,
          mf : ForestModel) : String is do
  mf.sut.sutVariables.each{ sv |
    if oclf.contains(sv.attName) then
      oclf := oclf.replace(sv.attName.toString, prefix  + sv.attName.toString)
    end
  }
  result := oclf
end
operation explicitContextTSStateVar(oclf: String, prefix : String, mf : ForestModel)
         : String is do
  mf.ts.testVariables.each{ sv |
    if oclf.contains(sv.attName) then
      oclf := oclf.replace(sv.attName.toString, prefix + sv.attName.toString)
    end
  }
  result := oclf
end
```

A.7.5 Final status constraints

```
operation final_status_constraints(mf : ForestModel, ma: AlloyModel) is do
  var fact : ALLOY::Fact init createFact(ma, "finalStatus",
     createConstraint("status.statusSeq/last.curPState = ps_final"))
  fact.factConstraint.add(createConstraint(" no statusSeq/last.nextStep.eventSent "))
end
```

A.7.6 Precondition constraints

```
operation precondition_constraints(mf : ForestModel, ma: AlloyModel) is do
  var predPrecond : ALLOY::Predicate init ALLOY::Predicate.new
  predPrecond.predName := "precond"
  predPrecond.predArgs.add(createArgument(ma, "status", ALLOY::Multiplicity.one,
      "Status"))
  predPrecond.predArgs.add(createArgument(ma, "t", ALLOY::Multiplicity.one,
      "PTransition"))
  mf.sut.ptransitions.each {pt |
    predPrecond.predConstraint.add(createConstraint("(t in pt_" + pt.transName + ")
        implies pre_pt_" + pt.transName + "[status]"))
  }
  ma.predicates.add(predPrecond)
  mf.sut.ptransitions.each {pt |
    var pred : ALLOY::Predicate init ALLOY::Predicate.new
    pred.predName := "pre_pt_" + pt.transName
    pred.predArgs.add(createArgument(ma, "status", ALLOY::Multiplicity.one, "Status"))
    if not pt.precond.isVoid then
      pred.predConstraint.add(createConstraint(
         explicitPrecondConstraint(pt.precond.exp, mf, pt)))
    end
    ma.predicates.add(pred)
  }
end
operation explicitPrecondConstraint(oclf: String, mf : ForestModel,
           pt : FOREST::ProtocolTransition) : String is do
  oclf := commonRules(oclf)
  oclf := explicitContextSUTStateVar(oclf, "status.sut.", mf)
  oclf := explicitContextTSStateVar(oclf, "status.ts.", mf)
  oclf := explicitContextSUTEventParameters(oclf,
           "status.nextStep.eventReceived.", pt.trigger)
  result := oclf
end
```

A.7.7 Postcondition constraints

```
operation postcondition_constraints(mf : ForestModel, ma: AlloyModel) is do
  var predPostcond : ALLOY::Predicate init ALLOY::Predicate.new
  predPostcond.predName := "postcond"
  predPostcond.predArgs.add(createArgument(ma, "status", ALLOY::Multiplicity.one,
      "Status"))
  predPostcond.predArgs.add(createArgument(ma, "status'", ALLOY::Multiplicity.one,
      "Status"))
  mf.sut.ptransitions.each {pt |
    predPostcond.predConstraint.add(createConstraint("(status.nextStep-
        .selectedPTransition in pt_" + pt.transName + ") implies
          post_pt_" + pt.transName + "[status,status']"))
  }
  ma.predicates.add(predPostcond)
  mf.sut.ptransitions.each {pt |
    var pred : ALLOY::Predicate init ALLOY::Predicate.new
    pred.predName := "post_pt_" + pt.transName
    pred.predArgs.add(createArgument(ma, "status", ALLOY::Multiplicity.one,
        "Status"))
    pred.predArgs.add(createArgument(ma, "status'", ALLOY::Multiplicity.one,
        "Status"))
    if not pt.postcond.isVoid then
      pred.predConstraint.add(createConstraint(
          explicitPostcondConstraint(pt.postcond.exp, mf, pt)))
    end
    ma.predicates.add(pred)
  }
end
operation explicitPostcondConstraint(oclf: String, mf : ForestModel,
            pt : FOREST::ProtocolTransition) : String is do
  oclf := commonRules(oclf)
```

```
    oclf := explicitContextSUTStateVar(oclf, "status.sut.", mf)
    oclf := explicitContextTSStateVar(oclf, "status.ts.", mf)
    oclf := explicitContextSUTEventParameters(oclf,
                "status.nextStep.eventReceived.", pt.trigger)
    oclf := translateEventSent(oclf, "status.nextStep.eventSent.", mf)
    result := oclf
  end
```

A.7.8 Common rules

```
  operation commonRules(oclf: String) : String is do
    oclf := oclf.replace("<>", "!=")
    oclf := translateNullAssignment(oclf)
    oclf := translateIfThenElse(oclf)
    oclf := translateBooleanAssignment(oclf)
    oclf := translatIsKindOf(oclf)
    result := oclf
  end
```

A.8 Miscellaneous operations

```
  operation getSigInModel(ma : AlloyModel, sigName: String) : ALLOY::Signature is do
    result:= ma.signatures.select{sig | sig.sigName.equals(sigName)}.one
  end
  operation addNewFieldToSig(ma : AlloyModel, sig: ALLOY::Signature,
      fieldName: String, fieldMult: ALLOY::Multiplicity, type: String) is do
    var field :  ALLOY::Field init  ALLOY::Field.new
    field.fieldName := fieldName
    field.fieldMult := fieldMult
    field.fieldType:= getSigInModel(ma, type)
    sig.fields.add(field)
  end
  operation createSig(ma : AlloyModel, sigName: String, isAbstract : Boolean,
          isSingleton : Boolean, extends: String) : ALLOY::Signature is do
    var sig : ALLOY::Signature init ALLOY::Signature.new
    sig.sigName := sigName
    sig.isAbstract := isAbstract
    sig.isSingleton := isSingleton
    if not extends.isVoid then
      sig.extends := getSigInModel(ma, extends)
    end
    ma.signatures.add(sig)
    result := sig
  end
  operation createConstraintToSig(sig: ALLOY::Signature, exp: String) is do
    var cons : ALLOY::Constraint init ALLOY::Constraint.new
    cons.consExp := exp
    sig.sigConstraint.add(cons)
```

```
end
operation createFact(ma : AlloyModel, factName: String, cons: ALLOY::Constraint)
       : ALLOY::Fact is do
  var fact : ALLOY::Fact init ALLOY::Fact.new
  fact.factConstraint.add(cons)
  if not factName.isVoid then
    fact.factName := factName
  end
  ma.facts.add(fact)
  result := fact
end
operation createConstraint(exp: String) : ALLOY::Constraint is do
  var cons : ALLOY::Constraint init ALLOY::Constraint.new
  cons.consExp := exp
  result := cons
end
operation createArgument(ma: AlloyModel, name : String, mult : ALLOY::Multiplicity,
         type: String) : ALLOY::Argument is do
  var arg : ALLOY::Argument init ALLOY::Argument.new
  arg.argName := name
  arg.argMult := mult
  arg.argType := getSigInModel(ma, type)
  result := arg
end
```

B. KERMETA IMPLEMENTATION OF SERELA SEMANTICS

B.1 Implementation of the main operation

```
@mainClass "TRANSFO_AM_TM::Main"
@mainOperation "main"
package TRANSFO_AM_TM;
require kermeta

//EMF Profiles
require "platform:/resource/SESAME_PHD/metamodels_emf/Ecore.profile.ecore"
//SESAME Profiles
require "platform:/resource/SESAME_PHD/metamodels_sesame/FOREST.ecore"
require "platform:/resource/SESAME_PHD/metamodels_sesame/SERELA.ecore"

using kermeta::persistence
using kermeta::standard
using FOREST
using SERELA

class Main {
operation main() is do
  //ATM files
  var inputModelFileName : String init "../models/atm/ATM_initial.forest.xmi"
  var testSelectionModelFileName : String init
      "../models/atm/ATM_test_selection.serela.xmi"
  var outputModelFileName : String init "../models/atm/ATM_constrained.forest.xmi"

  // Input class
  var analysisModel : ForestModel
  var testModel : ForestModel init ForestModel.new
  var testSelectionModel: TestSelectionModel
  // Create the repository, then the resource
  var repositoryIn : EMFRepository init EMFRepository.new
  var resourceIn : EMFResource
  resourceIn ?= repositoryIn.createResource(inputModelFileName,"http://FOREST")
  resourceIn.load
  // Load the model (we get the instance)
   analysisModel ?= resourceIn.instances.one
```

```
// Create the repository, then the resource
var repositoryIn2 : EMFRepository init EMFRepository.new
var resourceIn2 : EMFResource
resourceIn2 ?= repositoryIn2.createResource(testSelectionModelFileName,"http://SERELA")
resourceIn2.load
 // Load the model (we get the instance)
 testSelectionModel ?= resourceIn2.instances.one

