FAKULTÄT FÜR !NFORMATIK

Model Transformation from UML State Machines to Input/Output Symbolic Transition Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

ausgeführt von

Christopher Thurnher
Matrikelnummer 0125913

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Wien, 24.09.2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT FÜR !NFORMATIK

Model Transformation from UML State Machines to Input/Output Symbolic Transition Systems

MASTER'S THESIS

to obtain the academic degree

Master of Science

within the study

Software Engineering & Internet Computing

by

Christopher Thurnher
Matriculation Number 0125913

at the
Faculty of Informatics at the Vienna University of Technology

Supervision:
Adviser: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Vienna, 24.09.2008

(Signation Author) (Signation Adviser)

To my father

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Wien, 24. 09 2008

Christopher Thurnher

Acknowledgments

I would like to thank all persons who contributed in many different ways to my master's thesis.

Thanks to Bernhard Peischl, Martin Weiglhofer and Franz Wotawa who supervised me during this thesis. They always supported me with great ideas and fruitful discussions. Without their advice it would not have been possible to accomplish this work.

My special thanks goes to Armin Beer who offered me the opportunity to work at Siemens and always helped me with his substantiated knowledge and long time experience. Thanks to the people of the Test Support Center for the pleasant work atmosphere.

Very special thanks goes to my family. Without their support it would not have been possible to study computer science. They made it possible to write this thesis. Thanks to my girlfriend Silvia who always believed in me and supported me in an emotional way.

The research herein is 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!

Christopher Thurnher

Abstract

Testing is a very important part in software engineering. A lot of research is done in the field of automated test case generation. There are proper tools that are able to generate test cases from Input/Output Symbolic Transition Systems (IOSTS), a state/transition based model. This representation is however not used in industry for system specification purposes. UML has become the de facto standard in this area.

This thesis fills the gap between system specification with UML 2 and automated test case generation based on IOSTS. It presents a model transformation from UML 2 to Input/Output Symbolic Transition Systems. The generated IOSTS may then be used by tools that are capable of automated test case generation based on IOSTS like the Symbolic Test Generator (STG) that is also used in this thesis.

The test case generation process is based on conformance testing. Conformance testing means testing a system specification against an implementation according to a conformance relation. In the context of this thesis, the *sioco* (symbolic input output conformance) conformance relation is discussed, as well as the *ioco* (input output conformance) conformance relation, the *sioco* relation is based on. A detailed description of IOSTS is also part of this thesis as well as an explanation of the test case generation process.

UML is very general, there exist a lot of diagrams and elements for system specification purposes. The subset of UML that is used by the transformation is also described in this thesis.

The main part of this work is the transformation algorithm. There is a detailed description how elements of UML are mapped to elements of IOSTS. A pseudocode implementation of the algorithm is also part of this thesis. The transformation should not rely on the tool that uses the IOSTS, so a general IOSTS meta model is introduced. The transformation is split in two parts, a model to model transformation to generate an IOSTS model that conforms to the IOSTS meta model defined in this thesis and a model to text transformation to generate a specific textual representation an arbitrary tool can handle (like the STG tool).

The transformation is illustrated by a practical example. The Conference Protocol is used for this purpose. The example shows the whole transformation, starting with an UML specification. The result is an IOSTS system specification in the STG language. The STG tool is chosen since it is able to handle Input/Output Symbolic Transition Systems.

Last, some ideas about how the transformation may be improved in some future work are introduced.

Zusammenfassung

Testen ist sehr wichtig im Software Engineering Prozess. Automatische Testfallgenerierung ist ein aktuelles Forschungsgebiet. Es gibt sehr gute Tools, die Testfälle ausgehend von Input/Output Symbolic Transition Systems (IOSTS), ein auf State/Transition basierendes Modell, erzeugen können. Diese Repräsentation wird in der Industrie jedoch nicht zur Systemspezifikation verwendet. UML ist der de facto Standard in diesen Gebiet.

Diese Diplomarbeit schließt die Lücke zwischen der Systemspezifikation anhand von UML 2 und den Möglichkeiten der automatisierten Testfallgenerierung basierend auf IOSTS. Eine Modelltransformation von UML 2 zu IOSTS wird präsentiert. Tools, die die Fähigkeit besitzen anhand von IOSTS Spezifikationen Testfälle zu generieren, können diese dann nutzen. Der Symbolic Test Generator (STG), welcher auch im Rahmen dieser Diplomarbeit vorgestellt wird, ist eines dieser Tools.

Der automatisierte Testfallgenerierungsprozess basiert auf Conformance Testing. Dies bedeutet eine System Spezifikation anhand einer Conformance Relation mit einer Implementierung zu testen. Im Rahmen dieser Arbeit wird sowohl die *sioco* (symbolic input output conformance), als auch die *ioco* (input output conformance) Conformance Relation vorgestellt, da die *sioco* Relation auf *ioco* basiert. Des Weiteren werden sowohl IOSTS, als auch der Testfallgenerierungsprozess genau erläutert und definiert.

UML ist sehr allgemein, es existieren sehr viele Diagramme und Elemente mit denen Systeme spezifiziert werden können. Deshalb benutzt der Transformationsalgorithmus der in dieser Arbeit vorgestellt wird nur eine Teilmenge von UML. Diese Teilmenge wird im Rahmen der Diplomarbeit genau spezifiziert.

Der Hauptteil dieser Arbeit ist der Transformationsalgorithmus. Es wird sehr detailliert erklärt, wie die UML Elemente auf die IOSTS Elements abgebildet werden. Eine Pseudocode Implementierung des Algorithmus wird auch präsentiert. Da die Transformation nicht vom Tool, das die generierten IOSTS benützt, abhängen soll, wird ein allgemeines IOSTS Metamodell eingeführt. Die eigentliche Transformation ist aufgeteilt in eine Modell zu Modell Transformation, um ein IOSTS Modell zu generieren, welches dem allgemeinen Metamodell entspricht und in eine Modell zu Text Transformation, um die spezifische textuelle Repräsentation zu erstellen mit der ein beliebiges Tool umgehen kann.

Ein praktischen Beispiel, das Conference Protocol, veranschaulicht die Transformation. Dieses Beispiel zeigt den gesamten Transformationsprozess ausgehend von einer UML Spezifikation. Das Ergebnis ist eine IOSTS Spezifikation welche das STG Tool lesen kann. Dieses Tool wird verwendet da es mit IOSTS umgehen kann.

Abschließend werden einige Ideen präsentiert, welche die Transformation in zukünftigen Arbeiten verbessern können.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Motivation

Testing is a crucial part in almost any software engineering process. Especially in safety critical systems, testing should be a major step in order to improve the propriety of the systems. Nowadays, systems become more and more complex, so the effort that has to be taken in testing increases. Manual testing is not always the best procedure, a lot of research is made in the field of automatic test case generation.

One approach is test case generation from a formal system specification. There are various tools that are able to generate executable test cases from specification languages that are based on state/transition systems. Input/Output Labelled Transition Systems (IOLTS, [Tretmans, 1996]) and Input/Output Symbolic Transition Systems (IOSTS, [Rusu et al., 2000], [Frantzen et al., 2005]) are examples of these systems. The Symbolic Test Generator (STG, [IRISA, 2008a]) is a tool that is able to generate executable test cases from an IOSTS system specification.

Unfortunately, IOSTS are not really used in industry in order to formulate a system specification. One reason for this may be that there are no tools that simplify the generation of an IOSTS. This however is certainly not the only one. Nowadays, the Unified Modeling Language (UML, [OMG, 2007a], [OMG, 2007b]) has become the de facto standard for software specification purposes in industry. There exist various diagrams that help to formalize almost any aspect of a system.

The goal of this diploma thesis is to fill the gap between UML modelling and automatic test case generation from IOSTS (used by the STG tool). Since there exist tools that are able to generate proper test cases from an IOSTS specification, the transformation from an UML model to an IOSTS model in the STG language specification is a step towards generating test cases from UML. Thus, a model transformation is the main part of this thesis.

The UML specification is very complex and general. In order to perform the transformation only a subset of the UML specification is used. The selection of this subset, as well as the restrictions that have to be made is also part of this thesis. The main focus is to show that a transformation is feasible, not to incorporate as many parts of the UML specification as possible.

1.2 Task Outline

Figure 1.1 shows the overall task of generating executable test cases from UML models. First, an UML model has to be constructed with an UML modeling tool. After that, the model is passed to the IOSTS Generator. This is where the actual transformation takes place. The generator takes as argument an UML model and produces one or more IOSTS system specifications that conform to the STG language specification described in Chapter 5.4. This specification is then passed to the STG tool. The STG tool is able to generate executable test cases from an IOSTS system specification. Actually, the STG tool does not generate the executable test cases directly. It first generates symbolic test cases that are then transformed to executable test cases using a test driver. It also needs a test purpose in order to generate executable test cases (see Section 5.2), but generating the test purpose is not scope of this thesis, only the transformation from UML to an IOSTS

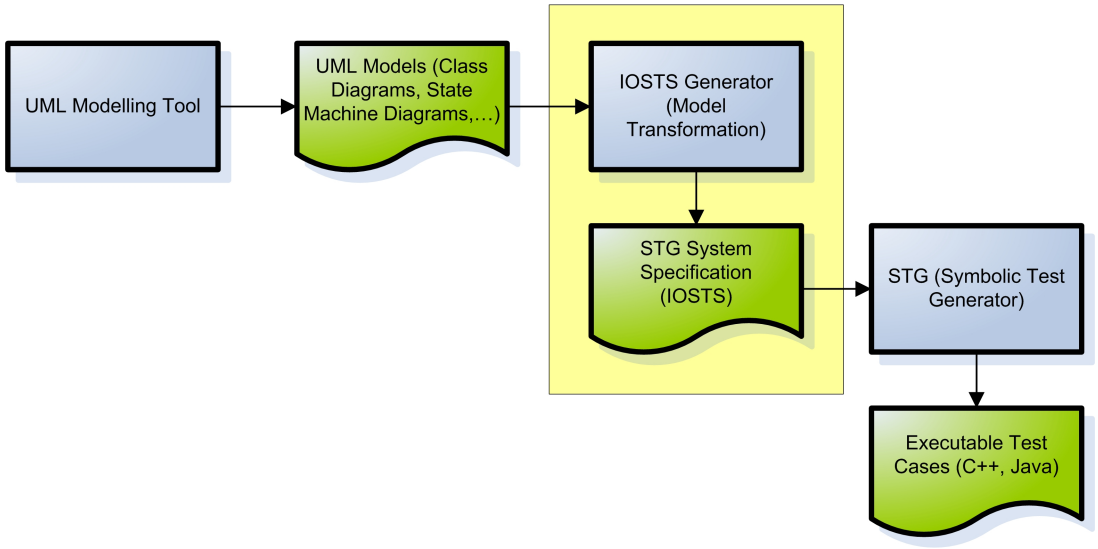


Figure 1.1: Overall Task

specification is taken into account. The yellow frame in Figure 1.1 denotes the part of the overall task that is covered in this thesis. It is supposed to fill the gap between UML modelling and test case generation from Input/Output Symbolic Transition Systems.

1.3 Thesis Outline

Chapter 2 gives an overview of some projects that comprise a transformation from UML to a state/transition based system in order to generate test cases. There are transformations to IOLTS as well as transformations to the *IF* (Intermediate Format) language [Bozga et al., 2002].

Chapter 3 comprises some theoretical background. Since an IOSTS that is used in the transformation as target representation is based on IOLTS, a definition of IOLTS as well as the introduction of the *ioco* conformance relation, an integral part in the theory of conformance testing, is given. An example is also part of that chapter.

The next chapter (Chapter 4) provides a definition of IOSTS. Conformance testing with IOSTS is also explained in this chapter. After that, the test case generation process from IOSTS is shown in detail.

The STG tool is introduced in Chapter 5. The overall workflow that is provided by this tool is explained. After that, an example for the test case generation process is given. Last, the IOSTS language specification used by the STG tool is introduced.

Chapter 6 gives an overview of UML. There is a detailed description of the parts of UML that are used by the transformation, i.e. *Class*, *State Machine* and *Activity Diagrams*. Not all elements of these diagrams are used in the transformation process, the restriction that have to be made are also part of this chapter. Finally, there is a brief introduction to the Object Constraint Language (OCL) [OMG, 2006b].

The transformation from UML to IOSTS is explained in detail in Chapter 7. First, an extension that has to be made to the UML model in terms of a stereotype is introduced.

Then, there is a description of how the elements of UML are mapped to the elements of IOSTS. This chapter also comprises a pseudo code implementation of the transformation algorithm.

Chapter 8 describes the implementation of the transformation. The algorithm is implemented in the eclipse framework, a description of the plugins used for this purpose is part of this chapter. The overall test case generation workflow from UML to IOSTS as well as the implemented transformation workflow that is part of the overall workflow is also discussed. Then, the IOSTS meta model that is used for the model transformation is introduced. At the end, some implementation details are illustrated.

An example of the transformation is given in Chapter 9. The Conference Protocol is used for this purpose. This protocol is explained first. Then, the UML specification of the protocol is illustrated. At the end, the resulting IOSTS is depicted.

Chapter 10 comprises some ideas about how the application can be improved in some future work.

Chapter 2

Related Work

Contents

2.1 Introduction

This chapter gives an overview of different projects that transform UML models into an intermediate representation in order to generate test cases. Section 2.2 describes a procedure of test case generation from UML 1.4 models with the aid of TGV and UMLAUT by transforming the UML models into an IOLTS representation. The AGEDIS framework presented in Section 2.3 is also based on UML 1.4. It uses the *IF* language as intermediate format. Section 2.4 introduces the IF toolset which is a part of the IST OMEGA project. This framework also uses the *IF* language as intermediate format taking timing concepts into account.

2.2 Test Case Generation with UMLAUT and TGV

[Pickin et al., 2002] describe a method for test case generation from UML 1.4 models with the assistance of the UMLAUT (**U**nified **M**odeling **L**anguage **A**ll p**U**rposes **T**ransformer, [Ho et al., 1999] and [IRISA, 2008b]) and TGV [Jard and Jéron, 2005] tool. The overall process is depicted in Figure 2.1. First of all, the system to be tested has to be modelled in UML. This model has to consist of at least one *Class Diagram*. Each main class has to provide a *State Machine Diagram* as well. The initial state of the system has to be described by means of an *Object Diagram*.

Another input for this process is the test objective. It describes the part of the overall system that has to be tested (like the test purpose described in Section 4.3). The test objectives for this application have to be provided in terms of an UML *Sequence Diagram* based notation, i.e. the *TeLa* language [Pickin et al., 2001] is used. The *TeLa* language enhances the UML 1.4 *Sequence Diagram* notation with some constructs needed for this application. The test objective has to comprise two parts, an accept scenario that describes the parts of the system the tester wants to test and a reject scenario that describes the parts of the system the tester wants to avoid.

The test case generation process is based on Input/Output Labelled Transition Systems (see Section 3.2). The UMLAUT tool is used to translate the UML model into an Labelled Transition System (LTS) (step 1 in Figure 2.1). In order to specify the inputs, outputs and internal actions needed for the IOLTS test case, the classes of the *Class Diagrams* are associated with external actors. The visible actions are those that are associated with an external actor. This mechanism helps to generate *.hide* and *.io* files (*.hide* files provide the internal, *.io* files the input and output actions). These files are used to transform the LTS test case into an IOLTS later on. The test objectives are also transformed to an LTS (step 2 in Figure 2.1).

