Graz University of Technology
Faculty of Computer Science
Institute for Software Technology

Dissertation

On Model-based Test Case Generation from Communicating UML Models

by

Valentin Constantin Chimisliu

Supervisor/First reviewer: Univ.-Prof. Dr. Franz Wotawa
Second reviewer: Prof. Dr. Manuel Núñez

Graz, 2013

Abstract

As the complexity of software systems increases, testing becomes more and more important and expensive. Thus, the need to automate as much as possible of this task also arises. The academic community offers promising tools providing amongst others capabilities for test case generation. Usually such tools require personnel skilled in the field of formal methods. The lack of such personnel often hinders the adoption of the tools in an industrial environment. This work attempts to breach the gap between academic model-based testing tools and their usage in industry. This is achieved by allowing the specification of the desired behavior of the system in a widely accepted industry notation (UML statecharts) and then deriving a formal specification of the system by means of the LOTOS language via a behind the scene model transformation.

The present work proposes an approach to generate test cases from deterministic distributed reactive systems specified as asynchronously communicating UML statecharts. Two approaches are presented for the generation process. The first one is fully automated and generates test cases aimed at structural coverage of the UML model. The second requires the intervention of the tester in order to annotate states and/or transitions partially describing a test scenario of interest.

In model-based testing the size of the model has a great impact on the time for computing test cases. In model checking, slicing of specifications is used to obtain reduced models pertinent to criteria of interest. In specifications described using state based formalisms slicing involves the removal of transitions and merging of states thus obtaining a structural modified specification. Using such a specification for model based test case generation activities where sequences of transitions represent test cases might provide traces that are not valid on a correctly behaving implementation. In order to avoid this, the present work suggests the use of control, data and communication dependencies for identifying parts of the model that can be excluded so that the remaining specification can be safely employed for test case generation.

Kurzbeschreibung

Da die Komplexität von Software-Systemen steigt, wird Testen wichtiger aber auch teurer. Somit entsteht die Notwendigkeit möglichst viel dieser Aufgabe zu automatisieren. Die akademische Gemeinschaft bietet vielversprechende Prototypen an die unter anderen Funktionen auch die Generierung von Testfällen ermöglichen. Normalerweise erfordern solche Werkzeuge Mitarbeiter die auf dem Gebiet der formalen Methoden qualifiziert sind. Das Fehlen einer solchen Personal hindert oft die Annahme diesen Werkzeuge in einer industriellen Umwelt. Diese Arbeit versucht, die Lücke zwischen akademischer modellbasierten Testtools und deren Verwendung in der Industrie zu schließen. Dies wird durch die Angabe der gewünschten Verhalten des Systems in einem allgemein anerkannten Industrie-Notation (UML Statecharts) erreicht. Das UML Modell wird dann verwendet, um daraus eine formale Spezifikation des Systems mittels der LOTOS Sprache zu extrahieren.

Die vorliegende Arbeit stellt ein Modell-basierte Testfall Generierung Ansatz vor. Die Modelle beschreiben deterministische verteilte reaktive Systeme die asynchron kommunizieren und als UML Statecharts spezifiziert sind. Es gibt zwei Möglichkeiten für die Generierung der Testfälle. Die erste ist voll automatisiert und erzeugt Testfälle die auf strukturelle Abdeckung des UML-Modells angestrebt sind. Die zweite erfordert die Hilfe des Testers, um Zustände und/oder Transitionen zu annotieren. So wird ein Testszenario teilweise beschreiben.

In modellbasierten Testen hat der Größe des Modells einen großen Einfluss auf die Testfallgenerierung Zeit. In Model Checking wird Slicing von Spezifikationen verwendet, um reduzierten Modellen zu erhalten. In Spezifikationen beschrieben mit Zustand-basierten Formalismen, Slicing beinhaltet die Entfernung von Transitionen und Zusammenführen von Zustände. So wird eine strukturell veränderte Spezifikation erzeugt. Bei Verwendung eines solchen Spezifikation für Modellbasierte Testfallgenerierung besteht die Gefahr ungültige Testfälle zu bekommen. Um dies zu vermeiden, schlägt die vorliegende Arbeit die Verwendung von Steuer-, Daten- und Kommunikations-Abhängigkeiten vor. Die Abhängigkeiten werden

eingesetzt zum Identifikation von Teile des Modells, die sicher während die Testfallgenerierung ausgeschlossen werden können.

Contents

Acknowledgments

This is where I give thanks to all those who have supported me during the writing of this thesis. First of all I thank God for all He has given me, for the things I am aware of and for those I am not. Quite a few people have helped me grow both professionally and personally during the last four and a half years. I would like to give special thanks to my supervisor Professor Franz Wotawa not only for his valuable comments on my work but also for his confidence and constant support. I would also like to express my appreciation for the members of my defense committee, Professor Manuel Núñez and Professor Denis Helic.

I would like to thank my parents and my sister Diana for their support throughout the years and for their contribution to my own personality. I would like to thank my wife Irina for her support and her patience during my study. I also thank her for proof reading this thesis. I thank Mihai for being my best friend for the last 19 years and for his valuable comments on my work. I thank Simona with whom I shared an office for the last two years and a beautiful friendship for the last 18. I also thank Iulia for her friendship and support. Further thanks go to my former colleague Martin Weiglhofer for fruitful discussions regarding the LOTOS language.

This thesis was partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT). Thank you!

Chapter 1

Introduction

1.1 Motivation

Today's software enabled-systems are becoming increasingly complex, distributed and highly reactive. In this respect the software's functional correctness with regard to its requirements is a major quality attribute. To ensure functional correctness in a practical setting - besides following appropriate software engineering methods - software testing is the predominant verification technique. However, designing appropriate test cases is regarded as a rather expensive task. Testing activities usually consume an important part (estimated to be of up to 50 % [Alb76, Mye04]) from the resources of software development projects. Thus it is desired to automate as much as possible from this task.

Activities like test case execution and evaluation already enjoy a high degree of automation. Even though there have been some advances in the automatic generation of test cases, there is still work to be done in this direction. The integration of this activity in the already existing testing process is of great importance. UML statecharts [OMG13] are the de-facto accepted industry standard for modeling the behavior of software systems. They are also part of the industrial setting in which the current work was conducted. Therefore the generation of test cases from UML statecharts is desired.

In most situations, test cases are created manually this being an error prone and time consuming process. Even though UML statecharts have enough expressive power offering building elements to describe different behavioral aspects, for the automatic generation of test cases we need a formal representation of the system and a sound testing theory. The test case generation approach proposed in this work is based on the formal language LOTOS [ISO89] and on the input output conformance (IOCO) relation [Tre08] as the underlaying theory of the test case generator TGV [JJ05].

Formal Description Techniques (FDT) like LOTOS encompass those methods used for the unambiguous and exact specification of telecommunication (and not only) services and protocols. The fact that these type of systems are generally distributed also suggests the use of such techniques in our setting regarding the description of distributed embedded systems.

The main advantages of FDT are abstractness, implementation-independence, formal semantics and support of verification methods [FH92]. In our particular context of model-based test case generation these are exactly the characteristics we need our test specification to have.

Abstractness and implementation-independence relate to the fact that in model-based test case generation, a specification should be at a higher level of abstraction than the implementation of the system. This means that the specification needs to be simpler than the implementation in order for it to be verified more easily than the implementation. If the same level of granularity existed between the implementation and specification, the effort of verifying the specification would be equivalent to that needed for the verification of the implementation.

Formal semantics and support for verification methods of FDT are important since these make the specification machine readable thus allowing an increased degree of automation in the process of test case generation (and not only).

Formal languages are not very user friendly and this is an important impediment in the adoption of academic model-based testing tools in an industrial context. The present work is also intended as an attempt to breach this gap by allowing the specification of the system in a widely accepted industry notation (UML statecharts) and the connection to academic tools (like the CADP framework [GMLS07]) by providing a behind the scene formal representation of the system using the formal language LOTOS. There are many examples [APWW07, SBBW09, Tan09, SZWH11] of the successful use of the CADP framework in industrial settings. As LOTOS contains no timing constructs the current work also presents an approach for abstracting timing information from the UML statechart model. In the generated LOTOS specification the proposed timing abstraction aims at keeping the visible output behavior of the system by preserving the order timeout transitions fire with respect to each other.

In model-based testing the size of the used model has a great impact on the time for computing test cases. In model checking, dependence relations have been used in slicing of specifications in order to obtain reduced models pertinent to a criteria of interest. Usually in specifications described using state based formalisms slicing involves the removal of transitions and merging of states thus obtaining a structural modified specification. Using such a specification for model based test case generation activities where sequences of transitions represent test cases might provide traces that are not valid on a correctly behaving implementation. In order to avoid this, the present work suggests the use of control, data

and communication dependencies for identifying parts of the model that can be excluded so that the remaining specification can be safely employed for test case generation. This information is included in test purposes which are then used in the test case generation process.

1.2 Problem Statement

Many software controlled functionalities in modern vehicles are distributed over several electronic control units (ECU). These ECUs communicate with each other in order to implement the required functionalities. Due to the different dependencies involved in such functionalities testing them becomes a complex and error prone task. A promising technique for supporting the test engineers in such tasks is the use of formal model based testing techniques. The generation of test cases from a formal model in our setting involves several challenges that need to be considered:

Testing Technique and Modeling Language In the academic domain there are several testing techniques each having different assumptions in order to be able to treat different issues. So finding and eventually extending the proper and most simple technique able to deal with the particularities of the considered type of systems are important issues.

In order to apply test case generation techniques one needs a formal model of the system. The direct use of a formal language in an industrial environment is usually hindered by the lack of trained personnel. UML is the de-facto modeling language used in industrial environments (including our project setting) thus being the obvious choice for the modeling process. However UML enjoys a formal syntax but lacks a formal semantic. A formalization fitted for the considered environment semantic needs to be achieved.

Another issue is the fact that the chosen testing technique might not accept the UML model as input. Thus the formalization of the semantics needs to be provided in a formalism compatible with the testing technique. This process however needs to be fully automated and hidden from the user.

Timing Given that the considered type of systems also use timing constructs to realize their functionalities, a method to accommodate this need has to be researched. Even though there are also modeling languages and testing techniques allowing the use of timing constructs, these were not always usable within our research context.

Test Case Selection Once the appropriate test case generation technique and formal model are in place, the question of how to select the appropriate set of test cases arises. There are several approaches tackling this issue. The use of test purposes has shown to be very promising for this task. Test purposes represent abstractions of scenarios that need to be tested. They are usually specified

manually by using some sort of formal representation.

The automatic generation of test purposes has great potential in reducing the complexity of the testing process. Structural coverage criteria can be used to achieve this. However a high coverage does not necessarily mean that the system is well tested. So providing the means for the test engineer to specify the test purposes and thus steer the test case generation process is also important. The specification of test purposes suited for use in an industrial environment needs to be explored.

State Space Explosion The exponential growth of the number of possible states representing the behavior of a system is a well known problem not only in the testing domain. This is also known as the state space explosion problem and is one of the biggest problems influencing the test case generation process. The generation time and the probability of finding the required test cases are directly influenced by the size of the possible behaviors of the system.

Depending on the type of the model, the size of its state space is usually influenced by the use of variables, loops and parallelism. A variable will generate a state of the system for each of its allowed values. In the case when several variables are used, the allowed value combinations of the variables must also be considered. Loops and parallelism influence the size of the state space through the number of possible executions they have.

The issue of state space explosion has yet to be fully resolved. So depending on the situation at hand different techniques need to be considered in order to alleviate this problem.

1.3 Thesis Statement

A model-based test case generation approach can be used in practice to generate test cases for deterministic systems whose desired behavior is described by means of asynchronously communicating UML Statecharts.

1.4 Contribution

This thesis describes a test case generation approach for asynchronously communicating UML Statecharts. A prototype tool has been developed implementing the proposed approach. The workflow describing the functionality of the tool is presented in Figure 1.1. The input for the tool is an UML model describing (by means of communicating statecharts) the desired behavior of the system under test. The model is then used to automatically derive a formal representation in form of a LOTOS specification. Another input is represented by the user defined annotations partially describing scenarios to be tested. These annotations are used internally to generate test purposes for the TGV test generator. The tool

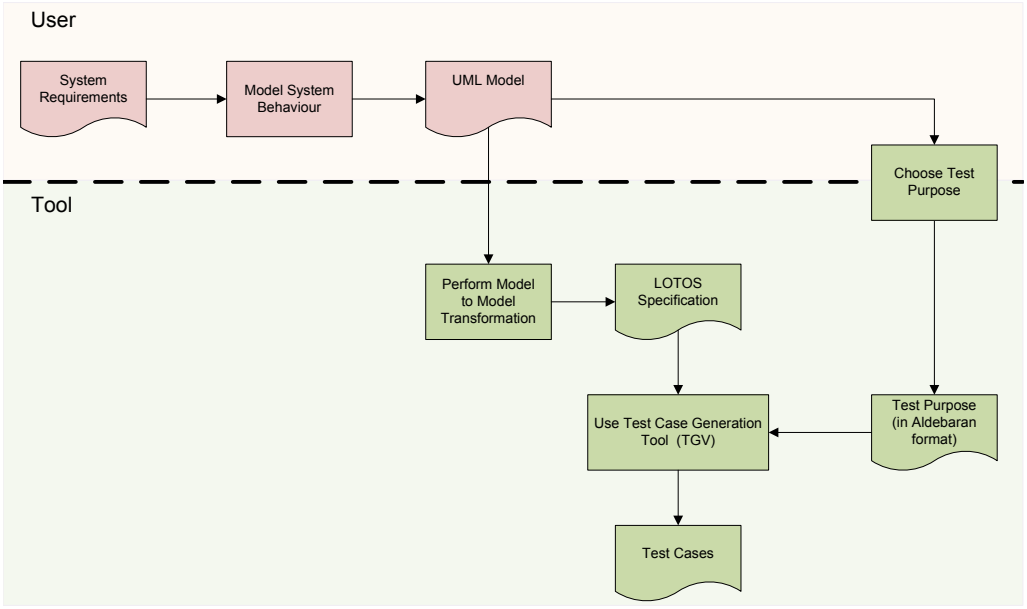

Figure 1.1: Tool workflow

can also be used to generate test cases without the user annotations. In this case test purposes are automatically generated and aimed at achieving state and/or transition coverage on the UML model.

The work described in this thesis has been partially published in international workshops and conferences:

- A first variant of the transformation of one UML statechart model into a LOTOS specification was presented in [CSP09a]. Also first considerations of how to include also several other statecharts models was described in [CSP09b]. These transformation variants were not automated and did not consider the use of timeouts in the original UML model.

- In [CW11] we presented an approach for the treatment of timeouts during the transformation. The approach abstracts the real timing by considering only the visible output of the system. Thus the order timeout transitions fire with respect to each other is preserved. The presence of a global clock is simulated through LOTOS abstract data types and several control points artificially inserted in the specification. The goal is to ensure that the transition triggered by the timeout with the smallest value will always fire before any other timeout transition.

- In order to steer the test case generation process, the mechanism of test purposes is used. In [CW12] we present two approaches for generating test purposes. One of which is aimed at providing structural coverage and is

fully automatic. The second approach to test purpose generation is partially automated requiring the intervention of the user for specifying desired test scenarios by annotating states and/or transitions of the statechart model.

- In [CW13] we introduce the usage of different statechart dependence relations in order to enhance test purposes with refuse states. These states are used by the TGV test-case generation tool in order to limit the searched state space during the test case search process. We use these refuse states in order to improve a previously presented test case generation technique [CW12] aimed at structural coverage (state and transition coverage) of the specification.

1.5 Research Context

Part of this thesis is the result of a joint research project by the Institute of Software Technology - Graz University of Technology, the Area E - Virtual Vehicle, and two industrial partners from the automotive industry.

The main project goal was to analyze the testing environment, existing models and tool chain in place at the industrial partner, propose an appropriate model based test case generation technique and implement this technique in a prototype tool. Another important goal was the integration of the technique with the already existing tool chain.

The models and tool chain of our industrial partners also impose some restrictions on the used models and proposed model-based test case generation technique. One of the limitations is the fact that only deterministic models are used. The communication scheme is also a deterministic one. So after a sequence of inputs the system will always return the same output. The need for deterministic behavior for embedded systems has also been acknowledged in [HK04].

The afore mentioned restrictions do limit the applicability of the proposed approach to the considered semantics. However the technique can be adapted to also accept nondeterministic systems. The used formalism (LOTOS) is very well suited to describe nondeterministic behavior. The test case generation technique will require future research for dealing with such nondeterministic behavior.

1.6 Organization

The remainder of this thesis is organized as follows: Chapter 2 presents in Section 2.1 topics regarding software testing and model based testing. In Sections 2.2 and 2.3 we present the UML statecharts used for modeling the system's behavior in our industrial setting and the LOTOS formal language respectively.

Chapter 3 presents a running example used in the rest of the thesis and also several other models we used in our experiments. The transformation of UML

statecharts into LOTOS is described in Chapter 4 while in Chapter 5 we introduce the treatment of timing concepts in the transformation.

Chapter 6 presents a method for automatically generating test cases aiming at structural coverage (state and transition coverage) of the model. It also shows how to semi-automatically generate test cases by making use of user provided annotations on the UML model. Chapter 7 describes an approach regarding the use of control, data and communication dependencies for identifying parts of the model that can be excluded during test case generation for a given test purpose. This information is used in order to enrich test purposes with refuse transitions. These transitions specify which parts of the model are not of interest (will not be explored) in the search for test cases satisfying the given test purposes.

Related work is discussed in Chapter 8 and the thesis is concluded in Chapter 9 where we present conclusions and directions for future work.

Chapter 2

Preliminaries

2.1 Software Testing

Throughout the literature there have been given several definitions for software testing. The general understanding for it [UPL06] is that "Testing aims at showing that the intended and actual behaviors of a system differ, or at gaining confidence that they do not. The goal of testing is failure detection: observable differences between the behaviors of implementation and what is expected on the basis of the specification."

Regarding the confidence gained through testing it is important to mention that "Testing shows the presence, not the absence of bugs" (Edsger Dijkstra). Thus testing cannot prove that a system will always behave correctly.

To better present this topic we need to define some of the most important terms in software testing. We shall further use the definitions given by [AO08]. The first terms we introduce are those of faults (Definition 2.1.1), errors (Definition 2.1.2) and failures (Definition 2.1.3). Basically, a failure is the observation of incorrect behavior of the implementation while a fault represents the cause of a failure. A fault could be for example an incorrect instruction in the implementation. The manifestation of a fault is called an error which can be for example an incorrect internal state of the implementation.

Definition 2.1.1. Software Fault: A static defect in software.

Definition 2.1.2. Software Error: An incorrect internal state that is the manifestation of some fault.

Definition 2.1.3. Software Failure: External, incorrect behavior with respect to the requirements or other description of the expected behavior.

It is also important to note that not all inputs will execute the faulty instruction and reveal the failure. Even if the instruction gets executed depending on the values of the inputs, the failure might or might not be revealed. In order to reveal the fault, the instruction containing the fault needs to be executed, the implementation needs to be in an incorrect state (error) and the error must propagate so that it causes the output of the software to differ from the expected one. Thus a failure has been revealed.

Software Testing assumes the execution of a system with the goal of detecting failures [UL07]. A differentiation needs to be made between testing and other quality improvement techniques such as static verification, code inspections, reviews and debugging. Debugging is the process of finding a fault once a failure has been identified through testing.

The process of testing usually implies the execution of the implementation against test cases. A test case can be seen as an experiment in providing inputs to the implementation in order to verify that it behaves according to the expected results. Definition 2.1.4 presents the definition of test cases as it was given by [AO08]. The test case values in the definition are the values needed to perform some execution of the implementation. The prefix values represent those values needed in order to take the implementation to a state where it can consume the test case values. The set of postfix values is composed of those values that need to be provided to the software after it has received the test case values.

Definition 2.1.4. Test Case: A test case is composed of the test case values, expected results, prefix values and postfix values necessary for a complete execution and evaluation of the software under test.

There are different kinds of testing classified according to various criteria [UL07]:

- **Scale of the system under test (SUT) :**
  - *Unit Testing* - a single unit at a time is tested e.g. a single method, class etc.
  - *Component Testing* - each component/subsystem is tested separately.
  - *Integration Testing* - aims at testing that several components work together correctly.
  - *System Testing* - the system is tested as a whole.
- **Characteristics being tested:**
  - *Functional Testing* - is the most common type of testing and aims at testing the functionality of the system e.g. given a set of inputs the correct outputs are obtained.

  - *Robustness Testing* - tests the system under invalid conditions e.g. unexpected inputs, hardware or network failure etc.
  - *Performance Testing* - tests how a system performs in terms of responsiveness and stability under heavy load.
  - *Usability Testing* - is focused in discovering user interface problems that might result in making the system difficult to use.

- **Tests derived from :**
  - *Black Box Testing* - does not consider any details regarding the internal structure of the SUT treating it as a *black box*. In this case, tests are derived from the system requirements describing the desired behavior of the SUT.
  - *White Box Testing* - uses the actual code of the implementation in order to design test cases e.g. design test cases such that each statement of a method is executed by at least one test case (statement coverage).

The classic testing process contains three main tasks [UL07]:

- **Test case design** activities assume the creation of test cases from the requirements of the system.
- **Test case execution and result analysis** activities relate to the execution of the test cases on the SUT. The failed runs are analyzed in order to determine the cause of the failure.
- **Verifying how the test cases cover the requirements** is usually a criterium used to measure the quality of the testing process.

In software development processes, testing activities encounter various difficulties. Different process oriented approaches have emerged in order to solve some of the problems related with testing activities. Depending on the used testing process, it is possible to differentiate between the following testing approaches and also points out their pros and cons [UL07]:

- **Manual Testing Process** was the first testing process used where all the activities were performed manually. It is however still used. Some of the advantages and disadvantages of this approach are:
  - **Pros:** Used for functional testing.
  - **Cons:** Imprecise coverage of SUT functionality, no capabilities for regression testing, very costly process, no effective measurement of test coverage.

- **Capture Replay Testing Process** is the process that saves the interactions of the SUT during test execution (*capture*) with the goal to *replay* them during following test executions. The main goal of this process is the reduction of the costs generated by test re-execution. The rest of the testing activities are still carried out manually. Pros and cons of this approach :
  - **Pros:** Provides the possibility to automatically execute the captured test cases.
  - **Cons:** Imprecise coverage of SUT functionality, weak capabilities for regression testing, costly process.
- **Script-Based Testing Process** assumes the use of test scripts to further increase the automation degree of the approach. The scripts are used in order to run test cases but also observe the behavior of the SUT.
  - **Pros:** Automates the execution and re-execution of test scripts.
  - **Cons:** Imprecise coverage of SUT functionality, complex scripts are difficult to write and maintain, requirements traceability is developed manually.
- **Keyword-Driven Automated Testing** raises the abstraction level of the test cases in order to overcome the maintenance problem of test scripts. This is achieved by using *action keywords* corresponding to different fragments of a test script. These *action keywords* are then used to define the test cases. The test execution framework translates the keyword and data values in the abstract test cases in order to generate executable test cases.
  - **Pros:** Higher level test scripts are easier to develop.
  - **Cons:** Imprecise coverage of SUT functionality, requirements traceability is developed manually.

2.1.1 Model-based Testing

The main understanding of model-based testing (MBT) (also valid in our setting) refers to the generation of executable test cases that include oracle information based on models of the SUT behavior [UL07].

Definition 2.1.5. Model-Based Testing is the automation of the design of black-box tests.

One advantage of MBT over the other methods mentioned in Section 2.1 is the fact that it offers a higher degree of automation of the different testing activities. Thus test cases are no longer designed manually but are automatically generated from the model of the SUT's behavior. By using different test selection criteria

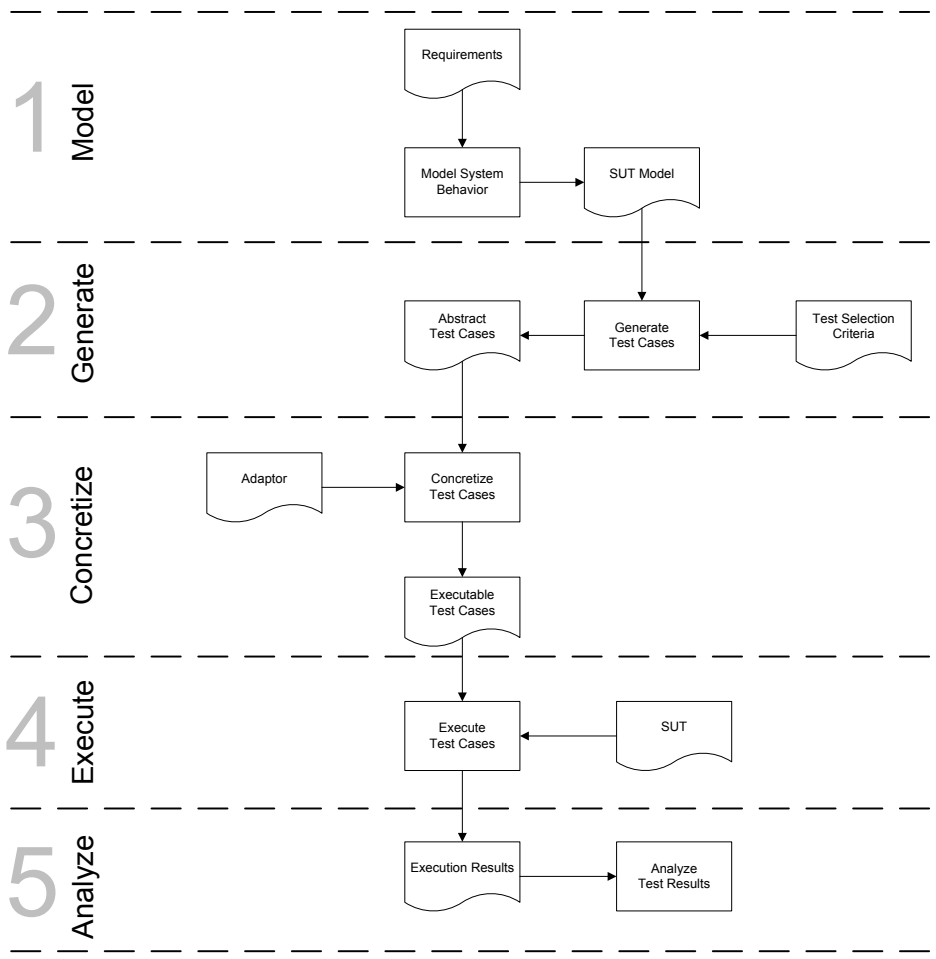

Figure 2.1: Model-Based Testing steps

the same model can be used in order to generate different test suites. Another advantage is the fact that the model is used as a **test oracle** and thus the expected results for the test cases are also automatically generated.

MBT assumes the usage of a model describing the desired behavior of the SUT and employs automatic tools in order to generate tests from the model. Since the models are used as the basis for generating the test cases it is obvious that they play a crucial role in MBT. The models need to be validated themselves this being a "reciprocal activity: validating the model usually means that the requirements themselves are scrutinized for consistency and completeness" [UPL06]. Two main characteristics of the models used in MBT can be identified [UPL06]:

- The model must be simpler (at a higher abstraction level) than the SUT or at least easier to check, modify and maintain. Otherwise, the efforts of validating the model would equal the efforts of validating the SUT.

- Even though the model is more abstract than the SUT it is crucial that it preserves enough details regarding the behavior that needs to be tested in order for it to be used for generating "meaningful" test cases.

Figure 2.1 presents the five activities involved in MBT. These activities [UL07] are:

1. **Model** the SUT and/or its environment. The models need to be simpler than the SUT and focus only on the aspects that need to be tested.

2. **Generate** abstract test cases. Since the used model is more abstract than the SUT, any test cases generated from it will also be at a higher abstraction level compared to the SUT. Hence these test cases can not be directly executed against the SUT. Since usually the amount of possible test cases is infinite, some kind of selection criteria is used in order to obtain a finite set of test cases. Such criteria can be structural coverage of the model (e.g. state and/or transition coverage) or "some *test case specifications* in some simple pattern language to specify the kinds of test cases we want generated"[UL07].

3. **Concretize** the abstract test cases into executable ones. This is usually achieved by employing "adapter code that wraps around the SUT and implements each abstract operation in terms of lower level SUT facilities"[UL07]. This step fills in the details that were abstracted during the modeling step thus bringing the abstract test cases at the same abstraction level as the SUT.

4. **Execute** the test cases on the SUT.

5. **Analyze** the results of the test execution.

Advantages of using MBT in industrial settings are also pointed out by several experience reports. One work [PERH04] analyzed a set of 15 case studies (most performed in industrial contexts) in order to evaluate the applicability of MBT. The survey indicated that MBT was able to find errors in systems that were considered mature and well tested. Other reports [FHP02, BLLP04, PPW+05] indicate that MBT found at least the same number of errors when compared to manual testing. The amount of uncovered errors variates form one case to another. This variation depends on the type of systems and also on the used tools. Another important factor in the success of MBT is the experience of the tester in modeling the SUT and choosing the appropriate test selection criteria. Further advantages indicated by the case studies include the fact that MBT can reduce the time needed for the design of test cases and also helps in discovering problems related to the requirements of the SUT.

Even though the application of MBT offers quite a few advantages over the classic testing process, it has however its limitations [UL07]:

- Requires different skills compared to manual test design. Such skills include the ability to model the SUT and apply appropriate abstractions.
- It is mainly used for functional testing. The models describe only behavioral aspects of the systems. Robustness, performance and usability testing are left to be done manually.
- The models need to be updated once the requirements change. But also in a manual testing process, once the requirements change the test cases need to be updated.
- Some parts of the SUT might be difficult to model so it might be better to manually design test cases to test such parts.
- Time to analyze failed tests relates to the fact that a failed test case might indicate an error in the SUT, adapter code (needed to concertize the abstract test cases) or in the model. Something like this happens also in other testing approaches where a test case might fail due to a fault in the SUT or an error in the test script.

So the application of MBT in practice depends on the context of the project and on the ratio between advantages and limitations of applying the technique.

2.2 UML Statecharts

A state machine describes the behavior of systems in terms of sequence of states by specifying its evolution as responses to external stimuli. This type of notation imposes the restriction that the described behavior is composed of a finite number of states. This due to the fact that such diagrams need different states to represent every valid combination of parameters describing the behavior of the system. The problem with this restriction is obvious in the case of more complex systems. Even in the case of systems of reduced complexity, such diagrams can become quite complex and difficult to understand. The description of more complex behaviors becomes impossible due to the fact that the number of states and transitions needed is very large and often infinite.

In order to alleviate these limitations and still preserve the useful aspects regarding the behavior description capabilities of state machines David Harel introduced in 1987 the concept of statecharts [Har87]. Statecharts extend the notion of state machines with the concepts of hierarchy, superstates, orthogonal regions, pseudostates and others which will be presented later in this section. In the following years statecharts evolved into more than 20 variants [vdB94].

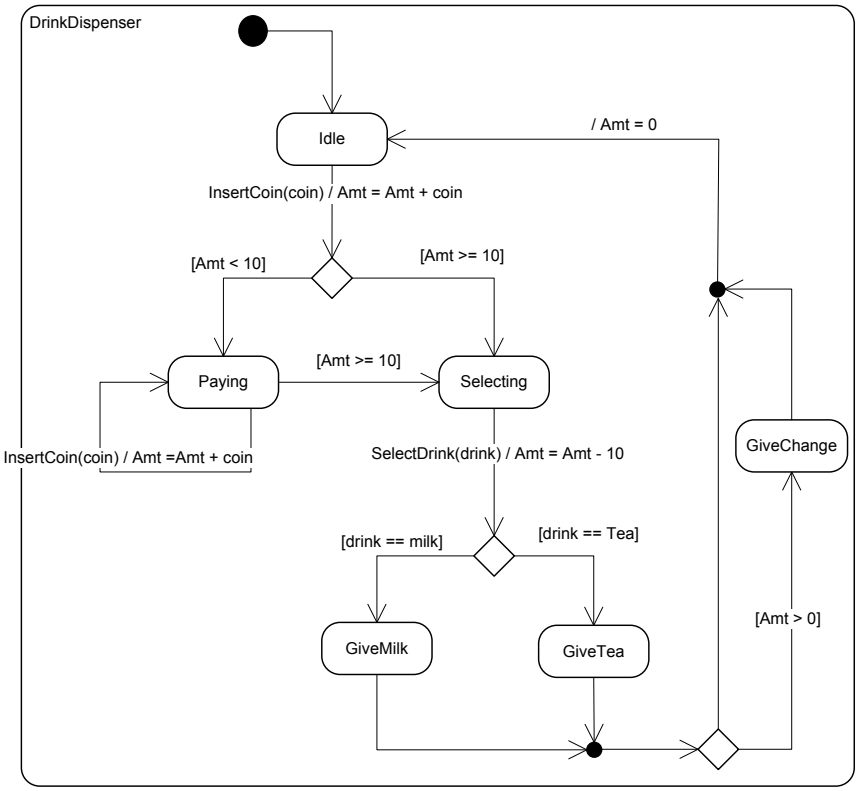

Figure 2.2: Statechart of a Drink Dispenser

Harel's statecharts have already achieved widespread use in industry. This is also due to the fact that they have been adopted and adapted by the UML standard [OMG13] as the basis for UML State Machines. Regarding the different variants of statecharts we will consider in the rest of this work the UML compliant variant implemented in the IBM Rhapsody tool [IBM13]. The semantics of these statecharts are described in previous work [HK04] and [HG96].