//Final State and FinalTrans instructions are interpreted first
//because of their correlation with RepeatEvent, RepeatState and RepeatTrans
testSelectionModel.instructions.each{ t |
do
  if (t.isKindOf(FinalState)) then
    tiFinalState(t.asType(FinalState), analysisModel)
  end
  if (t.isKindOf(FinalTrans)) then
    tiFinalTrans(t.asType(FinalTrans), analysisModel)
  end
end
}

testSelectionModel.instructions.each{ t |
do
  if (t.isKindOf(SelectEvent)) then
    tiSelectEvent(t.asType(SelectEvent), analysisModel)
  end
  if (t.isKindOf(SelectTrans)) then
    tiSelectTrans(t.asType(SelectTrans), analysisModel)
  end
  if (t.isKindOf(SelectData)) then
    tiSelectData(t.asType(SelectData), analysisModel)
  end
  if (t.isKindOf(RepeatEvent)) then
    tiRepeatEvent(t.asType(RepeatEvent), analysisModel)
  end
  if (t.isKindOf(RepeatState)) then
    tiRepeatState(t.asType(RepeatState), analysisModel)
  end
  if (t.isKindOf(RepeatTrans)) then
    tiRepeatTrans(t.asType(RepeatTrans), analysisModel)
  end
  if (t.isKindOf(Scope)) then
    tiScope(t.asType(Scope), analysisModel)
  end
  if (t.isKindOf(InitialState)) then
    tiInitialState(t.asType(InitialState), analysisModel)
  end
  if (t.isKindOf(InitialVal)) then
    tiInitialVal(t.asType(InitialVal), analysisModel)
    end
  end
  }
  //save the model
  var repositoryOut : EMFRepository init EMFRepository.new
  var resourceOut : Resource init repositoryOut.createResource(outputModelFileName,
      "http://FOREST")
  resourceOut.instances.add(analysisModel)
  resourceOut.save
end
```

B.2 Implementation of the test selection instruction SelectEvent

```
operation tiSelectEvent(tsi: SelectEvent, model : ForestModel) is do
  model.sut.eventsReceived.each{ evt |
  do
    if (evt.evtName.equals(tsi.evtNameSE)) then
      var c : Constraint init Constraint.new
      c.exp := tsi.expSE
      evt.event.add(c)
    end
  end
  }
  model.ts.eventsReceived.each{ evt |
  do
    if (evt.evtName.equals(tsi.evtNameSE)) then
      var c : Constraint init Constraint.new
      c.exp := tsi.expSE
      evt.event.add(c)
    end
  end
  }
end
```

B.3 Implementation of the test selection instruction SelectTrans

```
operation tiSelectTrans(tsi: SelectTrans, model : ForestModel) is do
  model.sut.ptransitions.each{ tr |
  do
    if (tr.transName.equals(tsi.transNameST)) then
      if (not tr.precond.isVoid) then
        if (tr.precond.exp.equals("") or tr.precond.exp.equals("true")) then
          tr.precond.exp := tsi.expST
        else
```

```
          tr.precond.exp := tr.precond.exp + " and (" + tsi.expST + ")"
        end
      end
    end
  end
  }
end
```

B.4 Implementation of the test selection instruction SelectData

```
operation tiSelectData(tsi: SelectData, model : ForestModel) is do
  model.datatypes.each{ dt |
  do
    if (dt.typeName.equals(tsi.typeNameSD)) then
      var c : Constraint init Constraint.new
      c.exp := tsi.expSD
      dt.datatype.add(c)
    end
  end
  }
end
```

B.5 Implementation of the test selection instruction RepeatEvent

```
operation tiRepeatEvent(tsi: RepeatEvent, model : ForestModel) is do
  model.sut.eventsReceived.each{ evt |
  do
    if (evt.evtName.equals(tsi.eventNameRE)) then
      //step 2
      var svCtr : StateVariable init StateVariable.new
      svCtr.attName := "ctr_evt_" + tsi.eventNameRE
      svCtr.attType := getIntDataType(model)
      svCtr.isReadOnly := false
      model.ts.testVariables.add(svCtr)
      //step 3
      if (model.initialValue.isVoid) then
        model.initialValue := Constraint.new
        model.initialValue.exp := svCtr.attName + " = 0 "
      else //step 4
        model.initialValue.exp := model.initialValue.exp + " AND "
                                   + svCtr.attName + " = 0 "
      end
      //step 5
      model.sut.ptransitions.each{pt |
      do
        if (not pt.trigger.isVoid) then
        if (pt.trigger.evtName.equals(tsi.eventNameRE)) then
          if (pt.postcond.isVoid) then
            pt.postcond := Constraint.new
            pt.postcond.exp := svCtr.attName + " = " + svCtr.attName
                             + "@pre + 1"
          else
            pt.postcond.exp := pt.postcond.exp + " AND " + svCtr.attName
                             + " = " + svCtr.attName + "@pre + 1"
          end
        end
        end
      end
      }
      //step 6
      if (not tsi.minRE.isVoid) then
        //6.a
        var svMin : StateVariable init StateVariable.new
        svMin.attName := "MIN_evt_" + tsi.eventNameRE
        svMin.attType := getIntDataType(model)
        svMin.isReadOnly := true
        model.ts.testVariables.add(svMin)
        //6.b
        svMin.constant := Constraint.new
        svMin.constant.exp := tsi.minRE
        //6.c
        model.sut.ptransitions.each{pt2 |
        do
          if pt2.target.stateName.equals("final") then
            if (pt2.precond.isVoid) then
              pt2.precond := Constraint.new
              pt2.precond.exp := svCtr.attName + " >= " + svMin.attName
            else
              if (pt2.precond.exp.equals("") or pt2.precond.exp.equals("true")) then
                pt2.precond.exp := svCtr.attName + " >= " + svMin.attName
              else
                pt2.precond.exp := pt2.precond.exp + " AND (" + svCtr.attName
                                        + " >= " + svMin.attName + ") "
              end
            end
          end
        end
        }
      end
      //step 7
      if (not tsi.maxRE.isVoid) then
        //7.a
        var svMax : StateVariable init StateVariable.new
        svMax.attName := "MAX_evt_" + tsi.eventNameRE
        svMax.attType := getIntDataType(model)
        svMax.isReadOnly := true
        model.ts.testVariables.add(svMax)
        //7.b
        svMax.constant := Constraint.new
        svMax.constant.exp := tsi.maxRE
        //7.c
        model.sut.ptransitions.each{pt3 |
        do
          if (not pt3.trigger.isVoid) then
          if pt3.trigger.evtName.equals(tsi.eventNameRE) then
            if (pt3.precond.isVoid) then
              pt3.precond := Constraint.new
              pt3.precond.exp := svCtr.attName + " < " + svMax.attName
            else
              if (pt3.precond.exp.equals("") or pt3.precond.exp.equals("true")) then
                pt3.precond.exp := svCtr.attName + " < " + svMax.attName
              else
                pt3.precond.exp := pt3.precond.exp + " AND (" + svCtr.attName
                                         + " < " + svMax.attName + ") "
              end
            end
          end
          end
        end
        }
      end
    end
  end
  }
end
```

B.6 Implementation of the test selection instruction RepeatState

```
operation tiRepeatState(tsi: RepeatState, model : ForestModel) is do
  model.sut.pstates.each{ ps |
  do
    if (ps.stateName.equals(tsi.stateNameRS)) then
      //step 2
      var svCtr : StateVariable init StateVariable.new
      svCtr.attName := "ctr_ps_" + tsi.stateNameRS
      svCtr.attType := getIntDataType(model)
      svCtr.isReadOnly := false
      model.ts.testVariables.add(svCtr)
      //step 3
      if (model.initialValue.isVoid) then
        model.initialValue := Constraint.new
        model.initialValue.exp := svCtr.attName + " = 0 "
      else //step 4
        model.initialValue.exp := model.initialValue.exp + " AND " + svCtr.attName
                                     + " = 0 "
      end
      //step 5
      model.sut.ptransitions.each{pt |
      do
        if (pt.trigger.evtName.equals(tsi.stateNameRS)) then
          if (pt.postcond.isVoid) then
            pt.postcond := Constraint.new
            pt.postcond.exp := svCtr.attName + " = " + svCtr.attName + "@pre + 1"
          else
            pt.postcond.exp := pt.postcond.exp + " AND " + svCtr.attName
                                   + " = " + svCtr.attName + "@pre + 1"
          end
        end
      end
      }
      //step 6
      if (not tsi.minRS.isVoid) then
        //6.a
        var svMin : StateVariable init StateVariable.new
        svMin.attName := "MIN_ps_" + tsi.stateNameRS
        svMin.attType := getIntDataType(model)
        svMin.isReadOnly := true
        model.ts.testVariables.add(svMin)
        //6.b
        svMin.constant := Constraint.new
        svMin.constant.exp := tsi.minRS
        //6.c
        model.sut.ptransitions.each{pt2 |
        do
          if pt2.target.stateName.equals("final") then
            if (pt2.precond.isVoid) then
              pt2.precond := Constraint.new
              pt2.precond.exp := svCtr.attName + " >= " + svMin.attName
            else
              if (pt2.precond.exp.equals("") or pt2.precond.exp.equals("true")) then
                pt2.precond.exp := svCtr.attName + " >= " + svMin.attName
              else
                pt2.precond.exp := pt2.precond.exp + " AND (" + svCtr.attName
                                     + " >= " + svMin.attName + ") "
              end
            end
          end
        end
        }
      end
      //step 7
      if (not tsi.maxRS.isVoid) then
        //7.a
        var svMax : StateVariable init StateVariable.new
        svMax.attName := "MAX_ps_" + tsi.stateNameRS
        svMax.attType := getIntDataType(model)
        svMax.isReadOnly := true
        model.ts.testVariables.add(svMax)
        //7.b
        svMax.constant := Constraint.new
        svMax.constant.exp := tsi.maxRS
        //7.c
        model.sut.ptransitions.each{pt3 |
        do
          if pt3.trigger.evtName.equals(tsi.stateNameRS) then
            if (pt3.precond.isVoid) then
              pt3.precond := Constraint.new
              pt3.precond.exp := svCtr.attName + " < " + svMax.attName
            else
              if (pt3.precond.exp.equals("") or pt3.precond.exp.equals("true")) then
                pt3.precond.exp := svCtr.attName + " < " + svMax.attName
              else
                pt3.precond.exp := pt3.precond.exp + " AND (" + svCtr.attName
                                      + " < " + svMax.attName + ") "
              end
```