After that (step 3 in Figure 2.1), the TGV tool takes care of generating an IOLTS test case from the LTS system specification, the LTS test objective and the *.hide* and *.io* files extracted from the system specification. The test case is a subgraph of the system specification enhanced with the verdict states *Pass*, *Inconclusive* and *Fail*. The meaning of these verdicts is the same as explained in Section 4.3.

Finally a test case represented in the *TeLa* language is derived from the IOLTS test case using the UMLAUT tool (step 4 in Figure 2.1). The test case also takes asynchronous communication and concurrency into account.

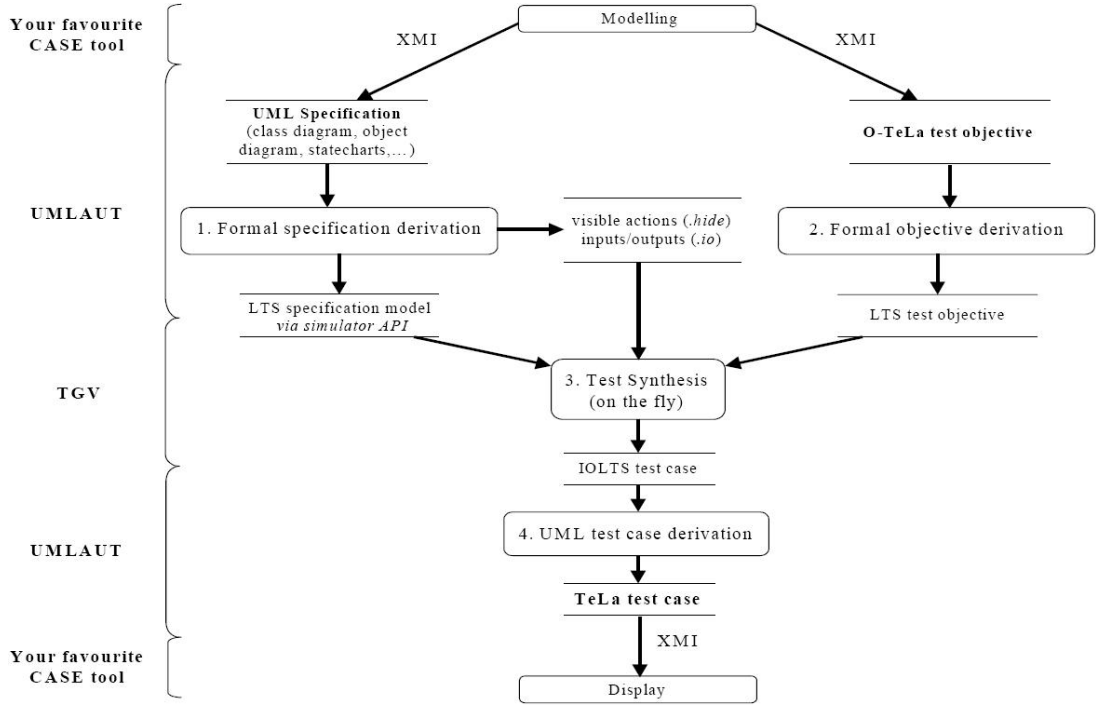


Figure 2.1: Test Case Generation Workflow with UMLAUT and TGV (source: [Pickin et al., 2002])

The method described in [Pickin et al., 2002] is very similar to the method described in this thesis. One difference is the use of Input/Output Symbolic Transition Systems instead of IOLTS (see Section 4.2). This application also uses UML 2 models instead of UML 1.4. Another difference is the use of the STG tool (see Chapter 5) in place of TGV for the test case generation purposes since it can handle IOSTS.

2.3 AGEDIS

The AGEDIS (**A**utomated **G**eneration and **E**xecution of Test Suites for **DI**stributed Component-based **S**oftware) tools [Hartman and Nagin, 2004] offer a whole framework for test case generation from UML 1.4 models. The UML model is transformed to an *IF* (intermediate format) specification [Bozga et al., 2002] from which the executable test cases can be generated.

The input for this process is a behavorial model of the system under test (SUT) and a test generation strategy. Both are UML models. The behavorial model comprises *Class*, *State Machine* and *Object Diagrams*. A detailed description of the modelling language used in the AGEDIS framework can be found in [Cavarra and Davies, 2001], actually, an UML profile that must be used is defined there.

The *Class Diagrams* are used to define the static structure of the SUT as well how the objects may interact. The operations and signals of the *Class Diagram* may be stereotyped with «controllable» and «observable» to determine whether the operation or signal may be called by the tester (like input actions in the context of IOSTS defined in Section 4.2)

or only be observed (like output actions in the context of IOSTS). Attributes may also be «observable».

The *State Machine Diagrams* are used to show how objects may evolve during their lifetime. The events that may be used here are call and signal events (see Section 6.4.2 for a detailed description of these events). The actions that describe the effect of a transition may be *call*, *send*, *assign*, *procedure* or *function* actions. Since this application uses the UML 1.4 notation, the effect of a transition, that is a behavorial feature in UML 2 (see Figure 6.6), is a simple chain of actions. *Assign*, *procedure* and *function* actions are actions and functions defined in the *IF* language.

The *Object Diagrams* show which objects are involved in the start state of the system under test.

It is also possible to add global functions and procedures defined in the *IF* action language to the model by adding them as a note in a separate, empty structure diagram.

The test generation strategy is defined either in terms of test directives or test purposes. Test directives must define start states, finish states, states that must be visited, states that must be avoided, operations that must be used and operations that must be avoided. Test purposes are defined in terms of *State Machine Diagrams*. These *State Machines* describe the chain of operations that must be performed by the test.

After that, the behavorial model and the test directive is compiled together to create a model execution interface described in [Bozga and Olvovsky, 2002]. This is the intermediate model defined in the *IF* language. Then, the test generator produces an abstract test suite from this execution interface with the aid of TGV [Jard and Jéron, 2005] and GOTCHA [Farchi et al., 2002]. The abstract test suite can then be transformed to an executable test suite using the AGEDIS execution engine.

The AGEDIS framework has been evaluated in some case studies, details can be found in [Hartman, 2004]. A difference between the AGEDIS test case generation process and the one provided in this thesis in the use of UML 1.4 instead of UML 2. The use of *IF* as action language is also very specific, nowadays OCL is more common in terms of UML. The test directive is quite similar to the test purpose in the context of STG (see Section 5.2). The *IF* language used to describe the intermediate model (model execution interface) is also very close to the STG language specification (see Section 5.4), actually STG used a subset of *IF* at the beginning and adopted it later on in order to take the difference between *IF* and IOSTS into account [IRISA, 2008a].

2.4 OMEGA

The IST OMEGA project [OMEGA, 2008] aims at building an UML-based methodology and a validation environment for real-time and embedded applications [Ober et al., 2006]. This project comprises amongst others a toolset called timed system verification, a subproject of IST OMEGA, also called IF toolset (refer to [Bozga et al., 2004] for a detailed description). The overall architecture of the IF toolset is depicted in Figure 2.2. Basically, it consists of three levels: the specification, the intermediate description and the LTS level:

- *specification level*: this level consists of components capable of translating UML

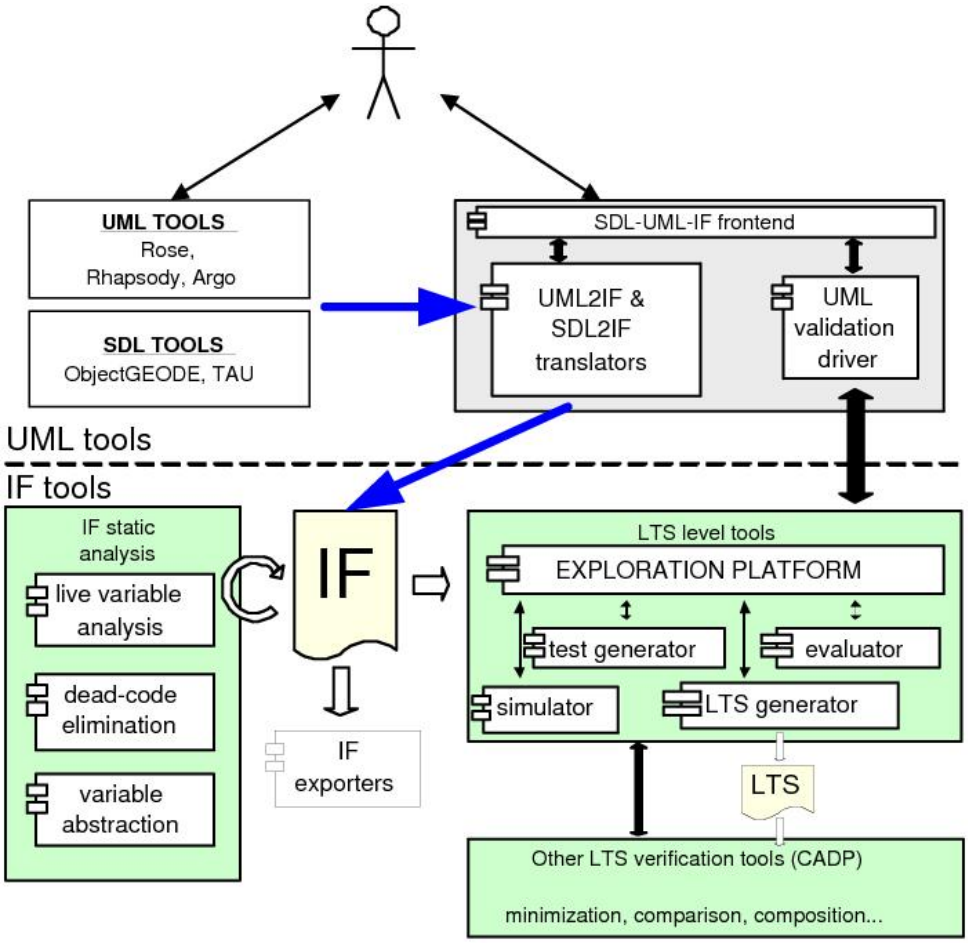


Figure 2.2: IF Toolset (source: [Bozga et al., 2004]

and SDL descriptions to the *IF* language. The UML specifications must conform to a predefined UML profile described in [Graf et al., 2006]. This profile also takes timing into account by defining concepts of time-triggered behavior and duration constraints. The main focus of the IF toolset is modeling real-time applications, thus the timing concepts defined in the UML profile play a crucial role.

- *intermediate description level*: the models of the specification level are transformed to the *IF* language [Bozga et al., 2002]. This is an intermediate format based on timed automata [Bozga et al., 2004].

- *LTS level*: a LTS describes an execution of an *IF* description. There are various tools available that can be used for evaluation or simulation of these models. The TGV tool can also be used for test case generation purposes.

This framework uses the *IF* language for defining the intermediate models that represent the UML (SDL) models. The application presented in this thesis uses IOSTS as intermediate representation. The transformation from UML to *IF* (UML2IF translator) is a basic component of the IF toolset but, unlike the transformation presented in this thesis, it takes timing properties into account.

2.5 Summary

This chapter presented three test case generation projects. All of them are based on a transformation from UML to an intermediate model representation the test cases are generated from. These intermediate models are either based on IOLTS or the IF language. This thesis presents a model transformation from UML to Input/Output Symbolic Transition Systems (IOSTS) [Rusu et al., 2000] (see chapter 4). IOSTS rely on symbolic data and the *sioco* conformance relation [Frantzen et al., 2006].

Another difference to the previously presented projects is that the transformation presented in this thesis is based on UML 2 models instead of UML 1.4. Most UML tools are already based on UML 2, so taking this version as basis for the transformation makes the whole process more acceptable.

Chapter 3

The *ioco* Test Theory

Contents

3.1 Introduction

This chapter describes the *ioco* (input output conformance) test theory. Conformance testing is based on the concept of testing an implementation against a specification according to a conformance relation ([Tretmans, 1994], [Tretmans, 1996]). To show the concept of a conformance relation, the *ioco* conformance relation is introduced in this chapter. Section 3.2 shows a definition of Input/Output Labelled Transition Systems (IOLTS). The *ioco* test theory is based on such systems. Section 3.3 describes the *ioco* test theory. Finally an example is given in Section 3.4 to show how *ioco* works. The *ioco* conformance relation is introduced because the test generation algorithm described in the next chapter is based on the the *sioco* conformance relation [Frantzen et al., 2006] which itself is based on Input/Output Symbolic Transition Systems (IOSTS) [Rusu et al., 2000]. To understand the concepts of an IOSTS, it is beneficial to be familiar with IOLTS. The same applies to *sioco*. *sioco* is based on the *ioco* conformance relation, it simply lifts the *ioco* conformance relation to the level of symbolic transition systems. So this chapter deals with the basic elements that are needed in order to become familiar with the definitions of the following chapters.

3.2 Definition of IOLTS

An Input/Output Labelled Transition System is a structure consisting of states and transitions. Each transition is labelled with either an input or an output action [Tretmans, 1996]. The definition of an IOLTS is as follows:

An IOLTS is a quintuple $\langle S, s_0, \Sigma_I, \Sigma_U, \rightarrow \rangle$ whereas

- S is a is a countable set of states
- $s_0 \in S$ is the initial state
- Σ_I is a countable set of input actions, Σ_U is a countable set of output actions. $\Sigma_I \cap \Sigma_U = \emptyset$.
- $\rightarrow \subseteq S \times \Sigma_\tau \times S$ is the transition relation. Σ_τ abbreviates the set $\Sigma \cup \{\tau\}$, whereas τ denotes an unobservable action and $\Sigma = \Sigma_I \cup \Sigma_U$.

3.3 The *ioco* Conformance Relation

Conformance testing means testing an implementation of a system against a specification according to a conformance relation [Tretmans, 1994]. Only the observable behavior of the implementation is tested, i.e. the outputs. This section describes the *ioco* conformance relation. The formulas and definitions in this section are taken from [Tretmans, 1996], [Frantzen et al., 2005] and [Frantzen et al., 2006].

In order to understand the *ioco* conformance relation, some additional notions have to be introduced:

- $s \xrightarrow{\mu}$ means $\exists s' \in S : s \xrightarrow{\mu} s'$ with $s \in S$ and $\mu \in \Sigma_\tau$. The meaning of this abbreviation is the existence of a simple transition with origin s, destination s' and action $\mu \in \Sigma_\tau$. $s \xrightarrow{\mu} s'$ abbreviates $(s, \mu, s') \in \rightarrow$.

- $s \stackrel{\sigma}{\Longrightarrow}$ means $\exists s' \in S : s \stackrel{\sigma}{\Longrightarrow} s'$ with $s \in S$ and $\sigma \in \Sigma^*$ (Σ^* denotes the set of all finite sequences of actions over Σ). This notation is almost similar to the one before. The σ also takes the unobservable action τ into account. $\sigma \in \Sigma^*$ means not a single action but an action sequence $\mu_0, \mu_1, \mu_2 ... \mu_n \in \Sigma_\tau$. So $s \stackrel{\sigma}{\Longrightarrow}$ denotes the existence of a path $s \xrightarrow{\mu_0} s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} ... \xrightarrow{\mu_n} s'$ with origin s and destination s'. $\Longrightarrow \subseteq S \times \Sigma^* \times S$ is also called generalized transition relation.