2.2.1 Syntax

Usually (and also in our case) when describing a system, a class diagram is used to represent parts of the system that interact with the environment and with each other. The dynamic part of the system (its behavior) is specified with the help of statecharts that are associated to the classes in the diagram. Thus every class has a statechart defining its reactions to the events it receives. Such classes are called **reactive** classes.

We present a formal definition of statecharts in Definition 2.2.1. Thus a statechart is represented as a tuple of states, transitions, variables and an initial state. The set of states S is composed of simple, composite states and pseudostates.

The states represent stable points in the evolution of the system. Pseudostates on the other hand do not and are used only to increase the expressive power of statecharts and reduce the modeling effort in some situations.

Definition 2.2.1. (Statechart) A statechart is a tuple $SC = (S, T, V, i_{ps0})$ where:

- $S \neq \emptyset$ is the set of states in the statechart. S is the union of three distinct sets, that is, $S = S_s \cup S_c \cup S_{ps}$ where:
  - S_s - simple states of the system.
  - S_c - the set of composite states. A composite state can contain other states and pseudostates.
  - $S_{ps} = I_{ps} \cup J_{ps} \cup En_{ps} \cup Ex_{ps} \cup C_{ps} \cup H_{ps}$ - the set of pseudostates.
  - I_{ps} - the initial pseudostates of the composite states. The transition originating from such a pseudostate points to the initial state of the containing composite state.
  - J_{ps} - the set of junction nodes. Junction nodes are semantic-free pseudostates used to chain together several transitions.
  - En_{ps} - the entry points into composite states.
  - Ex_{ps} - the exit points of a composite states.
  - C_{ps} - condition pseudostates allow splitting of transitions into multiple outgoing paths depending on guards on the outgoing transitions.
  - H_{ps} - history pseudostates are used to store the last active state configuration inside the containing composite state.
- $T \subseteq S \times L \times S$ - is the set of transitions in the statechart where: $L \subseteq E \times G \times A$ is the label of the transition. All components of a label are optional.
  - $E = E_{tr} \cup TM_{ev} \cup \{\epsilon\}$ - the triggers (events) that can fire the transitions. Where E_{tr} is the set of triggering events, TM_{ev} is the set of timeout events and ϵ is the empty sequence.
  - G - the guards (condition) that have to be true in order for the transition to be fired.
  - A - the actions that are to be performed when the transition is fired.
- V - the set of variables used in the statechart.
- i_{ps0} - the initial pseudostate that is the origin of the transition pointing to the initial state of the statechart.

The notions of states, pseudostates and transitions in Definition 2.2.1 are further defined and informally described in the rest of this section.

Example 2.2.1. We present as example in Figure 2.2 the statechart of a drink dispenser machine that sells only two types of drinks (milk and tea). After introducing an amount of money greater than 10 the machine allows the selection of a drink and depending on the inserted amount provides also change (in case the amount was greater than 10). This is just a simple example that depicts some of the characteristics of statecharts such as:

- states and transitions
- hierarchy - state *DrinkDispenser* contains all the other states, pseudostates and transitions.
- pseudostates - the condition node connected to state *Idle* where the decision is made if the inserted amount is sufficient for a drink (state *Selecting*) or more coins are needed (state *Paying*).

States

"A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur"[OMG13]. A state can be either active or inactive. It becomes active when a transition targeting the state is fired. An active state becomes inactive if a transition originating from it is fired.

Example 2.2.2. In the statechart depicting the DrinkDispenser in Figure 2.2 for the state *Selecting* the condition that holds is represented by the fact that the user has inserted a sufficient amount of coins to be allowed to select a drink. The state waits for the event *SelectDrink* with the appropriate drink in order for the system to move to one of the states *GiveMilk* or *GiveTea*.

Even though in the definition of a statechart (Definition 2.2.1) the set of states includes simple, composite states as well as pseudostates ($S = S_s \cup S_c \cup S_{ps}$) we make a clear differentiation between states (simple and composite) and pseudostates. As already mentioned a state models a situation where some condition holds. Pseudostates however do not represent such situations and are only semantic nodes used in the modeling of the system. This means that a system cannot "stay" in a pseudostate. Pseudostates are used to express parallel behavior (fork or join), conditional behavior, entry and exit in/from composite states, termination of behavior (final pseudostate) and others. We will discuss the different types of pseudostates later on in this section.

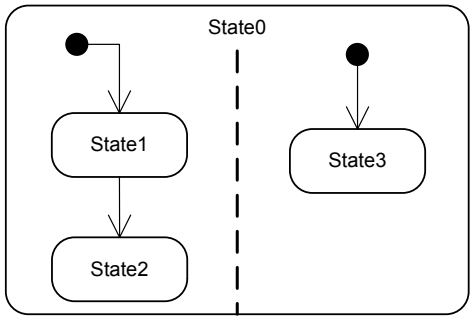

Figure 2.3: Example of an orthogonal state

Definition 2.2.2. (State) A state $s \in S/S_{ps}$ is a tuple $s = (name(s), type(s), entryAct(s), inAct(s), init(s), substates(s), exitAct(s))$ where:

- $name(s)$ is a string representing the name of state s and $\forall s_1, s_2 \in S : s_1 \neq s_2 \implies name(s_1) \neq name(s_2)$,
- $type(s) \in \{simple, orthogonal, sequential\}$,
- $entryAct$ is the set of actions that are executed when state s is entered,
- $inAct(s)$ is the set of actions that are executed while state s is active,
- $init(s) \in I_{ps} : type(s) = simple \implies init(s) = \emptyset$ - is the initial pseudostate of s,
- $substates(s) \subseteq S : type(s) = simple \implies substates(s) = \emptyset$ - is the set of substates contained by s,
- $exitAct$ is the set of actions that are executed when state s is exited.

In Definition 2.2.2 we present a formal definition of a state s as a tuple $(name(s), type(s), entryAct(s), inAct(s), init(s), substates(s), exitAct(s))$. The state is identified by a name, a type, entry and exit actions, initial state and a set of substates. Regarding the type of a state we identify three types of states:

- simple states (also called basic states) are states that do not contain any other states. State *Idle* in Figure 2.2 is such a state.
- sequential states (also called OR states) contain other states that are related to each other by exclusive OR - if the state is active, only one of its substates is going to be active. In the example in Figure 2.2 state *DrinkDispenser* is an OR state containing all the other basic states, transitions and pseudostates.

- orthogonal states (also called AND states) contain at least two concurrent regions in which several substates are located. An active AND state implies that all its regions are active (there is at least one active state in each of its regions). In the example in Figure 2.3, *State0* is such a state containing two orthogonal regions.

Orthogonal and sequential states are also called composite states due to the fact that they contain other substates. Graphically, states are represented as a rectangle with the state name shown inside it. There are several variants regarding the location of the name inside the rectangle. Orthogonal states are drawn as having several regions separated by each other through dotted lines.

Each state can have associated entry, during and exit actions. Entry and exit actions are executed when the state is entered and exited respectively. Internal (or during) actions (also called "static reactions") represent tasks that are to be executed as long as the system is in that respective state. These reactions have the same format as the label of a transition $ev[guard]/inAct(s)$ [HK04]. The execution of such static reactions assumes the occurrence event ev and the fulfillment of the guard. If these preconditions are met, then the actions $inAct(s)$ are going to be performed. However, the active configuration (the set of active states) of the system will not be changed and the state s will still be active after the execution of the actions.

Transitions

As specified in Definition 2.2.3 a transition is a connection between a source state and a target state. The source and target of a transition can be composite states, basic states or pseudostates. The transition has associated a label of the form $ev[guard]/actions$, where ev is the event that triggers the transition if $guard$ evaluates to $true$. Once the transition is fired, the actions defined on it are executed. All of the components of the label of a transition are optional.

Definition 2.2.3. (Transition) A transition tr is a tuple $tr = (source(tr), event(tr), guard(tr), actions(tr), target(tr))$ where:

- $source(tr) \in S$ is the source state from which tr originates.
- $event(tr) \in E$ is the trigger (event) that needs to happen in order to fire the transitions.
- $guard(tr) \in G$ represents the guard (condition) that has to be true in order for the transitions to be fired.
- $actions(tr) \subset A$ are the actions that are to be performed when the transition is fired.

- $target(tr) \in S$ is the state targeted by the transition.

IBM Rhapsody [IBM13] allows the usage of two types of triggers which are events and operation calls. Events are used for asynchronous communication while operation calls represent synchronous communication. In asynchronous communication the event gets enqueued into a FIFO queue and will be consumed as soon as the system reaches a stable state (we discuss this in more detail later in this section). This way the object that generated the event will continue its behavior without considering whether the event has been consumed or not. In the case of synchronous communication there is no enqueuing of the calls and the calling object (the one that issues the operation call) is blocked until the callee finishes processing the call.

Considering only the events, we can identify two types of events: normal events and timeout events. A timeout event is denoted in Rhapsody as *tm(t)* where t is the time in milliseconds until the transition is triggered. The timeout t starts from the activation of the source state of the timeout transition. When it elapses a timeout event is automatically generated causing the transition to be fired (assuming that its guard evaluates to true).

Remark. Event though there are two types of triggers that can fire a transition, we only consider events since the aimed domain is embedded distributed systems. Such systems usually use asynchronous communication.

Because in the rest of this work we use different names to identify different categories of transitions we give here the respective definitions. We define normal transitions (Definition 2.2.4) as those transitions explicitly triggered by an event other than a timeout event. The guard and actions part of the transition are still optional.

Definition 2.2.4. (Normal Transition) A normal transition is a tuple $tr_n = (source(tr_n), event(tr_n), guard(tr_n), actions(tr_n), target(tr_n))$ where:

- $source(tr_n) \in S$ is the source of tr_n.
- $event(tr_n) \in E_{tr}$ is the trigger event of the transitions.
- $guard(tr_n) \in G$ represents the guard (condition) that has to be true in order for the transitions to be fired.
- $actions(tr_n) \subset A$ are the actions that are to be performed when the transition is fired.
- $target(tr_n) \in S$ is the target state of the transition.

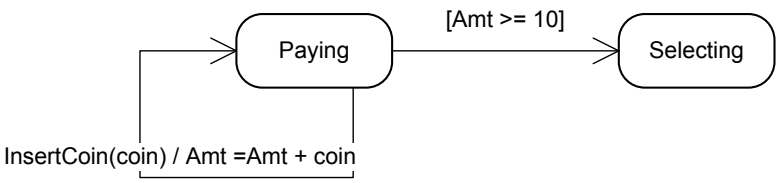

Figure 2.4: Example of normal and completion transitions

Since all the components from the label of a transition are optional (including the event) we can have transitions with no explicit triggering event. We call such transitions completion transitions (Definition 2.2.5). After entering the source state of such a transition, if its guard conditions are met then the transition is fired automatically without the need for an explicit trigging event.

Definition 2.2.5. (Completion Transition) A completion transition is a tuple $tr_{cmpl} = (source(tr_{cmpl}), event(tr_{cmpl}), guard(tr_{cmpl}), actions(tr_{cmpl}), target(tr_{cmpl}))$ where:

- $source(tr_{cmpl}) \in S$ is the source of tr_{cmpl}.
- $event(tr_{cmpl}) = \epsilon$.
- $guard(tr_{cmpl}) \in G$ represents the guard (condition) that has to be true in order for the transitions to be fired.
- $actions(tr_{cmpl}) \subset A$ are the actions that are to be performed when the transition is fired.
- $target(tr_{cmpl}) \in S$ is the target state of the transition.

Example 2.2.3. Figure 2.4 contains two states (*Paying* and *Selecting*). The transition looping at state *Paying* ($InsertCoin(coin)/Amt = Amt + coin$) is an example of a normal transition triggered by event *InsertCoin* and updating the amount of money accordingly as a result of its execution. The completion transition connecting the two states ($[Amt >= 10]$) will be fired as soon as a sufficient amount of coins has been inserted into the machine.

Depending on the type of the triggering event we also identify timeout transitions (Definition 2.2.6) as those transitions explicitly triggered by a timeout event. Similar to the other types of transitions, the guard and actions part of the transition are still optional.

Definition 2.2.6. (Timeout Transition) A timeout transition is a tuple $tr_{tml} = (source(tr_{tml}),\ event(tr_{tml}), guard(tr_{tml}), actions(tr_{tml}), target(tr_{tml}))$ where:

- $source(tr_{tml}) \in S$ is the source of tr_{tml}.
- $event(tr_{tml}) \in TM_{ev} : TM_{ev} \neq \emptyset$ is the trigger event of the transitions.
- $guard(tr_{tml}) \in G$ represents the guard (condition) that has to be true in order for the transitions to be fired.
- $actions(tr_{tml}) \subset A$ are the actions that are to be performed when the transition is fired.
- $target(tr_{tml}) \in S$ is the target state of the transition.

Pseudostates

A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph [OMG13]. There are several types of pseudostates defined in the UML standard allowing the modeling of different behaviors. They are divided into two main categories *AND* and *OR* [HK04].

OR pseudostates imply that out of all transitions connected to such a node only one incoming and one outgoing transitions will fire (depending of course on the arrival of the appropriate event and the truth value of the guard). The quoted definitions of the different *OR* pseudostates below are taken from the UML standard [OMG13]:

- The **initial** pseudostate represents a default vertex that is the source for a single transition to the default state of a composite state. There can be at most one initial vertex in a region. The outgoing transition from the initial vertex may have a behavior, but not a trigger or guard. An initial pseudostate is represented as a small solid filled circle and an example can be found in Figure 2.5.
- **Junction** vertices are semantic-free vertices that are used to chain together multiple transitions. A junction can be used to unite multiple incoming transitions into a single outgoing one. Semantically all incoming transitions have the same target state. They just share the outgoing transition of the junction node. The UML standard also allows junction nodes to be used for splitting incoming transitions into several outgoing ones. However the Rhapsody semantic (which we use) restricts the use of these nodes only to merging. Graphically, a junction is represented as a small colored circle. In Figure 2.5 there is an example of a junction node: the small dark circle targeted by the transitions from states *GiveMilk* and *GiveTea* and whose outgoing transition points to the choice pseudostate.
- **Choice** vertices, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a dynamic

conditional branch. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step. In deterministic systems the guards on the outgoing transitions must be mutually exclusive so that only one of the guards is true at any time during the execution of the model. In non-deterministic systems the standard specifies that if more than one guard evaluates as true, one of the outgoing transitions is selected. Graphically, a choice is represented as a "diamond-shaped symbol". An example of such a node can be found in Figure 2.5 splitting the transition originating from state *Selecting* ($SelectDrink(drink)/Amt = Amt - 10$) into the two transitions ($[drink == milk]$ and $[drink == tea]$) depending on the value of the *drink* parameter received with event *SelectDrink*.

- **Deep History** represents the most recent active configuration of the composite state that directly contains this pseudostate (e.g., the state configuration that was active when the composite state was last exited). There can be only one such node inside a composite state. It can have only one outgoing transition targeting the default history state (the state that will be activated at the first execution activating the parent state). Such a node is represented as a small circle containing an *H**.

- **Shallow History** represents the most recent active substate of its containing state (but not the substates of that substate). As in the case of deep history, there can only be at most one shallow history pseudostate in a composite state. There can also be only one outgoing transition from such a node pointing to the default history substate. This type of pseudostate is represented as a small circle containing an *H*.

- The **entry** pseudostate is an entry point of a composite state and has a single outgoing transition pointing to a state inside the containing superstate. Such a node is depicted as a small empty circle on the border of its composite state.

- An **exit** pseudostate is an exit point of a composite state. Graphically, such a node is represented as a small circle enclosing an *X*. The node is located on the edge of the composite state it belongs to.

- Entering a **terminate** pseudostate implies that the execution of this state machine by means of its context object is terminated. The state machine does not exit any states nor does it perform any exit actions other than those associated with the transition leading to the terminate pseudostate. The node is represented as a cross.

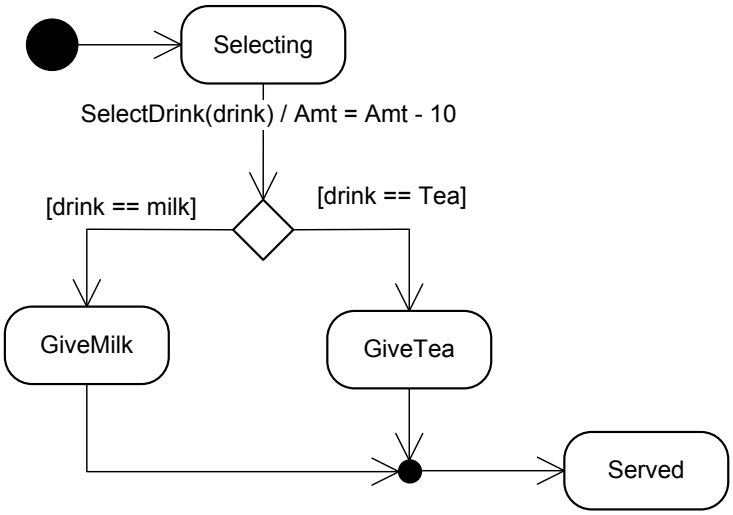

Figure 2.5: Example of Junction and Condition pseudostates

AND connectors are usually used in the presence of orthogonal states and imply that more than one (usually two) transitions segments (incoming or outgoing) are fired at the same time. These connectors come in two variants:

- **Join** vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards or triggers[OMG13]. Considering the example in Figure 2.6 only when the system is in states *State2* and *State3* and if the event *ev2* occurs will the system move to *State4*. If the active state configuration is for example *State1, State3* and *ev2* occurs there will be no state configuration change and the event will be ignored.

- **Fork** vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e., vertices in different regions of a composite state). The segments outgoing from a fork vertex must not have guards or triggers [OMG13]. An example of a fork pseudostate can be found in Figure 2.6. The pseudostate is targeted by the transition triggered by event *ev1* originating from state *State5*. If the active state configuration of the system is given by *State5* and *ev1* occurs, the new state configuration will be *State0* (the orthogonal state), *State1* and *State4*. Considering the occurrence of event *ev2* while the system is in state *State5* the new active state configuration will be *State0* (the orthogonal state), *State2* and *State3* (due to the fact that these last two states are the default states for their respective orthogonal regions).

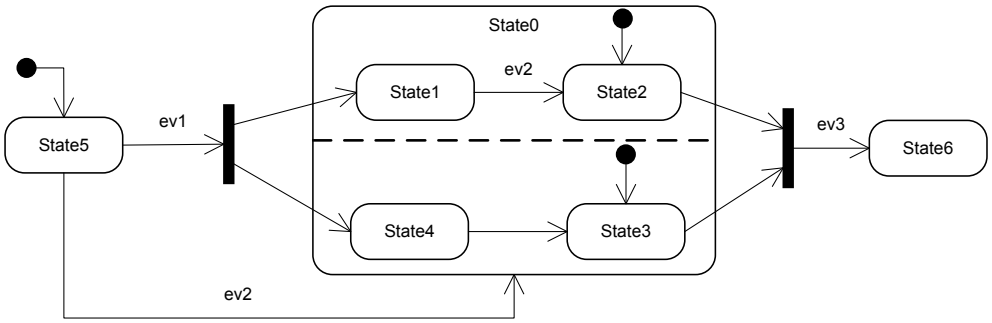

Figure 2.6: Example of Fork and Join pseudostates

2.2.2 Semantics

As already mentioned, since their introduction, several statecharts variants have been proposed [vdB94]. Out of these variants we use the one implemented in the UML modeling tool IBM Rhapsody. In this section we present the execution semantics considered for the rest of this work.

The UML standard does offer a detailed informal definition of the intended semantics. However, it allows some variations due to the fact that it is intended as a general purpose modeling language aiming to accommodate a wide range of domains. In addition, there are also some aspects that are not clearly defined when considering non-deterministic systems. For example the order of execution of transitions in orthogonal regions. Another example would be that in some cases of conflicting transitions (presented below), the standard allows for a nondeterministic choice to be made - a transition is arbitrarily chosen to fire.

Due to the fact that we consider test case generation for deterministic systems, such variations need to be properly handled in order not to provide invalid test cases. In Rhapsody, these cases of nondeterminism are either resolved to deterministic choices or not allowed.

The Run-to-Completion Step

According to previous work [HK04] the behavior of a system described in Rhapsody is a set of possible **runs**. A run consists of a series of detailed snapshots of the system's situation. Such a snapshot is called a **status**. The first in the sequence is the initial status, and each subsequent one is obtained from its predecessor by executing a **step** or **run-to completion step** (RTC) as it is called in the UML standard.

In order to define the run-to completion step execution semantics we first need to define the terms of active state configuration and stable state of the system.

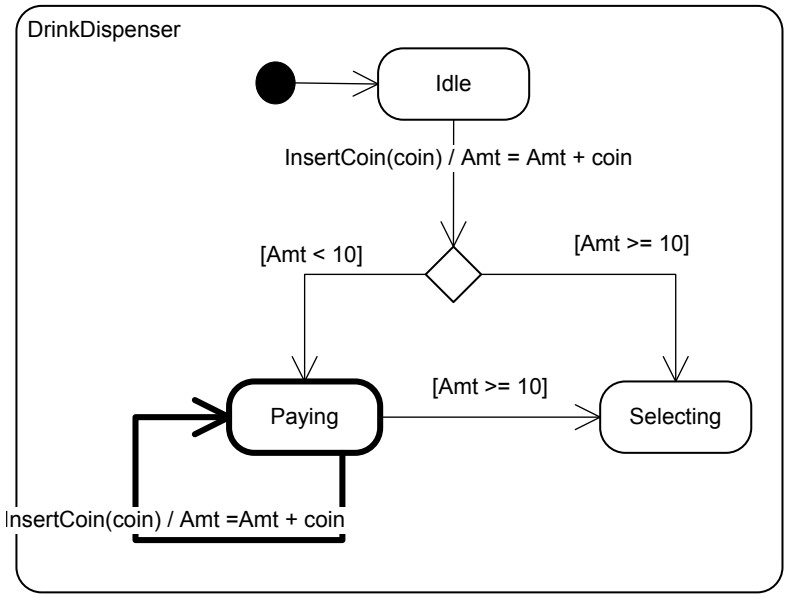

Figure 2.7: Partial specification of Drink Dispenser

Definition 2.2.7. (Active State Configuration) The active state configuration of a system is the set of active states at a given moment in the execution of the system.

Example 2.2.4. Considering the example in Figure 2.7 (a partial specification of the Drink Dispenser), after the initialization of the system (execution of the transition originating from the initial pseudostate) the active state configuration is $\{DrinkDispenser, Idle\}$.

Definition 2.2.8. (Stable State Configuration) A system S is considered to be in a stable state if the following conditions hold:

- All actions caused by the consumption of the last event have been performed.
- Its current active state configuration does not have any completion transitions that can be fired.
- No other type of transitions can be fired without consuming a new event.

According to Definition 2.2.8 the system can remain in the current active state configuration until the arrival of a new event.

Example 2.2.5. Considering the same example in Figure 2.7 and the active state configuration $\{DrinkDispenser, Paying\}$, the system is in a stable state only if after the consumption of the last *InsertCoin* event and the updating of the amount

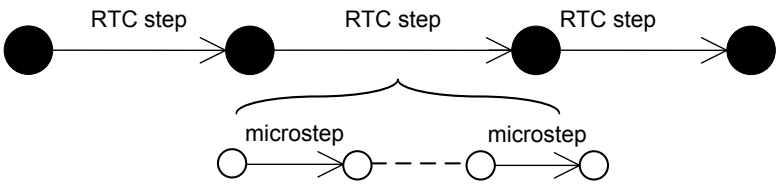

Figure 2.8: The Run-to-Completion Step

variable ($Amt = Amt+coin$) the value of Amt is smaller than ten (the completion transition targeting state *Selecting* can not fire). If however $Amt >= 10$ the completion transition will fire and the stable state configuration of the system will be $\{DrinkDispenser, Selecting\}$.

A **Run-to-completion step** represents the evolution of the system from the reception of an event until reaching the next stable state configuration. This means that the event has been consumed, the appropriate actions have been completed and all possible completion transitions have been fired. Thus an RTC step can be seen as the mechanism that makes the system evolve from one stable state configuration to the next one upon receiving an event.

An RTC step is said to be uninterruptable since upon starting to consume an event, the statechart can not be interrupted until it reaches a stable state. Thus the events that arrive for the statechart during a RTC step are enqueued in a FIFO event queue and will be delivered to the statechart when this reaches a stable state. If in the newly reached stable state configuration there is no transition that can be triggered by the event from the queue, the system will not change to another state configuration and the event will be discarded. This process continues with the other events in the event queue until the queue is emptied.

From the reception of an event and until the current RTC step is completed several actions need to be performed and completion transitions can be fired (if their guards evaluate to true). Thus an RTC step is described as being composed of several **microsteps**. The evolution of the statechart from one stable state to the next one is a sequence of microsteps.

Figure 2.8 [HK04] illustrates the decomposition of an RTC step into microsteps. The black circles in the figure represent the stable state configurations of the system between two RTC steps. The empty circles between two microsteps represent unstable active state configuration of the statechart.

Conflicting Transitions (Nondeterminism)

Two transitions are said to be in a conflict if there is some common state that would be exited if either of them were to be taken [HK04]. This situation appears

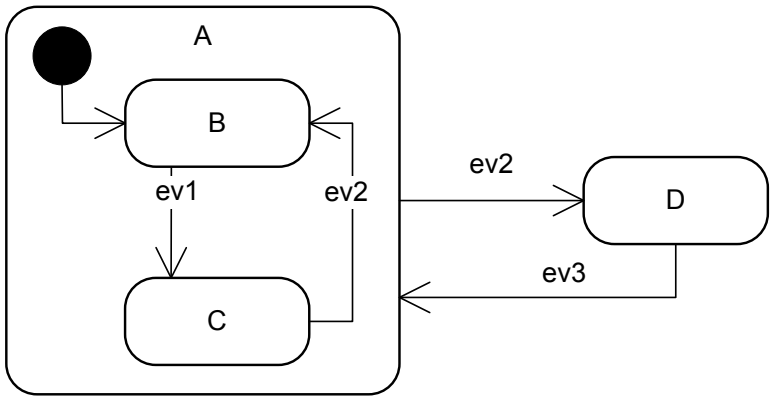

Figure 2.9: Transition conflict

in the case when more than one transitions available in the current active state configuration can be triggered by the same event. There are two types of conflicts between transitions depending on their source state and the hierarchical relations between their sources.

The first type of conflict refers to the situation where two transitions triggered by the same event (and whose guards evaluate to true) originate from the same state. So upon the reception of the triggering event both transitions would cause the source state to be exited. This is a clear case of nondeterminism since there are no rules defined in order to specify which of the transitions should be taken. Rhapsody does not allow this type of conflict since the diagrams are used for code generation and "for most embedded software systems such nondeterminism is not acceptable"[HK04].

The second type of conflict appears between two transitions having the same triggering event, whose source states are on different hierarchical levels in the statechart and such that the source state of one transition is a substate of the source of the other conflicting transition.

In this case a prioritization scheme is used in order to resolve the conflict and remove nondeterminism. In Rhapsody, in case of conflicting transitions the one with the lower level source state has the higher priority.

Example 2.2.6. Consider the example in Figure 2.9 where the transition originating from state C targeting state B is in conflict with the transition originating from state A to state D since $ev2$ is the trigger for both ones. If the active state configuration of the statechart is $\{A, C\}$ and event $ev2$ arrives the new active state configuration will be $\{A, B\}$ since C is lower in the state hierarchy than A.

System Reaction

The usage of events as transition triggers imply the use of an asynchronous communication mechanism that assumes the presence of a FIFO event queue. This queue is used to store the events that are sent to an object whose statechart has not yet reached a stable state configuration (the RTC step has not finished). Events stored in the queue are dispatched (in the order of their arrival) to their destination once the respective RTC step has been completed.

The afore mentioned event queue is managed by an event dispatcher that is responsible for delivering the event in the top of the queue to the object the event was sent to. Upon receiving the event, the object consumes it according to the RTC step semantics.

We describe below the reaction of the system during a RTC step and the processing of an event [HK04]:

1. The dispatcher extracts event *ev* from the top of the event queue.

2. The transitions that have *ev* as triggering events and guards evaluating to true are identified. This is achieved by traversing the states in the active state configuration starting with the ones lowest in the hierarchy upwards. This step identifies a maximal set of non-conflicting transitions that can fire.

3. The transitions identified at step 2 are fired. This is done by exiting the source states of the transition, execute the exit action of the states in the order the states are exited. The states are exited according to their level in the state hierarchy - from low to high. The actions of the transition are sequentially performed followed by the entry actions of the target states, this time starting with the states higher in the hierarchy to the lower ones. For the newly entered composite states recursively perform the default transitions until reaching simple states. Considering orthogonal states, the transition execution order is not defined (thus implementation dependent). This means that statecharts with orthogonal states might contain non-deterministic behavior.

4. At this point the event has been consumed and the system is in a new active state configuration. This configuration however might not be a stable one. So there might exist completion transitions that can be fired. These are fired (without needing an explicit triggering event) according to the step 3 described above. This process is repeated until the system reaches a stable state when it is ready to accept the next event from the event queue.

Example 2.2.7. Consider again the example in Figure 2.7 and that the stable state configuration is $\{DrinkDispenser, Paying\}$ (which means that $Amt < 10$).

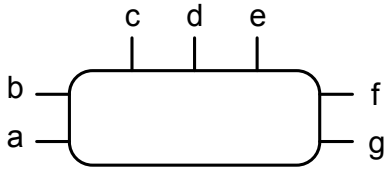

Figure 2.10: Lotos Specification as a black box

Consider also that the value of variable *Amt* equals eight. Upon receiving the event *InsertCoin*(5) the system exits state *Paying* (any exit actions would now be performed), updates the value of *Amt* to 13 and enters again state *Paying* (if there were any entry actions defined for the state these would be executed). Since the value of *Amt* is now greater than 10 the completion transition targeting state *Selecting* ([*Amt* >= 10]) needs to be fired. Thus according to the system reactions described above state *Paying* is exited and state *Selecting* is entered. At this point there are no more completion transitions that can be fired which means that the system is now in a stable state.

2.3 Language Of Temporal Ordering Specification

The Language Of Temporal Ordering Specification (LOTOS) [ISO89] is a formal description technique developed within ISO for the formal specification of open distributed systems.

LOTOS consists of a process algebraic [BK84, Hoa85] part based on Milner's Calculus of Communicating Systems (CCS) [Mil89, Mil82] and on Hoare's Communicating Sequential Processes (CSP) [Hoa85], and a data part based on the abstract data type (ADT) language ACT ONE [dMRV92].

In literature [BB87, FH92] the term **Basic LOTOS** is used to refer to the variant of the language that does not use any data values (defined by means of ADT) in its specifications. **Full Lotos** is used to refer to the variant using both the process algebraic and abstract data type parts. The semantics of the latter variant of the language builds on that of basic LOTOS by adding constructs and concepts enriching the language by allowing the use of data types for the description of systems. Since we use the full version of LOTOS, in the rest of this section we shall focus on this variant of the language.

The LOTOS model of a system can be thought of as a black box with a number of gates (interaction points) used for communication with its environment. Figure 2.10 [BB87] conceptually presents such a representation of a LOTOS specification communicating via the gates a through g with its environment.

A LOTOS specification describes a distributed system as a hierarchy of processes where each of the processes may contain further subprocesses. Such a

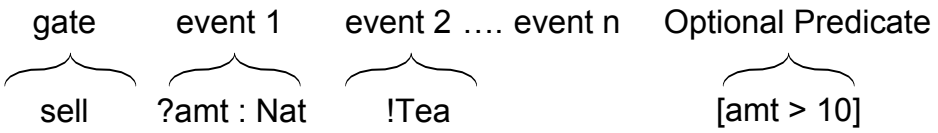

Figure 2.11: Structure of a LOTOS action

process represents "an entity able to perform internal, unobservable actions, and to interact with other processes, which form its environment" [BB87]. The interaction points used by the process for communicating with its environment are called gates. Such a gate represents an abstraction of an interface in an implementation.

In full LOTOS, an action has the structure presented in Figure 2.11 [FH92]. The upper part of the figure represents the structure while the lower part is an example of such an action ($sell\ ?amt : Nat\ !Tea\ [amt > 10]$). Thus an action is composed of a gate, a list of events and an optional predicate. An event can accept (?) or offer (!) values. The predicate (condition) has to be satisfied by the variables ($[amt > 10]$) accepted by the action. In case of basic LOTOS an action contains only the gate.