```
            end
          end
        end
        }
      end
    end
  end
  }
end
```

B.7 Implementation of the test selection instruction RepeatTrans

```
operation tiRepeatTrans(tsi: RepeatTrans, model : ForestModel) is do
  model.sut.ptransitions.each{ pt |
  do
    if (pt.transName.equals(tsi.transNameRT)) then
      //step 2
      var svCtr : StateVariable init StateVariable.new
      svCtr.attName := "ctr_pt_" + tsi.transNameRT
      svCtr.attType := getIntDataType(model)
      svCtr.isReadOnly := false
      model.ts.testVariables.add(svCtr)
      //step 3
      if (model.initialValue.isVoid) then
        model.initialValue := Constraint.new
        model.initialValue.exp := svCtr.attName + " = 0 "
      else //step 4
        model.initialValue.exp := model.initialValue.exp + " AND " + svCtr.attName
                                         + " = 0 "
      end
      //step 5
      model.sut.ptransitions.each{pt |
      do
        if (pt.trigger.evtName.equals(tsi.transNameRT)) then
          if (pt.postcond.isVoid) then
            pt.postcond := Constraint.new
            pt.postcond.exp := svCtr.attName + " = " + svCtr.attName + "@pre + 1"
          else
            pt.postcond.exp := pt.postcond.exp + " AND " + svCtr.attName
                                     + " = " + svCtr.attName + "@pre + 1"
          end
        end
      end
      }
      //step 6
      if (not tsi.minRT.isVoid) then
        //6.a
        var svMin : StateVariable init StateVariable.new
        svMin.attName := "MIN_pt_" + tsi.transNameRT
        svMin.attType := getIntDataType(model)
        svMin.isReadOnly := true
        model.ts.testVariables.add(svMin)
        //6.b
        svMin.constant := Constraint.new
        svMin.constant.exp := tsi.minRT
        //6.c
        model.sut.ptransitions.each{pt2 |
        do
          if pt2.target.stateName.equals("final") then
            if (pt2.precond.isVoid) then
              pt2.precond := Constraint.new
              pt2.precond.exp := svCtr.attName + " >= " + svMin.attName
            else
              if (pt2.precond.exp.equals("") or pt2.precond.exp.equals("true")) then
                pt2.precond.exp := svCtr.attName + " >= " + svMin.attName
              else
                pt2.precond.exp := pt2.precond.exp + " AND (" + svCtr.attName
                                        + " >= " + svMin.attName + ") "
              end
            end
          end
        end
        }
      end
      //step 7
      if (not tsi.maxRT.isVoid) then
        //7.a
        var svMax : StateVariable init StateVariable.new
        svMax.attName := "MAX_pt_" + tsi.transNameRT
        svMax.attType := getIntDataType(model)
        svMax.isReadOnly := true
        model.ts.testVariables.add(svMax)
        //7.b
        svMax.constant := Constraint.new
        svMax.constant.exp := tsi.maxRT
        //7.c
        model.sut.ptransitions.each{pt3 |
        do
          if pt3.trigger.evtName.equals(tsi.transNameRT) then
            if (pt3.precond.isVoid) then
              pt3.precond := Constraint.new
              pt3.precond.exp := svCtr.attName + " < " + svMax.attName
            else
              if (pt3.precond.exp.equals("") or pt3.precond.exp.equals("true")) then
                pt3.precond.exp := svCtr.attName + " < " + svMax.attName
              else
```

```
                pt3.precond.exp := pt3.precond.exp + " AND (" + svCtr.attName
                                         + " < " + svMax.attName + ") "
              end
            end
          end
        end
        }
      end
    end
  end
  }
end
```

B.8 Implementation of the test selection instruction Scope

```
operation tiScope(tsi: Scope, model : ForestModel) is do
  var scope : StateVariable init StateVariable.new
  scope.attName := "scope"
  scope.attType := getIntDataType(model)
  scope.isReadOnly := true
  scope.constant := Constraint.new
  scope.constant.exp := tsi.scope
  model.ts.testVariables.add(scope)
end
```

B.9 Implementation of the test selection instruction InitialState

```
operation tiInitialState(tsi: InitialState, model : ForestModel) is do
  model.sut.pstates.each{ps |
  do
    if (ps.stateName.equals(tsi.stateNameIS)) then
      model.sut.initialState := ps
    end
  end
  }
end
```

B.10 Implementation of the test selection instruction InitialVal

```
operation tiInitialVal(tsi: InitialVal, model : ForestModel) is do
  if (model.initialValue.isVoid) then
    model.initialValue := Constraint.new
  end
  model.initialValue.exp := tsi.expIV
end
```

B.11 Implementation of the test selection instruction FinalTrans

```
operation tiFinalState(tsi: FinalState, model : ForestModel) is do
  model.sut.pstates.each{ps |
  do
    if (ps.stateName.equals(tsi.stateNameFS)) then
      var finalPS : ProtocolState
      //if no final state in the PSM then we create one
      if (model.sut.pstates.select{ps2 | ps2.stateName.equals("final")}.one.isVoid) then
        finalPS := ProtocolState.new
        finalPS.stateName := "final"
        model.sut.pstates.add(finalPS)
      else
        finalPS := model.sut.pstates.select{ps2 | ps2.stateName.equals("final")}.one
      end
      var pt : ProtocolTransition init ProtocolTransition.new
      pt.source := ps
      pt.target := finalPS
      pt.transName := tsi.finalTransNameFS
      model.sut.ptransitions.add(pt)
    end
  end
  }
end
```

B.12 Implementation of the test selection instruction FinalState

```
operation tiFinalTrans(tsi: FinalTrans, model : ForestModel) is do
  model.sut.ptransitions.each{pt |
  do
    if (pt.transName.equals(tsi.transNameFT)) then
      var finalPS : ProtocolState
      //if no final state in the PSM then we create one
      if (model.sut.pstates.select{ps2 | ps2.stateName.equals("final")}.one.isVoid) then
        finalPS := ProtocolState.new
        finalPS.stateName := "final"
        model.sut.pstates.add(finalPS)
      else
        finalPS := model.sut.pstates.select{ps2 | ps2.stateName.equals("final")}.one
      end
      pt.target := finalPS
    end
  end
  }
end
```

C. SEMANTICS OF THE INITIAL MODEL OF THE VOCS EXAMPLE IN ALLOY

C.1 Alloy file header declaration

```
open util/boolean
open util/ordering[Status] as statusSeq
```

C.2 Static view of the Analysis Model

```
//----- State variables of the SUT -----//
sig SUT extends SUTComp {
  auth : lone Authentication,
  lastClass : lone ClassifiedImg,
  secondLastClass: lone ClassifiedImg
}

fact noUnusedSUT{
  all s : SUT | some status : Status | status.sut = s
}

fact distinctSUT {
  no disj sut, sut’: SUT | {
    sut.auth = sut’.auth
    sut.lastClass = sut’.lastClass
    sut.secondLastClass = sut’.secondLastClass
  }
}