- $\mathbf{traces}(s) =_{def} \left\{ \sigma \in \Sigma^* | s \stackrel{\sigma}{\Longrightarrow} \right\}$ with $s \in S$. This abbreviation simply specifies the set of paths with origin s.

Another concept that has to be explained is quiescence. An output action $\mu \in \Sigma_U$ at a given state of an IOLTS is called observation. A state where no observation can be made is called a quiescent state. So a quiescent state s is a state that has no outgoing transition labelled with an output action: $\forall \mu \in \Sigma_U \cup \{\tau\} : s \stackrel{\mu}{\not\rightarrow}$. A quiescent state is denoted by $\delta(s)$.

δ is a special action label not part of the action label set Σ. $\Sigma_\delta = \Sigma_I \cup \Sigma_U \cup \{\delta\}$. Now consider, every quiescent state s of an LTS `L` is extended with a self loop labelled δ. Σ_δ^* is called the set of extended traces that may also use the δ transitions of quiescent states. The suspension relation is defined by $\Longrightarrow_\delta \subseteq S \times \Sigma_\delta^* \times S$.

Three more functions are now being defined ($s \in S$, $C \subseteq S$ and $\sigma \in \Sigma_\delta^*$):

- $\mathbf{Straces}(s) =_{def} \left\{ \sigma \in \Sigma_\delta^* | s \stackrel{\sigma}{\Longrightarrow}_\delta \right\}$ is the set of suspension traces. The suspension traces are almost the same as the traces defined before, they just take quiescence into account, that means they also consider the δ transitions in quiescent states.

- C **after** $\sigma =_{def} \bigcup_{s \in C} s$ **after** σ, whereas s **after** $\sigma =_{def} \left\{ s' \in S | s \stackrel{\sigma}{\Longrightarrow}_\delta s' \right\}$. s **after** σ is the set of states that can be reached by a suspension trace starting in s. C **after** σ is simply the union of all these sets for each state $s \in C \subseteq S$.

- $\mathbf{out}(C) =_{def} \bigcup_{s \in C} \mathbf{out}(s)$, whereas $\mathbf{out}(s) =_{def} \left\{ \mu \in \Sigma_U | s \xrightarrow{\mu} \right\} \cup \{\delta | \delta(s)\}$. $\mathbf{out}(s)$ denotes all possible observations that can be made in state s. If there is no output action, the observation is δ. In that case state s is a quiescent state. $\mathbf{out}(C)$ is the union of the observations that can be made in all states $s \in C \subseteq S$.

Now the conformance relation *ioco* can be defined. As mentioned above, conformance testing means testing an implementation against a specification according to a conformance relation. This relation is defined by:

$$\texttt{P} \textbf{ ioco } \texttt{L} \textit{ iff } \forall \sigma \in \mathbf{Straces}(l_0) : \mathbf{out}(p_0 \textbf{ after } \sigma) \subseteq \mathbf{out}(l_0 \textbf{ after } \sigma)$$

`P` is a model of the physical implementation of the specification `L`. `P` is an input-enabled IOLTS defined by $\texttt{P} = \langle P, p_0, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$, whereas $\texttt{L} = \langle L, l_0, \Sigma_I, \Sigma_U, \rightarrow_L \rangle$. Input-enabled

means that the IOLTS must be able to react to any given input action $\mu \in \Sigma_I$ in any state. The *ioco* conformance relation says that all observations that can be made after any given path in the implementation must be a subset of the observations that can be made in the specification.

3.4 Example

This section gives a brief example to illustrate the *ioco* conformance relation. Three IOSTS are shown in Figure 3.1. The specification is defined by $\mathtt{L} = \langle L, l_0, \Sigma_I, \Sigma_U, \rightarrow \rangle$, whereas $L = \{l_0, l_1, l_2, l_3, l_4\}$ is the set of states of L. The initial state is l_0. The set of input actions is defined by $\Sigma_I = \{coin\}$, the set of output actions is defined by $\Sigma_U = \{coffee, milk, tea\}$. This specification actually denotes a vending machine, where the user can insert a coin and is then able to choose between coffee, milk and tea. The input actions of the specification L are denoted by ? in the diagram, the output actions by ! respectively.

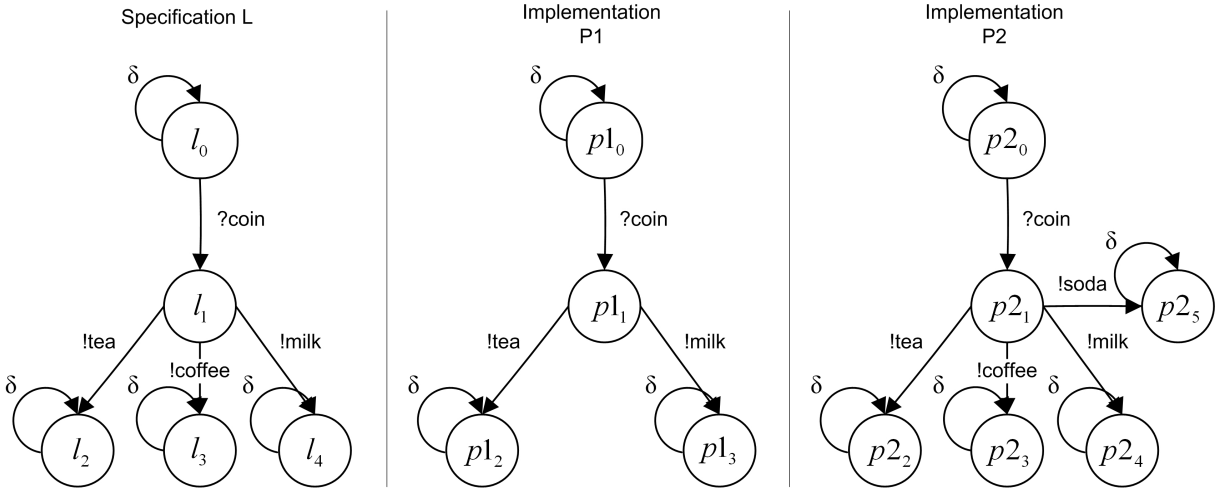


Figure 3.1: Three IOLTS, one specification L and two implementations P1 and P2

P1 describes one possible implementation of L. It is also an IOLTS defined by $\mathtt{P1} = \langle P1, p1_0, \Sigma_I, \Sigma_U, \rightarrow \rangle$, whereas $P1 = \{p1_0, p1_1, p1_2, p1_3\}$ is the set of states of $P1$. The initial state is $p1_0$. The set of input actions is defined by $\Sigma_I = \{coin\}$, the set of output actions is defined by $\Sigma_U = \{milk, tea\}$. To show that P1 *ioco* L, the observations of two different paths are considered (actually all possible paths have to be considered in order to show that P1 *ioco* L but in this small example this can be illustrated using only those two paths).

First: $\sigma = \{\} \Rightarrow$
out $(p1_0$ **after** $\sigma) = \{\delta\} \subseteq$
out $(l_0$ **after** $\sigma) = \{\delta\}$

Second: $\sigma = \{coin\} \Rightarrow$
out $(p1_0$ **after** $\sigma) = \{milk, tea\} \subseteq$

$\mathbf{out}\,(l_0\ \mathbf{after}\ \sigma) = \{coffee, milk, tea\}$

There are of course a lot more paths, it is always possible to add a δ transition in the path, but this does not really change anything in this short example. The observations that can be made after any path in the implementation `P1` are a subset of the observations of the specification `L`, so `P1` *ioco* `L`.

The second implementation `P2` is almost the same as the specification, there is just another transition from the state $p2_1$ to $p2_5$ labeled with the output action $!soda$. The set of states of `P2` is defined by $P2 = \{p2_0, p2_1, p2_2, p2_3, p2_4, p2_5\}$, the set of output actions Σ_U consists of $\{coffee, milk, soda, tea\}$. Consider again the two paths in the specification `L` and in the implementation `P2`:

First: $\sigma = \{\} \Rightarrow$
$\mathbf{out}\,(p2_0\ \mathbf{after}\ \sigma) = \{\delta\} \subseteq$
$\mathbf{out}\,(l_0\ \mathbf{after}\ \sigma) = \{\delta\}$

Second: $\sigma = \{coin\} \Rightarrow$
$\mathbf{out}\,(p2_0\ \mathbf{after}\ \sigma) = \{coffee, milk, soda, tea\} \not\subseteq$
$\mathbf{out}\,(l_0\ \mathbf{after}\ \sigma) = \{coffee, milk, tea\}$

The observations of the second path of the implementation `P2` are no subset of the specification `L`, so the *ioco* relation is not fulfilled for the implementation `P2` and `L`.

Chapter 4

Input Output Symbolic Transition Systems (IOSTS)

Contents

4.1 Introduction

This chapter describes the concepts of Input/Output Symbolic Transition Systems (IOSTS). There are basically two definitions of IOSTS in literature: [Rusu et al., 2000] and [Frantzen et al., 2005]. The IOSTS defined in Section 4.2 is based on the definition in [Rusu et al., 2000]. Section 4.3 describes the concept of conformance testing taking IOSTS into account. There is a brief introduction of the *sioco* conformance relation [Frantzen et al., 2006]. This relation defines the theoretical background of conformance testing with IOSTS. The terms test purpose and test case are also explained. The last section (Section 4.4) describes the process of generating test cases from an IOSTS system specification and an IOSTS test purpose.

4.2 Definition of IOSTS

An IOSTS [Rusu et al., 2000] is a tuple $\langle D, \Theta, Q, q^0, \Sigma, T \rangle$, whereas:

- D is a finite set of typed data. It consists of the disjoint union of a set V of variables, a set P of parameters and a set M of messages.
- Θ is the initial condition, a boolean expression on $V \cup P$.
- Q is a nonempty, finite set of states.
- q^0 is the initial state.
- Σ is a nonempty finite alphabet of actions. It is a disjoint union of a set Σ^i of input actions, a set Σ^o of output actions and a set Σ^{int} of internal actions. Each action has a (possibly empty) set of types assigned to it called the signature of the gate ($sig\,(a) = \langle \vartheta_1, ...\vartheta_k \rangle$; the signature of an internal action $a \in \Sigma^{int}$ is the empty tuple).
- T is a set of transitions. Each transition consists of:
  - An origin $q \in Q$.
  - A destination $q^{'} \in Q$.
  - An action $a \in \Sigma$. Each action has a tuple of messages $\mu = \langle m_1, ...m_k \rangle$ assigned to it, whereas $m_i \in M$. The signature of the messages corresponds to the signature of the action $a \in \Sigma$ ($sig\,(a) = \langle \vartheta_1, ...\vartheta_k \rangle \Rightarrow \forall i \in [1, k]\,, type\,(m_i) = \vartheta_i$).
  - A guard which is a boolean expression on $V \cup P \cup \mu$.
  - A set of assignments A. Each variable $x \in V$ has exactly one assignment in A, of the form $x := A^x$, whereas A^x is an expression on $V \cup P \cup \mu$.

An example of an IOSTS can be found in Section 5.3.

4.3 Conformance Testing with IOSTS

Conformance testing is the methodology of testing an implementation against a specification. This is done by means of a conformance relation. Chapter 3 describes the conformance relation *ioco*. However, this relation is based on Input/Output Labeled Transition Systems. The transformation algorithm explained in this thesis uses Input/Output Symbolic Transition Systems instead of that for generating test cases. [Frantzen et al., 2006] define the conformance relation *sioco* (symbolic input output conformance) for IOSTS. This relation is very similar to *ioco*, in fact they lift the *ioco* relation to the level of symbolic transition systems. *sioco* is basically a symbolic version of *ioco* by taking the enhancements of IOSTS into account (data-dependent control flow).

The specification of the system to be tested is an IOSTS defined in Section 4.2. In order to understand the test case generation process in Section 4.4, two more concepts have to be explained [Rusu et al., 2000]:

- *Test Purpose*: The test purpose is, like the specification, an IOSTS. It determines the parts of the specification that should be tested. So the test purpose is like a subset of the specification that only contains the relevant parts of it. There are additional states that have to be added to the test purpose, an *Accept* and a *Reject* state in order to determine whether the test purpose is fulfilled or not. The paths that lead to the *Reject* state do not mean that the specification behaves false, they are just not relevant in context of the test purpose. Thus, only paths that lead to the *Accept* state should be considered by the resulting test case. Formally, let $\mathtt{S} = \langle D_S, \Theta_S, Q_S, q_S^0, \Sigma_S, T_S \rangle$ be a specification, a test purpose is defined by $\mathtt{TP} = \langle D_{TP}, \Theta_{TP}, Q_{TP}, q_{TP}^0, \Sigma_{TP}, T_{TP} \rangle$. The set of states of the test purpose is extended by an *Accept* and a *Reject* state: $\{Accept, Reject\} \subseteq Q_{TP}$.

- *Test Case*: The test case is the result of the test case generation process explained in Section 4.4. The test case is also an IOSTS that has some additional states called verdicts; these are *Pass*, *Inconclusive* and *Fail*. The test case is then run parallel with the implementation. The state of the test case at the end is observed to determine whether the implementation behaves in conformance with the specification or not. The verdict *Pass* means, that the implementation behaves correctly. That means that it behaves in conformance with the specification and the test purpose is satisfied. *Inconclusive* means, the implementation behaves in conformance with the specification but the test purpose is not satisfied. It is just known that the implementation behaves right until it leaves the test purpose, it is not known if it behaves correct after that. So *inconclusiveness* means that it is not really clear if the behavior of the implementation is right or wrong. Remember the test purpose has an *Accept* and an *Reject* state. Paths that lead to a *Reject* state are just not considered by the test purpose, although they are conformant with the specification. Finally, *Fail* denotes that the implementation is not conformant with the specification, i.e. an output is observed at any state that is not allowed there. There are two attributes a test case should comprise

  - *input completeness*: A test case should be input complete, that means it can handle all inputs at all state where inputs are possible.

– *determinism*: The result of a test case should not depend on the path that is taken w.r.t. to a non deterministic choice, that means a state cannot have two outgoing transition labelled with the same action.

4.4 Test Case Generation Process

There are different approaches to generate test cases from an IOSTS. There is an algorithm for on-the-fly test case generation based on LTS [Tretmans, 1996] or STS [Frantzen et al., 2005] specifications. It produces test cases with verdicts *Pass* and *Fail*. This algorithm is not deterministic. It is also shown that the test cases produced with the algorithm are sound and complete. A test suite `T` is sound means that for all implementations `P` that conform to a specification `S` w.r.t $ioco_F$ ($ioco_F$ is just a general version of the *ioco* conformance relation described in Section 3.3), the implementations `P` pass the test suite `T`. Completeness means that the algorithm can produce test cases that show that an implementation is non conformant if it is not $ioco_F$-correct w.r.t. the specification. In general [Frantzen et al., 2005]:

- T *is sound* $=_{def} \forall \mathtt{P} : \mathtt{P}\ ioco_F\ \mathtt{S} \Rightarrow \mathtt{P}\ passes\ \mathtt{T}$
- T *is complete* $=_{def} \forall \mathtt{P} : \mathtt{P}\ ioco_F\ \mathtt{S} \Leftarrow \mathtt{P}\ passes\ \mathtt{T}$

whereas `S` is a specification, `P` an implementation and `T` a test suite. The algorithm is not described in detail here.