Example 2.3.1. Consider the example of the action in Figure 2.11 ($sell\ ?amt : Nat\ !Tea\ [amt > 10]$). This action alone would describe a communication through gate *sell* accepting a natural number ($?amt : Nat$) and offering the value $!Tea$ if the value of *amt* is greater than ten ($[amt > 10]$). Both Nat and the value Tea are defined by using ADT.

```
process name_proc [gate list] (parameter list) :
   functionality
    behavior expression
  where
    type definitions
    process definitions
endproc
```

Figure 2.12: Structure of a LOTOS process

The behavior of a process is specified by means of a behavior expression. Such expressions are created by using several LOTOS operators to combine actions and other behavioral expressions.

When the process communicates with its environment both the process and the environment participate equally to the action. This mechanism is called process synchronization and is based on the concept of process rendezvous. In order for

Table 2.1: LOTOS process synchronization types

Proc1	Proc2	Synchronization Condition	Interaction Type	Effect
g !E1	g !E2	value(E1) = value(E2)	value matching	synchronization
g !E1	g ?x:t	value(E1) is of sort t	value passing	x = value(E1)
g ?x:t	g ?y:u	t = u	value generation	x=y=v where v is a value of sort t

two (or more) processes to synchronize on an action they need to offer the same gate and "agree" on the gate events. Since an action can offer (**!**) or accept (**?**) values there are three types of interactions that can take place between two synchronizing processes. Table 2.1 [BB87] presents the three types of interaction for two processes *Proc1* and *Proc2*.

The structure of a LOTOS process is given in Figure 2.12 [BB87] where the name of the process *name_proc* is followed by the enumeration of the gates (*[gate list]*) and the list of parameters it uses. A process can have two types of functionality **exit** and **noexit**. The **exit** functionality denotes a process that can execute and exit thus terminating successfully while **noexit** represents a process that either does not terminate successfully or describes a behavior that will not terminate. The **behavior expression** describes the process behavior in terms of actions and other processes defined after the **where** keyword.

```
process drinkDispenser [g] (drinksLeft:Nat) : noexit

  g ?amt:Nat !Tea [(amt > 10) and (drinksLeft > 0)];
    drinkDispenser[g](drinksLeft-1)

endproc
```

Listing 2.1: Example of a LOTOS process for a simple drink dispenser

Example 2.3.2. Listing 2.1 contains the declaration of a process describing a simple drink dispenser selling only tea. It does not return any change in the case when more than the required amount of coins necessary for a tea has been inserted. The process *drinkDispenser* communicates with its environment through gate *sell* and has a parameter ($drinksLeft$) in order to keep track of the number of drinks left in the machine. In order to sell a tea ($!Tea$) the machine needs to have at least one more drink available ($drinksLeft > 0$) and receive a sufficient amount ($?amt : Nat$) of coins ($[amt > 10]$). The semicolon after the mentioned action is the LOTOS operator **action prefix** (which we will present later in this section)

```
specification spec_name [gate list] (parameter list) :
    functionality
    type definitions
  behavior
    behavior expression
  where
    type definitions
    process definitions
endspec
```

Figure 2.13: Structure of a LOTOS specification

and is used to represent sequential behavior. Thus, after the completion of the action, the process will be reinstatiated and the process parameter $drinksLeft$ will be updated to the value $drinksLeft - 1$.

The structure of a LOTOS specification (Figure 2.13) is similar to that of a process.

2.3.1 LOTOS Operators

The process behavior is specified through behavior expressions composed of actions and LOTOS operators. The operators are used to combine behavior expressions in order to form more complex expressions. As detailed information on the language and tutorials can be found in [ISO89, BB87, FH92], below we present only some of the LOTOS operators relevant to our work.

LOTOS offers two operators that can be considered behavioral expressions by themselves [FH92] : **stop** and **exit**. The **stop** operator is used to represent a deadlock situation where a process can not offer any action while **exit** denotes successful termination of an expression.

The **action prefix** operator represented as a semicolon "**;**" is used to describe sequential behavior. It composes an action with a behavioral expression denoting that after the action, the behavior continues according to the specified expression.

A generalization of the action prefix is the **enable** ("»") operator which has a similar semantic but is used when composing two behavioral expressions (as opposed to an action and an expression). When using this operator to compose two expressions, the second one will be executed only if the first one terminates successfully (by executing an **exit**).

The **choice** operator represented as "**[]**" offers a choice between two alternative behavior expressions. The choice is not necessarily a deterministic one.

Example 2.3.3. In Listing 2.2 we have enhanced the functionality of the drink dispenser machine. It still does not return any change if the user inserts more than the required amount of coins needed for purchasing a drink. It now allows

a **choice** ("**[]**") between purchasing a drink (coffee or tea), reseting and loading drinks into the machine. The **action prefix** is present in several places. One of them is on line 10 where it composes the action representing a reset request with that of returning the amount of coins that has already been inserted in the machine ($ret!amt$). Successful termination (line **exit**) follows after returning the coins and **enable**s the re-instantiation of the *drinkDispenser* process (11). The re-instantiation of the process on line 11 can only happen after a reset request (only in this case successful termination **exit** is executed).

```
1  process drinkDispenser [g, s, reset, ret, load]
       (teaLeft:Nat, coffeeLeft:Nat) : noexit
2    g ?amt:Nat;
3      (
4      [teaLeft > 0] -> s !tea [amt >= 10];
5        drinkDispenser[g, s, reset, ret, load](teaLeft-1,coffeeLeft)
6      []
7      [coffeeLeft > 0] -> s !coffeee [amt >= 15];
8        drinkDispenser[g, s, reset, ret, load](teaLeft,coffeeLeft-1)
9      []
10     reset; ret !amt; exit
11     )>> drinkDispenser[g, s, reset, ret, load](teaLeft, coffeeLeft)
12   []
13     load ?teaAmt:Nat ?coffee:Nat;
14     drinkDispenser[g, s, reset, ret, load]
           (teaLeft + teaAmt, coffeeLeft + coffeeeAmt)
15 endproc
```

Listing 2.2: LOTOS process describing a more complex drink dispenser

In LOTOS, parallelism is expressed through the operators **pure interleaving** ("|||"), **partial synchronization** ("|[< *gates* >]|") and **full synchronization** ("||").

When using the **interleaving** operator ("|||") to compose two processes (behavior expressions), the two behaviors are allowed to unfold completely and independently of each other by allowing all actions of the processes to interleave in any order.

Example 2.3.4. Consider the already introduced *drinkDispenser* process is composed by **interleaving** with another process representing a snack vending machine - $drinkDispenser[g, s, reset, ret, load](teaLeft : Nat, coffeeLeft : Nat) \; ||| \; snackDispenser[sell, reset, ret, load](chocolateLeft : Nat, peanutLeft : Nat)$. All the actions of the two machines are allowed to interleave. Thus, the functioning or the *drinkDispenser* is independent of that of the *snackDispenser* and vice versa.

We already mentioned that processes communicate with each other through gates and depending on the gate offerings there are several types of interactions (see Table 2.1) allowed for such a communication to succeed.

If the situation arises when two processes need to collaborate on some of their actions we can use the **partial synchronization** ("$|[< gates >]|$") operator. The term $< gates >$ in ("$|[< gates >]|$") is a place holder for the names of the gates on which the composed processes need to synchronize in order for the behavior to continue. The actions containing other gates not listed in the operator are allowed to freely interleave. The set of possible actions of the expression is composed of the set of all actions (from both processes) occurring at gates not in the operator and the set of actions the processes are able to synchronize on at gates listed inside the **partial synchronization** operator. In the case when one of the processes can only evolve by executing an action at one of the synchronization gates it will have to wait until the second process synchronizes on that particular action.

```
1  process buyer[g, s, talk, eat]
       (cash:Nat) : noexit
2    talk !phone;
3    (
4    [cash >= 10] -> g !10; s !tea;
5      buyer[g, s, talk, eat](cash - 10)
6    []
7    [cash >= 15] -> g !15; s !coffeee;
8      buyer[g, s, talk, eat](cash - 15)
9    )
10   []
11   eat !chocolate; talk;
12    buyer[g, s, talk, eat](cash)
13 endproc
```

Listing 2.3: LOTOS process describing a buyer

Example 2.3.5. Consider the process *buyer* from Listing 2.3 describing the behavior of a buyer that can either talk on the phone (line 2) and then purchase a drink or eat a chocolate and then talk (line 11). In order for the buyer to purchase a drink it needs to synchronize with the *drinkDispenser* on the gates $[g, s]$ - $drinkDispenser[g, s, reset, ret, load](teaLeft : Nat, coffeeLeft : Nat)$ $[g, s]$ $buyer[g, s, talk, eat](cash : Nat)$. When both processes are first initialized, the *drinkDispenser* can only execute the actions at gate *load* since the buyer has not yet performed the $talk!phone;$ action. Thus all actions occurring at gates *talk*, *eat*, *reset*, *ret* and *load* are allowed to interleave.

If the set of gates inside the **partial synchronization** operator contains all the gates of the composed processes the **partial synchronization** becomes **full**

synchronization. This operator is used when the processes need to cooperate on all their actions. **Full synchronization"** ("||") denotes the fact that the actions which occur in either of the behavior expressions have to synchronize. Thus the composition contains all actions on which the processes can synchronize on at all communication gates.

Example 2.3.6. Consider again the process of the buyer in Listing 2.3) and consider also that the line 2($talk!phone$;) has been deleted from the process definition. Now, the behavior of expression $drinkDispenser[g, s, reset, ret, load](teaLeft : Nat, coffeeLeft : Nat) \;||\; buyer[g, s, talk, eat](cash : Nat)$ will allow the selling of drinks. This is so since the actions at gates g and s are the only ones offered by both processes. The behavior will continue until either there are no more drinks in the drink dispenser or the buyer runs out of money. All other actions at gates other than g and s are not part of the behavior since they are offered by only one of the processes and never by both at the same time.

Other LOTOS operators and concepts not detailed in this work and more information about the language can be found in [BB87, FH92, ISO89]. As a description of algebraic specification of data types is outside the scope of this work we refer the interested reader to [Mañ88a, Mañ88b, EM85, EM90].

Chapter 3

Modeling

The behavior of the considered type of systems is described by means of asynchronously communicating UML statecharts. By asynchronous it is understood the fact that the events the statecharts use to communicate with each other are enqueued in an FIFO queue and consumed by the targeted statechart after it reaches a stable state. The events used in the communication with and within the system can also carry data values as event parameters.

As already mentioned in Section 2.1.1 a key point in using MBT is the model itself. It is very important that the model resides at a higher level of abstraction than the SUT. It is also important to use appropriate abstractions during the modeling process. However the resulting model needs to retain enough information about the described behavior of the SUT to enable it to be used for test case generation.

Since abstractions are so important for MBT, in this chapter we further discuss this topic in Section 3.1. In Section 3.2 we present a running example and in Section 3.3 we shortly discuss other models used in our experiments.

3.1 Model Abstractions

By abstractions we understand simplifications of the model. These make the model easier to understand and validate. Usually domain knowledge is required in order to find good abstractions so that the resulting model is an appropriate approximation of the SUT with regard to the aspects that need to be tested.

It is possible to identify the following types of abstractions that can be used during the modeling process of MBT [PP05]:

- **Functional abstractions** are used with the goal of focusing the test generation only on certain parts of the functionality of the SUT by not specifying

other parts (considered not important for the purpose at hand). An advantage of this abstraction in the context of MBT is the fact that several models can be built (each containing only a part of the functionality of the SUT) in order to test each functionality separately. An example for functional abstractions is also in our running example (Section 3.2) where the used model describes only a part of the functionality dedicated to direction indication on vehicles. The modeled behavior is focused on the emergency operation of the flash light backup controller.

- **Data abstractions** assume the mapping of concrete data types to logical or abstract data types. One common technique is to represent only equivalence classes and not the concrete data values. An example of such an abstraction is also used in one of our models describing the behavior of the Keyless Entry functionality in modern vehicles. In that model the locking and unlocking of the vehicle also depends on its speed. Thus if the speed value is greater than 20 km/h the car will be locked. Otherwise, depending on the location of the keys, the vehicle might be locked or unlocked. The equivalence classes for the value of the speed are $speed <= 20$ and $speed > 20$.

- **Communication abstractions** are used in order to map a complex interaction of the SUT to a more simple operation. In our case such an abstraction relates to the fact that real communication messages are mapped to data carrying events.

- **Temporal abstractions** consider that only the order of events is important for the functionality at hand. Thus the precise timing of such events is abstracted away. An example would be that of abstracting from a concrete timer by only considering its start and finish as two distinct events.

All of the abstractions mentioned above imply a loss of information. Thus, the models built using such abstractions can only be used for testing those parts of the SUT's behavior that are specified [PP05].

3.2 Running Example

In this section we present the running example used in the rest of this work. It describes part of the functionality dedicated to direction indication (blinking) on cars. As mentioned above, the structure of the system is given in the form of a class diagram (Figure 3.1) where each model is represented by means of a class. The behavior of each class is described by a statechart nested inside it handling different parts of the functionality.

Please note that due to confidentiality and readability issues the running example is not the actual industrial model but a simplified version of it. It is

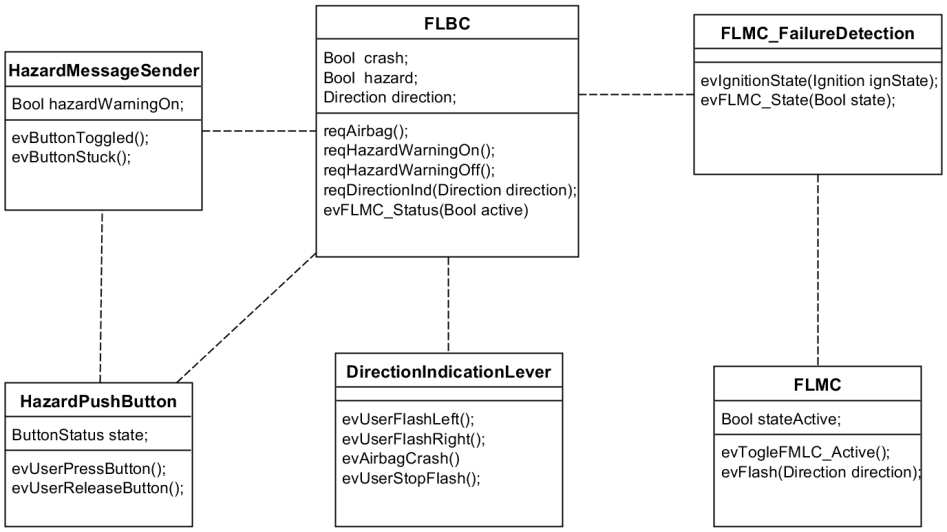

Figure 3.1: Class diagram for the Direction Indication system

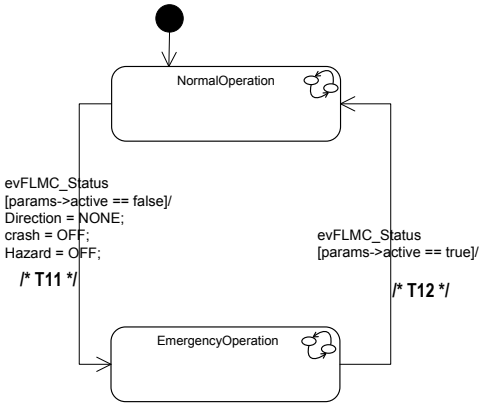

Figure 3.2: Statechart of the FLBC Class

a generic version of a blinker system encoding general functionality present on modern vehicles. The model used in our experiments is however the model from our industrial partner.

The structure of the system can be observed in Figure 3.1 and is composed of several models handling different parts of the functionality. The modeled behavior is focused on the emergency operation of the Flash Light Backup Controller (*FLBC*). The role of the *FLBC* is to provide basic functionality of the flashers in case the Flash Light Master Controller (*FLMC*) fails. The provided functionality contains direction indication, hazard warning and crash flashing.

In Figure 3.2 the symbols in the right conner of some states (e.g. *EmergencyOperation*) denote the fact that those are composite states containing further substates and transitions. The substates of the *EmergencyOperation* state in

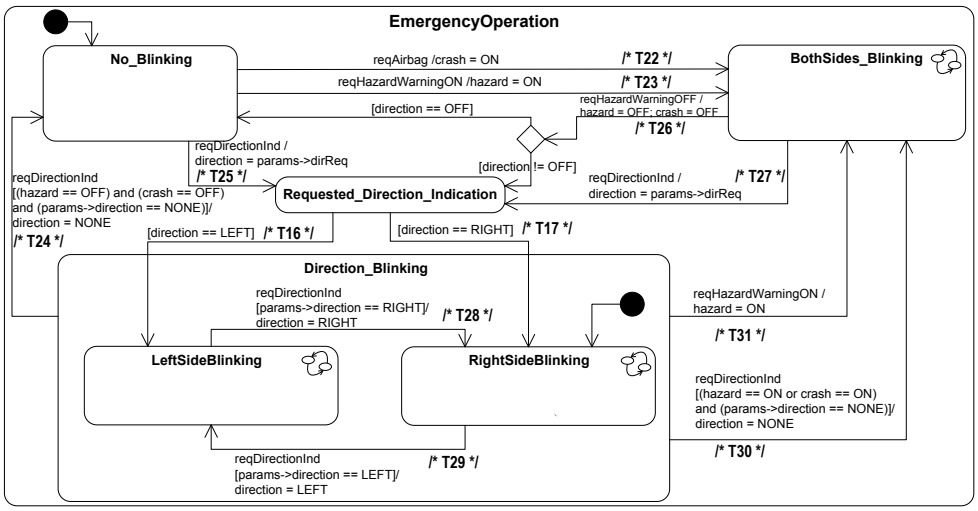

Figure 3.3: EmergencyOperation state

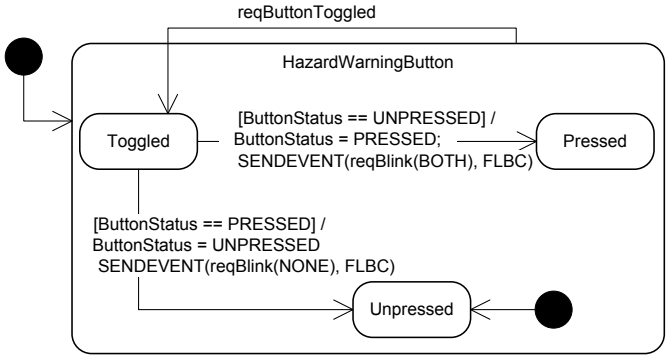

Figure 3.4: Statechart of the HazardWarningButton class

Figure 3.2 are depicted in Figure 3.3.

The *HazardPushButton* model describes the behavior of toggling the hazard button while the *HazardMessageSender* issues requests (via *reqHazardWarning* event) to the *FLBC* depending on the status received from the *HazardPushButton*.

The *DirectionIndicationLever* model represents the user activation of the direction indication lever and generates flashing requests to the *FLBC* accordingly. These models represent the interaction of the user with the system.

In the case of the *FLMC* the model describes only the situation when the *FLMC* fails and not the whole functionality of the controller. After this controller fails, the state of the *FLBC* changes (via the *evFLMC_ status* event) to *EmergencyOperation* state (see Figure 3.2). While in this state, after receiving requests for either hazard or crash warning the new state will be *BothSides_ Blinking* rep-

resenting the activation of all flashlights on the vehicle. The system can leave this state through a request for direction indication - *reqDirectionInd(LEFT/RIGHT)* or a *reqHazardWarningOFF()*. In the case of a direction indication request the flashing functionality shall continue only on the requested side (*LeftSideBlinking* or *RightSideBlinking*). After the direction indication has ceased (*reqDirectionInd(NONE)*), the system must be returned to the state previous to *Direction_ Blinking*.

3.3 Other Models

In the rest of this work we are going to use several models when we evaluate the proposed techniques (LOTOS transformation, test case generation and using dependence relations for improving test purposes). In this section we shortly present these models and discuss the applied abstractions. The first three models (Direction Indication, Diagnosis and Keyless Entry) are from our industrial partners and the others are taken from literature.

Direction Indication. The model of the directional indication was described above. The used abstractions are :

- Functional abstractions - the model focuses only on the emergency operation of the ECU controlling the blinking functionality. Thus not all the blinking functionality is specified.
- Communication and data abstractions - the actual messages the statecharts use to communicate are encoded into data carrying events. At the concrete level these are complex messages used by the actual communication network in the vehicle.

Diagnosis. The Diagnosis model [CSP09a] describes the diagnosis functionality of vehicles. Its purpose is to store the type, occurrence, and origin of errors during operation of the vehicle. Besides the diagnosis functionality the system also contains a model describing the behavior of the ignition switch of the vehicle and two other models defining the conditions needed for errors to be detected. When such an error has been detected it is communicated to the diagnosis model. The functionality, which is to be tested in this setting, defines how detected errors are to be treated and when to create an entry in the error memory.

Similar to the Direction Indication model communication and data abstractions are used here for mapping actual network messages to data carrying events.

Keyless Entry. This model [Sch12] describes the keyless entry functionality on modern vehicles. It is intended to allow a user to lock, unlock, enter and start the vehicle without using the remote of the car or its key. For example, if the key is inside the car the driver can start the vehicle without using the key to turn on the ignition switch. The model contains three communicating statecharts. As already

mentioned one functionality of the system is that the car will be automatically locked if its speed exceeds 20 km/h. Used abstractions:

- Data abstraction - we use equivalence classes for the value of the speed $speed <= 20$ and $speed > 20$.
- Communication and data abstractions - the actual messages the statecharts use to communicate are encoded into data carrying events.

Telephony Control Protocol. - This protocol [SIG13] defines how voice and data calls are set up between Bluetooth devices. It defines the call states between incoming and outgoing sides. These represent the two entities that take part in the call. We modeled the full specification of the protocol as opposed to the minimal subset of states (for use within power and memory restricted devices). For this model we used communication and data abstractions to map the actual communication messages to data carrying events.

Conference Protocol. This is the model of a multicast chatbox protocol [BFd+99]. It allows the participants of a conference to exchange messages. The participants of the conference can change dynamically by leaving and joining the conference.

For the model we used the following abstractions:

- Functional abstractions - Only one conference at a time is considered and only two participants are taking part in the conference.
- Data abstractions - The actual values of the messages exchanged between the participants are abstracted away. We use a constant value for it. Also the actual addresses and IDs of the participants are mapped to numerical values.
- Communication and data abstractions - the actual data units used to communicate between the conference protocol entity and the participants were mapped to data carrying events.

Loan Approval Web Service. This model [ZZK07] describes the behavior of a loan approval service that accepts loan requests from costumers. Once such a request has been received, depending on the requested amount, the system will approve or reject the request.

For the model we used the following abstractions:

- Functional and data abstraction - the evaluation of the request is based only on the requested amount. In a concrete implementation, the approval of such a request might involve a real loan expert in order to check the risk involved by individuals requesting the loan.

- Data abstraction - we limited the amount that can be requested to 150 and defined as limit for the invocation of an approver the amount of 130. Amounts smaller than that are approved.

Microwave Oven. This is a simple model [KDB11] of a microwave oven composed of two statecharts. One represents the way a user interacts (e.g. setting cooking time) with the oven and the other one describes how the system reacts when the user opens or closes the door.

In Table 3.1 we present some statistics regarding the models mentioned above. Thus the first column contains the name of the model while column $SCNo$ presents the number of communicating statecharts of the model. Columns $TrNo$ and $StNo$ contain the number of transitions and states for the respective model.

Table 3.1: Model Statistics

Model	$SCNo$	$TrNo$	$StNo$
Flasher	6	34	14
Diagnosis	4	38	17
KeylessEntry	3	35	22
MicrowaveOven	2	34	12
LoanApprovalWS	2	22	15
ConferenceProtocol	3	41	18
TelCtrlProtocol	2	55	26

Chapter 4

From UML Statecharts to LOTOS

Parts of this chapter have been published in "From UML Statecharts to LOTOS: A Semantics Preserving Model Transformation" [CSP09a] which is joint work with Christian Schwarzl and Bernhard Peischl, "Abstracting Timing Information in UML Statecharts via Temporal Ordering and LOTOS" [CW11] which is joint work with Franz Wotawa.

UML statecharts are often used in industry for constructing models of software systems. These models can be utilized in several ways including analysis of the system, code generation and even test case generation. They enjoy a high expressive power allowing the modeling of behaviors by use of concepts closer to the way of thinking of the human mind. Such concepts refer to hierarchy, concurrency, pseudostates, reactions to external events and also timing. But statecharts do not have a formal semantics and in order for automatic verification techniques to be applied, they need to be translated into a representation enjoying such formal semantics.

Since our goal is the automatic application of test case generation techniques, the existence of a well founded testing theory and the appropriate tool support [JJ05] is a big decision factor when choosing the formalism for the transformation. There are many examples of the successful use of the CADP framework in industrial settings. We have chosen LOTOS since it is the main input language for the tools in the CADP toolbox and also offers enough expressive power for representing the components and operational semantics of UML statecharts.

This chapter describes the transformation rules used to derive a LOTOS specification from the UML description of a distributed system. In Section 4.1 we provide an overview of the transformation process. Section 4.2 presents the first step of the transformation - the flattening of the statecharts while the transfor-

mation of the flattened statecharts into a LOTOS description is given in Section 4.3.

4.1 Overview

The first step of the transformation is represented by the flattening of the statechart in which the hierarchical structures and the pseudostates are removed. This step delivers a behavioral equivalent statechart described only in terms of simple states and transitions $SC = (S_s, T, V, i_{ps0})$.

During the flattening process transition copies are created for the transitions originating from composite states. Each such transition generates a copy of itself for each state contained by its source state.

After the flattening, each statechart in the system is mapped to a LOTOS process while the variables used in the statecharts become LOTOS process parameters. These processes are then composed using the interleaving operator ("|||"). Each such process (representing a statechart) will contain several subprocesses used to represent the states in the statechart. Every subprocess offers choices between several behavioral expressions generated from the transitions originating from the represented state. Such an expression preserves the id, triggering event, guard and action of the transition. Once such a behavior expression has been triggered, all the components of the transition (event, guard, action - value assignments and/or generation of events to other models) are executed.

During the transformation process, LOTOS abstract data types are used in order to preserve the traceability between the components of the UML model and the generated LOTOS constructs. Thus in the generated specification we can still identify the statecharts, states, transitions, triggering events and variables used. This information is required in order to map the generated test cases back to the original UML model.

The kind of systems we employ are asynchronously communicating distributed systems. So we also need to treat aspects such as asynchronous communication and run to completion step among communicating statechart models. For this purpose an extra process is inserted into the specification of the model. This process fully synchronizes with all the other models and dictates the communication mechanism with and within the system.

4.2 Flattening Statecharts

Transforming statecharts into a formal representation is not something new. The transformation process depends also on the targeted formalism. Usually the first step in such transformations [DMY02, BIKT01, RKRT01] is the elimination of hierarchy and pseudostates thus obtaining a flattened representation of the stat-

echart. After this step the behavior is expressed only in terms of simple states and transitions. The final formal representation is then obtained from this representation of the statechart.

In the literature there are two directions that have different understandings for the hierarchy concept during the flattening of a statechart. One of these defines statechart flattening as the removal of hierarchy and retention of concurrency [Was04, DMY02, BLA+02] thus obtaining a representation as a set of Mealy machines that operate concurrently.

The second view [BIKT01, SP10] regarding the hierarchy and flattening of statecharts lays closer to the semantics we described in Section 2.2. It views flattening as the elimination of hierarchy and also of concurrency. It is true that this second view suffers from exponential growth regarding the number of substates in orthogonal states but our models are aimed at being deterministic ones and also simpler (at a more abstract level than the implementation) and thus complex models involving a high number of orthogonal states might indicate a model that is too detailed to be used in MBT.

In our setting, the flattening of the statecharts is according to the second definition of the concept described above. In the absence of orthogonal states, the flattening process does not increase the number of states, it might actually reduce it since composite states are removed during this process. The number of transitions on the other hand increases due to the elimination of hierarchy structures and of pseudostates. New transitions are added to the model in order to preserve the behavior that would be lost by removing hierarchy and pseudostates.

The output of the flattening process is a homogeneous structure composed of simple states and transitions (see equation 4.1) preserving the behavior of the original hierarchical representation. The transition set T_{fl} in equation 4.1 represents the set of transitions generated during the flattening process.

$$\begin{aligned} flatten(SC) &= SC_{FL} \\ SC &= (S, T, V, i_{ps0}) \\ SC_{FL} &= (S_s, T \cup T_{fl}, V) \end{aligned} \tag{4.1}$$

In the rest of this section, when referring to the statecharts we will use the formal notations introduced in Section 2.2 namely in Definition 2.2.1. The definition describes an UML statechart as a tuple $SC = (S, T, V, i_{ps0})$ where $S = S_s \cup S_c \cup S_{ps}$ represents the set containing simple, composite states as well as pseudostates, T is the set of transitions, V that of the variables while i_{ps0} is the initial pseudostate (indicating the initial state of the statechart). The pseudostates set $S_{ps} = I_{ps} \cup J_{ps} \cup En_{ps} \cup Ex_{ps} \cup C_{ps} \cup H_{ps}$ contains the pseudostates (initial,junction, entry, exit, condition, history) of the statechart.

Input: j_{ps} - the junction pseudostate
Output: T_{FL_j} - a set of transitions replacing the use of j_{ps} in the statechart
1: $t_{out} \in Out_{tr}(j_{ps})$ //the outgoing transition of j_{ps} - $|Out_{tr}(j_{ps})| = 1$
2: **for all** $t_{in} \in In_{tr}(j_{ps})$ **do**
3: $target(t_{in}) = target(t_{out})$
4: $guard(t_{in}) = guard(t_{in}) \wedge guard(t_{out})$
5: $action(t_{in}) = action(t_{in}).action(t_{out})$
6: $T_{FL_j} = T_{FL_j} \cup t_{in}$
7: **end for**
8: $remove(j_{ps})$
9: $remove(t_{out})$

Figure 4.1: Algorithm for the removal of Junction pseudostates

Our approach for the flattening of statecharts is based on previous work in [SP10].The algorithm consists of two phases : removing pseudostates and eliminating hierarchy. We describe these steps separately below.

4.2.1 Removing Pseudostates

The first phase of the flattening process is the removal of pseudostates. Since most of the pseudostates also have a semantic meaning, care needs to be taken in order not to miss any of the behavior modeled with the help of pseudostates. The removal process consists of enumerating through all of the pseudostates and calling the appropriate removal function. We describe the mentioned functions below.

Initial pseudostates j_{ps} are eliminated by adding the action of their outgoing transition t_{out} to the actions of its target state s_{trg} ($entryAct(s_{trg}) = action(t_{out}).entryAct(s_{trg})$) and marking the state ($isInitial(s_{trg}) = true$) as an initial state. This information will be used later in the hierarchy removal step. From here on we will call such states initial states.

Junction pseudostates are semantic free nodes used to merge incoming transitions into a single outgoing transition. The equivalent behavior of such a merge is that all incoming transitions target the same state.

The algorithm in Figure 4.1 describes the steps used for the removal of such nodes. The functions $In_{tr}()$ (equation 4.2) and $Out_{tr}()$ (equation 4.3) used in the algorithm return the set of transitions targeting and respectively originating from the state passed as argument (j_{ps} in this case).

$$In_{tr} : S \to T$$
$$In_{tr}(s) = \left\{ \bigcup_{i=1..n} t_i | s \in S \wedge target(t_i) = s \right\} \tag{4.2}$$

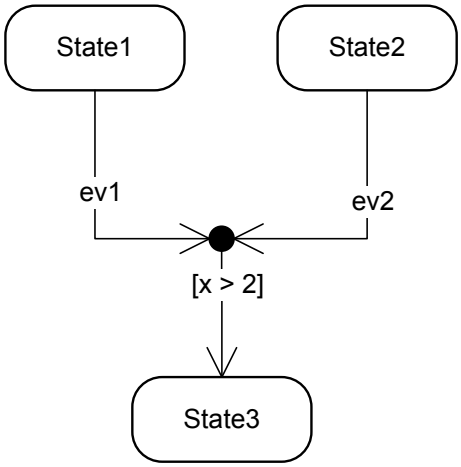

Figure 4.2: Junction node

$$Out_{tr} : S \to T$$
$$Out_{tr}(s) = \left\{ \bigcup_{i=1..m} t_o | s \in S \wedge source(t_o) = s \right\} \tag{4.3}$$

The elimination of the junction node j_{ps} assumes the *disconnection* of each incoming transition t_{in} from the junction node and reconnecting them to the target state of the outgoing transition (line 3). The guard of t_{in} is obtained uniting by conjunction its own guard with the guard of the outgoing transition t_{out} of j_{ps}. The actions of t_{out} are then appended to the actions of each of the newly connected transitions t_{in}. After these modifications, the pseudostate node and its outgoing transitions can be removed from the statechart. Figure 4.2 shows a statechart containing a junction node while Figure 4.3 the behaviorally equivalent version of the same statechart after the elimination of the junction node according to the described algorithm.