//----- State variables of the TS -----//
one sig TS extends TSComp {}

//----------- SUT EVENTS --------------//
sig authenticate extends SUTEvent {
  ad : one Authentication
}
one sig diagnosticReq extends SUTEvent {
  dp: one Diagnostic
}
sig image extends SUTEvent {
  imageCT : one Image
}
one sig ignitionOn extends SUTEvent {}
one sig ignitionOff extends SUTEvent {}
one sig wakeUp extends SUTEvent {}
one sig sleep extends SUTEvent {}
one sig crash extends SUTEvent {}
one sig timeoutSendClass extends SUTEvent {}
one sig timeoutImageReq extends SUTEvent {}

//----------- TS EVENTS --------------//
sig classification extends TSEvent {
  class: Class
}
sig diagnostic extends TSEvent {
  diag: Diagnostic
}
sig storeClassification extends TSEvent {
  ci: ClassifiedImg
}
one sig imageReq extends TSEvent {}

//----- constraints on events -----//
fact distinctEvents {
  no disj evt, evt’: image | evt.imageCT = evt’.imageCT
  no disj evt, evt’: authenticate | evt.ad = evt’.ad
  no disj evt, evt’: diagnostic | evt.diag = evt’.diag
  no disj evt, evt’: classification | evt.class = evt’.class
  no disj evt, evt’: storeClassification | evt.class = evt’.class
}
fact noUnusedSUTEvents {
  all evt: image | some status : Status | status.nextStep.eventReceived = evt
  all evt: authenticate | some status : Status | status.nextStep.eventReceived = evt
}
fact noUnusedTSEvents {
  all evt: classification | some status : Status | status.nextStep.eventSent = evt
  all evt: diagnostic | some status : Status | status.nextStep.eventSent = evt
  all evt: storeClassification| some status : Status | status.nextStep.eventSent = evt
}

//----------- DataTypes --------------//

abstract sig Authentication{}
one sig AuthValid, AuthInvalid extends Authentication{}

one sig Diagnostic {}

abstract sig PeriodicEvent{}
one sig Started, Stopped extends PeriodicEvent{}
```

```
sig ClassifiedImg {
  img : one Image,
  class : one Class
}

abstract sig Class{}
one sig Empty, ChildSeat, FifthFemale, Adult extends Class{}

sig Image{
  seat : one Seat,
  back : one Background,
  ooi : one OOI,
  po : set PO,
  focus : one Focus
}

abstract sig Focus{}
one sig Blurry, Sharp extends Focus{}

one sig PO {}

sig Background {
  illumination : one Illumination
}

abstract sig Illumination{}
one sig Bright, Dark, NormalIllumination extends Illumination{}

sig Seat {
  back : one Backrest,
  seat : one SittingPart
}

sig Backrest{
  pos : one BackrestPosition,
  materialBR : one MaterialRemission
}

abstract  sig BackrestPosition{}
one sig BRUpright, BRBended, BRLyingPos extends BackrestPosition{}

abstract sig MaterialRemission{}
one sig HighRemission, MediumRemission, LowRemission extends MaterialRemission{}

sig SittingPart {
  materialSP : one MaterialRemission,
  offset : one SeatOffset
}
```

```
abstract sig SeatOffset{}
one sig SeatInFront, SeatInBetween, SeatInTheBack extends SeatOffset{}


sig OOI {
  leg : seq Leg,
  torso :  one Torso,
  head : one Head
}

sig Leg {
  size : one LegSize,
  pos : one LegPosition,
  angle : one LegAngle
}

abstract sig LegSize{}
one sig ShortLeg, MediumLeg, LongLeg extends LegSize{}
abstract sig LegPosition{}
one sig LegBendedUp, LegBendedDown, LegStraigth extends LegPosition{}
abstract sig LegAngle{}
one sig LegForward, LegRight, LegLeft extends LegAngle{}

sig Torso {
  size : one TorsoSize,
  pos : one TorsoPosition,
  offset : one TorsoOffset
}
abstract sig TorsoPosition{}
one sig TorsoBended, TorsoUpright extends TorsoPosition{}
abstract sig TorsoSize{}
one sig SmallTorso, MediumTorso, LargeTorso extends TorsoSize{}
abstract sig TorsoOffset{}
one sig OffSeat, OnSeat extends TorsoOffset{}

sig Head{
  size : one HeadSize
}
abstract sig HeadSize{}
one sig SmallHead, MediumHead, BigHead extends HeadSize{}


fact noUnusedData {
  all dt: Leg | some dtp : OOI | dtp.leg[0] = dt or dtp.leg[1] = dt
  all dt: Torso | some dtp : OOI | dtp.torso = dt
  all dt: Head | some dtp : OOI | dtp.head = dt
  all dt: OOI | some dtp : Image | dtp.ooi = dt
  all dt: Seat | some dtp : Image | dtp.seat = dt
  all dt: Background | some dtp : Image | dtp.back = dt
  all dt: Backrest | some dtp : Seat | dtp.back = dt
  all dt: SittingPart | some dtp : Seat | dtp.seat = dt
  all dt: Image | some status : Status | status.nextStep.eventReceived in image
    and status.nextStep.eventReceived.imageCT=dt
  all dt: ClassifiedImg | some status : Status |
    (status.nextStep.eventSent in storeClassification
       and status.nextStep.eventSent.ci=dt)
     or (status.sut.lastClass = dt)
     or (status.sut.secondLastClass = dt)
}

fact distinctData {
  no disj dt, dt' : ClassifiedImg | dt.img = dt'.img and dt.class = dt'.class
  no disj dt, dt': Image | {
    dt.seat = dt'.seat
    dt.back = dt'.back
    dt.ooi = dt'.ooi
    dt.po = dt'.po
    dt.focus = dt'.focus
  }
  no disj dt, dt' : Background | dt.illumination = dt'.illumination
  no disj dt, dt' : Seat | dt.back=dt'.back and dt.seat=dt'.seat
  no disj dt, dt' : Backrest | dt.pos=dt'.pos and dt.materialBR=dt'.materialBR
  no disj dt, dt' : SittingPart | dt.materialSP=dt'.materialSP and dt.offset=dt'.offset
  no disj dt, dt' : OOI | {
    dt.leg=dt'.leg
    dt.torso=dt'.torso
    dt.head=dt'.head
  }
  no disj dt, dt' : Leg | {
    dt.size=dt'.size
    dt.pos=dt'.pos
    dt.angle=dt'.angle
  }
  no disj dt, dt' : Torso | {
    dt.size=dt'.size
    dt.pos=dt'.pos
    dt.offset=dt'.offset
  }
  no disj dt, dt' : Head | dt.size=dt'.size
}
```

C.3 Dynamic view of the Analysis Model

```
//----------- Protocol States -------------//
one sig ps_ES, ps_SL, ps_CM, ps_DM extends PState{}

//----------- Initial Status -------------//
fact initialStatus{
  statusSeq/first.curPState = ps_ES
no  statusSeq/first.sut.lastClass.class
no  statusSeq/first.sut.secondLastClass.class
}
//-------- State Variables InitialValue Constraints --------//

//----------- Final Status -------------//
fact finalStatus{
no statusSeq/last.nextStep.eventSent
}

//----------- Protocol Transitions ---------//
sig pt_tr1 extends PTransition{}{
  source = ps_ES
  trigger in ignitionOn
  target = ps_SL
}
sig pt_tr2 extends PTransition{}{
  source = ps_SL
  trigger in ignitionOff
  target = ps_ES
}
sig pt_tr3 extends PTransition{}{
  source = ps_CM
  trigger in ignitionOff
  target = ps_ES
}
sig pt_tr4 extends PTransition{}{
  source = ps_DM
  trigger in ignitionOff
  target = ps_ES
}
sig pt_tr5 extends PTransition{}{
  source = ps_SL
  trigger in wakeUp
  target = ps_CM
}
sig pt_tr6 extends PTransition{}{
  source = ps_CM
  trigger in sleep
  target = ps_SL
}
```

```
sig pt_tr7 extends PTransition{}{
  source = ps_DM
  trigger in sleep
  target = ps_SL
}
sig pt_tr8 extends PTransition{}{
  source = ps_CM
  trigger = crash
  target = ps_CM
}
sig pt_tr9 extends PTransition{}{
  source = ps_CM
  trigger = timeoutImageReq
  target = ps_CM
}
sig pt_tr10 extends PTransition{}{
  source = ps_CM
  trigger = timeoutSendClass
  target = ps_CM
}
sig pt_tr11 extends PTransition{}{
  source = ps_CM
  trigger = image
  target = ps_CM
}
sig pt_tr12 extends PTransition{}{
  source = ps_CM
  trigger = authenticate
  target = ps_DM
}
sig pt_tr13 extends PTransition{}{
  source = ps_DM
  trigger = diagnosticReq
  target = ps_DM
}
```

C.4 Constraints on the model