As explained in Section 4.3, the *sioco* conformance relation can be defined with the aid of *ioco*. The *ioco* conformance relation is just lifted to the level of STS. So soundness and completeness can also be achieved with this algorithm which is based on STS. Details about the algorithm can be found in [Frantzen et al., 2005], the algorithm is not discussed in detail here since it is not used in this application.

Another approach for generating test cases from IOSTS is described in [Rusu et al., 2000]. The algorithm uses some operations on IOSTS like product, closure, determinization,... The exact definitions of these operations are not shown here, only their meaning is explained. For a detailed description refer to [Rusu et al., 2000] and [Constant et al., 2007]. The process consists of the following steps (see Figure 4.1):

- *product between test purpose and specification*: The product operation is one of the operations defined in [Rusu et al., 2000] and [Constant et al., 2007]. It results in an IOSTS that combines the states and transitions of the specification and the test purpose, including the *Accept* and *Reject* states of the test purpose. Paths that lead to an *Accept* state fulfill, paths that lead to a *Reject* state are not considered by the test purpose. The result of this operation is a subgraph of the specification enriched with *Accept* and *Reject* states, the two IOSTS are somehow put together. The product operation cannot be applied to two arbitrary IOSTS, they have to fulfill some requirements: They have to share the same input and output actions as well as the same parameters. They must not have any variables or internal actions in common. A feature of the resulting IOSTS is that each input action of

the specification/test purpose becomes an output action in the result and vice versa (the renaming is actually a separate step not included in the product operation, but here it is assumed that this step is done after the product). This is due to the fact that the generation process produces a test case. The test case is then run parallel against an implementation, thus an observed output of the implementation serves as input to the test case.

- *closure*: The test case under construction has to run parallel to the implementation, so it has to react to inputs from the implementation at any given state. One problem that may occur is that a state of the resulting IOSTS, after the product operation described above, has only transitions leaving it labelled with output or internal actions. If there is no transition labelled with an input action, the test case cannot react to an input of the implementation. There is a requirement of the test case that says a test case, i.e. its states must be input complete (see Section 4.3), that means that a state must be able to react to all inputs at any given state. The closure operation takes care of this by eliminating internal actions. If a transition leaving a state is labelled with an internal action, its guard and assignments are added to the next transition (the state between the two transition can be eliminated). If the next transition is also labelled with an internal action, this is repeated until all internal actions are eliminated. The resulting IOSTS has no longer transitions with internal actions and can in that way better react to a given input of the implementation (note that the closure operation does not make the whole IOSTS input complete, this is done in the *selection* step). One problem concerning closure are cycles of internal actions. This issue is addressed in [Rusu et al., 2000].

- *determinization*: One characteristic of a test case is the fact that it should be deterministic. After the product and closure operation it is however not assured that the resulting IOSTS is deterministic, so a determinization step has to be applied on the result. A typical case of nondeterminism is a state that has two outgoing transition with the same action assigned to it. The result of the test case should not depend on which of the two transitions is taken because this could cause a tester to wrongly declare a conformant implementation to be non-conformant. [Rusu et al., 2000] and [Constant et al., 2007] propose a method to eliminate nondeterminism, a detailed description on this issue can be found in [Jéron et al., 2006].

- *selection*: The goal of the selection step is adding verdicts to the test case. As mentioned in Section 4.3, a test case contains states called verdicts. These are *Pass*, *Inconclusive* and *Fail*. The IOSTS created after the product, closure and determinization consists amongst others of states called *Accept* and *Reject*. Adding verdicts to this IOSTS is done as follows:

  - Every *Accept* state is labelled with *Pass*. These are the states in which the implementation behaves conformant to the specification and the test purpose is fulfilled
  - Every *Reject* state is labelled with *Inconclusive*. These are the states in which the implementation behaves conformant to the specification but the test purpose cannot be fulfilled.

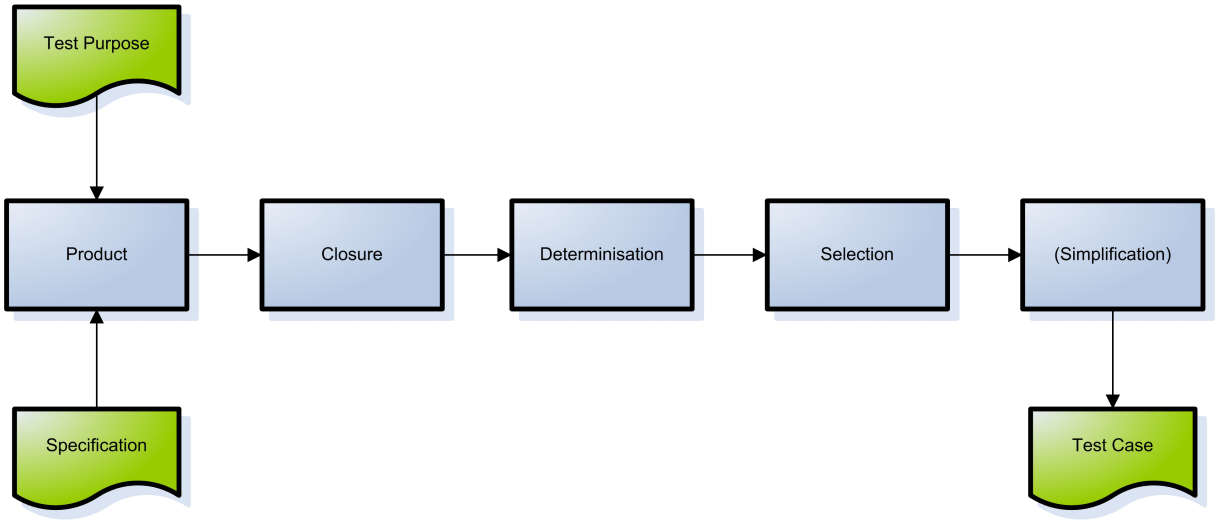


Figure 4.1: Test Case Generation Process

– Every other state is labelled with *Lead2Pass*.

Now there are three sets of states, *Pass*, *Inconclusive* and *Lead2Pass*. Although the closure step removed internal transitions, the IOSTS obtained so far is not input complete, i.e. there are states in the set *Lead2Pass* that cannot handle every input. To be able to make the test case input complete, a new state called *Fail* is added to the IOSTS. Every input in a state of the set *Lead2Pass* that is not handled yet does not conform to the specification, that means that this input is not allowed at the given state. So, for each of this inputs in every state a transition is added labelled with that input that leads to the new *Fail* state. The test case in now input complete. It can handle every input at every state and every path leads to either a *Pass*, *Inconclusive* or *Fail* state.

- *simplification*: The test case obtained so far comprises all properties defined in 4.3. However, due to the generation so far, the test case can contain some unreachable parts. [Rusu et al., 2000] refer to methods to detect and eliminate such unreachable paths. This makes the resulting test case much simpler and easier to understand, although this step is not mandatory.

[Rusu et al., 2000] also show that the resulting test case is correct, by defining that the test case must be *sound* for the specification and implementation and *relatively complete*, *accurate* and *conclusive* for the specification, test purpose and the implementation.

- *sound*: Soundness means a test case does not reject conformant implementations, only non conformant ones.

- *relatively complete*: Relatively completeness means that the test case can discover all implementations that are not conformant to the specification.

- *accurate*: Accuracy means that if in the implementation a trace is observed, that conforms to the implementation and is included in the test purpose (it ends in an *Accept* state), the test case always gives the verdict *Pass*.
- *conclusive*: Conclusiveness means that if the observed trace of an implementation which is also a trace in the specification leads to the verdict *Inconclusive*, this path cannot be extended in any way to produce the verdict *Pass* subsequently.

Chapter 5

The Symbolic Test Generator (STG)

Contents

5.1 Introduction

This section gives an overview of the **S**ymbolic **T**est **G**enerator (STG). This tool implements the test generation process described in Section 4.4 based on an IOSTS system specification and test purpose. It then generates executable test cases. The basic workflow of this tool is explained in Section 5.2. Section 5.3 gives a short example of a coffee machine that illustrates the process of generating test cases. Another example can be found in [Clarke et al., 2001a]. The last section (Section 5.4) introduces the STG language specification the STG tool uses for specifying IOSTS.

5.2 STG Workflow

The STG tool is introduced in [Clarke et al., 2001b] and [Clarke et al., 2002]. It is implemented in Ocaml. It implements the test case generation process described in [Rusu et al., 2000] and Chapter 4.4. The overall workflow of the STG tool is depicted in Figure 5.1. The STG tool takes as input two IOSTS, a specification of a system and a test purpose. They have to conform to the STG language specification described in Section 5.4. The STG tool does not take care of constructing these IOSTS, they have to be provided. Afterwards, the STG tool generates a symbolic test case by performing the test generation process described in Chapter 4.4. The simplification step of this process is done with the NBAC [Jeannet, 2008] tool. This tool provides the opportunity to analyze the test case and remove unreachable parts. After the test case generation process, the symbolic test case can be translated to either an executable C++ or JAVA test program. Since the symbolic test case provides no actual values to the symbolic data, the Lucky library [Verimag, 2008] is used to instantiate this data in the test case. The test program is able to interact with an implementation that is interface-compatible with the specification. The results of the test program are *Pass*, *Inconclusive* or *False*. Their meaning is described in the test case generation process (see Section 4.4). The STG tool can be obtained here [IRISA, 2008a].

5.3 STG Example

This section provides an example application of the test case generation process implemented in STG. First, the system specification is introduced. The next section provides the test purpose. Last, the resulting test case is depicted.

5.3.1 System Specification

The system specification depicted in Figure 5.2 describes a simple vending machine. A user may insert coins, choose a beverage or cancel the transaction. The first line of a transition always denotes the guard, surrounded by parenthesis (if there is no guard, true is depicted). The action of a transition is clarified through the *sync* keyword. Input actions are denoted by ?, output actions by ! respectively. The assignments of the transition are situated below the action.

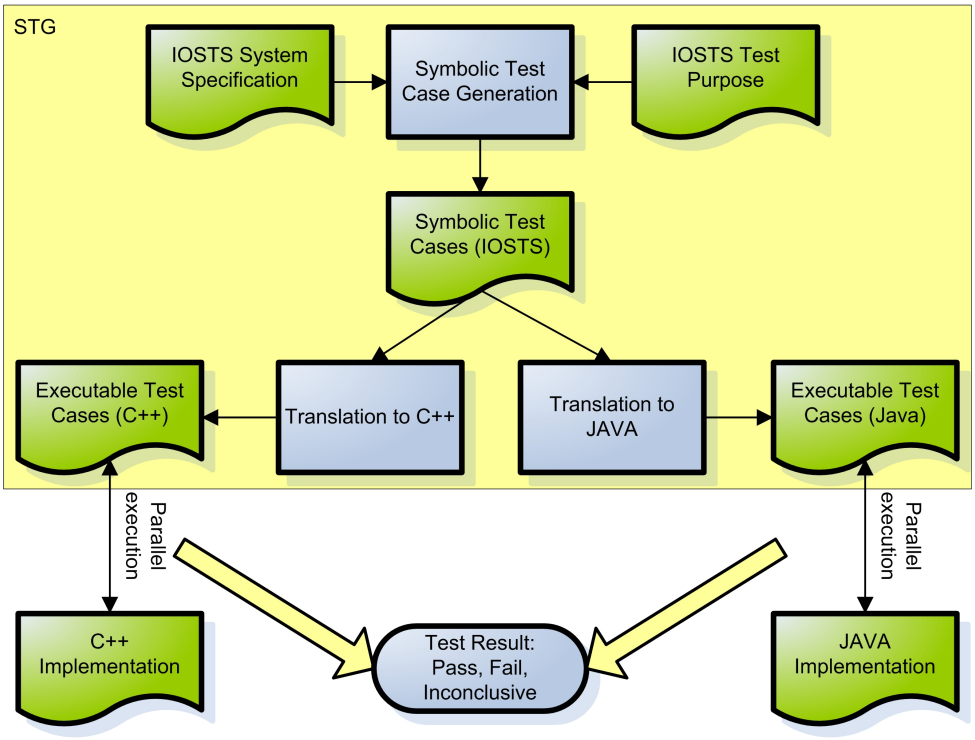


Figure 5.1: STG Workflow

The set of variables V is defined by $\{total, drink\}$, the set of parameters P is $\{price\}$. The set of states S contains $\{PreInit\_Start,\ Begin,\ Idle,\ Pay,\ Choose,\ Return,\ Delivery\}$, whereas the initial state is *PreInit_Start*. The set of input actions is defined by $\{Coin,\ ChooseBeverage,\ Cancel\}$, the set of output actions comprises $\{Return,\ Deliver\}$. The first transition from *PreInit_Start* to *Begin* just contains a guard ($price > 0$). This is the initial condition Θ (see Section 4.2). In STG, the first transition always defines the initial condition. In this case, the parameter *price* must not be negative. The transition from *Begin* to *Idle* just initializes the variable *total* (all variables must be initialized in STG). The transition from *Idle* to *Pay* simulates the insertion of a coin into the vending machine. So there is an input action *Coin* that takes as message the actual *coin* that is inserted. The value of the message *coin* must be greater that 0, this is checked in the guard. The value of the inserted coins is accumulated in the variable *total*. If this value is smaller than the price of the beverage stored in the parameter *price*, the transition from *Pay* to *Idle* is taken. The machine informs the user about the remaining amount of coins she has to enter (output action *Return*). Now the user can insert some more coins or cancel the transaction (transition from *Idle* to *Return*) by calling the input action *Cancel*. The transition from *Pay* to *Choose* is taken if the user entered enough money ($total \geq price$). The difference between the entered amount and the actual price is returned to the user by calling the output action *Return*. In the *Choose* state, the user may again cancel the transaction by calling the input action *Cancel* (transition from *Choose* to *Cancel*). Otherwise, the machine checks whether the entered amount of money equals the price of the beverage (guard on the transition from *Choose* to *Delivery*). The user is now able to choose the drink by performing the *ChooseBeverage* action. The *drinkRequest* message

is saved in the variable *drink*. After that, the transition from *Delivery* to *Begin* checks if the actual *drink* equals the selected and delivers the *drink* to the user by performing the *Deliver* output action. Finally, if the user chooses to abort the transaction in the *Idle* or *Choose* state, the transition from *Return* to *Begin* takes care of returning the already inserted money back to the user (output action *Return*).