The elimination of **choice** pseudostates (algorithm in Figure 4.4) is similar to that of the junction nodes. In this case the source of the outgoing transitions of the choice pseudostate becomes the source of its incoming transition. The guards and actions of the new transitions are obtained by adding those of the outgoing transitions to the incoming transition guard and actions respectively. The choice node and its incoming transitions are removed from the statechart after all outgoing transitions have been modified.

History nodes are used to represent the most recent active state configuration of the composite state that directly contains the pseudostate. There are two types of history nodes. Shallow history is used to store the last active substate (but not the substates of that substate) of the state containing the pseudostate. Deep

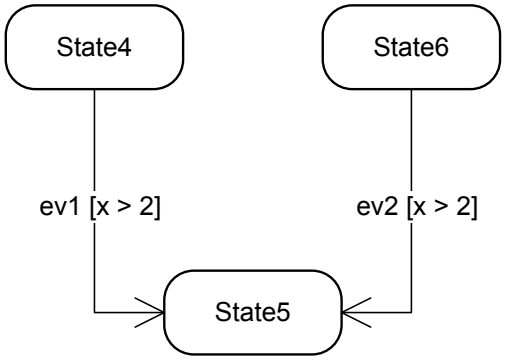

Figure 4.3: Removing Junction pseudostates

Input: ch_{ps} - the choice pseudostate
Output: $T_{FL_{ch}}$ - a set of transitions replacing the use of ch_{ps} in the statechart
1: $t_{in} \in In_{tr}(ch_{ps})$ //the incoming transition of ch_{ps} - $|In_{tr}(ch_{ps})| = 1$
2: **for all** $t_{out} \in Out_{tr}(ch_{ps})$ **do**
3: $source(t_{out}) = source(t_{in})$
4: $guard(t_{out}) = guard(t_{in}) \wedge guard(t_{out})$
5: $action(t_{out}) = action(t_{in}).action(t_{out})$
6: $T_{FL_{ch}} = T_{FL_{ch}} \cup t_{out}$
7: **end for**
8: $remove(ch_{ps})$
9: $remove(t_{in})$

Figure 4.4: Algorithm for removing Condition pseudostates

history keeps track of the last active state in the state hierarchy of its parent state.

Intuitively, a transition targeting a history node can be replaced through a number of transitions targeting the child states in the parent state. Depending on the type of history node the considered child states are at the same level (simple or composite) or are all the simple states (on all hierarchical levels) contained by the parent state of the history node. The new transitions share the source of the one targeting the history node. Such a transition fires when it receives the appropriate event, its guard evaluates to true and its target state is the last active child state of the state containing the history node. Only one such transition can fire at one time.

Figure 4.6 contains the algorithm used for removing a shallow history pseudostate hs_{shl} from a statechart. Thus for each transition t_{in} targeting hs_{shl} (line

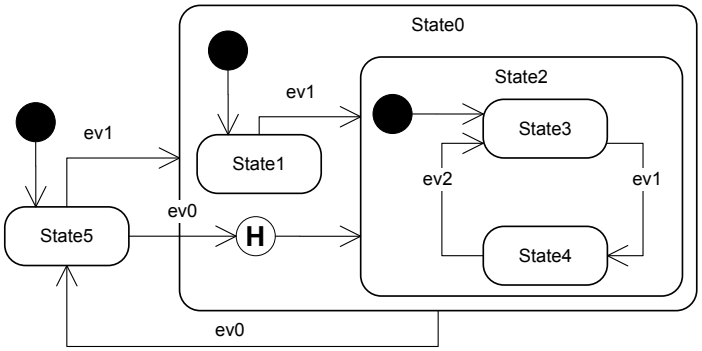

Figure 4.5: Example of a statechart containing a History pseudostate

Input: hs_{shl} - the shallow history pseudostate
Output: $T_{FL_{hs}}$ - a set of transitions replacing the use of hs_{shl} in the statechart

1: $t_{out} = getOutTr(hs_{shl})$
2: **for all** $s_{sbl} \in \{S - S_{ps}\} | parent(s_{sbl}) = parent(hs_{shl})$ **do**
3: **for all** $t_{in} \in In_{tr}(hs_{shl})$ **do**
4: $t_{new} = newTransition()$ //create new transition
5: **if** $type(s_{sbl}) == composite$ **then**
6: $st_{targ} = init(s_b)$
7: **else**
8: $st_{targ} = s_{sbl}$
9: **end if**
10: $source(t_{new}) = source(t_{in})$
11: $target(t_{new}) = st_{targ}$
12: $event(t_{new}) = event(t_{in})$
13: $action(t_{new}) = action(t_{in})$
14: $guard(t_{new}) = \text{“}(v_{hst} == st_{targ})'' \wedge guard(t_{in})$
15: $T_{FL_{hs}} = T_{FL_{hs}} \cup t_{new}$
16: **end for**
17: $entryAct(s_{targ}) = entryAct(s_{targ}).\text{“}v_{hst} = st''_{targ}$
18: **end for**
19: $remove(In_{tr}(hs_{shl}), t_{out}, hs_{shl})$

Figure 4.6: Algorithm for removing Shallow History pseudostates

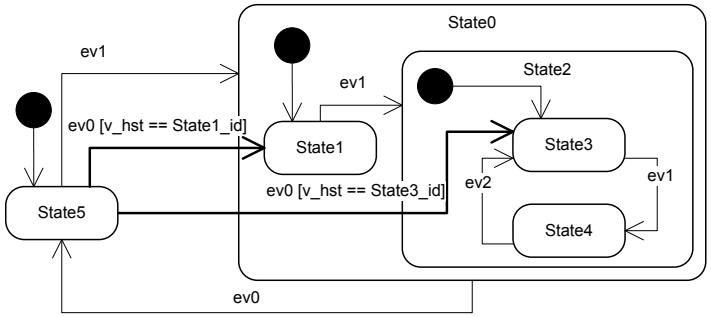

Figure 4.7: Removed Shallow History pseudostate

3) a number of transitions equal to the number of sibling states (line 2) of hs_{shl} will be generated. The newly created transitions t_{new} (line 4) will have the same source, triggering event and action as t_{in}. The target of t_{new} is computed depending on the type of the sibling state (line 5). If the sibling state s_{sbl} is a composite state, the target state of t_{new} is the initial state of s_{sbl}. Otherwise s_{sbl} itself is the target of t_{new}.

In order to keep track of last active state, we insert a new variable v_{hst} in the statechart model (line 17). This variable is updated every time a state s_{sbl} inside the $parent(hs_{shl})$ state is entered. If the state is a simple state v_{hst} is updated to point to the state itself. If the state is a composite state the variable will always point to the initial state of s_{sbl}. After the creation of the new transitions and updating of the appropriate entry actions, the history pseudostate and its outgoing transition are removed.

Example 4.2.1. Consider the statechart in Figure 4.5. After applying the algorithm in Figure 4.6 we obtain the representation in Figure 4.7. The two thicker transitions connecting $State5$ with $State1$ and $State3$ respectively are the ones replacing the use of the removed shallow history pseudostate. For readability reasons the entry and exit actions of the states are not explicitly depicted.

The algorithm used for removing **deep history** pseudostates in Figure 4.8 is similar to that for shallow history. The difference is that this time the active state configuration that needs to be recreated takes into consideration all nesting levels inside the parent of the node. Thus all simple states (at all hierarchical levels) inside the parent of the history node ($parent(hs_{deep})$ are targeted by the newly created transitions. This is achieved by recursively going through all the states of $parent(hs_{deep})$.

Example 4.2.2. Consider the same statechart in Figure 4.5 but this time consider that it contains a deep history node. After applying the algorithm in Figure

Input: hs_{deep} - the deep history pseudostate
Output: $T_{FL_{hs}}$ - a set of transitions replacing the use of hs_{deep} in the statechart

1: $t_{out} getOutTr(hs_{shl})$ //$|Out_{tr}(hs_{deep})| = 1$
2: **for all** $st \in \{S - S_{ps}\} | parent(hs_{deep}) \in ancestors(st)$ **do**
3: **if** $type(st) == simple$ **then**
4: **for all** $t_{in} \in In_{tr}(hs_{deep})$ **do**
5: $t_{new} = newTransition()$ //create new transition
6: $source(t_{new}) = source(t_{in})$
7: $target(t_{new}) = st$
8: $event(t_{new}) = event(t_{in})$
9: $action(t_{new}) = action(t_{in})$
10: $guard(t_{new}) =$ "$(v_{hst} == st)'' \wedge guard(t_{in})$
11: $T_{FL_{hs}} = T_{FL_{hs}} \cup t_{new}$
12: **end for**
13: $entryAct(st) = entryAct(st).$"$v_{hst} = st''$
14: **else**
15: continue from line 2
16: **end if**
17: **end for**
18: $remove(In_{tr}(hs_{deep}), t_{out}, hs_{deep})$

Figure 4.8: Algorithm for removing Deep History pseudostates

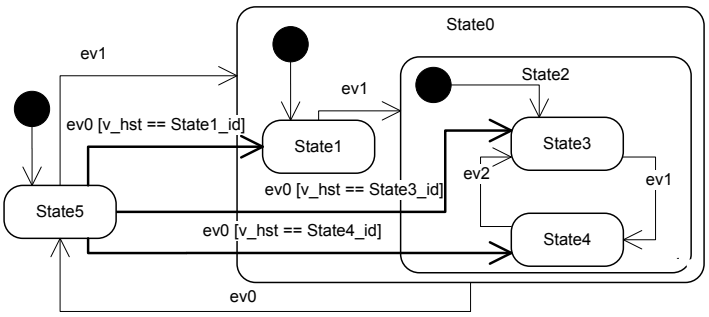

Figure 4.9: Removed Deep History pseudostate

4.8 we obtain the representation in Figure 4.9. This time the history node is replaced by the three thicker transitions connecting $State5$ every simple state inside $State0$. Again for readability reasons the entry and exit actions of the stats are not explicitly depicted.

We remove **terminate** pseudostates by replacing them with a simple state with no outgoing transitions and marking it as a final state. We do this since later when removing the hierarchical structures we will need to differentiate transitions targeting such states from transitions targeting other simple states. Upon triggering a transition targeting a final state no other states are exited (no exit

actions are performed). The behavior obtained this way is that of a system that can not evolve any more.

In order to remove **exit** and **entry** points we use an algorithm similar to that for removing junction pseudostates (see Figure 4.1). The algorithm is similar also due to the fact that entry and exit nodes have only one outgoing transition and may have several incoming ones. Since the outgoing transitions of such nodes do not have a triggering event, guard or actions, we simply disconnect every such transition from the pseudostate and reconnect it to the state targeted by the outgoing transition of the entry/exit node. After this step the entry/exit nodes and their outgoing transitions can be deleted.

4.2.2 Removing Hierarchy

The second phase of the flattening process is the removal of hierarchical structures from the statechart. At this point, all pseudostates have already been removed and the statechart models contain only states (simple and composite) and transitions $SC = (S_s \cup S_c, T, V, st_{init})$ where st_{init} is the initial state.

The algorithm used for the removal of hierarchy is presented in Figure 4.10 and consists of two steps.

The first step generates new transitions to replace all incoming and outgoing transitions of composite states. Internal actions of states and transition conflicts are also handled in this phase:

- Move the during actions defined for the states in the statechart to new transitions. As already mentioned, these actions have the form of a normal transition and they must have an explicit triggering event. The firing of such a transition does not cause its containing state to be exited but its actions are fulfilled every time it receives its triggering event and its guard evaluates to true. We treat such actions by generating for each of them a transition looping on the state containing the action (line 3). The label of such a transition is the internal action itself. The difference between these transitions and the other ones is the fact that during the hierarchy removal process, the entry and exit actions of the states they enter/exit are not added to the action part of the transition (because internal actions do not exit/enter states).

- For every outgoing transition $tr_{out} \in Out_{tr}(st)$ of a composite state st a new transition will be created for every simple substate for which st is an ancestor state (line 9). The newly created transitions will have as source state the substate for which it has been created. The rest of the elements of such a transition are an exact copy of tr_{out}. The rationale is that when fired, a transition originating from a composite state exits from all active

Input: SC - the statechart without pseudostates
Output: SC_{flat} - the flattened representation of SC

1: $T_{during} = \emptyset$ //set of transitions replacing during actions in states
2: **for all** $st \in S$ **do**
3: **for all** $act_{in} \in duringAct(st)$ **do**
4: $t_{drng} = newTransition(st, event(act_{in}), guard(act_{in}), action(act_{in}), st)$
5: $T_{during} = T_{during} \cup t_{drng}$
6: **end for**
7: **if** $st \in S_c$ **then**
8: $createDuringTransCopies(T_{during}, simpleSubstates(st))$
9: **for all** $tr_{out} \in Out_{tr}(st)$ **do** //create transition copies
10: $createTransCopies(Out_{tr}(parent(st)), simpleSubstates(st))$
11: **end for**
12: **for all** $tr_{in} \in In_{tr}(st)$ **do**
13: $recursiveReconnect(tr_{in}, initialSimpleState(st))$
14: **end for**
15: **end if**
16: **end for**
17: $removeAll(tr_{out} \in Out_{tr}(s) | s \in S_c)$
18: $resolveTransitionConflicts(SC)$
//adding entry and exit actions
19: **for all** $st \in S_s$ **do**
20: **for all** $tr_{out} \in Out_{tr}(st) \wedge \neg isDuringTrans(tr_{out})$ **do**
21: $ss_{prnt} = parent(st)$
22: **while** $ss_{prnt} \neq firstCommonAncester(st, target(tr_{out}))$ **do** //complete outgoing transitions
23: $actions_{exit} = actions_{exit}.exitAct(ss_{prnt})$
24: $ss_{prnt} = parent(ss_{prnt})$
25: **end while**
26: $actions(tr_{out}) = actions_{exit}.actions(tr_{out})$
27: $st_{prnt} = parent(target(t_{out}))$
28: **while** $st_{prnt} \neq ss_{prnt}$ **do** //complete incomming transitions
29: $inActions = entryAct(st_{prnt}).inActions$
30: $st_{prnt} = parent(st_{prnt})$
31: **end while**
32: $actions(tr_{out}) = actions(tr_{out}).inActions$
33: **end for**
34: **end for**
35: $removeAll(s \in S_c)$

Figure 4.10: Algorithm for elimination of the hierarchical structure of a statechart

substates of its source state. Thus it can be fired from any substate (on all levels) in its source state.

- All incoming transitions $tr_{in} \in In_{tr}(st)$ of a composite state st will be disconnected from st and connected to the first initial simple state of st (line 13). This is achieved by recursively enumerating through the initial composite states of st until encountering an initial simple state. The rationale this time is that an incoming transition will activate recursively the initial states of the composite state it targets.

- During the flattening process we also take care of solving **transition conflicts** (line 18) according to the rules already described in Section 2.2.2. Two transitions are in conflict if they have the same triggering event and if the firing of either of them would cause at least one common state to be exited. In Rhapsody, transitions whose source states are lower in the state hierarchy have priority over those with sources higher up in the hierarchy. The conflict resolution must be performed after the creation of the transition copies because we need to know the source state (which now is always a simple state) of the transitions in order to appropriately modify only the transitions that would cause the same simple state to be exited. In order to achieve this we preserve the information relating a transition copy to the original transition it was created from. Thus we can identify the original level of the source states of all transitions. We resolve such a conflict between two transitions t_l and t_h where $originalLevel(source(t_l)) < originalLevel(source(t_h))$ by adding the negated guard of t_l to the guard of t_h. Thus $guard(t_h) = \neg guard(t_l) \wedge guard(t_h)$ will impede t_h to be fired if t_l can also fire and the system is in state $source(t_l)$.

After the creation of the transition copies, the original transitions can be removed from the model (line 17).

The last phase of the algorithm in Figure 4.10 consists in moving the entry and exit actions from all states in the appropriate order on the incoming and outgoing transitions respectively (except the transitions created for internal actions - line 20). These actions need to be moved in the order specified in Section 2.2.2. That is first the exit actions of states exited by transitions $tr_{out} \in Out_{tr}(st) | st \in S_s$ from the lowest level up and then the entry actions of the states entered by tr_{out} from the higher level downwards. Thus, for the transition tr_{out} the exit actions are gathered recursively (line 22) in the order the states are exited by tr_{out} until the first common ancestor of st (the source of the transition) and $target(tr_{out})$. The gathered actions are then added before the actions of tr_{out} (line 26) preserving the order in which they should be executed. Similarly, the entry actions of the states entered by tr_{out} are added after the actions already present on the transition (lines 28 - 32). The composite states can now be removed from the model.

Example 4.2.3. Figure 4.11 contains the flattened variant of the statechart presented in Figure 4.5 (the variant containing a deep history node). After the removal of the pseudostates, also the hierarchy structures have been removed. The obtained model contains now only simple states and transitions. The state with the thicker border ($State5$) is the initial state of the statechart. The entry actions of the states have now been moved on the appropriate transitions.

4.3 From Statecharts to LOTOS

At this point of the transformation process the statechart models have been flattened (Section 4.2). The hierarchical structures and the pseudostates have been removed. The flattening phase delivers a behavioral equivalent statechart described only in terms of simple states and transitions $SC = (S_s, T \cup T_{fl}, V, i_{ps0})$ where T_{fl} is the set of new transitions that were created during the flattening process.

LOTOS describes a system through a hierarchy of communicating processes, so we represent each statechart model in the system through a process. Every such process will contain several other processes corresponding to the simple states (preserved in the flattened model) of the represented statechart.

As several aspects regarding the execution and asynchronous communication semantics of the UML model need to be preserved also in the LOTOS representation we insert an extra process in the specification. This process (*ComManager* process) dictates the communication rules and takes care of the run to completion step in the execution semantics. It also ensures that the actions of each transi-

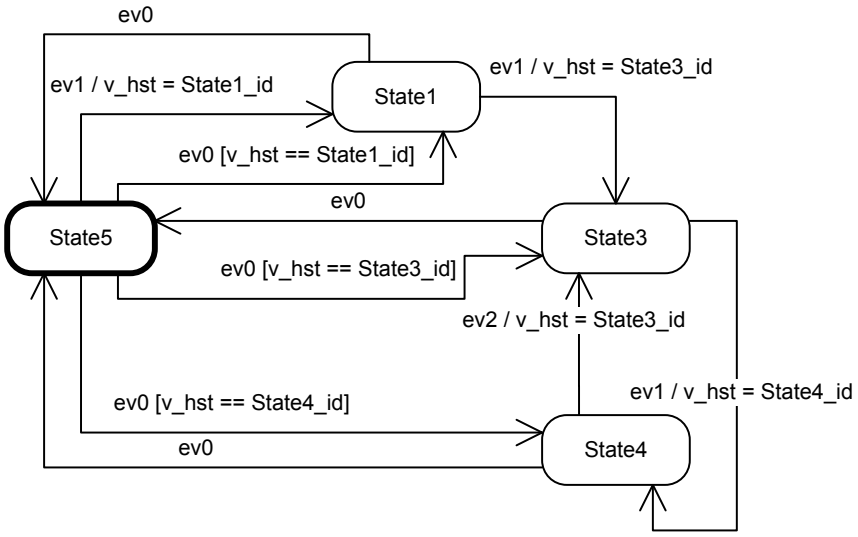

Figure 4.11: Flattened statechart

tion are treated as atomic execution entities. The LOTOS representation for the structure of the considered systems is given in Figure 4.12.

$$
\begin{aligned}
&< lotos\_spec >:= \ \text{`(`} < sc\_proc > \{\text{`|||`} < sc\_proc >\}\text{`)||`} \\
&\quad < synch\_proc > \\
&< sc\_proc >:= \ < sc\_name > \text{`[`} < gates > \text{`](`} < parameters > \text{`)`} \\
&< synch\_proc >:= \text{`}ComManager\text{[`} < gates > \text{`]`} \\
&\quad \text{(`} < synch\_params > \text{`)`} \\
&< sc\_name >:= \ \text{`}SC\_name_1\text{`} \ | \ \text{`}SC\_name_2\text{`} \ | \ \ldots \\
&< parameters >:= \ < param > \ \{\text{`,`} < param >\} \\
&< gates >:= \ \text{`}commGate, inGate, toQueue, fromQueue, outGate\text{`} \\
&< param >:= \ \text{`}param\_name_1\text{`} \ | \ \text{`}param\_name_2\text{`} \ | \ \ldots \\
&< synch\_params > \ := \ \text{`}synch\_param_1\text{`} \ | \ \text{`}synch\_param_2\text{`} \ | \ \ldots
\end{aligned}
$$

Figure 4.12: LOTOS system structure

We describe the structure by using an extended Backus-Naur notation where the symbols used are: ":=" - definition, "|"-alternation, "{" "}"- zero or more repetitions, "<" and ">" enclose non terminal symbols. For readability reasons terminal symbols are quoted (using ' ') and LOTOS keywords and reserved symbols are emphasized.

The $< synch\_params >$ token in Listing 4.12 contains a boolean parameter for each model in the system. These parameters are used by the *ComManager* process in order to monitor the state of the models in the system (more details in Section 4.3.4). Token $< parameters >$ contains the LOTOS parameters used to represent the variables in the statecharts.

The gates contained by $< gates >$ have the following functions:

- *commGate* is the control gate used to get information on the state of the models in the system. The role of this gate in the execution of the system will be further discussed in Section 4.3.4;

```
(DirectionLever[commGate, inGate, toQueue, outGate]() |||
HazardWarningB[commGate, inGate, toQueue, outGate](UNPRESSED) |||
FLBC[commGate, inGate, toQueue, outGate](false, false, NONE) |||
FLMC[commGate, inGate, toQueue, outGate](true) |||
FLMC_FailureDetection[commGate, inGate, toQueue, outGate]() )
||
ComManager[commGate, inGate, toQueue,outGate](true, true, true,
    true)
```

Listing 4.1: LOTOS system structure of the Direction Indication system

- *inGate* is the input gate used by the models in the system to accept events (either from the environment or from the event queue);
- *toQueue* is used to enqueue events in the event queue;
- *fromQueue* realizes the extraction of events from the event queue;
- *outGate* outputs the values of the process parameters (variables in the UML statechart).

The LOTOS specification of our running example is composed of several interleaving processes (one for each statechart model) fully synchronized with the *ComManager* process. This can be seen in Listing 4.1 where the parameters of the processes are initialized to their default values.

4.3.1 LOTOS Data Types

```
1  'type TRANSITIONS is ComModels, TransType'
2      'sorts Transition'
3      'opns'
4          < trans > ': ->Transition' { < trans > ': ->Transition'}
5          '_transEq_ :Transition, Transition ->Bool'
6          'fromModel :Transition − > Model'
7          'typeOfTr :Transition − > TrType'
8      'eqns'
9          'forall t1,t2 :Transition'
10         'ofsort Bool'
11             't1 transEq t1 = true;'
12             't1 transEq t2 = false;'
13     'ofsort Model'
14             'fromModel('< trans > ') =' < modelName > ';'
                   {'fromModel('< trans > ') =' < modelName >;}
15     'ofsort TrType'
16             'typeOfTr(' < trans > ') =' < EventsFired > ';' {'typeOfTr('
                   < trans > ') =' < EventsFired >;}
17 'endtype'
```

Figure 4.13: LOTOS data type definition for UML transitions

In order to help preserve the execution semantics of the UML Statecharts, we employ several LOTOS abstract data types. Another reason for these data types is to keep traceability between the original UML Statechart and the derived LOTOS representation. Thus after the transformation we can still identify UML states, transitions, events as well as Statechart variables in the LOTOS specification.

Every component of the UML model is represented by a corresponding LOTOS data type.

Transition Data Type

The declaration of the *Transition* data type is presented in Figure 4.13. There are several operations that we need to be able to perform on this data type. The first one is identifying the transition (line 4) which is actually an enumeration of the UML transitions IDs. The *transEq* operation is the equality operation for the data type while *fromModel* associates each transition ID with the corresponding statechart in the UML model.

In our UML models, when fired, a transition can generate more than one events. So we also need an operation to tell us how many events will be generated by $< trans >$. Operation *typeOfTr* does just that.

```
type TRANSITIONS is  ComModels, TransType
sorts Transition
  opns
    (* the ids of the transitions *)
    FLMC_T36 (*! constructor *): -> Transition
    FLMC_T40 (*! constructor *): -> Transition
  ...
    fromModel :Transition -> Model
    typeOfTr :Transition -> TrType
  eqns
  ...
    ofsort Model
      fromModel(FLMC_T36) = FLMC;
      fromModel(FLMC_T40) = FLMC;
  ....
    ofsort TrType
      typeOfTr(FLMC_T36) = Fires_0;
      typeOfTr(FLMC_T40) = Fires_2;
  ....
endtype
```

Listing 4.2: Example of LOTOS data type definition for transitions of the Direction Indication system

The example in Listing 4.2 contains a partial declaration of the *TRANSITIONS* data type. Only the constructors for transitions with ids *FLMC_ T36* and *FLMC_ T40* as well as equations for the operations *fromModel* and *typeOfTr* are presented.

Other data types appearing in Listing 4.13 are going to be briefly discussed in Section 4.3.1 however their names are self explanatory. For example the sort *Model* on line 6 identifies the communicating models.

Events Data Type

The communication with and within the system is done via data carrying events. The data type definition representing the triggering events is presented Figure 4.14.

```
1  'type EVENTS is ComModels, BOOLEAN, NATURAL, EventTypes'
2  'sorts Event'
3  'opns'
4        < event > ': ->Event' { < event > ': ->Event'}
5        'toModel :Event - > Model'
6        'ofType :Event - > EvType'
7         < getParamOperation >': Event ->' < paramDataType >
8  'eqns'
9        'forall mod: Model, ' < param > ':' < paramType > {, < param > ':'
             < paramType >}
10       'ofsort ' < paramType >
11         'getParam_'< param >'('< event >'('< model >',' < param > ','
               {< param >} '')) =' < param >';'
12         {'getParam_'< param >'('< event >'('< model >','
               < param >{< param >} '')) =' < param >';' }
13       'ofsort Model'
14         'toModel('< event >'('< model > {',' < param > } '') ='
               < model >';'
15         {'toModel('< event >'('< model > {',' < param > } '') ='
               < model >';'}
16       'ofSort EvType'
17         'ofSort('< event >'('< model > {',' < param > } '') ='
               < eventType >';'
18         {'ofSort('< event >'('< model > {',' < param > } '') ='
               < eventType >';'}
19 'endtype'
```

Figure 4.14: LOTOS data type definition for communication events

This data type provides operations for:

- Accessing the parameters of the event. Depending on the number and type of the event parameters we generate operations for retrieving the corresponding values. Parameter *state* from event *evFLMC_State* in Listing 4.3 can be accessed by using the operation *getParam_state* on events of type *T_evFLMC_State*;
- The model the event was sent to. The communicating events can be sent to specific models in the system. Thus operation *toModel* is used to identify

the model targeted by the event;

- Defining the type of the event. This is needed in order to identify the event in the LOTOS specification. For example in Listing 4.3 the type of event *evIgnitionState* is *T_evIgnitionState*.

```
type EVENTS is  ComModels, BOOLEAN, NATURAL, EventTypes
sorts Event
  opns
    evIgnitionState (*! constructor *): Model, Nat -> Event
    evFLMC_State (*! constructor *): Model, Nat -> Event
    ...
    ofType : Event -> EvType
      getParam_state: Event -> Nat
      toModel : Event -> Model
    ...
  eqns
    forall state: Nat,  mod: Model
    ofsort Nat
      getParam_state(evFLMC_State(mod, state)) = state;
      getParam_IgnState(evIgnitionState(mod, state)) = state;
      ...
    ofsort Model
      toModel(evIgnitionOn(mod)) = mod;
      toModel(evFLMC_State(mod, state)) = mod;
      ...
    ofsort EvType
      ofType(evIgnitionState(mod, state)) = T_evIgnitionState;
      ofType(evFLMC_State(mod, state)) = T_evFLMC_State;
    ...
```

Listing 4.3: LOTOS data type declaration example for the communication events

Event Queue Data Type

Since the communication within the system is asynchronous, we need to use a FIFO event queue that will store the events not yet consumed by the system. Such events need to be stored in the event queue if they are sent to a model that is not in a stable state (it is executing some transitions and the run to completion step has not finished yet).

So we define the abstract data type *Queue* for this purpose. The LOTOS declaration of the event queue is presented in Listing 4.4 and contains the usual queue operations:

- top - returns the event representing the head of the queue without removing it from the queue;
- add - adds an event to the tail of the queue;

```
type Queue is  NATURAL, Boolean, EVENTS
  sorts Queue
  opns
    nil (*! constructor *) : -> Queue
    add (*! constructor *) : Event, Queue -> Queue
    empty : Queue -> Bool
    pop : Queue -> Queue
    top : Queue -> Event
  eqns
    forall q: Queue, m: Event
    ofsort Bool
      empty(nil) = true;
      empty(add(m, q)) = false;
    ofsort Event
      top(add(m, nil)) = m;
      top(add(m, q)) = top(q);
    ofsort Queue
      pop(nil) = nil;
      pop(add(m, nil)) = nil;
      pop(add(m, q)) = add(m, pop(q));
endtype
```

Listing 4.4: LOTOS data type declaration for the event queue

- pop - extracts the event in the head of the queue;
- empty - returns true if there are no events stored in the queue and false otherwise.

Other Data Types

The definitions of the data types representing transitions, events and the event queue also make use of other data types. Since these are simpler data types (being usually the enumeration of the possible values and the equality operator on that set) we shortly present them below:

- *Model* - used in the *Transition* and *Event* data types for identifying the models containing the transition (*fromModel* operation) and destination of the event (*toModel* operation) respectively. It contains constructors for each model in the system as well as operations for testing equality between the elements of the *Model* sort.
- *TrType* - defines the number of events a transition can generate. It contains constructors for the number of events and equality operations;
- *Natural_Constants* - declares the constants used in the specification. Since the specification makes use of abstract data types all natural constants need to be specified in terms of already defined terms (usually numbers from 1

to 10). For example a constant Q with value 245 would be defined as $Q = 5 + (4 * 10) + (2 * 10 * 10)$;

- *StatesIds* - contains the ids of the states in each statechart of the system. These are used in the specification in order to keep track of the currently active state in each model in the evolving system. It also contains constructors for each defined state and the equality operation.

- *EventTypes* - used to identify the type of the communication events (*ofType* operation in *Events* data type).

4.3.2 Transition Transformation

Due to the fact that all components of a transition label are optional and that there is a conceptual difference between the transitions explicitly triggered by an event and the completion transitions we treat the two separately.

Completion Transitions

Completion transitions do not have an explicit triggering event and can have guards and/or actions. Such a transition is automatically fired when its source state is reached and its guard evaluates to true. If such a transition can be fired, the statechart is not in a stable state, thus the current run to completion step has not finished.

Considering the definition of statecharts in Section 2.2, the completion transitions are now defined as in Equation 4.4.

$$T_{cmpl} \subseteq S \times L \times S \;\; where \;\; L \subseteq G \times A \tag{4.4}$$

The LOTOS representation of such a transition is presented in Figure 4.15 where $< trans\_id >$ represents the id of the completion transition. The $< transition\_guard >$ represents the guard of the transition while $< output >$ offers to the environment the new active state $< newState >$ and the new values of the variables in the statechart (if any). The token $< next\_process >$ is the process corresponding to the state targeted by the completion transition and defines the behavior of the system after the completion transition has been fired. If the action part of the transition contains assignment statements, these are mapped to the parameters of the targeted process by updating the values accordingly.

An example of such a transition can be seen in Figure 4.16 (transition originating from state Toggled and targeting state Pressed) and its LOTOS representation is given in Listing 4.5. In Listing 4.5 the gate *commGate* is used for the execution of the run to completion step and other communication semantics which will be explained in more detail in Section 4.3.3. The only action of gate *inGate* is the id of the transition (*HazWarn_ T2*). When this transition if fired, a *reqBlink(FLBC,*

```
[ButtonStatus == UNPRESSED] ->
  commGate !HazardWarning !false;
  inGate !HazWarn_T2;
  toQueue !reqBlink(FLBC, BOTH);
  outGate !Pressed !PRESSED;
  Pressed[commGate, inGate, toQueue, fromQueue, outGate](PRESSED)
```

Listing 4.5: LOTOS representation of *HazWarn_ T2* completion transition

BOTH) is sent to the *FLBC* model by using the *toQueue* gate. Gate *outGate* outputs to the environment first the new active state of the model (corresponding to the *Pressed* state) and than the new values of the variables in the statechart (the state of the button is now *PRESSED*). The process corresponding to the state targeted by the transition is then instantiated and the new state of the hazard warning button is passed on as the value *PRESSED* for the parameter of process *Pressed*.

Event Triggered Transitions

These are transitions that are explicitly triggered by an event provided that their guards evaluate to *true*. Considering the definition of statecharts in Section 2.2, the normal transitions are defined as in Equation 4.5.

$$T_n \subseteq S \times L \times S \ \ where \ \ L \subseteq E_{tr} \times G \times A \ \ and \ \ E_{tr} \neq \emptyset \tag{4.5}$$

Such a transition is to be represented in the LOTOS specification by a behavioral expression presented in Figure 4.17.