```
//-------- State Variables ConstantConstraints --------//


//-------- Event Parameter Constraints --------//


//-------- DataType Constraints --------//
fact {
 all dt : OOI | #dt.leg = 2
}
fact {
  all dt : ClassifiedImg | {
    dt.class in FifthFemale implies {
      dt.img.ooi.leg[0].size in ShortLeg
      dt.img.ooi.leg[1].size in ShortLeg
      dt.img.ooi.torso.size in SmallTorso
      dt.img.ooi.head.size in SmallHead
    }
  }
}
//----------- PRE-CONDITION Constraints -------------//
pred precond(status: Status, t: PTransition) {
    (t in pt_tr1) implies pre_pt_tr1[status]
    (t in pt_tr2) implies pre_pt_tr2[status]
    (t in pt_tr3) implies pre_pt_tr3[status]
    (t in pt_tr4) implies pre_pt_tr4[status]
    (t in pt_tr5) implies pre_pt_tr5[status]
    (t in pt_tr6) implies pre_pt_tr6[status]
    (t in pt_tr7) implies pre_pt_tr7[status]
    (t in pt_tr8) implies pre_pt_tr8[status]
    (t in pt_tr9) implies pre_pt_tr9[status]
    (t in pt_tr10) implies pre_pt_tr10[status]
    (t in pt_tr11) implies pre_pt_tr11[status]
    (t in pt_tr12) implies pre_pt_tr12[status]
    (t in pt_tr13) implies pre_pt_tr13[status]
}
pred pre_pt_tr1(status: Status){}
pred pre_pt_tr2(status: Status){}
pred pre_pt_tr3(status: Status){}
pred pre_pt_tr4(status: Status){}
pred pre_pt_tr5(status: Status){}
pred pre_pt_tr6(status: Status){}
pred pre_pt_tr7(status: Status){}
pred pre_pt_tr8(status: Status){}
pred pre_pt_tr9(status: Status){}
pred pre_pt_tr10(status: Status){}
pred pre_pt_tr11(status: Status){}
pred pre_pt_tr12(status: Status){
  status.nextStep.eventReceived.ad = status.sut.auth
}
pred pre_pt_tr13(status: Status){}

//----------- POST-CONDITIONS Constraints -------------//
pred postcond(status, status': Status) {
  let t = status.nextStep.selectedPTransition | {
    (t in pt_tr1) implies post_pt_tr1[status, status']
    (t in pt_tr2) implies post_pt_tr2[status, status']
    (t in pt_tr3) implies post_pt_tr3[status, status']
    (t in pt_tr4) implies post_pt_tr4[status, status']
    (t in pt_tr5) implies post_pt_tr5[status, status']
    (t in pt_tr6) implies post_pt_tr6[status, status']
    (t in pt_tr7) implies post_pt_tr7[status, status']
    (t in pt_tr8) implies post_pt_tr8[status, status']
    (t in pt_tr9) implies post_pt_tr9[status, status']
    (t in pt_tr10) implies post_pt_tr10[status, status']
    (t in pt_tr11) implies post_pt_tr11[status, status']
    (t in pt_tr12) implies post_pt_tr12[status, status']
    (t in pt_tr13) implies post_pt_tr13[status, status']
  }
}

pred post_pt_tr1(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
pred post_pt_tr2(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
pred post_pt_tr3(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
pred post_pt_tr4(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
pred post_pt_tr5(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
```

```
pred post_pt_tr6(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
pred post_pt_tr7(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
pred post_pt_tr8(status, status': Status){
  some status.nextStep.eventSent
  status.nextStep.eventSent in storeClassification
  status.nextStep.eventSent.ci = status.sut.lastClass
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
pred post_pt_tr9(status, status': Status){
  some status.nextStep.eventSent
  status.nextStep.eventSent in imageReq
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
pred post_pt_tr10(status, status': Status){
  some status.nextStep.eventSent
  status.nextStep.eventSent in classification
  status.nextStep.eventSent.class = status.sut.lastClass.class
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}

pred post_pt_tr11(status, status': Status){
  (status.sut.lastClass != status.sut.secondLastClass) implies
    (status.nextStep.eventSent in classification
     and status.nextStep.eventSent.class = status.sut.lastClass.class)
  status'.sut.secondLastClass = status.sut.lastClass
}

pred post_pt_tr12(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}

pred post_pt_tr13(status, status': Status){
  status.nextStep.eventSent in diagnostic
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
}
```

D. SEMANTICS OF A CONSTRAINED MODEL OF THE VOCS EXAMPLE IN ALLOY

D.1 Alloy file header declaration

```
open util/boolean
open util/ordering[Status] as statusSeq
```

D.2 Static view of the constrained model

```
//----- State variables of the SUT -----//
sig SUT extends SUTComp {
  auth : lone Authentication,
  lastClass : lone ClassifiedImg,
  secondLastClass: lone ClassifiedImg
}

fact noUnusedSUT{
  all s : SUT | some status : Status | status.sut = s
}

fact distinctSUT {
  no disj sut, sut’: SUT | {
    sut.auth = sut’.auth
    sut.lastClass = sut’.lastClass
    sut.secondLastClass = sut’.secondLastClass
  }
}

//----- State variables of the TS -----//
sig TS extends TSComp {
  ctr_evt_image : one Int,
  MIN_evt_image : one Int,
}

fact noUnusedTS{
  all t : TS | some status : Status | status.ts = t
}

fact distinctTS {
  no disj ts, ts': TS | {
    ts.ctr_evt_image = ts'.ctr_evt_image
    ts.MIN_evt_image = ts'.MIN_evt_image
  }
}

//----------- SUT EVENTS --------------//
sig authenticate extends SUTEvent {
  ad : one Authentication
}
one sig diagnosticReq extends SUTEvent {
  dp: one Diagnostic
}
sig image extends SUTEvent {
  imageCT : one Image
}
one sig ignitionOn extends SUTEvent {}
one sig ignitionOff extends SUTEvent {}
one sig wakeUp extends SUTEvent {}
one sig sleep extends SUTEvent {}
one sig crash extends SUTEvent {}
one sig timeoutSendClass extends SUTEvent {}
one sig timeoutImageReq extends SUTEvent {}

//----------- TS EVENTS --------------//
sig classification extends TSEvent {
  class: Class
}
sig diagnostic extends TSEvent {
  diag: Diagnostic
}
sig storeClassification extends TSEvent {
  ci: ClassifiedImg
}
one sig imageReq extends TSEvent {}

//----- constraints on events -----//
fact distinctEvents {
  no disj evt, evt': image | evt.imageCT = evt'.imageCT
  no disj evt, evt': authenticate | evt.ad = evt'.ad
  no disj evt, evt': diagnostic | evt.diag = evt'.diag
  no disj evt, evt': classification | evt.class = evt'.class
  no disj evt, evt': storeClassification | evt.class = evt'.class
}

fact noUnusedSUTEvents {
  all evt: image | some status : Status | status.nextStep.eventReceived = evt
  all evt: authenticate | some status : Status | status.nextStep.eventReceived = evt
}
```

```
fact noUnusedTSEvents {
  all evt: classification | some status : Status | status.nextStep.eventSent = evt
  all evt: diagnostic | some status : Status | status.nextStep.eventSent = evt
  all evt: storeClassification| some status : Status | status.nextStep.eventSent = evt
}

//----------- DataTypes --------------//

abstract sig Authentication{}
one sig AuthValid, AuthInvalid extends Authentication{}

one sig Diagnostic {}

abstract sig PeriodicEvent{}
one sig Started, Stopped extends PeriodicEvent{}

sig ClassifiedImg {
  img : one Image,
  class : one Class
}

abstract sig Class{}
one sig Empty, ChildSeat, FifthFemale, Adult extends Class{}

sig Image{
  seat : one Seat,
  back : one Background,
  ooi : one OOI,
  po : set PO,
  focus : one Focus
}

abstract sig Focus{}
one sig Blurry, Sharp extends Focus{}

one sig PO {}

sig Background {
  illumination : one Illumination
}

abstract sig Illumination{}
one sig Bright, Dark, NormalIllumination extends Illumination{}

sig Seat {
  back : one Backrest,
  seat : one SittingPart
}
```

```
sig Backrest{
  pos : one BackrestPosition,
  materialBR : one MaterialRemission
}

abstract  sig BackrestPosition{}
one sig BRUpright, BRBended, BRLyingPos extends BackrestPosition{}

abstract sig MaterialRemission{}
one sig HighRemission, MediumRemission, LowRemission extends MaterialRemission{}

sig SittingPart {
  materialSP : one MaterialRemission,
  offset : one SeatOffset
}

abstract sig SeatOffset{}
one sig SeatInFront, SeatInBetween, SeatInTheBack extends SeatOffset{}


sig OOI {
  leg : seq Leg,
  torso :  one Torso,
  head : one Head
}

sig Leg {
  size : one LegSize,
  pos : one LegPosition,
  angle : one LegAngle
}

abstract sig LegSize{}
one sig ShortLeg, MediumLeg, LongLeg extends LegSize{}
abstract sig LegPosition{}
one sig LegBendedUp, LegBendedDown, LegStraigth extends LegPosition{}
abstract sig LegAngle{}
one sig LegForward, LegRight, LegLeft extends LegAngle{}

sig Torso {
  size : one TorsoSize,
  pos : one TorsoPosition,
  offset : one TorsoOffset
}
abstract sig TorsoPosition{}
one sig TorsoBended, TorsoUpright extends TorsoPosition{}
abstract sig TorsoSize{}
one sig SmallTorso, MediumTorso, LargeTorso extends TorsoSize{}
abstract sig TorsoOffset{}
```