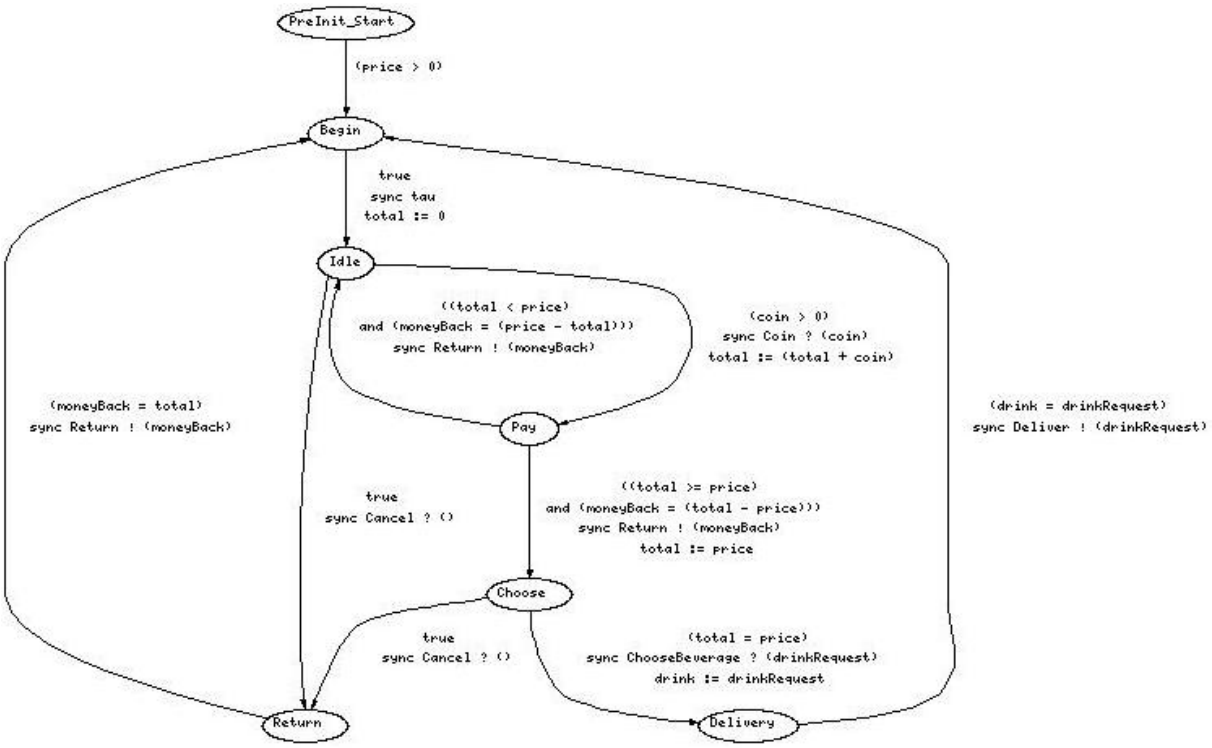


Figure 5.2: Coffee Machine Specification (source [IRISA, 2008a])

5.3.2 Test Purpose

Figure 5.3 shows one possible test purpose of the coffee machine system specification depicted in Figure 5.2. As mentioned in Section 4.3, a test purpose is a subgraph of the specification that only contains the elements that should be tested. The test purpose also comprises *Accept* and *Reject* states, denoting whether the test purpose is fulfilled or not. This test purpose is just interested whether the coffee machine is able to deliver coffee. Only by choosing coffee, the path ends in the accept state. If the user chooses tea, the path ends in the reject state, since the test purpose is not interested in that behavior (this behavior is not false, it is just not part of the test purpose). The user may also insert a coin only once, otherwise the test purpose is not fulfilled. Canceling the transaction is also not in the scope of the test purpose. So the test purpose is only interested in paths on which the user inserts a coin (the value of the coin must be big enough so that she does not have to insert a coin again) and chooses coffee, which is then delivered.

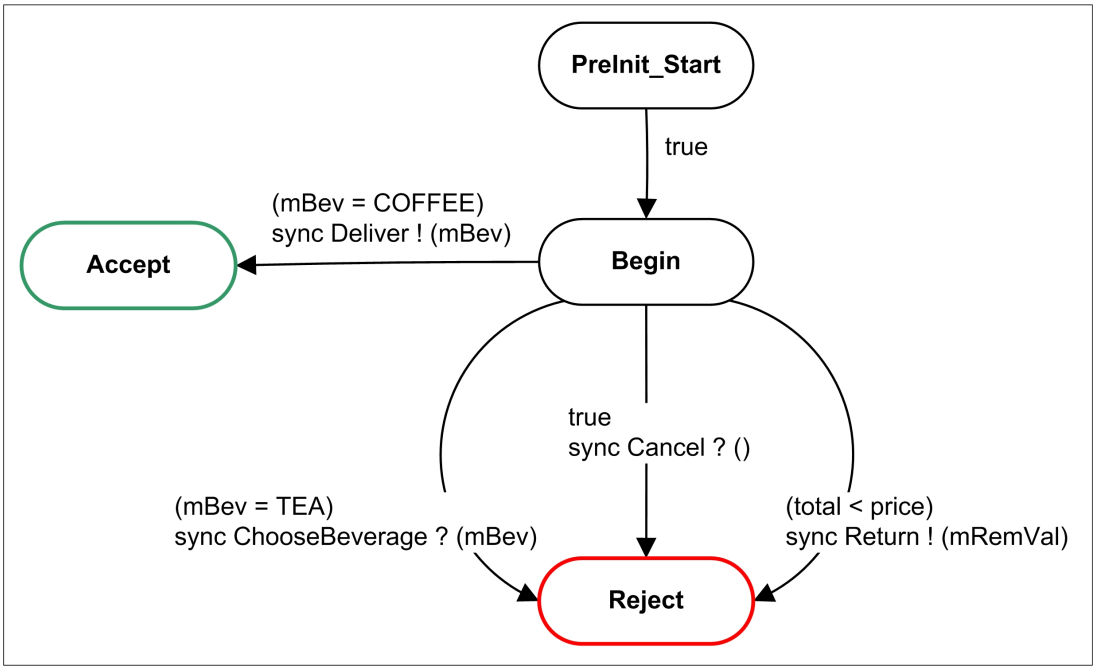


Figure 5.3: Coffee Machine Test Purpose [Clarke et al., 2002]

5.3.3 Test Case

Figure 5.4 shows the test case produced by the STG tool from the specification and test purpose after the test case generation process explained in Section 4.4. It consists of a path where the user inserts a coin and chooses coffee. Afterwards the coffee is delivered by the coffee machine. For readability reasons, the transitions that lead to a *Fail* state are removed in the figure. In this example the selection and simplification step of the test case generation process also removed the *Inconclusive* states. The guards of the transitions are not easy to understand, they seem to be very complex. This is due to the fact that the STG tool uses the NBAC tool for the simplification step. This tool does not deal with equalities, just with inequalities. So a simple equality like $x = 0$ is translated to $\neg((x - 1) >= 0) \wedge x >= 0$. This makes the guards seem to be a lot more complex than they actually are. This test case can now be translated to a executable test case in C++ or JAVA and the run parallel to the implementation to obtain the test results (*Pass*, *Inconclusive* or *Fail*).

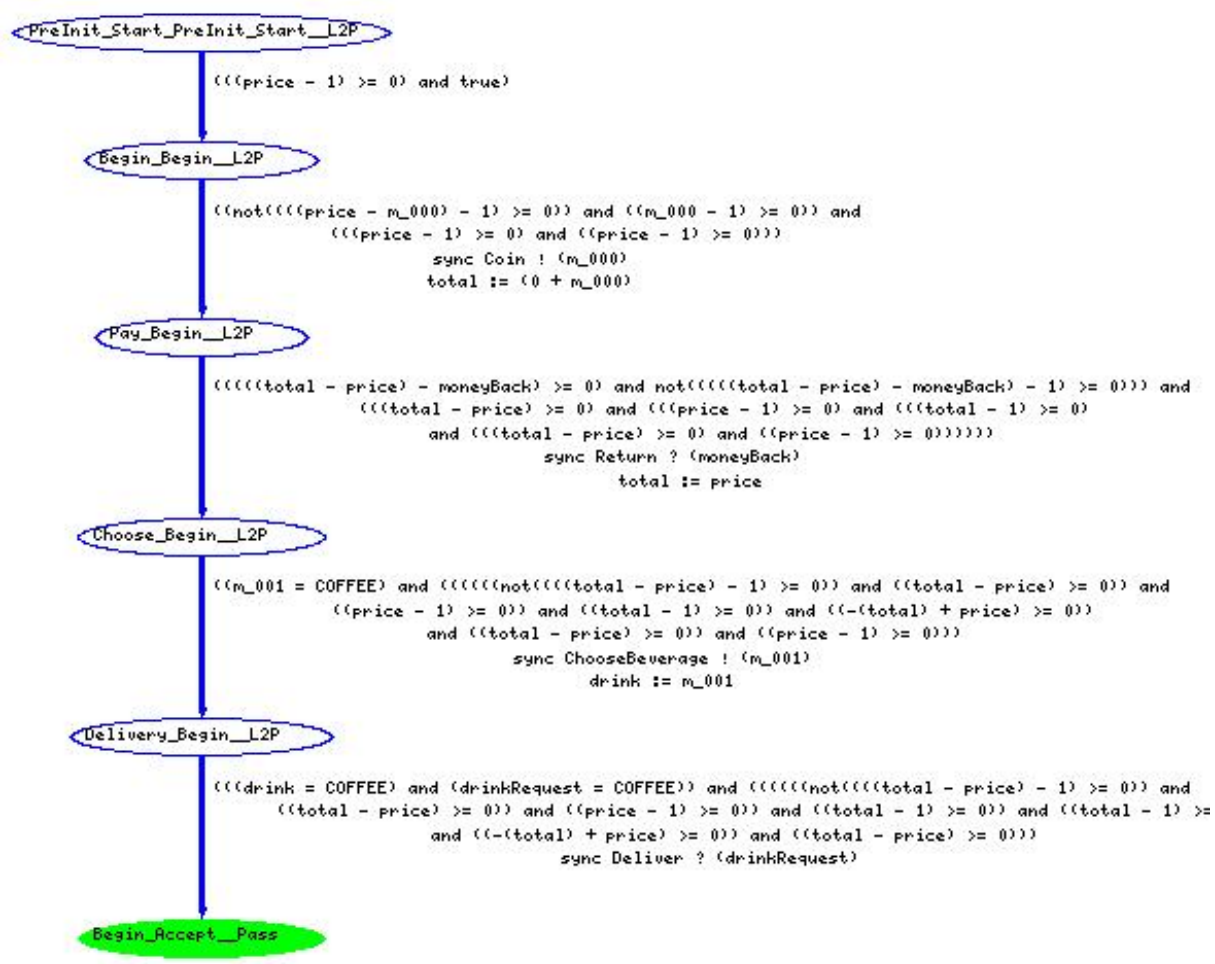


Figure 5.4: Coffee Machine Test Case (source [IRISA, 2008a])

5.4 STG Language

In STG the top level element is called system. Each system consists of a system identifier (a unique name), a set of constants, a set of types, a set of gates and a set of processes. Constants and types may be empty. Each process is an IOSTS which conforms to the definition in Chapter 4.2. All actions used in the processes must be defined in the top level system (actions are gates in the system). The distinction between input, output and internal actions is not made in the system. Each process may classify the predefined gates on its own. So the same gate may be an input action in one process and an output action in the other. Variables and parameters are also defined within each process. A detailed definition of the language used in STG can be found in Appendix A.

Chapter 6

UML

Contents

6.1 Introduction

This section describes general concepts of the **U**nified **M**odelling **L**anguage UML, especially those parts that are used in the transformation process. Section 6.2 describes the overall structure of UML. Section 6.3 introduces the structural elements of UML used in this application, especially the *Class Diagram*. The next section (Section 6.4) takes a closer look at the behavioral elements of UML. The main focus in this section are *State Machine* and *Activity Diagrams*. There is an overall description of these elements. According to that, the restrictions that have to be made in this application are also discussed. Section 6.5 introduces the extensibility mechanisms of UML. The last section (Section 6.6) shows some concepts of the OCL language that is also part of the UML specification.

6.2 General

The UML 2 specification is structured in four different parts:

- *UML Infrastructure:*

  The *UML Infrastructure* [OMG, 2007a] describes the main elements used in UML like classes, associations or packages. These elements are used by the UML meta model defined in the *UML Superstructure*, so this specification defines the meta meta model of UML.

- *UML Superstructure:*

  The *UML Superstructure* [OMG, 2007b] defines the UML meta model. It describes all model elements by means of the elements defined in the UML Infrastructure. All elements are introduced with a *Class Diagram* of the meta model and their meanings in prosa. The specification is divided in *structure*, *behavior* and *supplement*.

  Static elements that describe the overall architecture of the system are defined in the *structure* part of the *UML Superstructure* like classes, components, etc. Note that the UML specification describes model elements, not diagrams. However, the graphical definition used in diagrams of these model elements is also part of the specification. For example, the *UML Superstructure* defines the model element *State Machine* that contains other model elements like states, transitions, regions, etc. The graphical definition of these model elements is described later. It is divided in *graphic nodes* (e.g. like states) and graphic paths (like transitions).

  The *behavior* part describes the behavior of the system. There are elements that describe the life cycle of an object such as *State Machines* as well as elements to depict how objects interact with each other (*Interactions*). Elements that describe work flow are also part of the specification (*Activities*). The *UML Superstructure* also defines basic elements that are used by more than one diagram like *Common Behaviors* or *Actions*.

  Last, the *supplement* part of the specification describes *Auxiliary Constructs*, i.e. information flows, modules, primitive types and templates. Another part of this chapter deals with the use of *Profiles* to extend the UML specification.

Thus the *UML Superstructure* is structured in:

  - Structure
    - Classes
    - Components
    - Composite Structures
    - Deployments
  - Behavior
    - Actions
    - Activities
    - Commmon Behaviors
    - Interactions
    - State Machines
    - Use Cases
  - Supplement
    - Auxiliary Constructs
    - Profiles

- *UML Diagram Interchange Specification:*

  This specification [OMG, 2006a] extends the UML specification with some graph-oriented information to ease the interchange of UML models between different UML tools.

- *UML OCL Specification:*

  This specification [OMG, 2006b] defines the *Object Constraint Language*. OCL is an extension to UML that can be used to describe constraints on UML models. Section 6.6 gives a closer look to the OCL language.

The system to be transformed to an IOSTS is modeled in UML. Basically there are three diagram types involved in the transformation.

- A *Class Diagram* to describe the overall static architecture of the system.
- A *State Machine Diagram* to describe the behavior of the classes.
- A very basic *Activity Diagram* to describe an effect of a transition in a *State Machine*.

Since there are a lot of concepts in UML concerning these diagrams, only the basic parts of the UML specification are used in this application.

6.3 Structure

This section deals with the structural part of the *UML Superstructure*.

6.3.1 Structural Elements

Structural elements define static constructs that are contained in the structural diagrams of UML such as *Class Diagrams*, *Component Diagrams*, *Deployment Diagrams*. Examples for structural elements are classes, components, nodes artifacts... Since only *Class Diagrams* are part of the application, there won't be a detailed description of all these elements, only the main parts contained in *Class Diagrams* which are needed for the application are described next.

6.3.2 Class Diagram

A *Class Diagram* describes the overall static structure of a system [Pender, 2003]. A class consists of three compartments, the name compartment, the attributes compartment and the operations compartment.