In Figure 4.17 $<$ *condition* $>$ is the conjunction of the guard of the normal transition, the fact that the event is the one needed to trigger the transition (*ofType(msg)*) and that it was sent to the model containing the transition (*to-*

$<$ *compl_trans* $>$:= '[' $<$ *transition_guard* $>$ '] -> inGate !' $<$ *trans_id* $>$ ';' $<$ *output* $>$ ';' $<$ *next_process* $>$
$<$ *output* $>$:= 'outGate !' $<$ *newState* $>$ { '!' $<$ *paramValue* $>$ };
$<$ *next_process* $>$:= $<$ *state_name* $>$ '[' $<$ *gates* $>$ '](' $<$ *parameters* $>$ ')'

$<$ *parameters* $>$:= $<$ *param* $>$ { ',' $<$ *param* $>$ }
$<$ *gates* $>$:='commGate, inGate, toQueue, fromQueue, outGate'
$<$ *param* $>$:='$param\_name_1$' | '$param\_name_2$' | ...

Figure 4.15: LOTOS representation of completion transitions

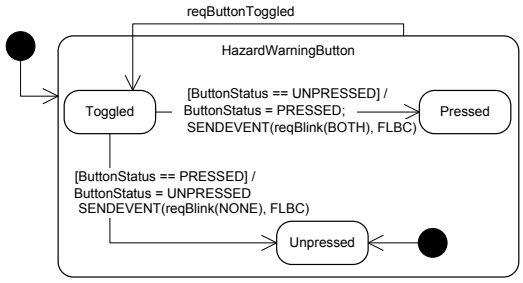

Figure 4.16: Statechart of the HazardWarningButton

$$
\begin{aligned}
&< normal\_trans >:= \text{`inGate !'} < trans\_id > \text{`?msg:Event'} \text{`['} \\
&\quad < condition > \text{`];'} < output > \text{`;'} < next\_process > \\
&< output >:= \text{`outGate !'} < newState > \{\text{`!'} < paramValue >\}; \\
&< condition >:= \text{`(ofType(msg) =='} < eventType > \text{`and (toModel} \\
&\quad \text{(msg) =='} < model > \text{`and'} < trnsition\_guard > \\
&< next\_process >:= < state\_name > \text{`['} < gates > \text{`]('} \\
&\quad < parameters > \text{`)'} \\
&< parameters >:= < param > \{\text{`,'} < param >\} \\
&< gates >:= \text{`}commGate, inGate, toQueue, fromQueue, outGate\text{'} \\
&< param >:= \text{`}param\_name_1\text{'} \mid \text{`}param\_name_2\text{'} \mid \ldots
\end{aligned}
$$

Figure 4.17: LOTOS representation of an event triggered transition

Model(msg)). So after receiving the event, the behavior of the system shall continue as specified by the $< next\_process >$ which is the process corresponding to the state targeted by the transition.

An example of a normal transition can be found in Figure 4.16 originating from state $HazardWarningButton$ and targeting state $Toggled$ and its LOTOS representation is given in Listing 4.6.

```
inGate !HazWarn_T3 ?msg:Event [(ofType(msg) eveq T_reqButonToggled)
  and (toModel(msg) meq FLBC)];
outGate !Toggled !buttonStatus
Toggled[commGate, inGate, toQueue, fromQueue, outGate]
  (buttonStatus)
```

Listing 4.6: LOTOS representation of *HazWarn_ T3* transition

4.3.3 State Transformation

Since at this point, the statechart has been flattened, only simple states are present in the model. Each state is represented in the LOTOS specification through a LOTOS process. Such a process can be seen as containing four parts, each one taking care of a different aspect in the execution semantics of the statechart. Figure 4.18 presents the process representing an UML state.

$< state\_process >$:= ‘process’ $< state\_name >$ ‘[’ $< gates >$ ‘]’
‘(’ $< parameters >$ ‘)’ $< process\_body >$ ‘endproc’
$< process\_body >$:= $< communication\_ctrl >$ ‘;’
{$< treat\_compl\_trans >$} ‘[ ]’ {$< treat\_other\_trans >$}
‘[ ]’ {$< make\_input\_complete >$}

$< communication\_ctrl >$:= ‘commGate !’ $< model\_id >$
‘!’ $< stable_state >$ ‘;’
$< treat\_compl\_trans >$:= {$< compl\_trans >$ { ‘[ ]’ $< compl\_trans >$
}}
$< treat\_other\_trans >$:= ‘[’ $< neg\_compl\_guards\_conj >$ ‘]’ ->
‘(’ {$< normal\_trans >$ { ‘[ ]’ $< normal\_trans >$ }} ‘)’

$< make\_input\_complete >$:= ‘[’ $< neg\_compl\_guards\_conj >$ ‘]’ ->
‘(’ ‘{’ $< artificial\_trans >$ { ‘[ ]’ $< artificial\_trans >$ }} ‘)’
$< artificial\_trans >$:= ‘inGate !’ $< dummy\_trans\_id >$
‘?msg:Event [’ $< neg\_guard >$ ‘];’ $< state\_process >$

Figure 4.18: LOTOS process representing a state from an UML statechart

The $< treat\_compl\_trans >$ token in the LOTOS representation of an UML state in Figure 4.18 takes care of offering to the environment a choice between the completion transitions originating from the state $< state\_process >$. In deterministic systems there will always be only one such transition that will be able to fire. In nondeterministic systems one such transition will be chosen or, depending on the semantics chosen for the statecharts, a prioritization algorithm can also be integrated as to differentiate between several completion transitions that can fire. As already mentioned, in our setting, nondeterminism is not allowed.

The semantics of the run to completion step impedes one model to consume an event if the model is not in a stable state, that is, if the model is in a state where at least one completion transition can fire (its guard is evaluated as *true*). In order to model this kind of behavior, a conjunction of the negated guards of the completion transitions $< neg\_compl\_guards\_conj >$ is added as a guard that restricts the behavior in the rest of the LOTOS process ($< treat\_other\_trans >$). The obtained behavior is that the gates corresponding to non completion transition

are offered to the environment only if $< neg\_compl\_guards\_conj >$ evaluates to true (no completion transition can fire).

The $< compl\_trans >$ and $< normal\_trans >$ are the ones described in Section 4.3.2 and are the LOTOS representation of the completion and normal event triggered transitions respectively.

The last part of the process $< make\_input\_complete >$ has the role of allowing this process to be able to handle any input event at any time. This is equivalent to making the state represented by the process input complete. This is needed in order to preserve the communication semantics of the system requiring that events that are not handled explicitly in the current state (during a run-to-completion step) need to be ignored. This is achieved by adding behavioral expressions representing dummy transitions that are triggered by events other than the ones triggering normal transitions originating from the current state. Such dummy transitions consume the events and reinstantiate the owner process without modifying the values of the process parameters.

```
process Toggled[commGate, inGate, toQueue, fromQueue, outGate]
    (ButtonStatus : Nat):noexit:=
(*-<treat_compl_trans>-*)
  [ButtonStatus == UNPRESSED]->
    commGate !HazardWarning !false;
    inGate !HazWarn_T2;
    toQueue !reqBlink(FLBC, BOTH);
    Pressed[commGate, inGate, toQueue](PRESSED)
  []
  [ButtonStatus == PRESSED]->
    commGate !HazardWarning !false;
    inGate !HazWarn_Trans3;
    toQueue !reqBlink(FLBC, NONE);
    Unpressed[commGate, inGate, toQueue](PRESSED)
  []
(*-<treat_other_trans>-*)
  [not(ButtonStatus == UNPRESSED ) and
    not(ButtonStatus == PRESSED)]->
    commGate !HazardWarning !true;
    inGate !HazWarn_Trans1 ?msg:Event [(ofType(msg) ==
        T_reqButtonToggled)
      and (toModel(msg) == HazardWarning)];
    outGate !ButtonStatus;
    Toggled[commGate, inGate, toQueue, fromQueue, outGate]
      (ButtonStatus)
endproc
```

Listing 4.7: Lotos representation of the Toggeled state

If the UML state represented by the process also has normal transitions with guards, dummy transitions will also be generated for each normal transition. These dummy transitions will have as triggering event the same event as the

normal one. The guard of such transitions will be complementary to the guard of the normal transition so that they will fire only when the normal ones can not consume the triggering event (due to guards evaluating to false). The modeled behavior in this case is that if the model receives an event that can trigger a transition in the current state but the guard of the transition evaluates to false, the model will ignore the event (by firing the corresponding dummy transition) and no state change will occur.

Listing 4.7 presents an example of a LOTOS representation for state *Toggeled* in Figure 4.16. As the only event triggered transition of state *Toggeled* does not have a guard and the triggering event (*reqButtonToggeled*) is also the only one that this statechart can receive, there is no need for making the state input complete (no $< make\_input\_complete >$ part required).

4.3.4 Communication Control

The transformation of one statechart model into a LOTOS representation has been presented in Sections 4.3.1, 4.3.2 and 4.3.3. This transformation can be used to represent a system composed of only one statechart and consuming the events it receives in a synchronous manner. However, the kind of systems we aim to handle are asynchronously communicating distributed systems. In order to integrate these semantics we need to extend the transformation in order to treat aspects such as asynchronous communication and run to completion step among communicating statechart models.

For this purpose an extra process is inserted into the specification of the model. This process fully synchronizes with all the other models and dictates the communication mechanism with and within the system. The structure of the *ComManager* process is presented in Figure 4.19 and can be described as being composed of several parts.

$< communication\_master\_process >$:= 'process ComManager
['$< gates >$']('$< synch\_params >$')'
$< synch\_process\_body >$ 'endproc'

$< synch\_process\_body >$:= $< get\_into\_stable\_state >$ '[]'
$< consume\_events >$
$< synch\_params >$:= 'stable_'$< model\_id >$ {',' 'stable_'
$< model\_id >$}
$< gates >$:= '$commGate, inGate, toQueue, fromQueue, outGate$'

Figure 4.19: Structure of the ComManager process

Where *commGate* is the control gate used to get information regarding the

state of the models in the system. This information is used to identify if the executing model has any completion transitions that could fire (if it is in a stable state). The *inGate* is the input gate used by the models in the system to accept events (either from the environment or from the event queue). The gates *toQueue* and *fromQueue* are used to put and respectively extract events from the event queue. The *outGate* gate outputs the values of the process parameters (variables in the UML statechart).

The $< synch\_params >$ contains a boolean parameter for each model in the system. The boolean parameters are used to store and monitor the state of the models in the system (stable or not). These values are transmitted through the *commGate* every time a new process (corresponding to an UML state) is instantiated. If the transmitted value is *!true* this means that there is at least one completion transition that can be fired from the current state of the system.

In order to extract events from the queue, the *ComManager* communicates via the *fromQueue* with the *EventQueue* process (depicted in Listing 4.8) as to receive the next event in the queue. If the queue is empty this is signaled by sending *!false* followed by the dummy event *!nullMsg*.

```
process EventQueue[toQueue, fromQueue](evQ: Queue): noexit:=
  toQueue ?ev: Event; EventQueue[toQueue, fromQueue](add(ev, evQ))
  []
  [not (empty(evQ))]->
    fromQueue !true !top(evQ); EventQueue[toQueue, fromQueue](pop(
        evQ))
  []
  [empty(evQ)]->
    fromQueue !false !nullMsg; EventQueue[toQueue, fromQueue](evQ)
endproc
```

Listing 4.8: The EventQueue process

Getting the Models into a Stable State

The first part of the *ComManager* process takes care of preserving the run-to-completion step semantic. This is achieved by first *forcing* the current executing model to trigger all its completion transitions whose guards evaluate to true. Thus, the *ComManager* will not allow the execution of other type of transitions but completion ones. After all the completion transitions have been fired, the next step is to extract and consume (one by one) all the events that have been enqueued in the event queue by previously fired transitions (events that were already in the queue before or that have been received during the current run to completion step).

Figure 4.20 contains the definition of the $< get\_into\_stable\_state >$ token introduced in Figure 4.19 and describes the mechanism of getting the model into a stable state.

```
1 < get_into_stable_state >= < fire_completion_trans > { '[ ]'
    < fire_completion_trans >}
2 < fire_completion_trans >:=
3   'commGate !'< model_id > '?stable: Bool;'
4     '[not(stable)]->'inGate ?tr:Transition;'
        < allow_trans_execution >
5     '[ ]'
6     '[stable]-> ComManager['< gates > ']( '< synch_params > ')
      '
7 < allow_trans_execution >:=
8     '(typeOfTr(tr) eq' < generated_events_no >
9      {'toQueue ?msg:Event; exit )>>'}
10    'outGate' {'?'< paramName > ':'< paramType > ;}'
11    ComManager['< gates > ']( '< synch_params > ')'
```

Figure 4.20: ComManager process - getting the system into a stable state

The behavior described by $< get\_into\_stable\_state >$ in Listing 4.20 is that of first allowing the execution of all available completion transitions (the ones that can be fired from the current state of the system) and then, after a stable state has been reached allow for the consumption of events from the event queue($<$ *consume_events* $>$ - explained in Section 4.3.4). The data *'!'*$< model\_id >$ *'?stable: Bool;'* exchanged at gate *commGate* (line 3) is used to get information on the state of the current executing model. All this data is provided every time a process (corresponding to an UML state) is instantiated. Thus $< model\_id >$ identifies the model, *stable* denotes if in that model there is at least one completion transition that can be fired.

If there is at least one completion transition that can fire, the *ComManager* process will allow this by providing the corresponding sequence of actions so that only the model identified by $< model\_id >$ can execute the behavioral expression representing a completion transition. This expression (line 4) is dynamically computed based on the type of transition (line 7), the number of events its execution will generate (line 8) and the model to which it belongs (10).

The example in Listing 4.9 partially depicts the first part of the *ComManager* process which takes care of getting the models in the Direction Indication system to a stable state. It contains only the part responsible with bringing the *FLBC* model in a stable state. If the *stable* gate offering is *false*, a completion transition will be allowed to fire, otherwise the *ComManager* will simply reinstantiate itself

```
process ComManager[commGate, inGate, toQueue, fromQueue, outGate](
    stable_FLBC: Bool, stable_FLMC: Bool, stable_FLMC_FailDetect:
    Bool, stable_DI_Lever: Bool, stable_HazW_Button: Bool,
    stable_HazW_ButtonMsgSender: Bool): noexit:=

[stable_FLBC ]-> commGate !FLBC ?state: StateId ?stable: Bool;
  (* if FLBC is not in a stable state, a completion transition
     needs fire *)
  [not stable]->
  ( inGate ?tr:Transition;
    (
      (
      [typeOfTr(tr) teq Fires_0] -> exit
      []
      [typeOfTr(tr) teq Fires_1] ->
        toQueue ?msg:Event; exit
      []
      ...   )>>
      (*Allow the transition to provide as output the current
          values of the attributes of the model*)
      outGate ?crash:Bool ?hazard:Bool ?direction:Nat;
      ComManager[commGate, inGate, toQueue, fromQueue, outGate](
          true, stable_FLMC, stable_FLMC_FailDetect,
          stable_DI_Lever, stable_HazW_Button,
          stable_HazW_ButtonMsgSender)
    ) []
  [stable]-> ComManager[commGate, inGate, toQueue, fromQueue,
     outGate]
      (true, stable_FLMC, stable_FLMC_FailDetect, stable_DI_Lever,
          stable_HazW_Button, stable_HazW_ButtonMsgSender)
  )
[] (* do the same for the other models and take them to a stable
    state *)
  ...
```

Listing 4.9: Example of ComManager process for the Direction Indication system

and mark its *stable_FLBC* parameter as true (indicating that the *FLBC* is in a stable state). This procedure is also applied to all the models in the system until none of them has any completion transition that can fire.

Consuming Events

After all models have fired their completion transitions, the *ComManager* allows the extraction of events from the event queue. Figure 4.21 describes the behavior of the *ComManager* process when doing this. If all models are in a stable state (line 2), *ComManager* attempts to extract an event from the event queue (line 3) by using the *fromQueue* gate. The first gate offering *?empty: Bool* is a marker that the process handling the event queue will offer when synchronizing with the *ComManager* process. The value of *empty* denotes if there is at least one event

in the queue which needs to be consumed. If so (line 5) *ComManager* allows the execution of a normal transition (line 6 - 7) triggered by the event extracted from the queue - *!msg*.

```
1 < consume_events >:= '[stable_' < model_id >'== true'
2     {'and' stable_' < model_id >'== true'}']->'
3     'fromQueue ?empty:Bool ?msg: Event;'
4     < empty_event_queue > '[ ]' < consume_external_event >
5 < empty_event_queue >:= '[empty == false]->'
6     inGate ?tr:Transition !msg;
7     < allow_trans_execution >
8 < consume_external_event >:= '[empty == true]->'
9     'inGate ?tr:Transition ?msg:Event;'
10    < allow_trans_execution >
```

Figure 4.21: ComManager process - consuming events

When the event queue has been emptied (line 8), *ComManager* simply allows the execution of a transition triggered by an event *?msg:Event* from the set of events that can be consumed in the current state of the system. Allowing also other events (from the ones that can not be consumed in the current state of the system) would not make much sense since these would simply be discarded by the system (by using the dummy transitions inserted at every state - see Section 4.3.3). The $< allow\_trans\_execution >$ in Figure 4.21 is the same one previously used in Figure 4.20.

A partial example of the *ComManager* process for the Direction Indication system taking care of the extraction and consumption of events from the event queue can be found in Listing 4.10. After the event queue has been emptied ($[empty]- >$) the process allows the reception of evens coming from outside the system (*inGate ?tr: Transition ?msg:Event;*). The rest of the transition execution is similar to the one when events are enqd from the event queue and thus not explicitly presented in the example.

```
....
(*After the models have reached a stable state, the events stored
    in the event queue can be consumed by the models.*)
  [ stable_FLBC and stable_FLMC and stable_FLMC_FailDetect and
      stable_DI_Lever and stable_HazW_Button and
      stable_HazW_ButtonMsgSender]->
  (*extract one event from the communication queue*)
  fromQueue ?empty:Bool ?msg: Event ;
  (
  [not(empty)]-> (
    inGate ?tr: Transition !msg;
    (*if the fired transition generates an event, this has to be
       inserted in the event queue*)
      ( [typeOfTr(tr) teq Fires_0] -> exit
        []
        [typeOfTr(tr) teq Fires_1] ->
          toQueue ?msg:Event; exit
      ...
        )>>(
        (*if the fired transition belongs to the FLBC model, the
            model needs to be returned to a stable state before
            another event can be consumed.     This is marked by
            changing the value of the stable_FLBC parameter to
            false *)
      [fromModel(tr) meq FLBC) ] ->
        outGate ?crash:Bool ?hazard:Bool ?direction:Nat;
        ComManager[commGate, inGate, toQueue, fromQueue, outGate](
            false, stable_FLMC, stable_FLMC_FailDetect,
            stable_DI_Lever, stable_HazW_Button,
            stable_HazW_ButtonMsgSender)
        []
        (*if the fired transition belongs to the FLMC model ... *)
      [fromModel(tr) meq FLMC ] ->
        outGate ?stateActive:Bool;
        ComManager[commGate, inGate, toQueue, fromQueue, outGate](
            stable_FLBC, false, stable_FLMC_FailDetect,
            stable_DI_Lever, stable_HazW_Button,
            stable_HazW_ButtonMsgSender)
        []
      (* do the same for the other models in the system depending
          on the model containing the fired transition *)
        ...
      )
      (*if there are no more events in the queue, then the system
          can receive events from the environment *)
  [empty]-> inGate ?tr: Transition ?msg:Event;
  ...
endproc
```

Listing 4.10: Example of ComManager process for the Direction Indication system (continued) - Consuming events

Chapter 5

Timing

Parts of this chapter have been published in "Abstracting Timing Information in UML Statecharts via Temporal Ordering and LOTOS" [CW11] which is joint work with Franz Wotawa.

In the previous chapter we described how to derive a LOTOS representation from an UML model composed of communicating statecharts. Our models represent also embedded systems. Since such systems usually involve the use of timing constructs for describing the desired behavior we also need to treat these in our transformation process.

The specification of timing aspects within the considered systems is done by means of timeouts. A timeout is specified with the help of timeout transitions (Definition 2.2.6). The triggering event of such a transition is a timeout event written as "tm" followed by an expression that evaluates to a time value e.g. "tm(90)". The intended semantic is that of a clock that starts when the originating state of the timeout transition is entered and elapses when it has measured the value equal to the timeout parameter. At that moment the transition is executed and the state of the system changes to the state targeted by the timeout transition.

5.1 Abstracting Timing Information

Unfortunately LOTOS does not contain any constructs to describe timing behavior. Other works also propose timing extensions to LOTOS. Some of them [Led92, RBC93, LL93, LGC96] are based on the introduction of new LOTOS operators. Others [Kho01, BD97] restrict the language to its basic form (the use of data values is not allowed). ETLOTOS [BDS95] and LOTOS NT [CCG+05], the newer versions of the language have timing constructs but still need a broader tool support.

So we propose a timing abstraction that aims to keep the visible output of the system by preserving the order timeout transitions fire with respect to each other.

The presence of a global clock is simulated through LOTOS abstract data types and several control points artificially inserted in the specification. The goal is to ensure that the transition triggered by the timeout with the smallest value will always fire before any other timeout transition. The employed data types for this purpose are *TmEvent* and *TimeContainer* to represent timeout events and the timeout event container. The afore mentioned control points are inserted in the processes representing UML states and also in the communication manager process that will now dictate also the timing behavior of the system. It does this by allowing only the timeout with the smallest value to trigger its transition. In order to keep track of all the possible timeouts in the system and always provide the smallest one, the process has as one of its parameters a data structure of type *TimeContainer* used to store all possible timeouts at the current active state of the system.

The *TmEvent* data type is similar to that used for representing normal events (see Section 4.3.1) with the difference that it contains as its parameters the timeout value and the id of the transition it triggers. This data type offers the operations *getParam_ timeout*, *getTmTransId* and *decrTm*. The last operation receives as parameter a natural number subtracting its value from the value of the timeout parameter. This operation is used for updating the value of the timeouts when another (smaller) timeout in the system has elapsed.

The *TimeContainer* data type has similarities with the event queue data type in that it is also a container for triggering events (timeout events) however the event insertion order in the time container does not matter. The operations offered by this data type are :

- *min: TimeContainer -> TmEvent* - returns the timeout event whose timeout parameter has the smallest value among all the timeout events in the time container.
- *pop_ min : TimeContainer -> TimeContainer* - removes from the container the timeout event returned by the *min* operation.
- *decr_ queue : Nat, TimeContainer -> TmQueue* - similar with the *decrTm* operation from the *TmEvent* data type but decreases all timeouts in the container with the value of the provided parameter.
- *containsTrans: Transition, TimeContainer -> Bool* - returns true if the container contains a timeout event of the provided transition.
- *getTrans: Transition, TimeContainer -> TmEvent* - returns the timeout event of the specified transition.

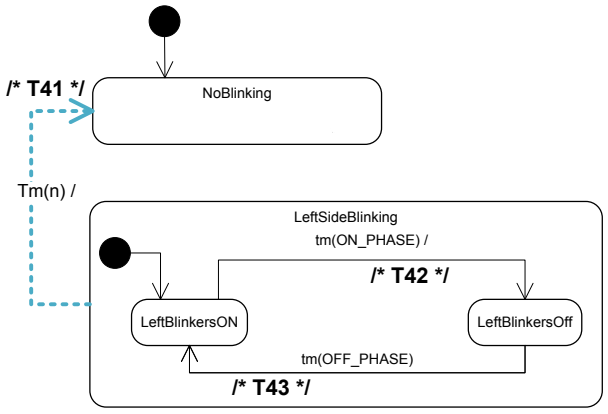

Figure 5.1: Timeout transition example

- *popTrans: Transition, TimeContainer -> TimeContainer* - removes the timeout event of the given transition

5.1.1 Timeout Transitions

Timeout transitions are transitions triggered by a timeout event. Such an event is received after entering the source state of the timeout transition and the timeout has elapsed. However, this behavior is abstracted in the transformation by preserving the ordering relation between the timeout transitions in different models (the smallest timeout will **elapse** before the other timeouts)

$$T_{tm} \subseteq S \times L \times S \ \ where \ \ L \subseteq TM_{ev} \times G \times A \wedge TM_{ev} \neq \emptyset. \tag{5.1}$$

The resulting LOTOS behavioral expression for this kind of transitions is similar to that for normal transitions. The main difference is the fact that after the consumption of a timeout event, this information needs to be propagated to the rest of the system. This is achieved through a dedicated gate - "*timeGate*".

```
<tm_trans>:='inGate !' <trans_id> '?Tmsg:TmEvent'
   '[' <condition> ']; ' <broadcast_tm> '; ' <next_process>
<broadcast_tm>:= 'timeGate !getParam(Tmsg) !' <model_id>
```

Figure 5.2: LOTOS Representation of a timeout transition

The triggering of such a transition represents the abstraction for the passing of a period of time equal to the parameter of the timeout event. All the other models in the system need to be informed of this event in order to update their

timing information accordingly. This is realized through the gate **timeGate** that offers the value of the timeout parameter and the id of the model in which the timeout transition has been fired. All other models shall use this information to decrease the timeout parameters in their time containers.

In Figure 5.1 we present an example of a timeout transitions (connecting state LeftBlinkersOn with the state LeftBlinkersOff) and its LOTOS representation in Listing 5.1.

5.1.2 State Transformation

```
process LeftBlinkersOn[alfaGate, inGate, toQueue, time]
    (tmQueue: TmQueue, timeSynch: Nat):noexit:=
  [(timeSynch eq 0)]-> LeftBlinkersOn[alfaGate, inGate, toQueue,
      outGate, timeGate]
    ( add(tm(FLBC_T42, ON_PHASE), tmQueue) ,  1)
  []
  [timeSynch eq 1]->
  alfaGate !Blinker !false !min(tmQueue);
  (
 .....
  (*handle timeout transition*)
  inGate !FLBC_T42 ?tmMsg:TmEvent [get_param(tmMsg)==ON_PHASE];
  timeGate  !get_param(tmMsg) !FLBC;
  toQueue !reqLightOff(InstrCluster, LEFT);
  LeftBlinkersOff[alfaGate, inGate, toQueue, timeGate]
    (removeTmEv(tmMsg,tmQueue), timeSynch)
 []
 .....
  (*updating timing information*)
  timeGate ?tmt:Nat !FLBC;
  LeftBlinkersOn[alfaGate, inGate, toQueue, timeGate]
    (decr_queue(tmt, tmQueue), timeSynch)
 )
endproc
```

Listing 5.1: LOTOS representation of state LeftBlinkersOn

In order to handle timing information, each process representing a state has as parameter a data structure used to hold all timing events that can fire from that state.

In case of process instantiations representing changes of state inside a composite state (at the non flattened statechart) care needs to be taken in order not to loose time progress of transitions originating from the super states.

Consider the example in Figure 5.1. If the active state is $LeftBlinkersON$ and the timeout transition $T42$ (targetting $LeftBlinkersOFF$) is taken, the timeout transition $T41$ must be updated with the value of the fired timeout and its timeout event preserved in the time container. In the new active state (the

$LeftBlinkersOFF$ process) the transition $T41$ will be kept in the container only if its guard still evaluates to true.

```
<state_process>:= 'process'<state_name>'['<gates>']'
   '('<parameters>')'  <process_body> 'endproc'

<process_body>:= <update_timing_container> '->'
  <communication_ctrl>';'<treat_compl_trans>'[ ]'
  <treat_other_trans>'[ ]'<update_timing_info> '[ ]'
  <make_input_complete>

<update_timing_container>:=
  { '[(timeSynch eq '<tm_tr_no>') and '<tmTrGuard>']->'
  <state_name>'['<gates>']'
   '( add(tm('<tm_trans_id>, <tm_value>)', tmCont), '
   timeSynch '+1' , <other_params> ')'
   '[]'
   '[(timeSynch eq '<tm_tr_no>') and not'<tmTrGuard>']->'
  <state_name>'['<gates>']'
   '( remove(tm('<tm_trans_id>, <tm_value>)', tmCont), '
   timeSynch '+1' , <other_params> ')'
   }

<communication_ctrl>:= 'commGate !'<model_id> '!'<
    is_stable>
   '!'<min_timeout>';'
<update_timing_info>:= 'timeGate ?tmt:Nat !'<model_id>';'
<tm_tr_no> := tmTransition_0 | tmTransition_1 ...
<parameters>:= 'tmCont: TimeContainer, timeSynch: Nat' {,
    <other_params>}
```

Figure 5.3: LOTOS Process Definition with timing constructs

Figure 5.3 presents the structure of a state with originating timeout transitions. When the process is first instantiated it updates its timing container by inserting and/or removing the timeouts of its timeout transitions. The *<tm_ tr_ no>* item represents ids given to the timeout transitions originating from the current state. Thus the timeouts corresponding to these transitions are added in the local time container of the process.

Once the time container has been updated and contains the available timeouts, the communication gate *commGate* (part of *<communication_ ctrl>*) transmits to the communication master process only the timeout with the smallest value which is possible in the current state. The communication master keeps track of the smallest timeouts of each statechart in the system and allows the execution

only for the minimum amongst them.

The <*treat_compl_trans*> and <*make_input_complete*> in Figure 5.3 are the same described in Section 4.3.3. The <*treat_other_trans*> is similar to the treatment of normal transitions in Section 4.3.2 but this time it also contains timeout transitions.

The <*update_timing_info*> part is used to update the values of the timeouts that can elapse in the current state. When a timeout transition is fired in another model it will broadcast (via the communication master) the value of the timeout to all the other models (by using the LOTOS multi way synchronization mechanism). The value will be subtracted from the parameters of all other timeouts possible at that moment. An example illustrating this can be found in Listing 5.1.

5.1.3 Communication Master and Timing

The communication master that dictates the execution and communication semantics of the system receives several additions in order to handle the timing aspects. The new structure of the process is presented in Figure 5.4. The process uses an extra parameter of type *TimeContainer* in order to keep track of the timeouts available in the system. This will contain only the smallest timeout that can elapse at the current active state of every statechart model. The timeout values are received via the communication gate *commGate* every time a new process (corresponding to a state change) is instantiated. Using this information the communication master updates *tmContainer* by removing and/or adding timeout events depending on the newly entered states of the executing model.

```
<ComMgr_process>:= 'process ComManager['<comm_gates>']'
   '('<synch_params>'): noexit:='
   <synch_process_body> 'endproc'

<synch_process_body> := <get_into_stable_state>'[ ]'
   <consume_timeout_event> '[ ]' <consume_external_evnets>

<comm_gates>:= 'commGate, inGate, toQueue, fromQueue,
     outGate, timeGate'
<synch_params>:= 'stable_'<model_id>
   {',stable_'<model_id>} ', tmContainer:TimeContainer'
```

Figure 5.4: Communication master process definition with timing constructs

The execution of timeout transitions (Figure 5.5) is similar to that of normal events but after the consumption of the timeout event and before instantiating the communication master again, the value of the elapsed timeout is broadcasted to all the models in the system. Such a transition can be fired only if at least

one of the models in the system is in a state from where a timeout transition can fire ("not(t_empty(SyncTimeQueue))") and the timeout has the smallest value among all the other possible timeouts ("msg == min(tmContainer)"). After firing the transition, the communication master will broadcast the value of the timeout to all other models in the system ("<propagate_timing_info>") and update the timing container accordingly (subtract the value of the timeout from all the other parameters of the timeout events and remove the *elapsed* timeout from the container).

```
<consume_timeout_event>:=
   'inGate' '?tr:Transition' '?msg:TmEvent'
   '['<tm_guard>']';<propagate_timing_info>';'
    <allow_trans_execution>

<tm_guard>:= 'not(t_empty(tmContainer))' 'and'
   '( msg = min(tmContainer))'
<propagate_timing_info>:=
   'time' '!getParam_timeout(min(tmContainer))'
     '!'<model_id>';'
   {'timeGate' '!getParam_timeout(min(tmContainer))'
     '!'<model_id>';'}
```

Figure 5.5: Communication master process consuming timeout events

5.2 Experimental Results

We used the proposed transformation (presented in Chapters 4 and 5) for deriving LOTOS representations of several UML models. We used three real-world examples (Flasher, Diagnosis and KeylessEntry) originating from the automotive domain and four more from literature. Table 5.1 presents the data regarding the obtained specifications.

Table 5.1: LOTOS Transformation - Model Statistics

Model	*SCNo*	*TrNo*	*StNo*	Tr_{flat}	St_{flat}	*Procs*	*LoC*
Flasher	6	34	14	72	19	30	2800
Diagnosis	4	38	17	44	14	21	1800
KeylessEntry	3	35	22	43	13	19	1470
Microwave	2	34	12	37	10	15	1170
LoanApproval	2	22	15	22	15	20	1210
ConferenceProt	3	41	18	41	18	24	1660
TelCtrlProt	2	55	26	70	21	26	2100

The first column of the table contains the name of the model. Column $SCNo$ presents the number of communicating statecharts of the model. Columns $TrNo$ and $StNo$ contain the number of transitions and states of the non flattened models. The number of transitions and states of the flattened version of the models can be found in columns Tr_{flat} and St_{flat} respectively. Column $Procs$ contains the number of LOTOS processes derived from the model. The last column (LoC) contains an approximation of the number of lines of code in the LOTOS specification.