```
one sig OffSeat, OnSeat extends TorsoOffset{}

sig Head{
  size : one HeadSize
}
abstract sig HeadSize{}
one sig SmallHead, MediumHead, BigHead extends HeadSize{}


fact noUnusedData {
  all dt: Leg | some dtp : OOI | dtp.leg[0] = dt or dtp.leg[1] = dt
  all dt: Torso | some dtp : OOI | dtp.torso = dt
  all dt: Head | some dtp : OOI | dtp.head = dt
  all dt: OOI | some dtp : Image | dtp.ooi = dt
  all dt: Seat | some dtp : Image | dtp.seat = dt
  all dt: Background | some dtp : Image | dtp.back = dt
  all dt: Backrest | some dtp : Seat | dtp.back = dt
  all dt: SittingPart | some dtp : Seat | dtp.seat = dt
  all dt: Image | some status:Status |
      status.nextStep.eventReceived in image
        and status.nextStep.eventReceived.imageCT=dt
  all dt: ClassifiedImg | some status:Status |
      (status.nextStep.eventSent in storeClassification
            and status.nextStep.eventSent.ci=dt)
       or (status.sut.lastClass = dt)
       or (status.sut.secondLastClass = dt)
}

fact distinctData {
  no disj dt, dt’ : ClassifiedImg | dt.img = dt’.img and dt.class = dt’.class
  no disj dt, dt’: Image | {
    dt.seat = dt’.seat
    dt.back = dt’.back
    dt.ooi = dt’.ooi
    dt.po = dt’.po
    dt.focus = dt’.focus
  }
  no disj dt, dt’ : Background | dt.illumination = dt’.illumination
  no disj dt, dt’ : Seat | dt.back = dt’.back and dt.seat = dt’.seat
  no disj dt, dt’ : Backrest | dt.pos = dt’.pos and dt.materialBR = dt’.materialBR
  no disj dt, dt’ : SittingPart | dt.materialSP = dt’.materialSP
                                    and dt.offset = dt’.offset
  no disj dt, dt’ : OOI | {
    dt.leg = dt’.leg
    dt.torso = dt’.torso
    dt.head = dt’.head
  }
  no disj dt, dt’ : Leg | {
    dt.size = dt’.size
    dt.pos = dt'.pos
    dt.angle = dt'.angle
  }
  no disj dt, dt' : Torso | {
    dt.size = dt'.size
    dt.pos = dt'.pos
    dt.offset = dt'.offset
  }
  no disj dt, dt' : Head | dt.size = dt'.size
}
```

D.3 Dynamic view of the constrained model

```
//----------- Protocol States -------------//
one sig ps_ES, ps_SL, ps_CM, ps_DM extends PState{}

//----------- Initial Status -------------//
fact initialStatus{
  statusSeq/first.curPState = ps_CM
  statusSeq/first.ts.ctr_evt_image = 0
  no  statusSeq/first.sut.lastClass.class
  no  statusSeq/first.sut.secondLastClass.class
}
//-------- State Variables InitialValue Constraints --------//

//----------- Final Status -------------//
fact finalStatus{
  statusSeq/last.curPState = ps_final
no statusSeq/last.nextStep.eventSent
}

//----------- Protocol Transitions ---------//
sig pt_tr1 extends PTransition{}{
  source = ps_ES
  trigger in ignitionOn
  target = ps_SL
}
sig pt_tr2 extends PTransition{}{
  source = ps_SL
  trigger in ignitionOff
  target = ps_ES
}
sig pt_tr3 extends PTransition{}{
  source = ps_CM
  trigger in ignitionOff
  target = ps_ES
}
sig pt_tr4 extends PTransition{}{
  source = ps_DM
  trigger in ignitionOff
  target = ps_ES
}
sig pt_tr5 extends PTransition{}{
  source = ps_SL
  trigger in wakeUp
  target = ps_CM
}
sig pt_tr6 extends PTransition{}{
  source = ps_CM
  trigger in sleep
  target = ps_SL
}
sig pt_tr7 extends PTransition{}{
  source = ps_DM
  trigger in sleep
  target = ps_SL
}
sig pt_tr8 extends PTransition{}{
  source = ps_CM
  trigger = crash
  target = ps_CM
}
sig pt_tr9 extends PTransition{}{
  source = ps_CM
  trigger = timeoutImageReq
  target = ps_CM
}
sig pt_tr10 extends PTransition{}{
  source = ps_CM
  trigger = timeoutSendClass
  target = ps_CM
}
sig pt_tr11 extends PTransition{}{
  source = ps_CM
  trigger = image
  target = ps_CM
}
sig pt_tr12 extends PTransition{}{
  source = ps_CM
  trigger = authenticate
  target = ps_DM
}
sig pt_tr13 extends PTransition{}{
  source = ps_DM
  trigger = diagnosticReq
  target = ps_DM
}
sig pt_tr14 extends PTransition{}{
  source = ps_CM
  no trigger
  target = ps_final
}
```

D.4 Constraints of the constrained model

```
//-------- State Variables ConstantConstraints --------//
fact {all status : Status |  status.ts.MIN_evt_image = 1}
fact {all status : Status |  no status.sut.auth}

//-------- Event Parameter Constraints --------//


//-------- DataType Constraints --------//
fact {
 all dt : OOI | #dt.leg = 2
}

fact {
  all dt : ClassifiedImg | {
    dt.class in FifthFemale implies {
      dt.img.ooi.leg[0].size in ShortLeg
      dt.img.ooi.leg[1].size in ShortLeg
      dt.img.ooi.torso.size in SmallTorso
      dt.img.ooi.head.size in SmallHead
    }
  }
}
fact testSelection {
  all dt : ClassifiedImg | {
    dt.class in Adult or dt.class in FifthFemale
  }
  all dt : Image | {
    dt.seat.back.pos in BRUpright
    dt.seat.seat.offset in SeatInBetween
    dt.ooi.leg[0].pos in LegBendedDown
    dt.ooi.leg[0].angle in LegForward
    dt.ooi.leg[1].pos in LegBendedDown
    dt.ooi.leg[1].angle in LegForward
    dt.ooi.torso.pos in TorsoUpright
    dt.ooi.torso.offset in OnSeat
    no dt.po
//TODO delete
//    dt.ooi.leg[0].size in MediumLeg
//    dt.ooi.leg[1].size in MediumLeg
//    dt.ooi.head.size in MediumHead
//    dt.ooi.torso.size in MediumTorso
//    dt.seat.back.materialBR in MediumRemission
//    dt.seat.seat.materialSP in MediumRemission
//    dt.focus in Sharp
  }
}
```

```
//----------- PRE-CONDITION Constraints -------------//
pred precond(status: Status, t: PTransition) {
    (t in pt_tr1) implies pre_pt_tr1[status]
    (t in pt_tr2) implies pre_pt_tr2[status]
    (t in pt_tr3) implies pre_pt_tr3[status]
    (t in pt_tr4) implies pre_pt_tr4[status]
    (t in pt_tr5) implies pre_pt_tr5[status]
    (t in pt_tr6) implies pre_pt_tr6[status]
    (t in pt_tr7) implies pre_pt_tr7[status]
    (t in pt_tr8) implies pre_pt_tr8[status]
    (t in pt_tr9) implies pre_pt_tr9[status]
    (t in pt_tr10) implies pre_pt_tr10[status]
    (t in pt_tr11) implies pre_pt_tr11[status]
    (t in pt_tr12) implies pre_pt_tr12[status]
    (t in pt_tr13) implies pre_pt_tr13[status]
    (t in pt_tr14) implies pre_pt_tr14[status]
}
pred pre_pt_tr1(status: Status){}
pred pre_pt_tr2(status: Status){}
pred pre_pt_tr3(status: Status){}
pred pre_pt_tr4(status: Status){}
pred pre_pt_tr5(status: Status){}
pred pre_pt_tr6(status: Status){}
pred pre_pt_tr7(status: Status){}
pred pre_pt_tr8(status: Status){}
pred pre_pt_tr9(status: Status){}
pred pre_pt_tr10(status: Status){}
pred pre_pt_tr11(status: Status){}
pred pre_pt_tr12(status: Status){
  status.nextStep.eventReceived.ad = status.sut.auth
}
pred pre_pt_tr13(status: Status){}
pred pre_pt_tr14(status: Status){
status.ts.ctr_evt_image >= 1
}