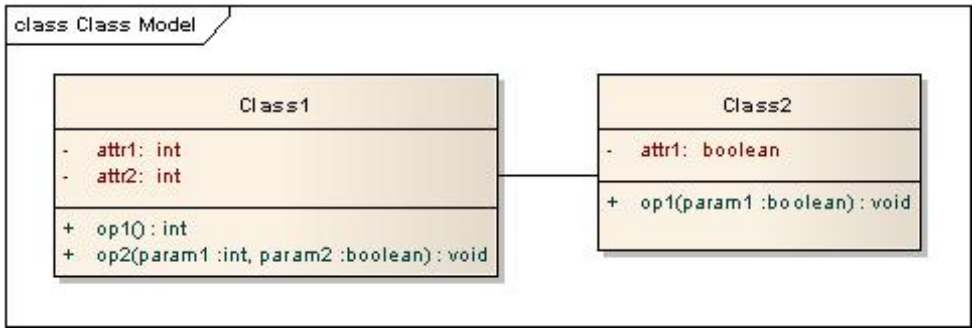


Figure 6.1: Simple Class Model

Name: This compartment contains the class name that must be unique within the *Class Diagram*.

Attributes: This compartment contains a list of the attributes of the class. An attribute is defined by its name and a type. The syntax of an attribute declaration is:

```
[visibility] name [: data-type]
```

whereas the visibility is either *public (+)*, *private (−)*, *protected (#)* or *package (∼)*. The name of the attribute must be unique within the class. In UML it is also possible to model derived attributes. Derived attributes depend on other attributes. They cannot be set manually; they are computed on the fly. This concept is not provided in this application.

Operations: This compartment contains the operations of the class. Operations can be called from other objects (if they are public) and may cause a change in the state of the object. The notation of an operation is:

```
[visibility] name ([parameter-list]) ':' [return-result]

parameter-list := name [':' data-type ]
```

The visibility of the operation is the same as of attributes. Each operation has a name and optionally a return value. The parameters of an operation are also defined by a name and a data type.

Associations: Relations between classes are described using associations. An association is represented by a solid line between classes. Objects of classes may only interact with each other if they share an association in the *Class Diagram*.

6.4 Behavior

This section deals with the behavioral part of the *UML Superstructure*.

6.4.1 Behavioral Elements

UML provides a lot of behavioral elements to describe behavioral diagrams such as *State Machine Diagrams*, *Sequence Diagrams*, *Activity Diagrams*,... Only those needed for the application are described in this chapter.

Figure 6.2 shows a *Package Diagram* of those packages that support behavioral modeling in UML. The *CommonBehaviors* package comprises the *BasicBehaviors* package shown in Figure 6.3, which contains the *Behaviors*, as well as the *Communication* package that includes constructs like *Event* (see Figure 6.5) and *Trigger* (see Figure 6.4). The package *Activities* contains elements to construct *Activity Diagrams*. *Sequence Diagrams* use elements of the *Interactions* package. *State Machine* and *Use Case Diagrams* use constructs of their respective package. The *Action* package contains various elements describing actions in UML.

BasicBehaviors

The class *Behavior* of the *BasicBehaviors* package (see Figure 6.3) is very important for the application since transitions and states of the *State Machine Diagram* (see Figure 6.6) reference it. Each behavior has a specification, which is a *BehavioralFeature* (e.g. an operation of a class or a signal), associated with it. Subclasses of *Behavior* are *OpaqueBehavior* (which itself is a superclass of *FunctionBehavior*). *State Machine*, *ProtocolStateMachine*, *Activity* and *Interaction* are also generalized from *Behavior* (not depicted in Figure 6.3), so whenever a *Behavior* is needed, one of these elements can be used.

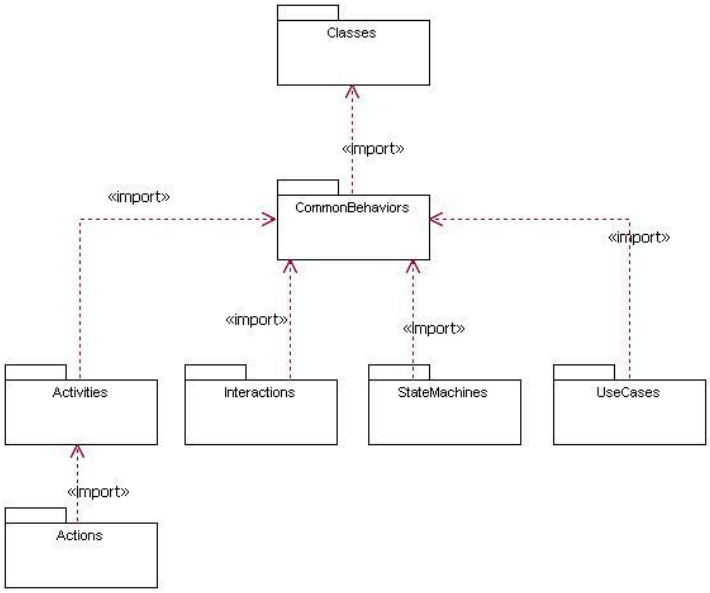


Figure 6.2: UML Packages that support Behavioral Modeling (source: [OMG, 2007b], page 215)

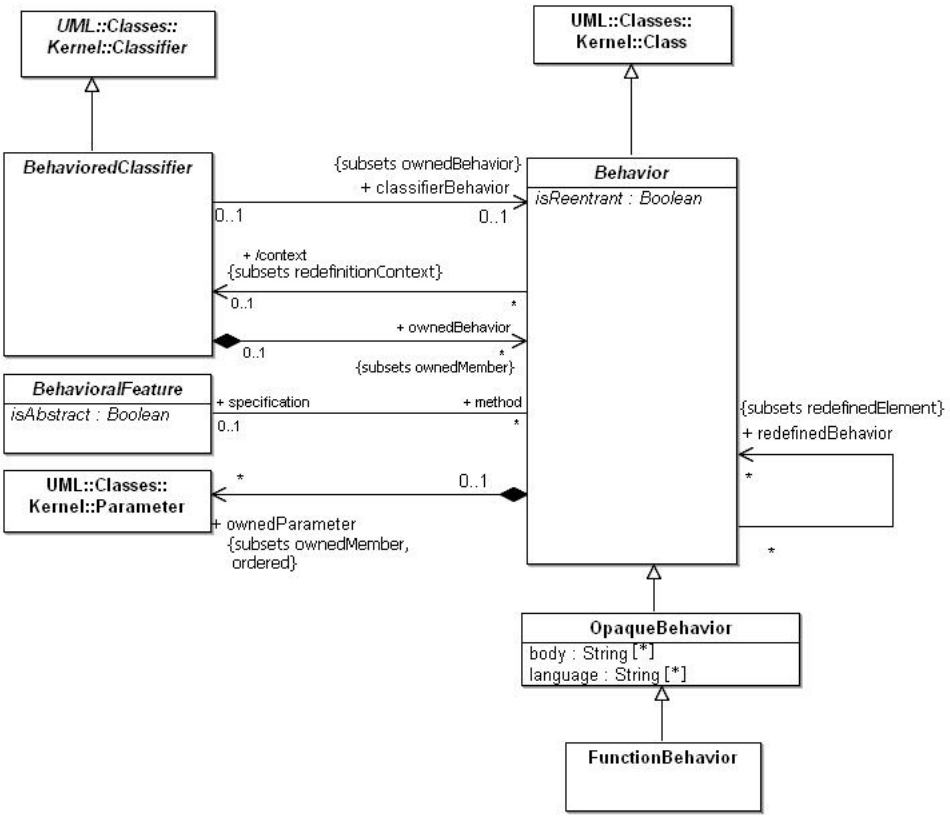


Figure 6.3: Basic Behaviors (source: [OMG, 2007b], page 424)

Trigger

Triggers are contained in the *Communications* package contained in the *CommonBehaviors* package of Figure 6.2. In *State Machines*, transitions and deferred events reference triggers. A trigger itself is a named element which is always associated with exactly one event. Events are explained in the next section.

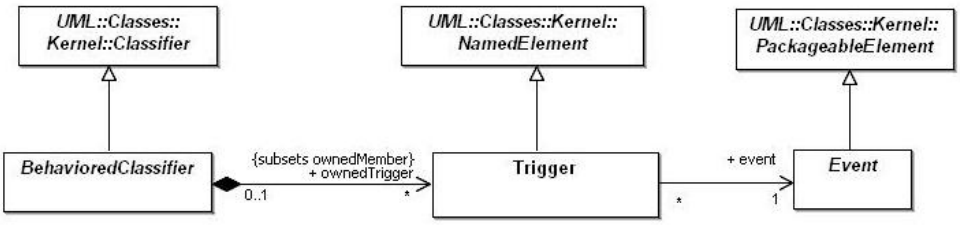


Figure 6.4: Triggers (source: [OMG, 2007b], page 426)

Event

UML defines many kinds of events, not all of them are supported by the application. Since STG does not deal with timing, *TimeEvents* are omitted. *AnyReceiveEvents* are also not part of the application. *SignalEvents* are always associated with a given signal of the UML model. *CallEvents* on the other hand are associated with an operation of the given *Class Diagram*. *ChangeEvents* contain a *ValueSpecification* which is basically a boolean expression. They are also not part of this application. So, only *CallEvents* and *SignalEvents* are considered.

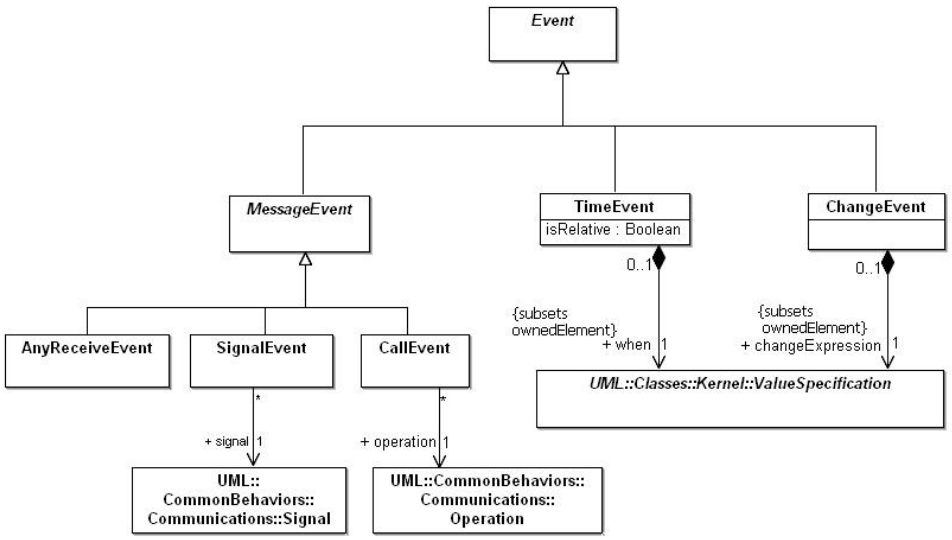


Figure 6.5: Events (source: [OMG, 2007b], page 426)

6.4.2 State Machine Diagram

A *State Machine Diagram* describes the life cycle of an object. It depicts the states of an object as well as its behavior, i.e. how the object reacts to events (events are always contained in a trigger). A *State Machine* is associated with a class, so every active class is supposed to have a *State Machine Diagram* associated to it. The basic components of a *State Machine* are states and transitions. Figure 6.6 shows the complete meta model of a *State Machine*. A *State Machine* consists of at least one region. Each region contains several vertices and transitions. Vertices are either states or pseudostates. The type of the pseudostate is enumerated in the *PseudostateKind* enumeration. This Application only allows initial and choice pseudostates. The number of regions is also restricted to one since more regions imply concurrency which is not really supported in the target model (IOSTS). States and transitions are basically the main parts of a *State Machine*, so they are described in the next sections.

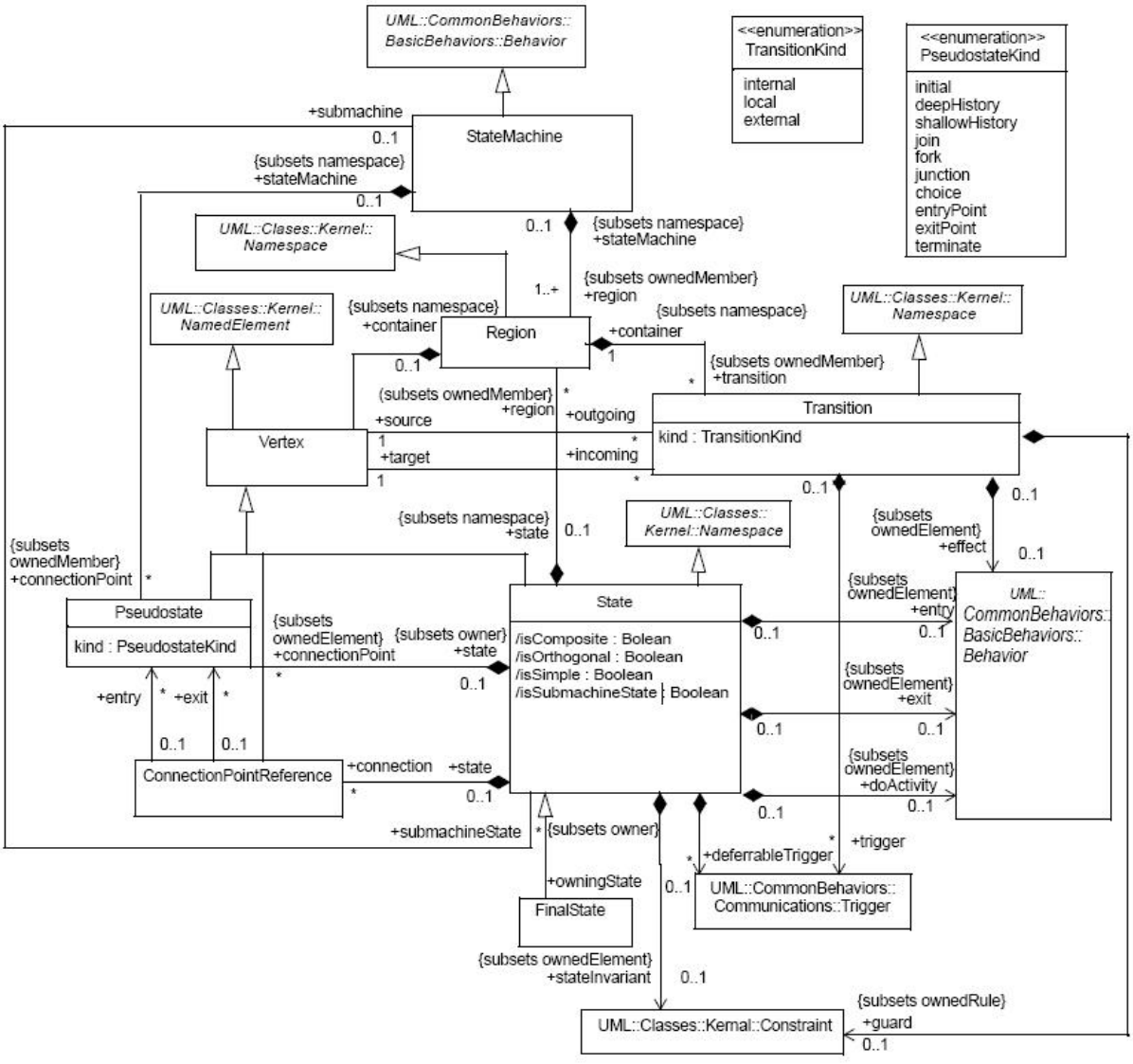


Figure 6.6: State Machine Meta Model (source: [OMG, 2007b], page 525)

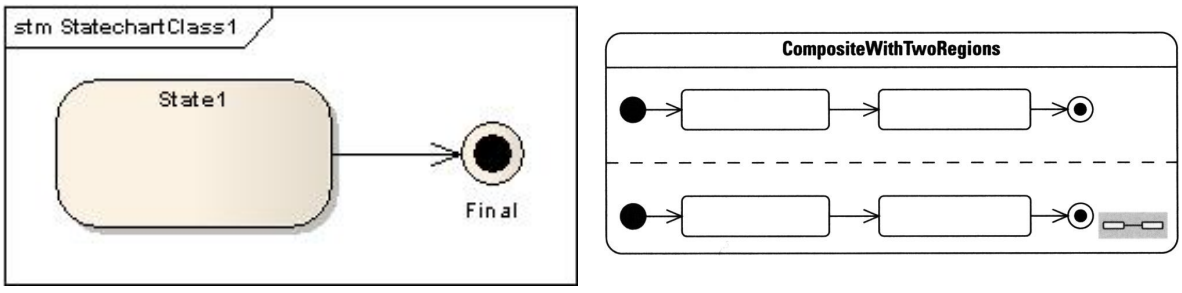
States

A state describes the current status of an object, i.e. the values of its attributes at a certain period of its life cycle. There are different kinds of states that are described next.