Chapter 6

Test Case Generation

Parts of this chapter have been published in "Model Based Test Case Generation for Distributed Embedded Systems" [CW12] which is joint work with Franz Wotawa.

Testing activities usually consume an important part from the resources of software development projects (estimated to be of up to 50% [Mye04]). Thus it is desired to automate as much as possible from this task. Activities like test case design are a good candidate for increasing the automation degree of the testing process.

As already mentioned, one important advantage of MBT is the fact that it offers a higher degree of automation of the different testing activities. Thus, test cases are no longer designed manually but are automatically generated from the model of SUT's behavior. In the rest of this chapter we describe a method for automatically generating test cases aiming at structural coverage (state and transition coverage) of the model. We also show how to semi-automatically generate test cases by making use of user provided annotations on the UML model.

6.1 Input Output Conformance Relation and TGV

The Input Output Conformance theory (IOCO) as described by Tretman [Tre08] formalizes a set of implementations that behave consistently with a specification. The observations of a system during testing (also called traces) represent the visible behavior of a system. Informally, the IOCO relation states that an implementation I conforms to specification S if after every trace, I exhibits at least the same outputs as S.

The IOCO relation represents the base of the conformance testing theory used by the test case generation tool TGV. A formal and thorough description of this theory can be found in [Cal05]. Conformance testing aims at checking that the

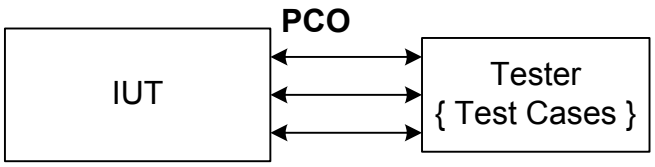

Figure 6.1: Testing and PCOs

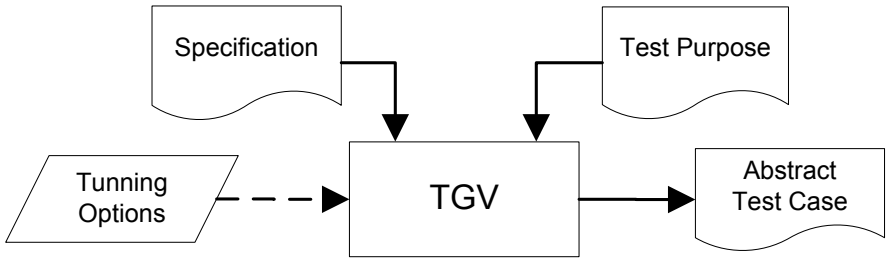

Figure 6.2: Functional view of TGV

visible behavior of an implementation under test (IUT) is correct with respect to a specification.

TGV allows the automatic generation of conformance test cases from formal specifications of reactive systems. By reactive we understand a software system which reacts to stimuli coming from its environment.

The IUT is considered a black box whose behavior is only observable by interaction with its environment. During the testing process, the environment is represented by the tester that controls and observes the behavior of the IUT through dedicated interfaces called points of control and observation (PCO). This can be observed in Figure 6.1 [JJ05]. Thus conformance testing is a type of functional testing of a black box nature.

In order to synthesize test cases, TGV (see Figure 6.2) requires the specification of the system under test. This specification is usually provided using a formal language like LOTOS whose syntax can be represented in terms of a labeled transition system (LTS).

TGV makes use of enumerative techniques (a labeled transition system representing the semantic of the LOTOS specification is generated on the fly) for test-case generation. This means that one of the biggest challenges is the well known state space explosion problem. In order to alleviate this problem and focus the generation process on particular aspects that need to be tested, TGV uses the concept of *test purposes*. A test purpose is a more abstract (simplified) description of such a scenario. A test purpose contains less information than the

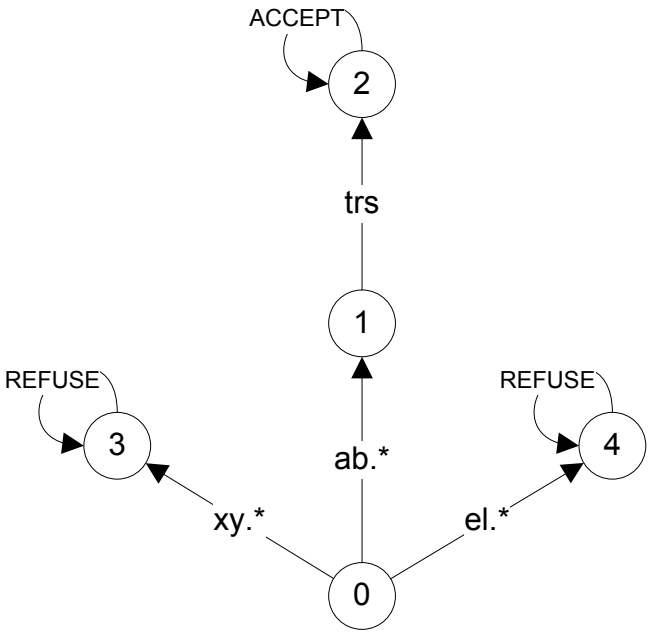

Figure 6.3: Test purpose example

test case it describes. For example in our setting a test purpose might specify a few transitions and the order they need to be visited in order to obtain a test case satisfying the test purpose.

An important part of such a test purposes are the *ACCEPT* and *REFUSE* states. *ACCEPT* states are used to identify desired behaviors, while *REFUSE* states are used to limit the exploration of the state space of the SUT by specifying which parts are not of interest for the current test purpose. It is obvious that the presence of more refuse states leads to a faster computation of test cases. Hence, the chances of running into the state space explosion problem are reduced.

A test purpose can be represented as an Input Output Labeled Transition System (IOLTS). The labels of the IOLTS can be specified as strings of characters. These strings can also contain regular expressions (e.g. *chx.** will match all actions starting with *chx*. We present an example for a test purpose in Figure 6.3. The described test purpose accepts sequences of actions starting with *ab* (*ab.**). Once such an action has been encountered, the next accepted one is *trs*. The test purpose refuses actions starting with *xy* or *el* (*xy.**, *el.**).

The basic components of a test case are interactions through PCOs [Cal05]:

- **outputs** are stimuli used to control the IUT's input events;
- **inputs** are observations of the IUT's outputs.

Another required input is the definition of the input/output alphabets of the specification representing the input and output actions of the system.

The inputs of the test case may lead to different verdicts:

- **Fail** - is returned when the IUT does not conform to its specification. In such cases we say that the IUT is rejected by the test case.
- **Pass** - is returned if the observation of the behavior of the IUT is correct thus fulfilling the test purpose.
- **Inconclusive** - is returned if the IUT behaves correctly but it is impossible to fulfill the test purpose. This can happen in case of non deterministic systems that may have a choice between several outputs to the same input.

6.2 Test Purpose Generation

The specification of good test purposes is not an easy task and requires significant effort. In previous work [dBRS+00], the authors were not able to design test purposes (even after ten hours) good enough as to uncover all seeded faults in a specification. One approach to this problem that reduces the effort required to design the test purposes is the generation of test purposes aimed at structural coverage of the specification.

6.2.1 Coverage aimed Test Purpose Generation

Considering coverage metrics we focus on transition and state coverage on the flattened representation of the statechart. Transition coverage on the flattened model. Flat Transition Coverage (FTC), has the advantage that it subsumes transition coverage on the original model.

Under the FTC metric a transition T originating from composite state S is considered covered if all of its copies generated during the flattening process are covered. On the non flattened version of the statechart this is equivalent to the firing of T from every state inside S. Under normal transition coverage metric (at the non flattened level) T is considered covered after being fired from at least one state inside S.

Even if it subsumes normal transition coverage FTC has the disadvantage that some transition copies are not reachable in the flattened model. This originates from the fact that the firing of some transitions originating from a composite state is conditioned by the current simple state inside the composite one. This means that certain combinations between simple states and transition copies are not possible even if the flattened model contains them.

Due to the fact that during the transformation we preserve the traceability between the UML model and the LOTOS specification we are able to generate

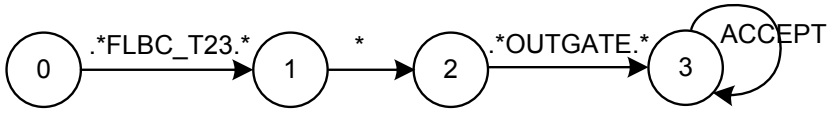

Figure 6.4: Test purpose for transition FLBC_T23

test purposes aimed at covering the original UML model. The specification of a test purpose aimed at covering a specific transition in the UML model comes quite straight forward on the LOTOS specification. Figure 6.4 contains the LTS representation of a test purpose generated for covering transition *FLBC_T23* from state *EmergencyOperation* in the *FLBC* statechart (Figure 6.7).

In the definition of the test purpose for covering transition $T23$, the label *.*FLBC_T23.** contains the id of the searched transition while *.*OUTGATE.** represents the action needed to get the values of the variables used in the statechart after triggering the transition *FLBC_T23*.

The generation process will stop as soon as an action matching the expression *.*FLBC_T23.** followed by one matching *.*OUTGATE.** is encountered. The edge *(1, *, 2)* is inserted as to accept several actions between the two described above. This is needed due to the fact that between firing a transition *(.*FLBC_T23.*)* and providing the output *.*OUTGATE.**, the model might also provide actions for enqueuing events in the event queue.

6.2.2 Test Purposes for Scenario based Testing

Due to the fact that a test suite providing high coverage of the model does not guarantee that the system is well tested further mechanisms for the specification of test purposes are needed.

In this section we present an approach for test case generation where test purposes are derived from annotations made on the model by the user. These annotations provided the order in which certain UML elements (states and/or transitions) needed to be visited in order to fulfill a specific scenario of interest.

Test Purpose Specification

The specification of test purposes is done by annotations on the model providing the order in which certain UML elements need to be visited. This can be regarded as an abstraction of the scenario of interest where the amount of information provided is always dependent on the user. It is the task of the tool to compute the test case containing the transitions need to be fired in all models so that the test scenario is fulfilled.

The annotation of the model is done using UML tags which are pairs consisting of two elements - a name and a value. The name of the tag will represent the id of

the scenario to cover with the generated test case. The annotated UML elements can be states and/or transitions. In order to specify a test scenario there are two types of annotations: inclusion and exclusion of UML statechart elements.

The description of such a test scenario is given in Figure 6.5.

```
<tp_scenario>:= <uml_tag>‘,’{<uml_tag>}
<uml_Tag>:= <included_item>|<excluded_item>
<included_item>:=‘IN =’ <number>
<excluded_item>:=‘EX =’ <number>‘−’<number>|<number>
<number> = ‘1’|‘2’|‘3’| ....
```

Figure 6.5: Representation of test purpose scenario

The semantic intended for the **included elements** is that they need to be part of the test case in the order provided by the IN annotation with regard to the other annotated elements.

The **excluded items** are used to cut off parts of the specification that are not desired for the current test scenario but can also be used to limit the dimension of the searched state space. The annotation *EX* specifies the ranges of items for which the current annotated element is excluded. For example *EX = 3 - 6* denotes that the excluded item and the behavior it triggers will be ignored during the test case search while investigating items with IDs between 3 and 6.

We formally define a test purpose *tp* (Definition 6.2.2) as an ordered sequence of test purpose items tpi_j (Definition 6.2.1)

Definition 6.2.1. (Test Purpose Item). A **test purpose item** *tpi* is a pair (IN, EX) where $IN \subset TR$ and $EX \subset TR$ are the included and excluded transitions of *tpi*. TR is the set of transitions in the model (from all statecharts).

Definition 6.2.2. (Test Purpose). A **test purpose** *tp* is a sequence $TPI = (tpi_0, tpi_1 ... tpi_n)$ of test purpose items where $\forall tpi_j : 0 \leq j \leq n : order(tpi_j) < order(tpi_{j+1})$.

Consider for example the scenario for which we need a test case to test the functionality where during a hazard warning blinking the user requests direction indication for the left hand side (see Figure 6.7). In order to provide the information for the test purpose, we need to annotate the transitions:

- *FLBC_ T23* - activate the hazard warning.
- *FLBC_ T16* - activate left side direction indication.

In order to help the search for the test case, we also mark as excluded item the transition *FLBC_ T12* - from state *Emergency_ Operation* to *Normal_ Operation*

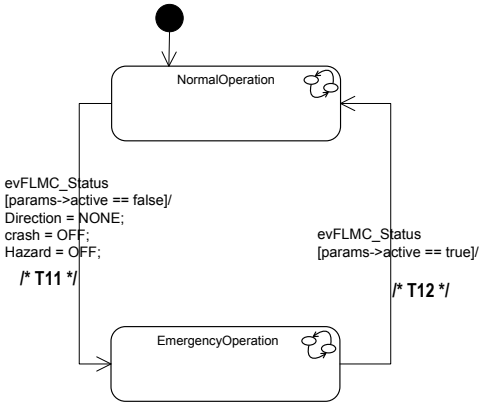

Figure 6.6: Statechart of the FLBC class

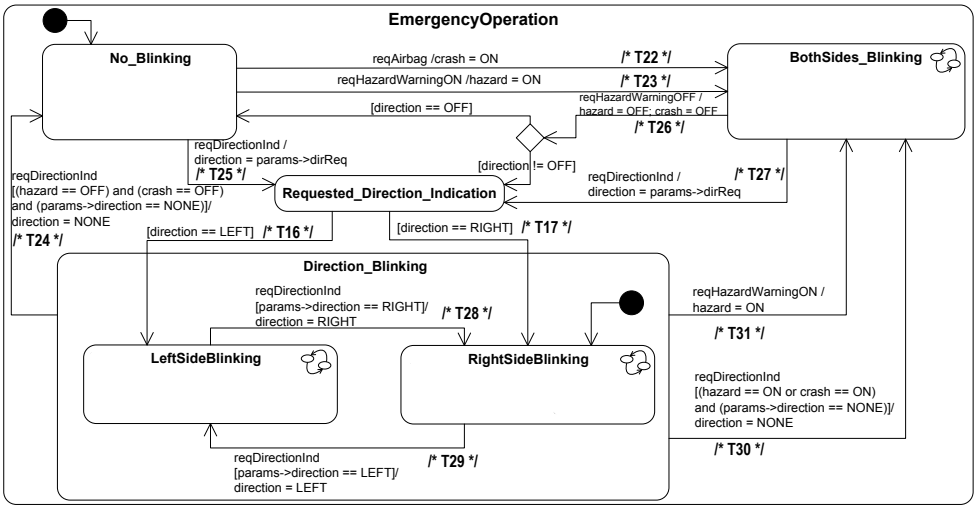

Figure 6.7: FLBC EmergencyOperation state

(in Figure 6.6) in order to prevent the system to return to the normal state after the hazard warning has been enabled. This transition is excluded only for *FLBC_ T16*.

In Figure 6.8 we present the annotations for the considered scenario. Each line in the example contains the id of the item on which the annotation was made (*FLBC_ T23*) followed by the tag name (*R_ Id_ 1*) and its value (*"IN = 1"*).

Test Purpose Representation

Test purposes are generated by using the annotations describing the test scenarios. A first step for the transformation of the annotations into a test purpose is done by moving the annotations on the states to all the transitions targeting them.

```
FLBC_T23 : R_Id_1 = ''IN = 1'';
FLBC_T16 : R_Id_1 = ''IN = 2'';
FLBC_T12 : R_Id_1 = ''EX = 2'';
```

Figure 6.8: Example of test scenario annotations

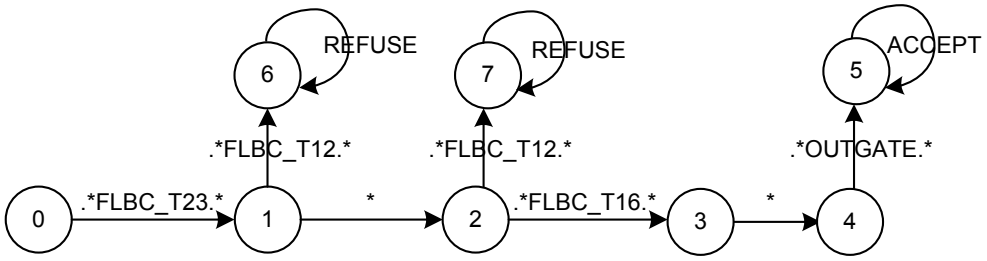

Figure 6.9: Test purpose for the scenario hazard warning followed by left side direction indication

Thus we obtain a specification of the test scenario only in terms of desired and excluded transitions.

The test purpose corresponding to the annotations for the example considered in Section 6.2.2 can be seen in Figure 6.9.

The included and excluded UML transitions are represented by labels on the edges of the IOLTS defining the test purpose. Edges representing excluded items lead to *REFUSE* states so that, during the test case generation TGV will not explore the parts of the model reachable via these edges. The node targeted by the last included transition leads to an *ACCEPT* node providing the termination criterium for the test case generation process.

The edges of the test purpose with label * accept all actions (except the excluded ones) during the generation and specify the fact that between two included items, there can be an arbitrary sequence of edges.

Figure 6.10 depicts a partial representation of the test case generated for the considered test scenario.

6.3 Experimental Results

In this section we present the results obtained by using the presented test purpose generation techniques. For this task we have used three real-world examples (Flasher, Diagnosis and KeylessEntry) originating from the automotive domain and four more from literature (described in Chapter 3).

Table 6.1 contains the results obtained when using the current test purpose generation technique aimed at transition coverage.

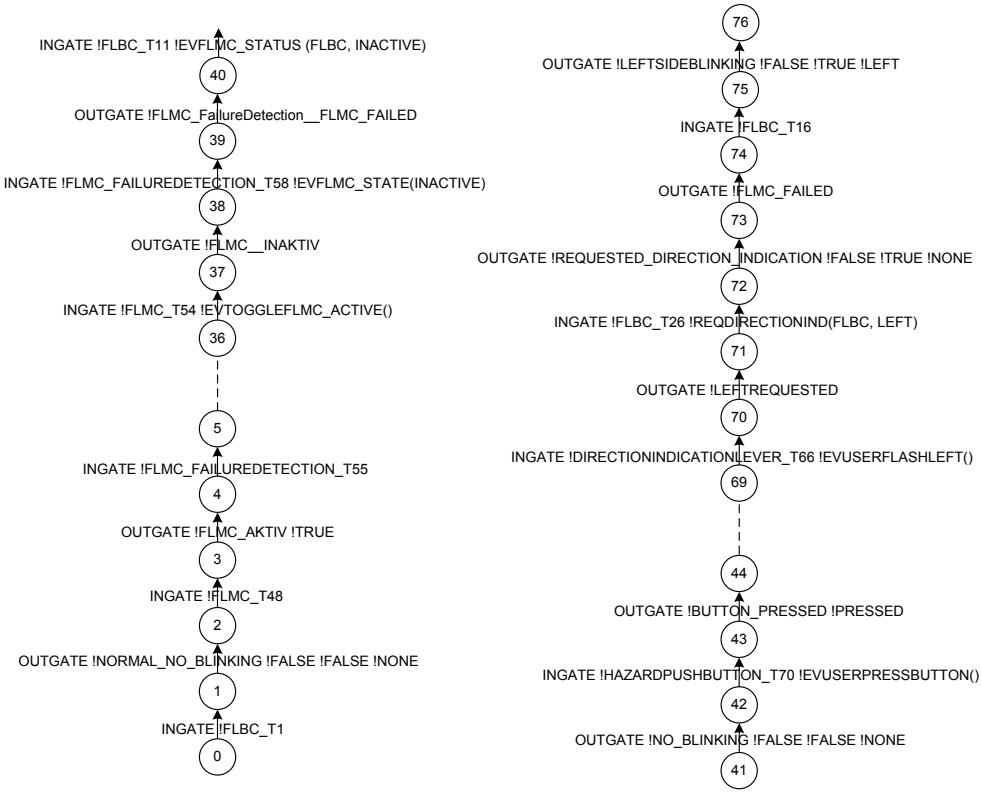

Figure 6.10: Test case for hazard warning - left side direction indication

The first column of the table contains the name of the model for which the test purposes were generated. The next column (TPs) contains the total number of generated test purposes. Column $ValidTPs$ presents the number of valid test purposes. By valid test purpose we mean test purposes not targeting transition copies that are not reachable on the flattened model.

In column TCs we give the number of generated test cases. We imposed a limit of 25 minutes per test purpose. If no test case was generated within this time, the generation process is stopped. Column $CompTime$ contains the time TGV was allowed to run for the generation process. The last two columns ($TCov_{flat}$ and $TCov$) contain the transition coverage on the flattened and non flattened model respectively.

In the case of three of the models (Diagnosis, MicrowaveOven and ConferenceProtocol), the search for some valid test purposes did not succeed within the imposed time limit. This happened due to the size of the searched state space. In order to cover the searched transition we manually specified the required test purposes equipped with several refuse states limiting the searched state space.

We present the results of using the presented technique of deriving test purposes from user annotations in Table 6.2. As expected, the generation for test

Table 6.1: Coverage aimed test case generation results

Model	*TPs*	*ValidTPs*	*TCs*	*CompTime*	$TCov_{flat}$	*TCov*
Flasher	72	70	70	58m10s	97%	100%
Diagnosis	44	42	38	1h43m50s	95%	97%
KeylessEntry	43	39	39	2m38s	91%	100%
MicrowaveOven	37	37	36	27m20s	97%	97%
LoanApprovalWS	22	22	22	1m20s	100%	100%
ConferenceProtocol	41	41	39	1h42m27s	95%	95%
TelCtrlProtocol	65	65	65	4m6s	100%	100%

purposes with manually defined *REFUSE* states is faster than for the generated ones which are missing such states. Of course the result of the generation process when using refuse transitions depend on the excluded items in test purposes and also on the size of the state space they define.

Table 6.2: Scenario based test purposes results

Model	*TPs*	*TCs*	*Time*
Flasher	51	51	10m17s
Diagnosis	32.	23	4m41s
KeylessEntry	15	15	52s
MicrowaveOven	15	15	57s
LoanApprovalWS	15	15	56s
ConferenceProtocol	15	15	15m5s
TelCtrlProtocol	15	15	55s

Chapter 7

Enhancing Test Purposes

Parts of this chapter are taken from "Using Dependency Relations to Improve Test Case Generation from UML Statecharts" [CW13] which is joint work with Franz Wotawa.

In model-based testing the size of the used model has a great impact on the time for computing test cases. In model checking, dependence relations have been used in slicing of specifications in order to obtain reduced models pertinent to a criterium of interest. Usually in specifications described using state based formalisms slicing involves the removal of transitions and merging of states thus obtaining a structural modified specification. Using such a specification for model based test case generation activities where sequences of transitions represent test cases might provide traces that are not valid on a correctly behaving implementation. In order to avoid such trouble, we suggest the use of control, data and communication dependences for identifying parts of the model that can be excluded so that the remaining specification can be safely employed for test case generation. This information is included in test purposes.

As already mentioned in Chapter 6 TGV requires test purposes in order to focus the generation of test cases on particular aspects of the system. This also allows for reducing the search of the model's state space during test case generation. As already mentioned, TGV uses special predefined labels in the test purposes in order to control the test case generation process. One of these is the *REFUSE* label, which is used to mark parts of the model that should not be explored in a particular test-case generation process. From here on we refer to transitions leading to refuse states as simply refuse transitions.

Because TGV makes use of enumerative techniques (an IOLTS representing the semantic of the LOTOS specification is generated on the fly) for test-case generation it is obvious that the presence of more refuse transitions leads to a faster computation of test cases. Hence, the chances of running into the state space explosion problem are reduced.

In Chapter 6 we presented a method for automatically generating test cases aiming at structural coverage (state and transition coverage) of the model. We also showed how to semi-automatically generate test cases by making use of user provided annotations on the UML model. The coverage generated test cases did not contain any refuse transitions. Therefore, the generation process was not as efficient as in the case when the user provides annotations that can be used as refuse transitions in the test purpose.

In this chapter we describe how to use different dependence relations (control, data, and communication) in order to automatically identify parts of the models that can be omitted during the generation process. This is done by computing direct and indirect dependences for the transitions we aim to cover. The dependence information is used to insert refuse transitions in the test purposes, and thus reducing the searched state space during the generation process.

The way of using these dependences differs from other approaches [ACH+09, LG08, JW02] where the models are sliced according to a slicing criterium. These approaches have proven their worth in situations involving techniques like model checking and debugging. In our approach we do not modify the structure of the model. Instead we enhance the test purpose with refuse transitions that capture those parts of the model that are not going to be explored. The modified test purposes focus test case generation and thus indirectly cut out transitions and states. Thus the specification is sliced during the generation process by not exploring its excluded parts. This is different to slicing approaches which eliminate transitions (by merging their source and target states) on which the slicing criteria (transitions and/or variables) do not depend.

By using classical slicing techniques for test case generation, the obtained sequences representing test cases might not be valid on the original model. Another drawback is the need for generating a model for each considered criteria and then using it for the purpose at hand. Hence, depending on the complexity of the model the effort needed for compiling every variant of the model might be time consuming. In our approach we use the same model, which needs to be compiled only once. There is however in our approach the possibility that the number of eliminated transitions might be smaller than in the case of the other slicing approaches. This depends however on the size and structure of the model.

7.1 Dependences

In literature there are several different control, data and communication dependence definitions for state based models. This variation in definitions has several causes. Some of these are due to the different syntax and/or semantics of the models (Input Output Transition Systems [LG08], Extended Finite State Machines (EFSM) [ACH+09], IF language [BFG03]etc.)). Some are caused by the

type of system whose behavior has to be described for example, if embedded systems with cyclic behavior are considered.

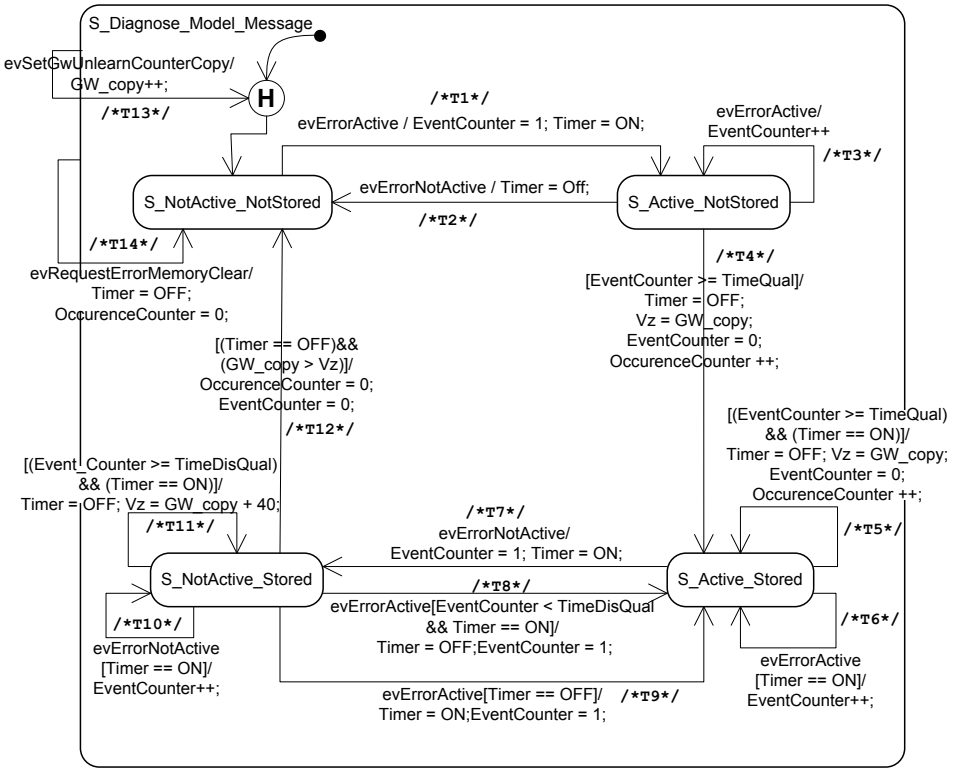

Figure 7.1: Statechart of Diagnosis functionality

As running example for the rest of this chapter we are going to use the model depicted in Figure 7.1, which describes the diagnosis functionality of modern vehicles. Its purpose is to store the type, occurrence, and origin of errors occurring during operation of the vehicle. Since the diagnosis functionality is distributed over several ECUs, the description of its behavior is also modeled using communicating UML statecharts. Besides the diagnosis functionality our system also contains models describing the behavior of the ignition switch of the vehicle. Two other models define the conditions needed for errors to be detected. When such an error has been detected it is communicated to the diagnosis model. The functionality, which is to be tested in this setting defines how detected errors are to be treated and when to create an entry in the error memory. The statechart model consists of five states and accepts four messages namely *evErrorActive*, *evErrorNotActive*, *evRequestErrorMemoryClear* and *evSetGwUnlearnCounterCopy*.

The state *S_ NotActive_ NotStored* corresponds to normal functioning when no error is detected. After an error is detected, the system moves to the state *S_ Active_ NotStored*, which means that an error has been detected but is not

yet stored. The error is stored after receiving five *evErrorActive* events, and the system moves to the *S_ ActiveStored* state. This means that the real ECU has now stored the error in its buffer, which might be read out using a dedicated diagnosis hardware. The diagnosis module shall leave this state and move to the *S_ NotActive_ Stored* state only after receiving an *evErrorNotActive* event.

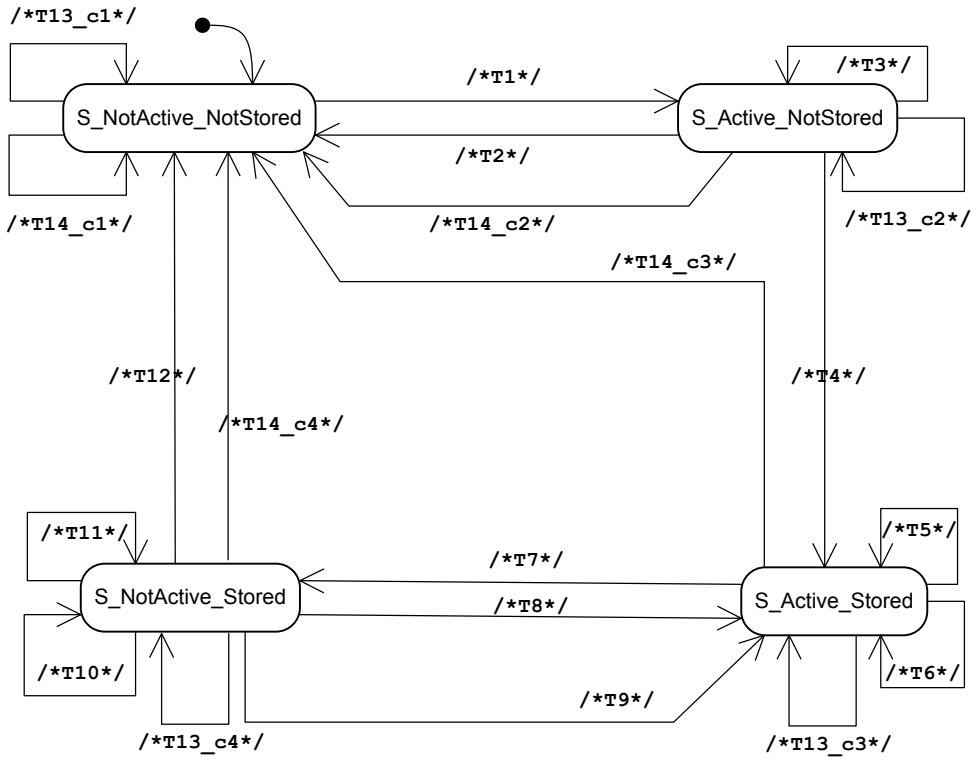

Figure 7.2: Flattened representation of Diagnosis

We use the transformation rules defined in Chapter 4 in order to derive a LOTOS specification from the diagnosis model. As already mentioned, the flattening process removes the hierarchical structures and the pseudostates preserving the execution semantics of the models. This step delivers a behavioral equivalent statechart using only simple states and transitions $SC = (S_s, T, V, i_{ps0})$. Figure 7.2 presents the flattened representation of the diagnosis statechart from Figure 7.1. Due to readability reasons the labels of the transitions in Figure 7.2 have been omitted, however they remain the same as in the original model. During the flattening process transition copies are created for the transitions originating from composite states. Each such transition generates a copy of itself for each state contained by the composite state. In our running example (Figures 7.1 and 7.2) transition $T14$ generates the copies $T14\_c1$, $T14\_c2$, $T14\_c3$ and $T14\_c4$.

7.1.1 Control Dependence

Informally, in classical definitions of control dependence for sequential programs a statement s_j is control dependent on a statement s_i if statement s_i causes the execution of statement s_j. Such definitions impose that the control flow graph (CFG) of the program meets certain properties. The presence of a final node in the CFG is one such restriction. However in the case of EFSM there might be the case where either no such node exists or several such nodes are present.