//----------- POST-CONDITIONS Constraints -------------//
pred postcond(status, status': Status) {
  let t = status.nextStep.selectedPTransition | {
    (t in pt_tr1) implies post_pt_tr1[status, status']
    (t in pt_tr2) implies post_pt_tr2[status, status']
    (t in pt_tr3) implies post_pt_tr3[status, status']
    (t in pt_tr4) implies post_pt_tr4[status, status']
    (t in pt_tr5) implies post_pt_tr5[status, status']
    (t in pt_tr6) implies post_pt_tr6[status, status']
    (t in pt_tr7) implies post_pt_tr7[status, status']
    (t in pt_tr8) implies post_pt_tr8[status, status']
    (t in pt_tr9) implies post_pt_tr9[status, status']
    (t in pt_tr10) implies post_pt_tr10[status, status']
    (t in pt_tr11) implies post_pt_tr11[status, status’]
    (t in pt_tr12) implies post_pt_tr12[status, status’]
    (t in pt_tr13) implies post_pt_tr13[status, status’]
    (t in pt_tr14) implies post_pt_tr14[status, status’]
  }
}

pred post_pt_tr1(status, status’: Status){
  no status.nextStep.eventSent
  status’.sut.lastClass = status.sut.lastClass
  status’.sut.secondLastClass = status.sut.secondLastClass
  status’.ts.ctr_evt_image = status.ts.ctr_evt_image
}
pred post_pt_tr2(status, status’: Status){
  no status.nextStep.eventSent
  status’.sut.lastClass = status.sut.lastClass
  status’.sut.secondLastClass = status.sut.secondLastClass
  status’.ts.ctr_evt_image = status.ts.ctr_evt_image
}
pred post_pt_tr3(status, status’: Status){
  no status.nextStep.eventSent
  status’.sut.lastClass = status.sut.lastClass
  status’.sut.secondLastClass = status.sut.secondLastClass
  status’.ts.ctr_evt_image = status.ts.ctr_evt_image
}
pred post_pt_tr4(status, status’: Status){
  no status.nextStep.eventSent
  status’.sut.lastClass = status.sut.lastClass
  status’.sut.secondLastClass = status.sut.secondLastClass
  status’.ts.ctr_evt_image = status.ts.ctr_evt_image
}
pred post_pt_tr5(status, status’: Status){
  no status.nextStep.eventSent
  status’.sut.lastClass = status.sut.lastClass
  status’.sut.secondLastClass = status.sut.secondLastClass
  status’.ts.ctr_evt_image = status.ts.ctr_evt_image
}
pred post_pt_tr6(status, status’: Status){
  no status.nextStep.eventSent
  status’.sut.lastClass = status.sut.lastClass
  status’.sut.secondLastClass = status.sut.secondLastClass
  status’.ts.ctr_evt_image = status.ts.ctr_evt_image
}
pred post_pt_tr7(status, status’: Status){
  no status.nextStep.eventSent
  status’.sut.lastClass = status.sut.lastClass
  status’.sut.secondLastClass = status.sut.secondLastClass
  status’.ts.ctr_evt_image = status.ts.ctr_evt_image
}
```

```
pred post_pt_tr8(status, status': Status){
  some status.nextStep.eventSent
  status.nextStep.eventSent in storeClassification
  status.nextStep.eventSent.ci = status.sut.lastClass
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
  status'.ts.ctr_evt_image = status.ts.ctr_evt_image
}
pred post_pt_tr9(status, status': Status){
  some status.nextStep.eventSent
  status.nextStep.eventSent in imageReq
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
  status'.ts.ctr_evt_image = status.ts.ctr_evt_image
}
pred post_pt_tr10(status, status': Status){
  some status.nextStep.eventSent
  status.nextStep.eventSent in classification
  status.nextStep.eventSent.class = status.sut.lastClass.class
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
  status'.ts.ctr_evt_image = status.ts.ctr_evt_image
}

pred post_pt_tr11(status, status': Status){
  status'.ts.ctr_evt_image = status.ts.ctr_evt_image + 1
  (status.sut.lastClass != status.sut.secondLastClass)
   implies (some status.nextStep.eventSent
      and status.nextStep.eventSent in classification
      and status.nextStep.eventSent.class = status.sut.lastClass.class)
  (status.sut.lastClass = status.sut.secondLastClass)
   implies no status.nextStep.eventSent
  status'.sut.lastClass.img = status.nextStep.eventReceived.imageCT
  status'.sut.secondLastClass = status.sut.lastClass
}

pred post_pt_tr12(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
  status'.ts.ctr_evt_image = status.ts.ctr_evt_image
}

pred post_pt_tr13(status, status': Status){
  some status.nextStep.eventSent
  status.nextStep.eventSent in diagnostic
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
  status'.ts.ctr_evt_image = status.ts.ctr_evt_image
}

pred post_pt_tr14(status, status': Status){
  no status.nextStep.eventSent
  status'.sut.lastClass = status.sut.lastClass
  status'.sut.secondLastClass = status.sut.secondLastClass
  status'.ts.ctr_evt_image = status.ts.ctr_evt_image
}
```

E. SEMANTICS OF THE METRICS DEFINED WITHIN THE INDUSTRIAL CASE STUDY IN ALLOY

E.1 State coverage metrics

```
run {some status : Status | status.curPState = ps_ES} for 10 but 3 Status
run {some status : Status | status.curPState = ps_SL} for 10 but 3 Status
run {some status : Status | status.curPState = ps_CM} for 10 but 3 Status
run {some status : Status | status.curPState = ps_DM} for 10 but 3 Status
```

E.2 Event coverage metrics

```
run {some status : Status | status.nextStep.eventReceived in ignitionOn}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in ignitionOff}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in wakeUp}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in sleep}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in image}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in authenticate}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in diagnosticReq}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in crash}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in timeoutImageReq}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventReceived in timeoutSendClass}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventSent in diagnostic}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventSent in storeClassification}
  for 10 but 3 Status
run {some status : Status | status.nextStep.eventSent in imageReq}
  for 10 but 3 Status
```

```
run {some status : Status | status.nextStep.eventSent in classification}
  for 10 but 3 Status
```

E.3 Use-case coverage metrics

```
run {pathUC1} for 10 but 3 Status
run {stateVarUC2} for 10 but 3 Status
run {pathUC3} for 10 but 3 Status
run {pathUC4} for 10 but 3 Status

pred pathUC1 {
 some disj status1, status2, status3, status4, status5 : Status |
    status1.nextStep.selectedPTransition = pt_tr1
    and status2.nextStep.selectedPTransition = pt_tr5
    and status2 in statusSeq/nexts[status1]
    and status3.nextStep.selectedPTransition = pt_tr9
    and status3 in statusSeq/nexts[status2]
    and status4.nextStep.selectedPTransition = pt_tr11
    and status4 in statusSeq/nexts[status3]
    and status5.nextStep.selectedPTransition = pt_tr10
    and status5 in statusSeq/nexts[status4]
}

pred stateVarUC2 {
  some status : Status | status.sut.lastClass != status.sut.secondLastClass
}

pred pathUC3 {
 some status1 : Status |
    status1.nextStep.selectedPTransition = pt_tr8
}

pred pathUC4 {
 some disj status1, status2, status3 : Status |
    status1.nextStep.selectedPTransition = pt_tr8
    and status2.nextStep.selectedPTransition = pt_tr12
    and status2 in statusSeq/nexts[status1]
    and status3.nextStep.selectedPTransition = pt_tr13
    and status3 in statusSeq/nexts[status2]
}
```

E.4 Output classification data coverage metrics

```
run {coveredClass[Empty]} for 10 but 3 Status
run {coveredClass[ChildSeat]} for 10 but 3 Status
run {coveredClass[FifthFemale]} for 10 but 3 Status
run {coveredClass[Adult]} for 10 but 3 Status

pred coveredClass (c: Class){
  some status : Status |  some dt : ClassifiedImg |
    dt.class in c and (
      (status.nextStep.eventSent in storeClassification
            and status.nextStep.eventSent.ci=dt)
      or (status.sut.lastClass = dt)
      or (status.sut.secondLastClass = dt)
    )
}
```

E.5 VOCS project-specific metrics