Simple State: A simple state is a state with the boolean flag *isSimple* set to true. As mentioned before, a simple state describes the state of the object, i.e. the values of its attributes at a certain period in its life cycle. Since a state is generalized from vertex, it may have incoming and outgoing transitions. There are some additional concepts a simple state may contain. If an action, i.e. a behavior occurs at every event that transitions into the state, it may be modeled as an entry action in the given state. The same concept can be applied to exit actions. Another construct that may be applied to simple states is a do activity. Do activities are performed from the time the object enters the state until it leaves it. As depicted in Figure 6.6, entry actions, exit actions and do activities are behaviors in the UML 2 meta model. As described in Chapter 6.4.1, a behavior may be an *OpaqueBehavior*, a *FunctionBehavior*, an *Activity*, an *Interaction*, a *State Machine* or a *Protocol State Machine*. This application does not include entry and exit actions since a *State Machine* can be modeled without them not changing its semantic. It also does not support the concept of do activities. Deferred events are events the object cannot respond to at the given state. The event is then passed to the next state until the object can respond to it. The syntax of these concepts is shown next:

```
Entry actions:    ['entry /'[behavior]]
Exit actions:     ['exit /'[behavior]]
Do activities:    ['do /'[behavior]]
Deferred event:   [Trigger '/defer']
```

Final State: A *State Machine* may optionally have a final state. This state must not have outgoing transitions. The final state does not mark the destruction of the object, but being in the final state implies never leaving it again. Figure 6.7 shows the graphical UML notation of a final state.



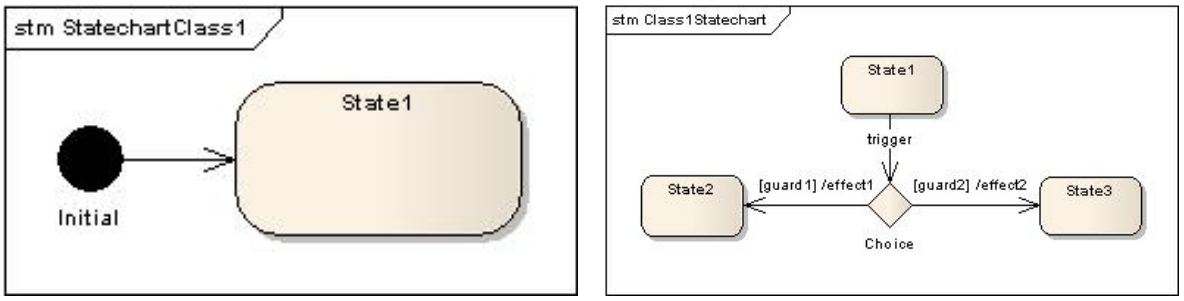
Final State Composite State

Figure 6.7: Final and Composite State

Composite State: States that contain other states are called composite states. They have the same features as simple states. There are sequential and concurrent composite states. A sequential composite state is like a simple state containing a *State Machine* within its boundaries. A state that contains more than one *State Machine* in different regions that are supposed to perform concurrently is called concurrent composite state. Composite states will not be considered in this application. Figure 6.7 shows the graphical notation of a concurrent composite state with two regions. The symbol in the lower right corner denotes a composite state.

Initial Pseudostate: Each *State Machine* must have an initial state. An initial state in UML is a pseudostate. A pseudostate does not have all features of a simple state, e.g. it may neither contain entry nor exit actions. A transition leaving an initial state must not have a trigger, but it can have a guard and an effect. Figure 6.8 shows the graphical notation of an initial pseudostate.

Choice Pseudostate: A choice pseudostate is like a decision point in the *State Machine Diagram*. If the same event contained in a trigger occurs more than once from one state but with different guards and actions, a choice pseudostate may be used to simplify the diagram. Figure 6.8 shows the graphical notation of a choice pseudostate.



Initial Pseudostate Choice Pseudostate

Figure 6.8: Initial and Choice Pseudostate

Transitions

A transition may have one or more triggers, a guard and an effect (see Figure 6.6). All of these properties are optional. This application allows at most one trigger associated to a transition. The syntax of a transition is (source [OMG, 2007b], page 574):

```
<transition> ::= [<trigger>[','<trigger>]* ['['<guard>']'] ['/'<effect>]]
```

Regarding Figure 6.4, each trigger is associated with exactly one event. The *guard* of the trigger is a boolean expression that must be true before the transition can be fired. The *effect* is the behavior of the transition and may be any subclass of *UML:: CommonBehaviors::BasicBehaviors::Behavior*. Only simple activities are allowed in this

application.

Trigger

UML defines a lot of events that can be associated with a trigger, not all of them are shown in Figure 6.5. Basically there are four kinds of events that are intended to be associated to triggers in *State Machine Diagrams*: call events, signal events, change events and time events.

- *Call Event*

  This event is basically a synchronous operation call of the current object from another object. A call event must be associated to a specific operation of the object. The syntax of a call event is defined as (source: [OMG, 2007b], page 425):

  ```
  <call-event> :: <name> ['(' [<assignment-specification>] ')']
  <assignment-specification> ::= <attr-name> [',' <attr-name>]*
  ```

  A call event consists of a *name* and an *assignment-specification*. The *name* is the name of the specified operation associated with the call event. The *assignment-specification* assigns the parameters of the operation defined by the *name* to an attribute that is defined by a name (*attr-name*). Since the *assignment-specification* is optional, it may be omitted even if the associated operation has parameters.

- *Signal Event*

  A signal event is an asynchronous event. It is associated with a signal and does not have a corresponding operation defined in any object. A signal is specified in the *Class Diagram* as a special class with the stereotype «signal»; its parameters are attributes of this class (no operations may be defined). A signal event is defined as (source: [OMG, 2007b], page 450):

  ```
  <signal-event> ::= <name> ['(' [<assignment-specification>] ')']
  <assignment-specification> ::= <attr-name> [','<attr-name>]*
  ```

  The elements of a signal event are quite the same as of a call event.

  Figure 6.9 shows the graphical notation of two signals defined in a *Class Diagram*.

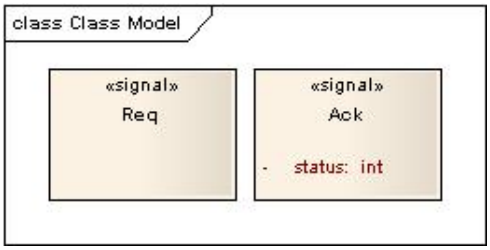


Figure 6.9: Two Signals

- *Change Event*

  A change event is triggered if a predefined condition becomes true, no operation or signal has to be considered. A condition could be a special assignment to one attribute of the class, e.g. attribute x becomes lower than zero. The keyword *when* specifies a change event (source: [OMG, 2007b], page 436):

  ```
  <change-event> ::= 'when' <expression>
  ```

  The change event is not considered in this application.

- *Time Event*

  This event deals with timing. There are relative and absolute time events. A transition of a relative time event is fired after a predefined amount of time, a transition of an absolute time event is fired at a certain point in time. The specification of this event is defined by (source: [OMG, 2007b], page 452):

  ```
  <time-event> ::= <relative-time-event> | <absolute-time-event>
  <relative-time-event> ::= 'after' <expression>
  <absolute-time-event> ::= 'at' <expression>
  ```

  The time event is not considered in this application.

Guard

As mentioned before, a guard is a condition which must be fulfilled at the time the event occurs. It is basically a boolean expression. If the condition of the guard is not true, the transition cannot be fired. Guards are optional. In UML, guards are constraints. Each constraint has an attribute called specification, which is a *ValueSpecification*. Figure 6.10 shows the meta model of the class *ValueSpecification*. There are a lot of subclasses generalized from *ValueSpecification*. In this application, the specification of a constraint of a guard is a *LiteralString*. The value of the *LiteralString* describes the boolean expression of the guard. This application restricts the values of the *LiteralString* to be valid OCL expressions.

Effect

As mentioned in Section 6.4.1, an effect is a behavior, thus it can be an *OpaqueBehavior*, *FunctionBehavior*, *State Machine*, *ProtocolStateMachine*, *Activity* or *Interaction*. In this application, only simple activities are allowed. A detailed description about activities that may be used in this application follows in Chapter 6.4.3.

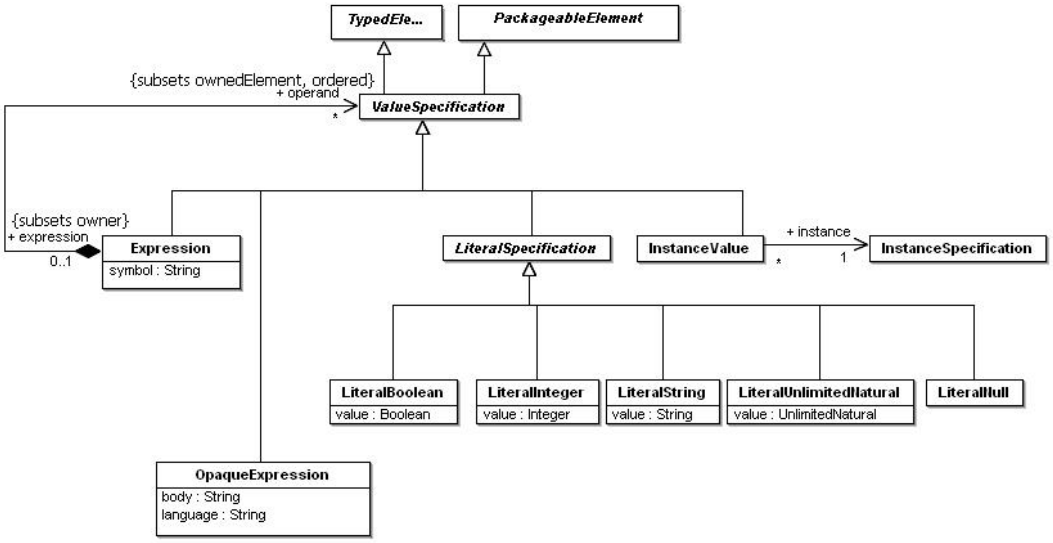
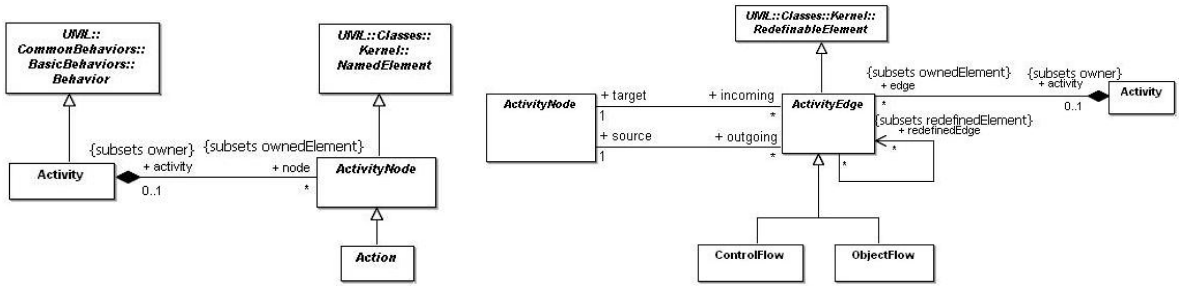


Figure 6.10: ValueSpecification (source: [OMG, 2007b], page 28)

6.4.3 Activity Diagram

This chapter deals with *Activities* in UML. Recall the *State Machine* meta model (see Figure 6.6). Each transition or state is associated with elements from the class *UML:: CommonBehaviors::BasicBehaviors::Behavior*. Subclasses of this class are *OpaqueBehavior*, *FunctionBehavior*, *Activity*, *Interaction*, *State Machine* and *Protocol State Machine*. In a *State Machine Diagram*, behaviors are used to describe the effect of a transition, entry actions, exit actions and do activities. This application only allows very simple *Activity Diagrams*.



Meta Model of Activity Nodes Meta Model of Activity Edges

Figure 6.11: Part of the Activity Meta Model (source: [OMG, 2007b], page 298 and 299)

Figure 6.11 shows the meta model of the basic elements used in *Activity Diagrams*. An *Activity Diagram* describes workflow; it describes the sequence of actions defined by an object. Each *Activity* contains several *ActivityNodes* and *ActivityEdges*. Nodes are connected by edges which describe the flow of the *Activity Diagram*. This application only allows very basic *Activity Diagrams*. Each diagram has an initial node and an activity final node. There are several other types of nodes like fork, join or merge nodes, but they

are not allowed in this application. Nodes can also be actions. Again UML defines a huge amount of actions. For this application, only three actions are permitted: call operation action, send signal action and value specification action. A call operation action may call any operation of any class in the *Class Diagram* (of course the objects have to share an association in the *Class Diagram* and the called operation of the associated object needs to be public). A send signal action is almost the same as a call operation action, it just sends signals except of calling a specific operation. The signal must be defined within the *Class Diagram*; it does not have to correspond to a specific operation. Both actions are subtypes of the class *InvocationAction* in the UML meta model. Value specification actions describe expressions. They are used to specify changes of the attributes of the object. They are not of the type *InvocationAction*, but they contain exactly one value of the type *UML::Classes::Kernel::ValueSpecification* (see Figure 6.10). There are different subtypes of *ValueSpecification*, like *LiteralString*, *LiteralInteger*, *Expression*... This application uses an *OpaqueExpression* to describe the value of the value specification action. An *OpaqueExpression* comprises an attribute called body, that contains the expressions. It is also possible to define the language used by the expressions in the language attribute. Note that it is not possible to have two consecutive value specification actions in an activity diagram in this application because all assignments can be done in one value specification action; there is no need to split it. Edges that connect the nodes may only be *ControlFlow* edges in this application (see Figure 6.11). They connect the edges in a straight line. Consequently, the *Activity Diagram* is just a sequence of the three permitted actions with no partition of the control flow. That means that each node must have exactly one incoming and one outgoing edge (except initial and final node).