This restriction can be lifted [RAB+07] by considering two definitions for control dependence for structures with zero or more end nodes. The new control dependences are not given in terms of paths to one final node but in terms of maximal (see Definition 7.1.1) or sink bounded paths (see Definition 7.1.3) of the CFG.

The first definition *Nontermination-Sensitive Control Dependence* is given in terms of maximal paths (which can also be infinite paths) and takes into consideration the fact that a potential infinite execution of a loop (an infinite path) may impede the execution of other nodes and thus providing a control dependence. Informally a node n_j is control dependent on node n_i if the execution of one branch of n_i will always lead to n_j and the execution of another branch (a maximal path) might not lead to n_j (n_j might not be executed). In Equation 7.1, $MaximalPahts(SC)$ denotes the set of maximal paths in the flattened statechart SC.

Definition 7.1.1. (Maximal Path). A path π is maximal if it terminates in an end state (state with no outgoing transitions) or is infinite.

$$\begin{aligned} MaximalPahts(st) = \{\pi | \pi = (t_i, t_{i+1}, ..., t_n) : source(t_i) = st \wedge \\ \pi \in MaximalPahts(SC)\} \end{aligned} \tag{7.1}$$

The second control definition is similar to the first one differing only in terms of the considered paths. Making use of sink bounded paths (Definition 7.1.3) it does not consider the situation where a loop might execute infinitely and thus it is nontermination-insensitive. Informally a control sink (Definition 7.1.2) is a region of the EFSM that once entered is never left. This can also be an end node of the EFSM or a state with no outgoing transitions. This definition is closer to the classical definition of control dependence given that the paths are always finite.

Definition 7.1.2. (Control Sink). A *control sink*, K, is a set of nodes that form a strongly connected component (SCC).

Definition 7.1.3. (Sink-bounded Paths). A maximal path π is sink bounded iff there exists a control sink K such that:

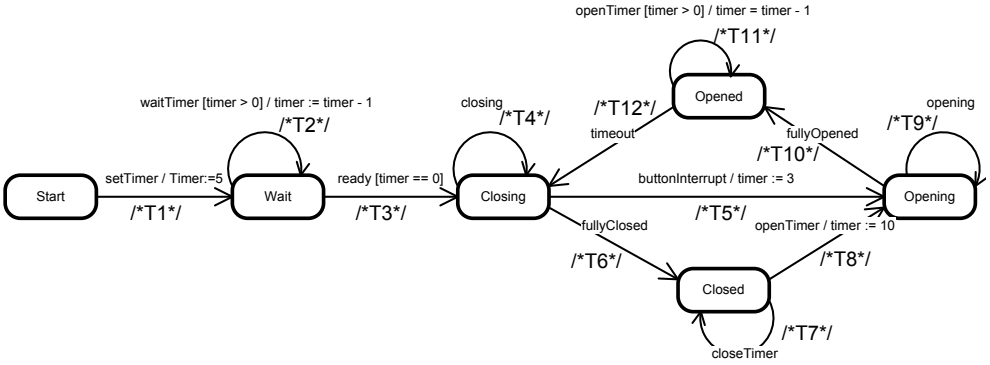

Figure 7.3: EFSM of Elevator Door Control system

1. π contains a node form K and

2. if π is infinite then all nodes in K occur infinitely often.

Figure 7.3 depicts the EFSM specification for a door control of an elevator system [ACH+09]. We are going to use it for exemplifying the differences between the control dependence definitions. In Figure 7.3 the SCC formed by the states *Closing*, *Closed*, *Opening*, *Opened* is such a control sink and the path *Wait*, *Closing*, *Closed*, *Opening*, *Opened* is a sink-bounded path. The path *Closing*, *Opening*, *Opened*, *Closing* is a maximal path. It is not a sink bounded path since the SCC *Closing*, *Opening*, *Opened* can be exited by going to state *Closed*. To be self-contained we informally introduce SCC. A SCC is a directed sub-graph of the original graph where there is a path from every node to every other node of the sub-graph.

Both afore mentioned control definitions are computed on CFGs and provide dependences between nodes of the CFG. EFSMs however have different semantics and properties and thus the definitions need to be adapted. Decisions (Boolean conditions) are made in CFGs at node level whereas in EFSMs they are made at the level of transitions. Therefore, [ACH+09] adapted and extended the nontermination-insensitive control dependence for EFSM providing the *Unfair Non-termination Insensitive Control Dependence* (Definition 7.1.5). They do this by removing the fairness condition of Definition 7.1.3 (the second condition) and give the definition of unfair sink-bounded paths. In Equation 7.2 below, $UnfairSinkPaths(SC)$ denotes the set of unfair sink-bounded paths in the flattened statechart SC.

Definition 7.1.4. (Unfair Sink-bounded Paths). A maximal path π is an unfair sink bounded path iff there exists a control sink K such that π contains a transition from K.

$$UnfairSinkPaths(st) = \{\pi | \pi = (t_i, t_{i+1}, ..., t_n) : source(t_i) = st \wedge \pi \in UnfairSinkPaths(SC)\} \quad (7.2)$$

Definition 7.1.5. (Unfair Non-termination Insensitive Control Dependence (UNTICD)). $t_i \xrightarrow{UNTICD} t_j$ means that t_j is control dependent on a transition t_i iff t_i has at least one sibling t_k such that:

1. for all paths $\pi \in UnfairSinkPaths(target(t_i))$, the node $source(t_j)$ belongs to π;
2. there exists a path $\pi \in UnfairSinkPaths(source(t_k))$ such that $source(t_j)$ does not belong to π.

According to Definition 7.1.5 in Figure 7.3 we have the following control dependences : $T5 \xrightarrow{UNTICD} T9, T10$, $T6 \xrightarrow{UNTICD} T7, T8$, $T8 \xrightarrow{UNTICD} T9, T10$, $T10 \xrightarrow{UNTICD} T11, T12$ and $T12 \xrightarrow{UNTICD} T4, T5, T6$.

As already mentioned, the usage of maximal paths (especially infinite ones) is supported by the fact that a potentially infinite execution of a loop may impede the execution of other nodes. Another important observation of [RAB+07] (also acknowledged in [LG08]) is that reaching a start node in a reactive system is analogous to reaching an end node in a program, i.e., the behavior will start again. Since these observations are also valid in our setting we are going to use the adapted version of the nontermination-sensitive control dependence (Definition 7.1.6) in order to identify control dependences of our models.

Definition 7.1.6. (Non-termination Sensitive Control Dependence (NTSCD)). $t_i \xrightarrow{NTSCD} t_j$ means that t_j is non termination sensitive control dependent on a transition t_i iff t_i has at least one sibling t_k such that:

1. for all paths $\pi \in MaximalPahts(target(t_i))$, the $source(t_j)$ belongs to π;
2. there exists $\pi \in MaximalPahts(source(t_k))$ a path such that $source(t_j)$ does not belong to π.

Considering the flattened representation of the diagnosis model (Figure 7.1) both UNTICD and NTSCD would deliver the same control dependences. This is so due to the fact that the whole model is a Strongly Connected Component (SCC). In such a case both definitions deliver the same results.

Compared to UNTICD, in the elevator door control EFSM in Figure 7.3 we have the same control definitions in the SCC *Closing*, *Closed*, *Opening*, *Opened* but also some other relations outside the SCC. The extra $T3 \xrightarrow{NTSCD} T4, T5, T6$ relations (not given in the case of UNTICD) are provided by NTSCD due to the presence of the infinite loop of transition $T2$ at state $Wait$.

It is possible to prove that inside control sinks NTSCD and UNTICD provide the same results [ACH+09]. Outside the control sinks UNTICD finds control dependences only if it can find alternative sink bounded paths bypassing the targeted transitions. If UNTICD is used for slicing, it might provide smaller slices because it does not consider infinite paths that do not end in control sinks. NTSCD on the contrary might finds more dependences (providing larger slices) exactly due to the fact that it also takes into consideration infinite loops. This is a good thing in our setting since any alternative path that might not influence the targeted transition is a potential refuse transition for our test purpose.

Computing Control Dependence

Input: $SC = (S_s, T, V, st_i)$
Output: CD(T), PI(T)

for all $t \in \{t | t \in T(SC) \wedge \exists t_s \in sibling(t) : target(t_s) \neq target(t)\}$ **do**
  for all $s_c \in \{s_c | \exists t_i \in \bigcap maxPaths(t) : source(t_i) = s_c \vee target(t_i) = s_c\}$ **do** // for common nodes on all paths
    if $\forall t_{st} \in sibling(t) \exists \pi \in maxPaths(t_{st}) : \forall t_i \in outgTr(s_c) \rightarrow t_i \notin \pi$ **then**
      for all $t_o \in \{t_o | t_o \in T(SC) \wedge source(t_o) = s_c\}$ **do**
        $CD(t_o) \leftarrow CD(t_o) \cup t$
        for all $t_{st} \in \{t_{st} | t_{st} \in sibling(t) \wedge \exists \pi \in maxPaths(t_{st}) : \forall t_i \in outgTr(s_c), t_i \notin \pi\}$ **do**
          $PI(t_o) \leftarrow PI(t_o) \cup t_{st}$
        end for
      end for
    end if
  end for
end for

Figure 7.4: NTSCD computation algorithm

The algorithm for computing the NTSCD is depicted in Figure 7.4. The used symbols and an informal description are presented bellow. We apply this algorithm for every flattened statechart $SC = (S_s, T, V, st_i)$ in our model.

Because we consider maximal paths (which include also infinite paths) we need to identify end states (states with no outgoing transitions) and cycles in our model SC. The end states can be easily found by enumerating through the states of the model and testing for the absence of outgoing transitions. For the identification of cycles we use a slightly modified version of depth first search (DFS) that at

every re-occurrence of an already visited state s_i saves in a list of sets all the states present on the stack starting with the first occurrence of s_i. After this step we obtain the set $CYCLES(SC)$ whose elements are sets of states representing the cycles in the model SC.

For transitions that are not part of a cycle, all nodes of the cycle that are targeted by transitions whose sources are not part of the cycle can be considered sink states. Thus we define $SINKS(t)$ (Equation 7.3) as the set containing all such states, end states and also the initial state st_i [RAB+07, LG08]. In the algorithm we also make use of the function $maxPaths(t)$ (Equation 7.4) which provides all maximal paths starting with transition t.

$$\begin{aligned} SINKS(t) = &\{s | s \in S_s(SC) \wedge |outTr(s)| = 0\} \cup \{st_i\} \cup \\ &\{s | s \in S_s(SC) \wedge \exists C \in CYCLES(SC) : \\ &(s \in C \wedge source(t) \notin C)\} \end{aligned} \tag{7.3}$$

$$maxPaths(t) = \{(t_1, t_2, ...t_n) | t_1 = t \wedge target(t_n) \in SINKS(t)\} \tag{7.4}$$

The algorithm for computing the NTSCD (Figure 7.4) requires as input a flattened statechart $SC = (S_s, T, V, st_i)$ and provides as output two key value maps:

1. CD(T) - key value map containing as key a transition and as value a set of transitions on which t is control dependent on;

2. PI(T) - key value map containing as key a transition and as value transitions that potentially do not influence t (from the NTSCD point of view).

The transitions with at least one sibling (statement 1) might NTSCD control other transitions. Only transitions originating from the states that appear in all maximal paths (statement 2) of transition t might be control dependent on t.

If there exists at least a sibling t_{st} of transition t that has at least one maximal path, which does not contain the considered state s_c, all the outgoing transitions t_o of s_c ($outgTr(s_c)$) are NTSCD control dependent on transition t (statements 4 - 5). Since we need as test cases sequences of transitions that are valid on the specification, the transitions on maximal paths starting at t and containing s_c are also added to the list of transitions $outgTr(s_c)$. For simplicity reasons this is not explicitly depicted in the algorithm. Also all the sibling of t that possess at least one maximal path bypassing s_c are added as potential independent transitions for t_o (statements 6 - 7).

7.1.2 Data Dependence

Classical data dependence definitions are given in terms of variable definitions and uses. Thus in terms of EFSM a variable is used on a transition if its value appears in the guard of the transition or appears on the right hand side of an assignment in the action of the transition. A variable is defined if it is assigned a value when the respective transition is fired.

We adopt the data dependence definition of [ACH+09] since the used formalism of EFSM is very similar to the representation we obtain after the flattening of the statecharts. Other data dependence definitions are also available in the literature (IOSTS [LG08]) but they are adapted to other formalisms that differ from the one we use.

Definition 7.1.7. (Data Dependence(DD)). $t_i \xrightarrow{DD}_v t_k$ means that transition t_i and t_k are data dependent with respect to variable v if the following conditions hold:

1. $v \in D(t_i)$, where $D(t_i)$ is the set of variables defined by actions of transition t_i;

2. $v \in U(t_k)$, where $U(t_k)$ is the set of variables used in the guard and actions of transition t_k;

3. there exists a path in the EFSM from source(t_i) to the target(t_k) whereby v is not modified.

Due to the fact that we do not modify the structure of the specification, in addition to computing the data dependence between a transition t_i and t_k with respect to a variable v we are also interested in the definition free paths, i.e., paths from t_i to t_k along which v is not redefined. We compute these paths by using DFS to explore the model backwards starting from t_k and following the incoming transitions of $source(t_k)$. Each time a variable v used by t_k is defined we save the respective path and add the transitions on it to the set of transitions t_k is dependent on. The considered paths are all simple paths (are not allowed to visit the same state twice). Thus after the execution of such an algorithm the map $DD(t_k)$ will contain the transitions on which t_k is data dependent on.

Since we are only interested in the execution of certain transitions, we reduce the set of variables of interest to the ones used in the guards of the transitions (including the variables that directly or indirectly influence them). The rationale behind this is the fact that the truth value of the guards is the one that allows for the execution of the transitions. Thus we reduce the set $DD(t_k)$ to the set of transitions that directly or indirectly might influence (by defining variables in guards) the truth value of the guard of t_k.

7.1.3 Communication Dependence

Depending on the state based formalism used, there are several definitions for communication dependence under different names: synchronization [JW02], global synchronization [VLH07], or communication [LG08] dependence.

Out of these, the one closest to what we need in our setting is the one given by [JW02] called synchronization dependence. This definition is more general and is given in terms of states and transitions in concurrent models. Informally it states that if the trigger event of some transition in an element x (x can be a state or transition) is generated by the action of an element y, and the automatons of x and y are concurrent, then x is synchronization-dependent on y.

In our particular case the communication dependence only relates to transitions within concurrent models. Thus we adapt the definition of [JW02] to Definition 7.1.8.

Definition 7.1.8. (Communication Dependence(COMD)). Given two transitions $t_i \in T(SC_1)$ and $t_k \in T(SC_2)$, $t_i \xrightarrow{COMD}_v t_k$ means that transition t_k is communication dependent on t_i iff the following conditions hold:

1. SC_1 and SC_2 are two concurrent statecharts and

2. $trigger(t_k)$ - the triggering event of t_k is generated by the actions of t_i.

Informally a transition t_k is communication dependent on a transition t_i in a concurrent statechart if the execution of t_i will generate the triggering event of t_k.

The direct communication dependences are computed by iterating through the transitions whose actions generate events and adding these transitions to the key value map $COMD(t)$ where t is the transition triggered by the generated event.

7.1.4 Computing Indirect Dependence

After computing the direct dependences for each model in our specification we compute the indirect dependences given a set of transitions of interest. Informally transition t_j is indirectly dependent on transition t_i if there exists a sequence of dependences leading from t_i to t_j [LG08]. This represents the transitive closure between t_i and t_j considering the ID relation.

Definition 7.1.9. (Indirect Dependence (ID)). $t_i \xrightarrow{ID} t_j$ means that t_j is indirectly dependent on t_i iff there exists a sequence $(t_1, ..., t_k)$ where $t_1 = t_i$ and $t_k = t_j$ such that for all $1 \leq n \leq k$: $t_n \xrightarrow{NTSCD} t_{n+1}$ or $t_n \xrightarrow{DD} t_{n+1}$ or $t_n \xrightarrow{COMD} t_{n+1}$.

Input: $T(M)$, CD, DD, $COMD$, TI
Output: $T_{IND}(t)$ - set of transition t does not depend on
1: **for all** $t \in T(M)$ **do**
2: $T_{CTRL}(t) \leftarrow T_{CTRL}(t) \cup t$
3: $T_{IND}(t) \leftarrow PI(t)$
4: $finished \leftarrow$ **true**
5: **repeat**
6: $finished =$ **true**
7: **for all** $t_i \in \{T(M) \setminus T_{CTRL}(t)\}$ **do**
8: **if** $controls(t_i, T_{CTRL}(t)) =$ **true then**
9: $finished \leftarrow$ **false**
10: $T_{CTRL}(t) \leftarrow T_{CTRL}(t) \cup t_i$
11: $T_{IND}(t) \leftarrow T_{IND}(D) \cup PI(t:i)$
12: **end if**
13: **end for**
14: **until** $finished$
15: $T_{IND}(t) \leftarrow T_{IND}(t) \setminus T_{CTRL}(t)$
16: **end for**

Figure 7.5: Independent transitions computation algorithm

In Figure 7.5 we present the algorithm for computing the independent transitions (T_{IND}) as well as the indirect dependences (T_{CTRL}) for each transition t in the model M (containing the communicating statecharts). The inputs for the algorithm are the set of transitions in the model $T(M)$ for which the dependence relations (CD, DD, $COMD$) and potential independents (TI) have been previously computed. The algorithm computes $T_{IND}(t)$ - the set of transitions that do not influence t for every transition in the model.

The function $controls(t_i, T_{CTRL}(t))$ (statement 8) returns true if at least a transition in T_{CTRL} is control, data or communication dependent on t_i. The algorithm iterates through the transitions of M and when it finds a transition t_i for which $controls(t_i, T_{CTRL}(t))$ returns *true* it updates the sets T_{CTRL} and T_{IND} accordingly. The algorithm is repeated until no more transitions are found such that $controls(t_i, T_{CTRL}(t)) = true$. The set difference between T_{IND} and T_{CTRL} represents the transitions that do not influence (directly or indirectly) the transition t.

The algorithm finishes as soon as all indirect dependences have been found. In the worst case all transitions of the models are directly or indirectly dependent on each other. Thus, in this case T_{CTRL} will contan all transitions in the model and $T_{IND} = \emptyset$.

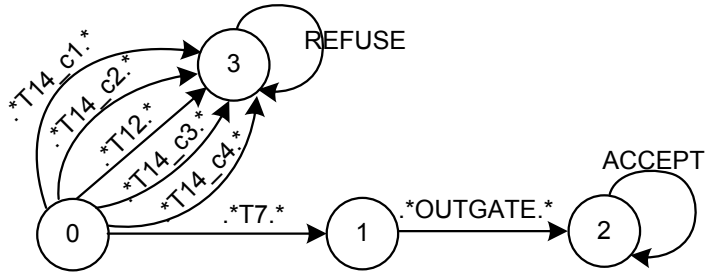

Figure 7.6: Test purpose for T7

7.2 Test Purpose Generation

In this section we describe the usage of the computed dependences for the test purpose generation. The generation process is similar to the one we presented in Chapter 6. The test purposes are also generated with the goal of achieving high transition coverage. This time we use the dependences in order to augment the test purposes with refuse transitions.

Since during the transformation we preserve the traceability between the UML model and the LOTOS specification we are able to generate test purposes aimed at covering the original UML model. Thus, Figure 7.6 contains the IOLTS representation of a test purpose generated for covering transition *T7* in Figure 7.2.

The refuse transitions in Figure 7.6 are identified by applying the algorithm presented in Figure 7.5 for computing the indirect dependences and independent transitions for each transition in the model. Figure 7.7 contains the test case generated by using the afore mentioned test purpose.

In the test purpose definition, labels of transitions are denoted by strings that can also be stated using regular expressions (e.g. *".*"* or *".*T7.*"*). The label *".*T7.*"* contains the id of the searched transition whereas *".*OUTGATE.*"* represents the action needed to get the values of the variables used in the statechart after triggering the transition *T7*.

The edges leading to state 3 are labeled with the IDs of the transitions *T7* does not depend on. The edge with the label "*REFUSE*" is used to mark parts of the model that will not be explored during the test generation process. Basically edges in the specification whose labels fit the (regular expression of) the labels of edges in the test purpose leading to the source of the *REFUSE* edge will not be explored (and thus neither the behavior that they lead to). The generation process will stop as soon as an action matching the expression *.*T7.** followed by one matching *.*OUTGATE.** is encountered.

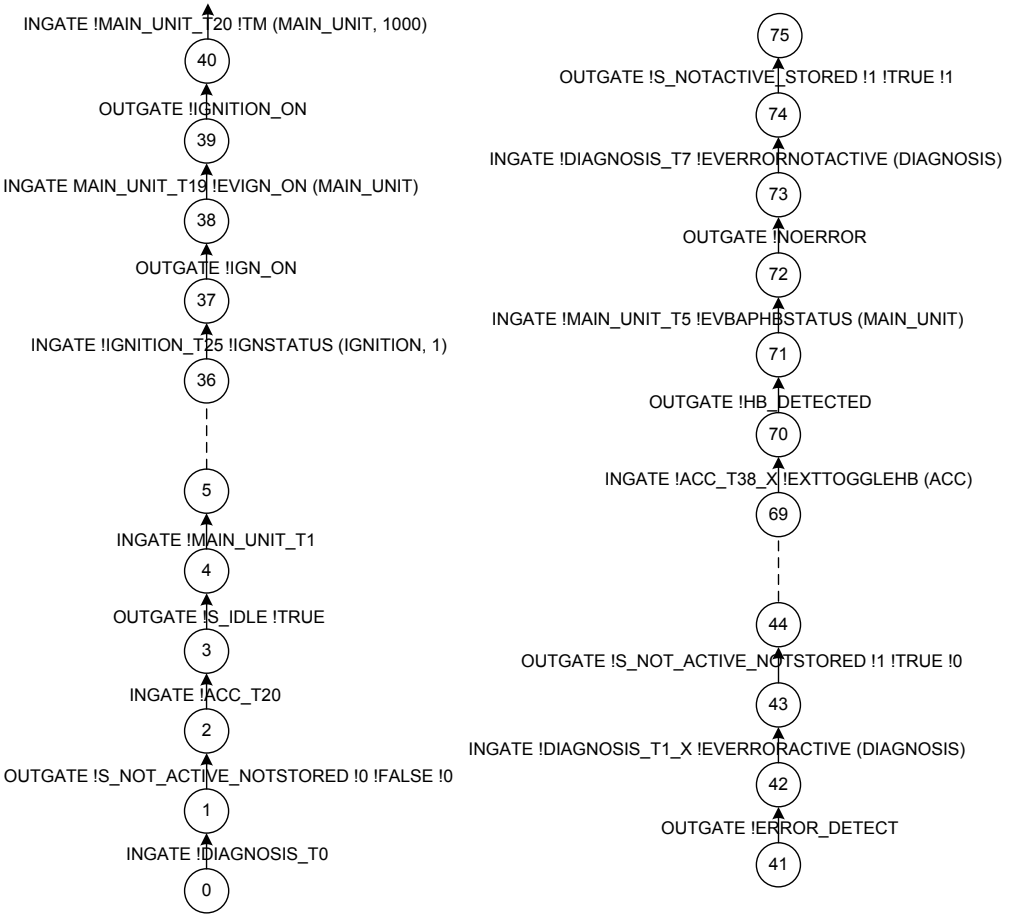

Figure 7.7: Test case covering transition T7

7.3 Experimental Results

We evaluated the proposed approach using three real-world examples (Flasher, Diagnosis and KeylessEntry) originating from the automotive domain and four more from literature (presented in Chapter 3.3). We used the identified refuse transitions in order to improve the previously presented test case generation technique (Chapter 6). The present approach complements the test purposes with the refuse transitions identified by using the dependence relations.

In Table 7.1 we present the results obtained when using the current test purpose generation technique aimed at transition coverage.

The first column of the table contains the name of the model for which the test purposes were generated. Column *Approach* contains the test case generation approach where *Deps.* stands for the current approach and *Apr1* for the previous one (the one in Chapter 6). The next column TPs contains the total number of generated test purposes. Column $ValidTPs$ presents the number of valid test

purposes. By valid test purpose we mean test purposes not targeting transitions copies that are not reachable on the flattened model.

In column TCs we give the number of generated test cases. Also this time we imposed the limit of 25 minutes per test purpose. If no test case was generated within this time, the generation process is stopped. Column $DepsCompTm$ contains the time needed for computing the dependence relations while column $GenTime$ contains the time TGV was allowed to run for the generation process.

The last two columns ($TCov_{flat}$ and $TCov$) contain the transition coverage on the flattened and non flattened model respectively.

For most of the models, the current approach delivered better results than the previous one. This was to be expected because one eliminated transition might translate to a (more or less) large part of the behavior (at IOLTS level - the enumerated behavior of the specification) that is not considered during the test case generation process.

In some cases (*Diagnosis*, *MicrowaveOven* and *ConferenceProtocol*) the dependences helped in finding transitions that the old approach was not able to find.

Even equipped with refused transitions the current approach failed in finding test cases to cover three transitions in the *Diagnosis* model. However the previous approach failed in finding four such test cases. Also in this case the current approach outperforms the old one.

In our experiments we observed a variation in the generation time of several test cases. Even though the test purposes of these test cases had refuse transitions the differences between the generation times can be explained through the size variation of the behavior reachable through the excluded transitions.

Table 7.1: Coverage aimed test case generation results

Model	*Approach*	TPs	$ValidTPs$	TCs	$DepsCompTm$	*GenTime*	$TCov_{flat}$	$TCov$
Flasher	Deps.	72	70	70	8s	14m30s	97%	100%
	Apr1	72	70	70	-	58m10s	97%	100%
Diagnosis	Deps.	44	42	39	2s	1h20m	95%	97%
	Apr1	44	42	38	-	1h43m50s	95%	97%
KeylessEntry	Deps.	43	39	39	2s	2m42s	91%	100%
	Apr1	43	39	39	-	2m38s	91%	100%
MicrowaveOven	Deps.	37	37	37	1s	2m10s	100%	100%
	Apr1	37	37	36	-	27m20s	97%	97%
LoanApprovalWS	Deps.	22	22	22	1s	1m9s	100%	100%
	Apr1	22	22	22	-	1m20s	100%	100%
ConferenceProtocol	Deps.	41	41	41	3s	6m7s	100%	100%
	Apr1	41	41	39	-	1h42m27s	95%	95%
TelCtrlProtocol	Deps.	65	65	65	2s	3m52s	100%	100%
	Apr1	65	65	65	-	4m6s	100%	100%

Chapter 8

Related work

MBT has received a lot of attention form the academic community. There are quite a few approaches dealing with test case generation from different kinds of models. Since an exhaustive survey about the general topic of MBT is out of the scope of this work we shall focus more on the topics related to the main parts of this thesis. Thus, in Section 8.1 we give some references regarding the topic of MBT applied to distributed reactive systems and in Section 8.2 we present work related to the transformation of statecharts into LOTOS specifications. In Sections 8.3 and 8.4 we discuss work related to the generation of test purposes and the usage of dependence relations for improving test case generation respectively.

8.1 Model Based Testing

Lutess is a tool used to generate tests for synchronous reactive systems [dBORZ99]. The tool makes use of a random generator (built on the basis of a Lustre [CPHP87] specification), a unit under test and a test oracle. Lustre is a formal declarative language for specifying synchronous reactive systems. The specification is a model of the valid environment behaviors defining environment of interest for the testing process. The unit under test and test oracle are synchronous reactive programs using only boolean inputs and outputs.

Another model based test generation approach [FHNS02] is based on finite state machine specifications, a set of coverage criteria and testing constraints. The generated test suites are aimed to cover the provided criteria while the test constraints are used in order to alleviate the state space explosion problem by specifying what should be avoided during the test case generation. The approach uses the GOTCHA [EFP02] test generation tool.

TORX [TB03] is an on-the-fly random test case generation tool. It is based

on the IOCO test theory (as well as TGV) and uses as input model transition system-based specifications. The models can be given in the formal languages LOTOS and Promela. The tool uses enumerative techniques (similar as TGV). TORX integrates together the test generation and execution. The inputs of the test cases are generated and executed on the SUT while its outputs are checked immediately.

The AGEDIS [HN04] project proposes a methodology for automated model driven test generation and execution for distributed systems. It is an integrated environment offering support for modeling, test generation and test execution. The tool needs as input a model of the SUT in the form of a model specified with the AGEDIS UML profile (using the AGEDIS defined modeling semantics) and annotated with directives given in the IF formal language [BFG03]. Other inputs are so called test execution directives describing the testing architecture of the SUT and also the test strategies to be used in the generation process. The AGEDIS test generator engine is based on a combination of concepts from TGV and GOTCHA test generators and also uses enumerative techniques. Several case studies [CSH03] have been carried out in the project covering different types of systems: Java programming interface to a messaging protocol (IBM UK), a web-based e-tendering application (Intrasoft International), and a piece of middleware in a message distribution system (France Telecom). "The overall conclusions from the case studies were mixed. There was a clear recommendation to pursue model-based testing further, citing benefits obtained simply by the act of modeling. The creation of a model by testers served to highlight inaccuracies in the specifications and in several cases exposed bugs at a very early stage in the development process. There was also much praise for the integrated nature of the tools and their interfaces. The abstract test suite and test execution trace format were instrumental in the integration and interoperability of a wide variety of tools all focused on the testing of distributed systems. The test execution framework was also seen as providing important automation services in an easily accessible manner. On the other hand, the industrial testers were critical of the modeling language and the test generator. The use of statecharts as the main behavioral description of the SUT was seen as useful in some contexts but not natural in others. The choice of IF as the action language was also criticized, since it did not provide sufficient high level programming constructs for effective high level modeling"[HN04].

A stress test methodology for finding failures related to network traffic in distributed systems based on UML models was presented [GBL06]. The used model is composed of different types of diagrams (Class, Sequence, Context, Network Deployment and Modified Interaction Overview Diagrams) augmented with timing information. A test model is built and together with different stress test parameters (objectives) is used to automatically (via an optimization algorithm) derive stress requirements. The requirements are then used in order to specify

test cases aiming to stress test the system.

Model checkers like SPIN [Hol97] have been used to verify [LMM99] software systems. In MBT model checkers are used in order to derive test cases. This is achieved by providing a proposition usually in some sort of logic (Linear Temporal Logic) that represents a negation of a test scenario of interest. Thus the model checker will return a trace (counterexample) that violates the property representing actually a test case to verify the test scenario.

Other MBT approaches use symbolic approaches for handling the use of data in the models. As opposed to enumerative techniques that enumerate the possible values of such data variables, symbolic techniques use constraint solvers in order to provide a valuation of the variables during the test case generation process. STG [CJRZ02] is a symbolic test generation tool using concepts from TGV (e.g. test purposes). Another symbolic test case generation technique is STSIMULATOR [FTW05] based on the TORX test case generator and provides on-the-fly random test generation. It implements a symbolic variant of the IOCO theory.

Another approach [Sch12] uses constraint solvers for the generation of test cases. The system in this case is specified communicating UML statecharts. The model is further used in order to derive its representation in the form of communicating Extended Symbolic Transition Systems (ESTS) (introduced in the same work). The test cases in this case are computed by searching the ESTS model and using the constraint solver to find valid traces in the model. Concepts similar to test purposes are also used in this approach. Here the test purposes specify also included and excluded item but also further state space limitations like the number of times a loop is allowed to be unrolled. These further specification are intended to also alleviate the state space explosion problem.

Other works [VCG+08] propose a symbolic model based test generation technique also used in Object-oriented reactive systems. It presents the SpecExplorer tool which is used by Microsoft product groups for testing operating system components and .NET framework components. The tool explores the state space of a model program and generates test cases from the model automaton that it built during the exploration process. For controlling the generation process, SpecExplorer offers restrictions mechanisms used to impose limitations on the parameters used to call an action, preconditions and state filters. These restrictions are used in order to limit the searched state space. The tool offers on-line and off-line test case generation capabilities. It also makes use of several search algorithms like Chinese postman, shortest path, random walks etc..

Also symbolic techniques suffer form the size of the state space. For example extensions to test purposes are needed to limit the number of symbolic traces in the used models [Sch12]. Other limitations of symbolic techniques are related to the limitations of the used constraint solvers [JWAW10]. Some of these relate to the data types that are allowed to be employed within the constraint solver

as well as the fact that the extracted constraint system might not always be solvable (there exists no valuation of the variables in order for the constraints to be satisfied).

UPPAAL [HLM+08] is another tool for MBT. It uses as input a model of the SUT given by means of compositions of concurrent timed automata. It implements a timed variant of the IOCO conformance relation as the basis of the used testing theory. The tool used in the approach is called UPPAAL TRON [LMN05] and is used for on line black-box testing of real-time embedded systems from non-deterministic timed automata specifications. It uses a randomize algorithm and symbolic techniques in order to represent sets of clock valuations during the generation process. Since the test cases are generated on-line the state space explosion problem is alleviated and by dynamical exploration of the states of the system, the approach can also deal with nondeterministic behaviors. However the length of the obtained test cases tends to increase due to the random nature of the used algorithm.