```
run {posture1} for 10 but 3 Status
run {posture2} for 10 but 3 Status

pred posture1 {
  some status : Status |  status.nextStep.eventReceived in image
    and image.imageCT.seat.back.pos in BRUpright
    and image.imageCT.seat.seat.offset in SeatInBetween
    and image.imageCT.ooi.leg[0].pos in LegBendedDown
    and image.imageCT.ooi.leg[0].angle in LegForward
    and image.imageCT.ooi.leg[1].pos in LegBendedDown
    and image.imageCT.ooi.leg[1].angle in LegForward
    and image.imageCT.ooi.torso.pos in TorsoUpright
    and image.imageCT.ooi.torso.offset in OnSeat
    and no image.imageCT.po
}

pred posture2 {
  some status : Status |  status.nextStep.eventReceived in image
    and image.imageCT.seat.back.pos in BRUpright
    and image.imageCT.seat.seat.offset in SeatInBetween
    and image.imageCT.ooi.leg[0].pos in LegBendedDown
    and image.imageCT.ooi.leg[0].angle in LegForward
    and image.imageCT.ooi.leg[1].pos in LegBendedDown
    and image.imageCT.ooi.leg[1].angle in LegForward
    and image.imageCT.ooi.torso.pos in TorsoBended
    and image.imageCT.ooi.torso.offset in OnSeat
    and no image.imageCT.po
}
```

F. USE-CASES DESCRIBING THE REQUIREMENTS OF THE INDUSTRIAL CASE STUDY

F.1 Primary Use Case - Classification of occupant

Trigger: Power is applied

Primary actor: Airbag controller on CAN bus

Goal: Determine whether airbag should be enabled or not based on occupant classification

1.1 System is initialized by physical signal `12V Klemme 30b = ON`

1.2 Network Management is activated by receiving `NM_ACSM` message

1.3 Normal operation is started by receiving message `Klemmen` with `KL R/15 = ON` before `VSM_TMO_-SHUTDOWN_DELAY` (5 seconds)

1.4 Image is requested via SPI message to DARSS.

1.5 Image is received via SPI messages from DARSS.

1.6 Occupant is classified

1.7 If `640ms` since last send then, Send classification result `Status Uberwachung Sitzbelegung ID=$30E`

Variation - Occupant could be [empty seat, Rear Facing Child Seat, 5^{th} percentile female, bigger than 5^{th}'ile female, other object on seat]. Occupant may vary in position on seat. Occupant may be moving. Occupant may have varying clothes, hair color and size.

F.2 Extension Use Case - Change in classification of occupant

Precondition: Occupant is classified and is different from last classification result.

Primary actor: Airbag controller on CAN bus

Goal: Determine whether airbag should be enabled or not based on occupant classification

2.1 immediately Send classification result `Status Uberwachung Sitzbelegung ID=$30E`

F.3 Extension Use Case - Storage of data in event of crash

Trigger: Crash message sent from chassis module

Primary actor: Airbag controller on CAN bus

Goal: Store record of data that was used in the decision to fire or not fire the airbag

3.1 Crash message received via CAN `Crash-Telegram ID=$1FE`

3.2 Last classification result is stored in NVM

3.3 Last received image is stored in NVM

F.4 Extension Use Case - Retrieval of data after a crash

Trigger: Diagnostic request sent by diagnostic tool

Primary actor: Diagnostic technician

Goal: Retrieval of data that was used in the decision to fire or not fire the airbag

4.1 System is initialized by physical signal `12V Klemme 30b = ON` and power is applied

  4.1.1 Authenticate diagnostics implies diag mode.

4.2 Diagnostic request received via CAN to retrieve crash data.

4.3 Stored classification is sent via CAN.

4.4 Stored image is sent via CAN.

G. COMPLEMENTARY CLASS DIAGRAMS FOR THE INDUSTRIAL CASE STUDY

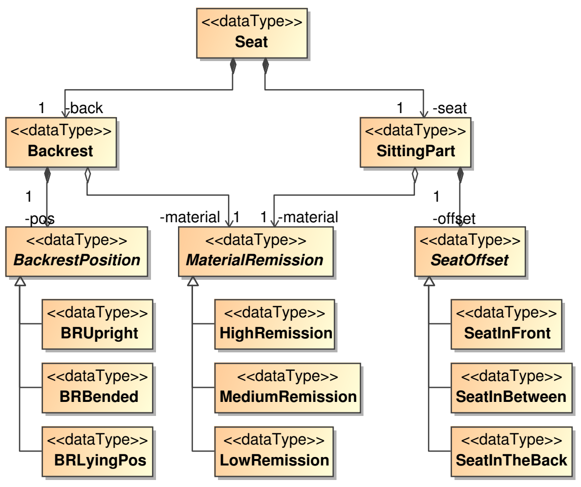


Fig. G.1: VOCS industrial case study: Seat data type definition

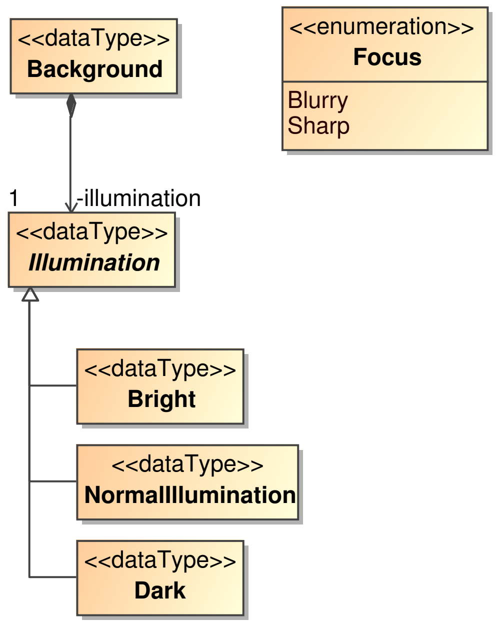


Fig. G.2: VOCS industrial case study: Background data type definition

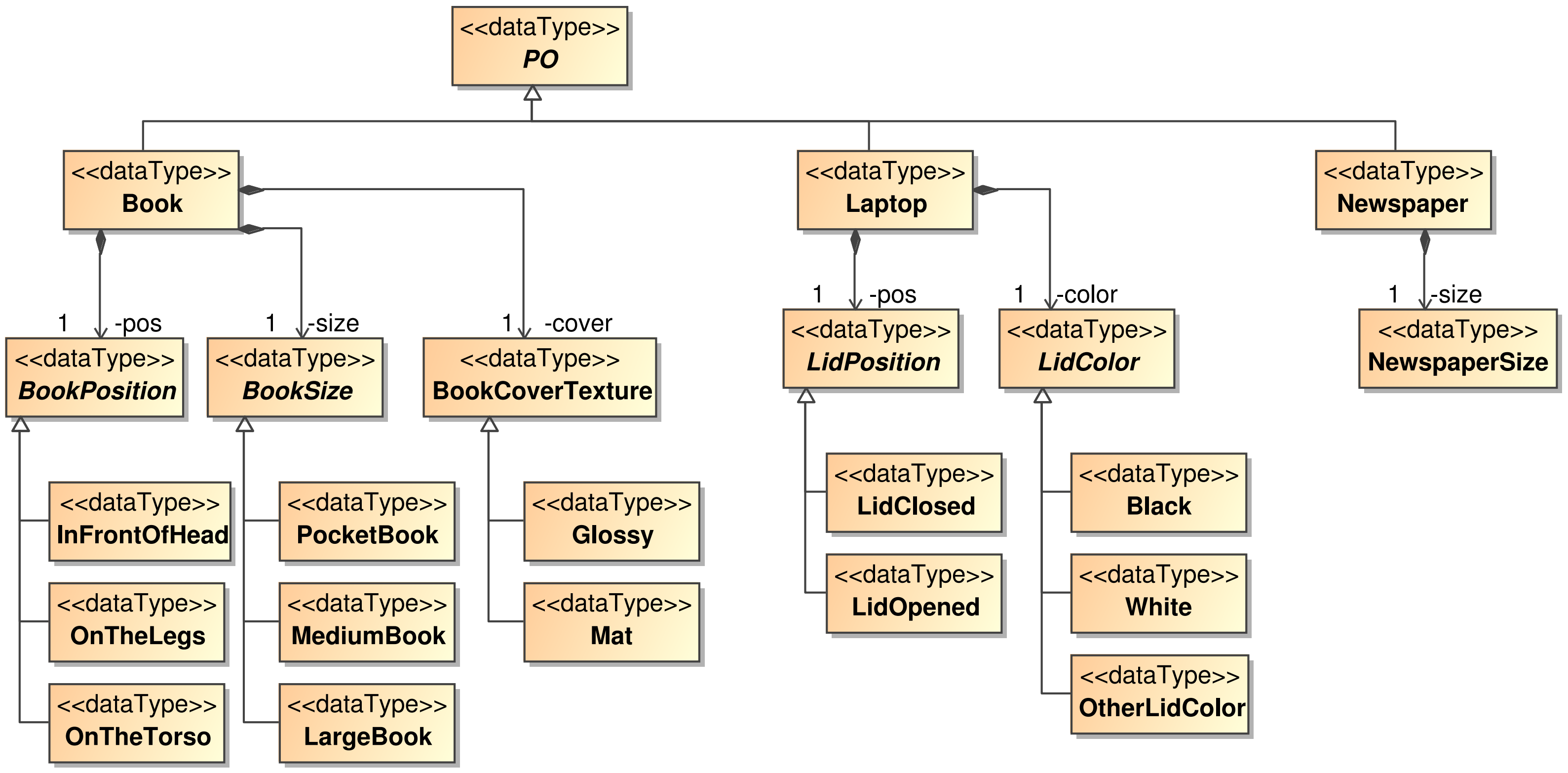


Fig. G.3: VOCS industrial case study: PO data type definition