The attention MBT has received form the academic community and the advantages it has proved have led to the steady but sure adoption of different such techniques in industrial MBT tools:

- CONFORMIQ [CS13] uses models of the SUT constructed by means of the QML language (UML state machines annotated with time properties). Test cases are traces starting from an initial node to a final node in the used model.

- Reactis [RS13] provides test case generation capabilities for for Simulink/Stateflow models [Mat13]. The tool can be used for testing conformance between a model and its implementation.

- CERTIFYIT [Sma13] is a tool that uses a functional model of the SUT specified by means of UML class, object and statechart diagrams enriched with OCL for the automatic generation of test cases. Test are generated based on requirement coverage and other user defined strategies.

- TVEC [Tec13] is a tool suite for model-based functional test case generation. The used models are given in a proprietary language called T-VEC Linear Form. The generated test cases also called test vectors and include traceability information to their associated requirements thus offering requirement coverage.

- TESTWEAVER [Qtr13] is a tool aimed at automatic validation of systems. It offers only on-line test generation capabilities. It allows the usage of reactive and continuous models specified in different languages e.g. Simlulik [Mat13], Modelica [mod13].

Several surveys considering different MBT approaches and further aspects of the topic [AKEV08, DNSVT07, ST08, PERH04] are also available.

8.2 UML Statecharts Formal Semantics via LOTOS

The transformation of UML statecharts into LOTOS was tackled in older work [HH01]. This approach has several limitations making it unfesable for our needs. First it does not allow the use of data variables in the statechart using only basic LOTOS for the transformation. Thus, there are no actions on transitions, entry, exit and internal actions of states or completion transitions. Regarding the use of pseudostates, only initial and final nodes are allowed. Communication, timing and conflicting transitions are also not supported.

Another approach [dS01] of deriving a LOTOS representation from UML constructs was proposed. Starting from the UML metamodel, the authors define mapping rules for some of the structural and behavioral diagrams. Concerning the behavioral aspects the focus is on the transformation of activity diagrams to LOTOS. Even though they allow the use of several pseudostates, the operational semantics of activity diagrams differ from the ones used by statecharts. There are no concepts of history, asynchronous communication, timing, run to completion step.

More recently transformation rules for deriving a LOTOS specification from an UML statechart [MS08] have also been proposed. The work considers first mapping rules of flat state machines to LOTOS and then tackles compositional (to be understood as hierarchical relations) semantics regarding the passing of control from subprocesses to their containing process. Regarding the second part of the paper, the authors base their approach on the introduction of a new operator allowing control passing from subprocess to parent process. By using full LOTOS, the work allows the use of variables in the statechart and on transition triggering events. This approach does not consider the treatment of history pseudostates, asynchronously communicating models and timing. The treatment of the run to completion step and conflicting transitions are also not mentioned.

Other approaches [BB11] consider transforming finite state machines to LOTOS. The disadvantages of finite state machines compared to statecharts have already been discussed in Section 2.2. We only mention some of them : no data values are accepted, no pseudostates, different operational semantics, timing etc..

Regarding the addition of constructs representing timed behavior in LOTOS specifications, there are several approaches proposing such an extension. Some of them [Led92, RBC93, LL93, LGC96] are based on the introduction of new operators or restrict [Kho01, BD97] the language to only its basic form (the use of data values is not allowed). We use the standardized full version of the language (including the data part). Thus, the insertion of new operators and lack

of the appropriate tool support was an impediment in using the afore mentioned approaches.

Our treatment of timing is closer to approaches [Has01] where a separate process is used to handle timing aspects. However our approach does not enumerate time but uses a concept inspired from that of time intervals (the different timeouts available in the system) and urgency. However in our case actions in LOTOS (execution of timeout transitions) occur at time t_{max} form the time interval $[t_{min}, t_{max}]$ (as opposed to any time t such that $t_{min} <= t <= t_{max}$) where t_{min} is the moment the source state of the transition was entered. t_{min} and t_{max} are not explicitly modeled however they can be computed from the resulting test cases.

E-LOTOS [LL97, Ver99], the newer and improved variant of LOTOS even though it supports the use of time does not yet enjoy the required tool support. LOTOS NT [CCG+05], an earlier variant of E-LOTOS is accepted in the CADP toolset but no handling of time is yet offered. Time handling was planned but not yet implemented.

8.3 Test Purpose Generation

TGV has been used in various case studies for automatic test generation. For example TGV was used for test case generation and was provided with manually-designed test purposes and randomly-generated test purposes [dBRS+00]. The use of TGV on the cache coherency protocol [KVZ98] was presented. These works don't use a structured way of automatic test purpose generation aimed at providing coverage metrics on the used specifications. The design of the test purposes is also not very well detailed.

A strategy for generating test purposes aimed at decision/condition coverage of a LOTOS specification of a SIP Registrar [APWW07] was also presented. This is achieved through the insertion of probes in the specification and using them as goals in the test purposes. In our case the initial representation of the system is as UML communicating statecharts. So concerning structural aspects of the specification, of interest for us are coverage metrics on the statechart model like state and transition coverage.

Another work [DT02] proposes a formalization of the notion of test purpose as Message Sequence Charts (MSC). Compared to our approach, this one requires extra effort for modeling the test purposes using another type of diagrams (MSC) than the ones used for describing the desired behavior of the system.

8.4 Improving Test Purposes

In our work, the way of using the dependences differs from other approaches [ACH+09, LG08, JW02] where the models are sliced according to a slicing criteria. These approaches have proven their worth in situations involving techniques like model checking and debugging. We use the relations in order to identify refuse transitions and do not modify the structure of the model. We also do not generate a separate specification for every criterium of interest.

Regarding the control, data and communication dependences, we discussed the related work at the respective sections where we introduced each of the used definitions. Moreover, there is a very good survey regarding dependency relations and slicing of state based systems available [And12]. Hence, we focus more on the use of dependences and slicing in the context of test case generation in this section.

Slicing has been used for the purpose of test case generation from UML activity diagrams [SM09]. There, the generated test cases are aimed at path coverage (their diagrams contain an end node) by computing dynamic slices corresponding to each conditional predicate on the edges of the diagram. In their work no static control dependences are used and only data dependences are employed so that only the nodes that affect the truth value of the predicate on the edge at run time are kept in the slices. Some differences to our work include the used formalisms (UML activity diagrams vs. UML statecharts), the type of systems (non distributed systems vs. distributed systems using asynchronous communication) and type of slicing (dynamic slicing vs. static slicing).

Another approach [WW09] also tries to generate test purposes and extended them with refuse states. In this case the refused states are computed using data flow graphs that are extracted from LOTOS specifications. The dependence relations (data flow graphs), type and behavioral semantic (synchronously communicating processes) of the systems are some of the differences to the current approach.

An approach using slicing for test case generation [BFG03] computes slices from specifications given in the formal language IF. The slices are calculated with respect to sets of signals (inputs or outputs) and also require external data in the form of test purposes or feeds. Our approach uses different formalisms and there is no need for external user provided data. We also do not generate a new specification for each criterium.

The derivation of test purposes from temporal logic properties specifications [dSM06] was also proposed. The approach uses modified model checking algorithms to extract examples and counterexamples from the state space of the specification. Test purposes are then constructed by analyzing the extracted behaviors.

Chapter 9

Conclusions

9.1 Summary

In the current work we presented an approach for automatic model-based test case generation in an industrial environment. The models used for the generation process are composed of asynchronously communicating components (UML Statecharts).

The proposed approach can increase the automation degree of the testing process and thus reduce the cost of this activity. In our specific project setting it was of utmost importance to provide a solution that can easily be integrated into the test automation set-up in place, the employed modeling paradigms (i.e. UML statecharts), and the test engineer's domain expertise. The UML enjoys a formal syntax but lacks a formal semantic. This was a problem since in order to use automatic test case generation tools, a formal semantic is also required. So we presented a semantics-preserving model transformation from UML Statecharts to the formal language LOTOS - the primary input language of the test case generator TGV. This process is fully automated and hidden from the user.

Given that the considered type of systems also use timing constructs to realize their functionalities, a method to accommodate this need was also presented. Furthermore as LOTOS has no constructs for representing time, the concept presented in the current work can also be adapted for other types of systems requiring timing to be specified in LOTOS.

Once the appropriate test case generation technique and formal model are in place, the question of how to select the appropriate set of test cases arises. There are several approaches tackling this issue. The use of test purposes has shown to be very promising for this task. Test purposes represent abstractions of scenarios that need to be tested. These are usually specified by using some sort of formal representation. For the test case generation we presented two different ways of

generating test purposes. One of them is aimed at providing structural coverage of the model and is fully automatic. The second approach to test purpose generation is partially automated requiring the intervention of the user for specifying desired test scenarios by annotating states and/or transitions of the UML model.

The exponential growth of the number of possible states representing the behavior of a system is a well known problem not only in the testing domain. This is also known as the state space explosion problem and is one of the biggest factors influencing the test case generation process. The generation time and the probability of finding the required test cases are directly influenced by the size of the possible behaviors of the system. The issue of state space explosion has yet to be fully resolved. So depending on the situation at hand different techniques need to be considered in order to alleviate this problem.

In model checking, dependency relations have been used in slicing of specifications in order to obtain reduced models pertinent to a criterium of interest. In specifications described using state based formalisms slicing involves the removal of transitions and merging of states thus obtaining a structural modified specification. Using such a specification for model based test case generation activities where sequences of transitions represent test cases might provide traces that are not valid on a correctly behaving implementation. In order to avoid such trouble, we also propose the use of control, data and communication dependence for identifying parts of the model that can be excluded so that the remaining specification can be safely employed for test case generation. This information is used in order to enhance test purposes with refuse states. These states are used by TGV to limit the searched state space during the generation process.

The proposed technique is an attempt at closing the gap between the academic tools and their usage in an industrial context. We implemented the presented approach in a prototype tool.

9.2 Limitations

The proposed technique can increase the automation degree of the testing process. However in order to successfully use the approach also its limitations need to be considered.

State space explosion. As already mentioned probably the biggest problem for applying automatic test case generation techniques is the exponential growth of the state space describing the behavior of the model. Some of the factors influencing the size of the state space are data variables (all possible values generate new states) and parallelism (different behaviors combinations need to be considered).

Model and used abstractions. In model based testing the model describing the desired behavior of the SUT is the basis for the generation of test cases. The

models need to be validated themselves. Thus the model must be simpler (at a higher abstraction level) than the SUT "or at least easier to check, modify and maintain. Otherwise, the efforts of validating the model would equal the efforts of validating the SUT" [UPL06]. Even though the model is more abstract than the SUT it is crucial that it preserves enough details regarding the behavior that needs to be tested in order for it to be used for generating "meaningful" test cases. So the appropriate trade off between the two afore mentioned factors needs to be found. This is not always an easy task requiring knowledge about the test case generation tool limitations, modeling experience as well as domain knowledge in order to find and apply the appropriate abstractions.

Deterministic models. The proposed approach only considers deterministic models. This restriction was imposed by our research context. However it also reduces the state space since after an input sequence the same output will be provided. As opposed to nondeterministic systems where in different runs the same inputs might cause different responses (output) form the SUT thus increasing the number of possible executions.

Communication scheme. Another particularity of our research context is the use of an asynchronous communication between the components of the system. This employs the usage of a FIFO queue which is used to store events received by the system during execution. The events will be consumed by the system once it reaches a stable state. The communication is also deterministic, that is during a run to completion step the other statecharts in the system are not evolving (no changes of state happen and no new events are generated).

UML modeling elements. Due to ambiguities in the UML standard referring to execution within orthogonal states in a statechart, these and the corresponding pseudostates (*AND* connectors - *join* and *fork*) are not considered in the proposed transformation. This restriction relates to the need of deterministic systems since the UML does not specify the exact order in which transitions are fired in orthogonal states if more than one transition in the state can be fired at one time.

9.3 Future Work

Some of the directions of future research aim to alleviate the restrictions presented in Section 9.2:

- Investigating the application of the proposed approaches in case of nondeterministic systems. The transformation approach can be modified in order to derive a LOTOS specification that allows non-deterministic behavior of the systems. Thus extending the transformation technique in order to accommodate also UML orthogonal states and the *join* and *fork* pseudostates is also desired. However the state space of the models will also increase so

in order to apply test case generation techniques, further abstractions might be needed during the modeling process.

- Further test case generation techniques are also of interest. For example applying a randomized test case generation and comparing the results with those obtained with the current technique.

- Regarding the use of dependences for adding refuse states to test purposes, another direction of interest for future work is the investigation of the happens-before relation for communication dependence. As also mentioned in [ACH+09] the communication dependence is not transitive thus reducing the precision of the obtained slices. In our case this means a possible increase of the number of refuse transitions identified and included in the test purpose. The happens-before relation [Lam78] helps by ensuring that dependences exist only between transitions where the source transition can happen before the target transition. In our case this means a possible increase of the number of identified refuse transitions.

Bibliography

[ACH+09] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt. Control dependence for extended finite state machines. In *12th Int. Conf. on Fundamental Approaches to Software Engineering, FASE '09*, pages 216–230. Springer-Verlag, 2009. [cited at p. 98, 102, 104, 106, 119, 124]

[AKEV08] B. Aichernig, W. Krenn, H. Eriksson, and J. Vinter. D 1.2 - state of the art survey - part a: Model-based test case generation. Technical report, 2008. [cited at p. 117]

[Alb76] D.S. Alberts. The economics of software quality assurance. In *National Computer Conference and Exposition*, pages 433–442. AFIPS Press, 1976. [cited at p. 3]

[And12] K. Androutsopoulos. Dependence for slicing state-based models: A survey. Research note, UCL Department of Computer Science, 2012. [cited at p. 119]

[AO08] P. Ammann and J. Offutt. *Introduction to software testing.* Cambridge University Press, 2008. [cited at p. 11, 12]

[APWW07] B. K. Aichernig, B. Peischl, M. Weiglhofer, and F. Wotawa. Protocol conformance testing a SIP registrar: An industrial application of formal methods. In *5th IEEE Int. Conf. on Software Engineering and Formal Methods, SEFM'07*, pages 215–224. IEEE, 2007. [cited at p. 4, 118]

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987. [cited at p. 33, 34, 35, 36, 39]

[BB11] R. Babaee and S. Babamir. From UML state machines to verifiable LOTOS specifications. In *Int. Conf. on Innovative Computing*

Technology, INTECH'11, volume 241, pages 121–129. Springer Berlin Heidelberg, 2011. [cited at p. 117]

[BD97] H. Bowman and J. Derrick. Extending LOTOS with time: True concurrency perspective. In *AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software, ARTS'97*, pages 382–399. Springer-Verlag, 1997. [cited at p. 79, 117]

[BDS95] J. Bryans, J. Davies, and S. Schneider. Towards a denotational semantics for ET-LOTOS. In *6th Int. Conf. on Concurrency Theory, CONCUR'95, LNCS 962*, pages 269–283. Springer, 1995. [cited at p. 79]

[BFd+99] A. F. E. Belinfante, J. Feenstra, R. G. de Vries, G. J. Tretmans, N. Goga, L. M. G. Feijs, S. Mauw, and A. W. Heerink. Formal test automation: A simple experiment. In *12th IFIP TC6 Int. Workshop on Testing Communicating Systems: Method and Applications*, pages 179–196. Kluwer Academic Publishers, 1999. [cited at p. 46]

[BFG03] M. Bozga, J.C. Fernandez, and L. Ghirvu. Using static analysis to improve automatic test generation. *International Journal on Software Tools for Technology Transfer*, 4:142–152, 2003. [cited at p. 98, 114, 119]

[BIKT01] G. W. Bond, F. Ivancic, N. Klarlund, and R. Trefler. Eclipse feature logic analysis. In *2nd IP-Telephony Workshop, IPTEL'01*, pages 100–107, 2001. [cited at p. 50, 51]

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(3):109–137, 1984. [cited at p. 33]

[BLA+02] G. Behrmann, K.G. Larsen, H.R. Andersen, H. Hulgaard, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. *Formal Methods in System Design*, 21:225–244, 2002. [cited at p. 51]

[BLLP04] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Software Practice and Experience*, 34(10):915–948, 2004. [cited at p. 16]

[Cal05] J. R. Calamé. Abstract specification-based test generation with TGV, 2005. [cited at p. 87, 89]

[CCG+05] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LOTOS NT to

LOTOS translator (version 5.8 of march 12, 2013). Technical report, 2005. [cited at p. 79, 118]

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'02*, Lecture Notes in Computer Science, pages 470–475. Springer-Verlag, 2002. [cited at p. 115]

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL'87*, pages 178–188. ACM, 1987. [cited at p. 113]

[CS13] Limited Conformiq Software. Conformiq Test Generator, http://www.conformiq.com/, Last Visited 2013. [cited at p. 116]

[CSH03] I. Craggs, M. Sardis, and T. Heuillard. AGEDIS case studies: Model-based testing in industry. In *1st European Conference on Model Driven Software Engineering*, pages 106–117, 2003. [cited at p. 114]

[CSP09a] V. Chimisliu, C. Schwarzl, and B. Peischl. From UML statecharts to LOTOS: A semantics preserving model transformation. In *9th Int. Conf. on Quality Software, QSIC'09*, pages 173–178. IEEE Computer Society, 2009. [cited at p. 7, 45, 49]

[CSP09b] V. Chimisliu, C. Schwarzl, and B. Peischl. Test case generation for embedded automotive systems: A semantics preserving model transformation. In *2nd Workshop on Model-based Testing in Practice, MoTiP'09*, pages 43–52, 2009. [cited at p. 7]

[CW11] V. Chimisliu and F. Wotawa. Abstracting timing information in UML statecharts via temporal ordering and LOTOS. In *6th Int. Workshop on Automation of Software Test, AST'11*, pages 8–14. ACM, 2011. [cited at p. 7, 49, 79]

[CW12] V. Chimisliu and F. Wotawa. Model based test case generation for distributed embedded systems. In *Int. Conf. on Industrial Technology, ICIT'12*, pages 656 – 661. IEEE, 2012. [cited at p. 7, 8, 87]

[CW13] V. Chimisliu and F. Wotawa. Using dependency relations to improve test case generation from uml statecharts. In *5th IEEE Int. Workshop on Software Test Automation (to appear)*. IEEE Computer Society, 2013. [cited at p. 8, 97]

[dBORZ99] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous

software. In *21st Int. Conf. on Software Engineering, ICSE'99*, pages 267–276, 1999. [cited at p. 113]

[dBRS+00] L. du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R.G. de Vries. Formal test automation: The conference protocol with TGV/TORX. In *IFIP TC6/WG6.1 13th Int. Conf. on Testing Communicating Systems: Tools and Techniques, TESTCOM'00*, pages 221–228. Kluwer Academic Publishers, 2000. [cited at p. 90, 118]

[dMRV92] J. de Meer, R. Roth, and S. Vuong. Introduction to algebraic specifications based on the language ACT ONE. *Computer Networks and ISDN*, 23:363–392, 1992. [cited at p. 33]

[DMY02] A. David, M.O. Möller, and W. Yi. Formal verification of UML statecharts with real-time extensions. In *Fundamental Approaches to Software Engineering, LNCS 2306*, pages 218–232. Springer Berlin Heidelberg, 2002. [cited at p. 50, 51]

[DNSVT07] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *1st ACM Int. Workshop on Empirical Assessment of Software Engineering Languages and Technologies, WEASELTech '07*, pages 31–36. ACM, 2007. [cited at p. 117]

[dS01] P.P. da Silva. A proposal for a LOTOS-based semantics for UML. Technical Report UMCS-01-06-1, Department of Computer Science, University of Manchester, 2001. [cited at p. 117]

[dSM06] D.A. da Silva and P.D.L. Machado. Towards test purpose generation from CTL properties for reactive systems. *Electronic Notes in Theoretical Computer Science*, 164(4):29 – 40, 2006. [cited at p. 119]

[DT02] P.H. Deussen and S. Tobies. Formal test purposes and the validity of test cases. In *22nd IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'02*, pages 114–129. Springer Berlin Heidelberg, 2002. [cited at p. 118]

[EFP02] A. Hartman E. Farchi and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41:89–110, 2002. [cited at p. 113]

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag, 1985. [cited at p. 39]

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer-Verlag, 1990. [cited at p. 39]

[FH92] M. Faci and M. Haj-Hussein. An introduction to LOTOS: Learning by examples 1, 1992. [cited at p. 4, 33, 34, 36, 39]

[FHNS02] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. *SIGSOFT Software Engineering Notes*, 27(4):134–143, 2002. [cited at p. 113]

[FHP02] E. Farchi, A. Hartman, and S.S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002. [cited at p. 16]

[FTW05] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In *Formal Approaches to Software Testing, LNCS 3395*, pages 1–15. Springer Berlin Heidelberg, 2005. [cited at p. 115]

[GBL06] V. Garousi, L.C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed systems based on UML models. In *28th Int. Conf. on Software Engineering, ICSE '06*, pages 391–400. ACM, 2006. [cited at p. 114]

[GMLS07] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *19th Int. Conf. on Computer Aided Verification, CAV'07*, pages 158–163. Springer-Verlag, 2007. [cited at p. 4]

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. [cited at p. 17]

[Has01] W.A. Hassan. DLOTOS: a LOTOS extension for clock synchronization in distributed systems. In *2nd Asia-Pacific Conference on Quality Software*, pages 149–153. IEEE, 2001. [cited at p. 118]

[HG96] D. Harel and E. Gery. Executable object modeling with statecharts. In *18th Int. Conf. on Software Engineering, ICSE'96.*, pages 246 –257. IEEE, 1996. [cited at p. 18]

[HH01] B. Hnatkowska and Z. Huzar. Transformation of dynamic aspects of uml models into LOTOS behaviour expressions. *International Journal of Applied Mathematics and Computer Science*, 11(2):537–556, 2001. [cited at p. 117]

[HK04] D. Harel and H. Kugler. The Rhapsody semantics of statecharts (or, on the executable core of the UML) - preliminary version. In *SoftSpez Final Report*, pages 325–354. Springer-Verlag, 2004. [cited at p. 8, 18, 22, 25, 28, 30, 31, 32]

[HLM+08] A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In *Formal Methods and Testing, LNCS 4949*, pages 77–117. Springer Berlin Heidelberg, 2008. [cited at p. 116]

[HN04] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. In *2004 SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'04*, pages 129–132. ACM, 2004. [cited at p. 114]

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. [cited at p. 33]

[Hol97] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997. [cited at p. 115]

[IBM13] IBM. Rational Rhapsody, http://www.ibm.com/software/awdtools/rhapsody/, Last Visited 2013. [cited at p. 18, 23]

[ISO89] ISO. ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989. [cited at p. 3, 33, 36, 39]

[JJ05] C. Jard and T. Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. Journal on Software Tools for Technology*, 7(4):297–315, 2005. [cited at p. 3, 49, 88]

[JW02] W. Ji and Q. Wei, D.and Zhi-Chang. Slicing hierarchical automata for model checking UML statecharts. In *4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, ICFEM '02*, pages 435–446. Springer Berlin Heidelberg, 2002. [cited at p. 98, 107, 119]

[JWAW10] E. Jöbstl, M. Weiglhofer, B.K. Aichernig, and F. Wotawa. When BDDs fail: Conformance testing with symbolic execution and SMT solving. In *3rd Int. Conf. on Software Testing, Verification and Validation, ICST'10*, pages 479–488. IEEE Computer Society, 2010. [cited at p. 115]

[KDB11] HJ. Kim, V. Debroy, and DH. Bae. Identifying properties of UML state machine diagrams that affect data and control dependence. In *2011 ACM Symposium on Applied Computing, SAC '11*, pages 1464–1469. ACM, 2011. [cited at p. 47]

[Kho01] A. Khoumsi. Synthesizing distributed real-time systems modeled by a timed version of a subset of LOTOS. In *14th Int. Symposium on Systems synthesis, ISSS'01*, pages 268–273. ACM, 2001. [cited at p. 79, 117]

[KVZ98] H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In *Int. Workshop on Testing of Communicating Systems, IWTCS'98*, pages 211–226, 1998. [cited at p. 118]

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. [cited at p. 124]

[Led92] G. Leduc. An upward compatible timed extension to LOTOS. In *IFIP TC6/WG6.1 4th Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols: Formal Description Techniques, IV, FORTE '91*, pages 217–232. Elsevier Science, 1992. [cited at p. 79, 117]

[LG08] S. Labbé and J. P. Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Form. Asp. Comput.*, 20(6):563–595, 2008. [cited at p. 98, 103, 105, 106, 107, 119]

[LGC96] A. Lakas, G. S. Gordon, and A. Chetwynd. Specification and verification of real-time properties using LOTOS and SQTL. In *Proceedings of the 8th International Workshop on Software Specification and Design, IWSSD '96*, pages 75–84. IEEE Computer Society, 1996. [cited at p. 79, 117]

[LL93] G. Leduc and L. Léonard. A timed LOTOS supporting a dense time domain and including new timed operators. In *5th IFIP WG6.1 Int. Conf. on Formal Description Techniques, FORTE'92*, pages 87–102. North-Holland, 1993. [cited at p. 79, 117]

[LL97] L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29(3):271 – 292, 1997. [cited at p. 118]

[LMM99] D. Latella, I. Majzik, and M. 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. [cited at p. 115]

[LMN05] K.G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Software*

Testing, LNCS 3395, pages 79–94. Springer Berlin Heidelberg, 2005. [cited at p. 116]

[Mañ88a] J.A. Mañas. A tutorial on ADT semantics for LOTOS users - part I: Fundamental concepts, 1988. [cited at p. 39]

[Mañ88b] J.A. Mañas. A tutorial on ADT semantics for LOTOS users - part II: Operations on types, 1988. [cited at p. 39]

[Mat13] Mathworks. Simulink, http://www.mathworks.de/products/simulink/, Last Visited 2013. [cited at p. 116]

[Mil82] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1982. [cited at p. 33]

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989. [cited at p. 33]

[mod13] Modelica, https://www.modelica.org/, Last Visited 2013. [cited at p. 116]

[MS08] R. Mrowka and T. Szmuc. UML statecharts compositional semantics in LOTOS. In *Proc. of the 2008 Int. Symposium on Parallel and Distributed Computing, ISPDC '08*, pages 459–463. IEEE Computer Society, 2008. [cited at p. 117]

[Mye04] G.J. Myers. *The Art of Software Testing, Second Edition.* John Wiley & Sons, 2004. [cited at p. 3, 87]

[OMG13] OMG. Uml superstructure reference, http://www.omg.org/spec/uml/2.0, Last Visited 2013. [cited at p. 3, 18, 20, 25, 27]

[PERH04] W. Prenninger, M. El-Ramly, and M. Horstmann. Case studies. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, pages 439–461. Springer, 2004. [cited at p. 16, 117]

[PP05] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. *Electron. Notes Theor. Comput. Sci.*, 116:59–71, 2005. [cited at p. 41, 42]

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *27th Int. Conf. on Software engineering, ICSE '05*, pages 392–401. ACM, 2005. [cited at p. 16]

[Qtr13] Qtronic. Testweaver ,http://www.qtronic.de, Last Visited 2013. [cited at p. 116]

[RAB+07] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems*, 29(5):77–93, 2007. [cited at p. 101, 103, 105]

[RBC93] N. Rico, G. Bochmann, and O. Cherkaoui. Model-checking for real-time systems specified in LOTOS. In *Computer Aided Verification, LNCS 663*, pages 288–301. Springer Berlin Heidelberg, 1993. [cited at p. 79, 117]

[RKRT01] E. Roubtsova, J. van Katwijk, R. C. M. de Rooij, and H. Toetenel. Transformation of UML specification to XTG. In *4th Int. Conf. on Perspectives of System Informatics, PSI '02, LNCS 2244*, pages 247–254. Springer Berlin Heidelberg, 2001. [cited at p. 50]

[RS13] Incorporated Reactive Systems. Reactis, http://www.reactive-systems.com/products.msp, Last Visited 2013. [cited at p. 116]

[SBBW09] L. Su, H. Bowman, P. Barnard, and B. Wyble. Process algebraic modelling of attentional capture and human electrophysiology in interactive systems. *Formsl Aspects of Computting*, 21(6):513–539, 2009. [cited at p. 4]

[Sch12] Christian Schwarzl. *Symbolic Model-based Test Case Generation for Distributed Systems*. PhD thesis, Institute for Software Technology - Graz University of Technology, Graz, December 2012. [cited at p. 45, 115]

[SIG13] Bluetooth SIG. Telephony control protocol specification, http://www.m2mgsm.com/download/bt/docs/descr2/tcsbinary.pdf, Last Visited 2013. [cited at p. 46]

[SM09] P. Samuel and R. Mall. Slicing-based test case generation from uml activity diagrams. *SIGSOFT Software Engineering Notes*, 34(6):1–14, 2009. [cited at p. 119]

[Sma13] Smartesting. Smartesting certifyit , http://www.smartesting.com, Last Visited 2013. [cited at p. 116]

[SP10] C. Schwarzl and B. Peischl. Test sequence generation from communicating UML state charts: An industrial application of symbolic transition systems. In *International Conference on Quality Software,*

QSIC'10, pages 122–131. IEEE Computer Society, 2010. [cited at p. 51, 52]

[ST08] J. Schmaltz and J. Tretmans. On conformance testing for timed systems. In *Formal Modeling and Analysis of Timed Systems, LNCS 5215*, pages 250–264. Springer Berlin Heidelberg, 2008. [cited at p. 117]

[SZWH11] J. Sun, H. Zhao, W. Wang, and G. Hu. Atomicity maintenance in EPCreport of ALE. In *10th Int. Conf. on Computer and Information Science, ICIS'11*, pages 224–229. IEEE, 2011. [cited at p. 4]

[Tan09] K.L.L. Tan. *Case Studies Using CRESS to Develop Web and Grid Services.* Technical report. Department of Computing Science and Mathematics, University of Stirling, 2009. [cited at p. 4]

[TB03] J. Tretmans and E. Brinksma. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003. [cited at p. 113]

[Tec13] T-VEC Technologies. T-vec test vector generation system , http://www.t-vec.com/solutions/tvec.php, Last Visited 2013. [cited at p. 116]

[Tre08] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, LNCS 4949*, pages 1–38. Springer, 2008. [cited at p. 3, 87]

[UL07] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. [cited at p. 12, 13, 14, 16, 17]

[UPL06] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing, 2006. [cited at p. 11, 15, 123]

[VCG+08] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal Methods and Testing, LNCS 4949*, pages 39–76. Springer Berlin Heidelberg, 2008. [cited at p. 115]

[vdB94] M. von der Beeck. A comparison of statecharts variants. In *3rd Int. Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems, ProCoS'94*, pages 128–148. Springer-Verlag, 1994. [cited at p. 17, 28]

[Ver99] A. Verdejo. E-LOTOS: Tutorial and semantics. Master's thesis, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 1999. [cited at p. 118]

[VLH07] S. Van Langenhove and A. Hoogewijs. SVtL: system verification through logic tool support for verifying sliced hierarchical statecharts. In *18th Int. Conf. on Recent Trends in Algebraic Development Techniques, WADT'06*, pages 142–155. Springer-Verlag, 2007. [cited at p. 107]

[Was04] A. Wasowski. Flattening statecharts without explosions. In *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems, LCTES '04*, pages 257–266. ACM, 2004. [cited at p. 51]

[WW09] M. Weiglhofer and F. Wotawa. Improving coverage based test purposes. In *9th Int. Conf. on Quality Software, QSIC'09*, pages 219 –228. IEEE, 2009. [cited at p. 119]

[ZZK07] Y. Zheng, J. Zhou, and P. Krause. A model checking based test case generation framework for web services. In *4th. Int. Conf. on Information Technology, ITNG '07*, pages 715–722. IEEE Computer Society, 2007. [cited at p. 46]

Appendices

List of Abbreviations

Abbreviation	Description	Definition
ADT	Abstract Data Type	page 33
CCS	Calculus of Communicating Systems	page 33
CFG	Control Flow Graph	page 101
COMD	Communication Dependence	page 107
CSP	Communicating Sequential Processes	page 33
DD	Data Dependence	page 106
ECU	Electronic Control Unit	page 5
EFSM	Extended Finite State Machines	page 98
ESTS	Extended Symbolic Transition Systems	page 115
FIFO	First In First Out	page 30
FDT	Formal Description Techniques	page 4
FLBC	Flash Light Backup Controller	page 43
FLMC	Flash Light Master Controller	page 43
FTC	Flat Transition Coverage	page 90
IOCO	Input Output Conformance	page 3
IOLTS	Input Output Labeled Transition System	page 89
ISO	International Organization for Standardization	page 33
IUT	Implementation Under Test	page 88
LTS	Labeled Transition System	page 88
LOTOS	Language Of Temporal Ordering Specification	page 33
MBT	Model-based Testing	page 14
NTSCD	Non-termination Sensitive Control Dependence	page 103
PCO	Points of Control and Observation	page 88
RTC	Run-to-Completion	page 28
SUT	System Under Test	page 12
SCC	Strongly Connected Component	page 101
UNTICD	Unfair Non-termination Insensitive Control Dependence	page 103

List of Figures

List of Tables

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)