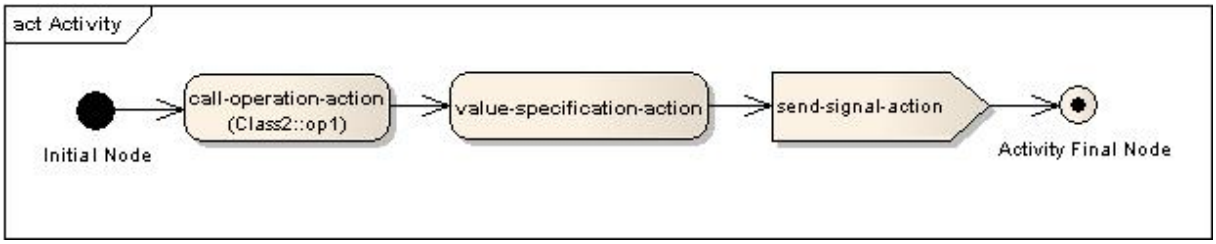


Figure 6.12: Example of an Activity Diagram

Figure 6.12 shows an example of an *Activity Diagram* allowed in the application. It is a simple control flow without branching. There are three actions called consecutively, one of each action type defined in this section.

6.5 Extensibility Mechanisms

UML provides three constructs that can be used for extension [Pender, 2003]: Stereotypes, tagged values and constraints.

6.5.1 Stereotypes

Stereotypes extend the semantic of the meta model although they must be based on existing types or classes. In a diagram, a stereotype is written between guillemots (≪...≫). UML provides some predefined stereotypes like ≪entity≫ or ≪subsystem≫, but it is also possible to define other stereotypes as needed, e.g. to indicate all user interface classes with a stereotype ≪userInterface≫.

6.5.2 Tagged Values

Tagged values are additional features that may be added in addition to those already defined in the meta model. They consist of a tag and an assigned value in the form *name=value*, in some diagram types they are also enclosed within curly braces. The author of a class for example may be included in a class diagram through a tagged value in the name compartment of a class by writing name="Christopher".

6.5.3 Constraints

A constraint defines a condition that must hold true for the duration of the context in which it is defined, e.g. a constraint of an attribute must hold true for the lifetime of an object whereas a constraint of an operation must only hold true for the duration of the operation. Constraints are placed within curly braces and can be defined in any given language. It is quite common to use OCL (see Chapter 6.6).

6.6 OCL

OCL is an extension to UML that can be used to describe constraints on UML models. It is a pure specification language which means it cannot change anything in the model. The state of a system cannot be altered through an OCL expression, though it can specify how the state of a system may change during execution. OCL just describes expressions, it is no programming language, thus it cannot be used to write program logic.

An OCL expression may be used for the following purposes [OMG, 2006b]:

- as a query language
- to specify invariants on classes and types in the class model
- to specify type invariant for stereotypes
- to describe pre- and post- conditions on operations and methods
- to describe guards

- to specify target (sets) for messages and actions
- to specify constraints on operations
- to specify derivation rules for attributes for any expression over a UML model

Some basic OCL constructs to describe these purposes are explained next.

Context: Each OCL expression is defined in a context. This can for example be a class or just a method of a class in a *Class Diagram*. It could also be a transition in a *State Machine Diagram*. The context is always defined at the beginning of an OCL expression. If the expression is defined in a diagram and the context is obvious, it can be omitted. The context declaration is optional. For example:

```
context c : className
```

defines an OCL expression for a class, whereas

```
context className::operationName(param1 : Type1, ...): ReturnType
```

defines an OCL constraint for an operation of a class.

Invariants: An invariant must always be true for all instances of the given type at any time. In OCL, this is done by the *inv* keyword. For example:

```
context c : className inv:
  c.var > 0
```

defines, that the variable in class c must always be positive.

Pre- and Postconditions: In OCL it is also possible to define pre- and postconditions on behavioral elements like operations. This is done by the *pre* and *post* keyword. For example:

```
context className::operationName(param1 : Type1, ...): ReturnType
pre : preconditionName: param1 > ...
post : postconditionName: result = ...
```

describes a pre- and a postcondition for the given operation. The reserved keyword *result* denotes the result of the operation. The names of the conditions are optional.

Chapter 7

Transformation from UML to IOSTS

Contents

7.1 Introduction

This chapter describes the basic transformation from UML to IOSTS. For this purpose, an extension to the UML model in terms of a stereotype has to be provided. This is explained in Section 7.2. Section 7.3 describes how the elements of the UML model (remember the UML model consists of *Class*, *State Machine* and *Activity Diagrams*) are transformed to elements in an IOSTS specification. A detailed pseudo code of the algorithm is presented in Section 7.4.

7.2 UML Extension

In an IOSTS, there is a distinction between input, output and internal actions. An IOSTS action corresponds to an operation or a signal in the UML model as explained in Section 7.3. However, UML does not really support this distinction. Thus, a mechanism is needed to provide this to the UML model. As explained in Section 6.5, it is possible to extend the UML model in terms of stereotypes. This application introduces an IO stereotype. This stereotype must be applied to all operations and signals of the given model. The value of the stereotype may be either «input», «output» or «internal».

7.3 Transformation of the Elements

This section describes how elements from the UML model are transformed to elements in the IOSTS. The notation of the IOSTS diagrams used in this section in depicted in Figure 7.1. States are identified by their name (<*state*>). A transition may have a guard (<*guard*>) which is illustrated with the keyword *if*. The action of a transition is identified with the keyword *sync* (*tau* denotes an internal action). Input actions are marked with *?*, output actions with *!* respectively. The keyword *do* designates the assignment block (an IOSTS transition may have more that one assignment, see Section 4.2). The assignment block is enclosed in curly brackets separated by |.

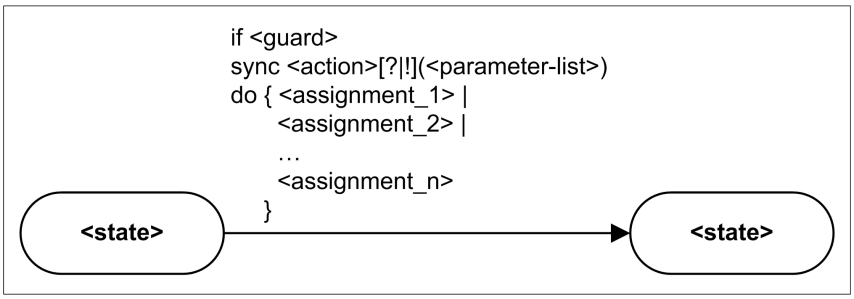


Figure 7.1: Notation of the IOSTS Diagrams

7.3.1 States

Four kinds of states and pseudostates respectively in UML *State Machines* are considered in the application.

- *Initial Pseudostate*

  An initial pseudostate in UML is transformed to an init state in IOSTS (see Figure 7.2).

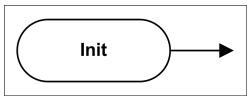


Figure 7.2: IOSTS Initial State

- *Final State*

  A final state in UML is transformed to a state in IOSTS that has no outgoing transitions (see Figure 7.3).

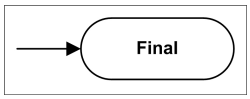


Figure 7.3: IOSTS Final State

- *Simple State*

  A simple state in UML is transformed to a state in IOSTS. Entry and exit actions, as well as do activities and deferred events are not allowed in this application.

- *Choice Pseudostate*

  A choice pseudostate is not transformed to a state in IOSTS. Each guard and effect of an outgoing transition of the choice pseudostate is joined with the trigger of the incoming transition and a new transition is created in IOSTS containing these elements. Consider the choice pseudostate of Figure 6.8, it is transformed to the IOSTS depicted in Figure 7.4.

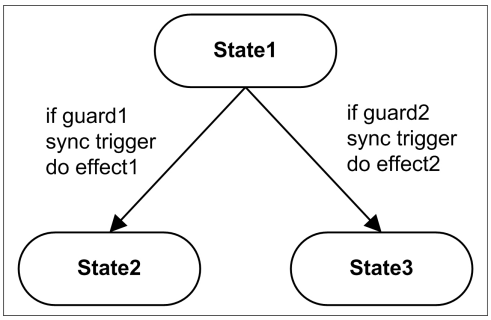


Figure 7.4: Transformation of an UML Choice Pseudostate to IOSTS

An incoming transition of a choice pseudostate must not have a guard or effect associated with it, whereas outgoing transitions are not allowed to contain a trigger.

7.3.2 Transitions

In an IOSTS each transition has either an input, output or internal action. In UML, a transition may have an trigger that contains either a call or a signal event and is so either associated with an operation or a signal. According to the stereotype of the operation/signal, the trigger is transformed to an input, output or internal action in the context of IOSTS. An effect is an activity. Activities may contain call operation or send signal actions (see Chapter 6.4.3) that are also associated with an operation or signal respectively. So these actions in an activity are also transformed to input, output or internal actions according to the stereotype of the operation/signal of the event. Activities may also comprise value specification actions. They are transformed to assignments in IOSTS. Each expression in the body of the value attribute of the value specification action (the value of the value specification action is actually an opaque expression) is added to the statement of the transition in the IOSTS. Guards in UML and IOSTS are quite the same, they are just added to the guard of the transition in IOSTS. Here is the syntax of a transition in STG defined in Appendix A:

```
<transition> := from <state-id>
                  [<guard>]
                  [<action>]
                  [<statement>]
                to <state-id>;
```

The next two sections describe how transitions are transformed when there is either a trigger or an effect associated to it. Afterwards, the transformation for the case that both, trigger and effect, are part of a transition is described.

Trigger

There are different types of events that can be associated to triggers; their transformation to IOSTS is described in this section. Call events and signal events are operation calls or signals from outside the object. They are transformed to actions (gates) in context of an IOSTS. The type of the action is determined through the stereotype of the operation/signal that is associated to the call/signal event of the trigger. Figure 7.5 shows the graphical notation of an UML trigger in a *State Machine Diagram*.

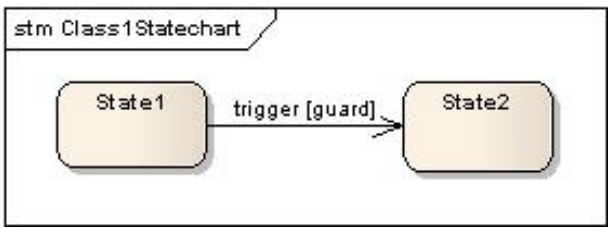


Figure 7.5: Call or Signal Event

- *Call Event*

  A trigger that contains a call event in UML is transformed to an action in IOSTS. The guard of the transition is also added to the transition in IOSTS. Figure 7.5 shows the graphical notation of a trigger. The event of the trigger is specified by its property *event* in UML.

  The transition in Figure 7.5 is transformed to the IOSTS depicted in Figure 7.6.

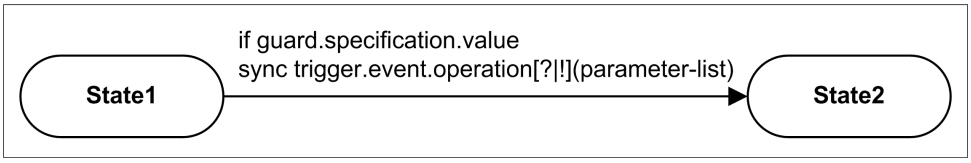


Figure 7.6: Transformation of an UML Call Event to IOSTS

- *Signal Event*

  Signal events are treated the same way as call events, except that the signal event is associated to a signal, not to an operation. Considering the trigger in Figure 7.5 contains a signal event, it is transformed to the IOSTS depicted in Figure 7.7.

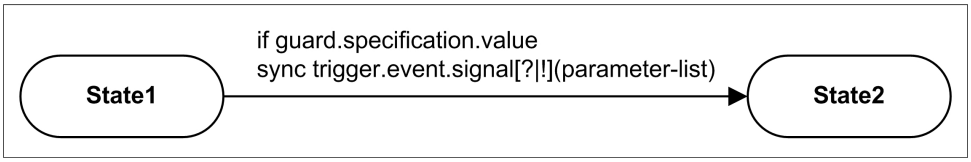


Figure 7.7: Transformation of an UML Signal Event to IOSTS

- *Change Event*

  Change events are not considered in this application.

- *Time Event*

  Time events are not considered in this application.

Guard

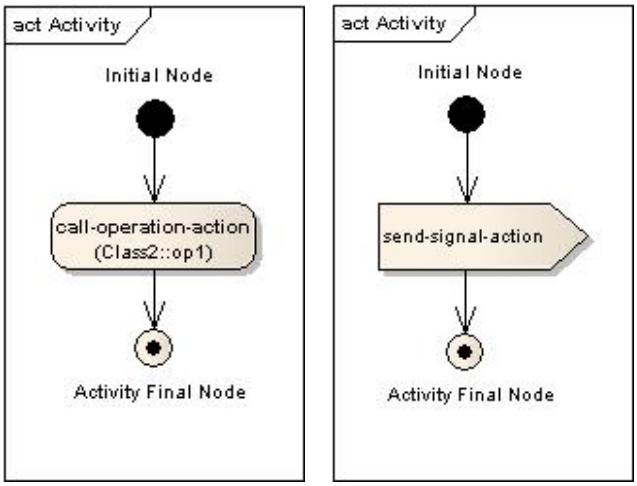
A guard in UML and IOSTS is quite the same. It is simply added to the guard of the transformed transition. If the transition has to be split in more than one transition, the guard is added to the first one. As mentioned in Chapter 6.4.2, the guard in UML is the value of a *LiteralString* which itself is the specification of a constraint (guard).

Effect

In this application effects are *Activities*. As explained in Section 6.4.3 an activity is a simple chain of three different kinds of actions. First of all, the transformation of each kind of action is described, that means activities with only one action (of course these *Activities* have an initial and an activity final node). The transformation of multiple actions is explained next.

- *Call Operation Action*

  A call operation action in an activity that describes an effect of a transition in a *State Machine Diagram* becomes an input, output or internal action in the context of IOSTS according to the stereotype of the associated operation. The operation of the call operation action is defined in the *operation* property of the action.



Call Operation Send Signal Action

Figure 7.8: Two different Actions

The transition in Figure 7.8 is transformed to the IOSTS depicted in Figure 7.9.

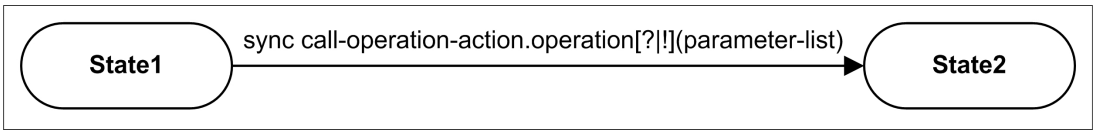


Figure 7.9: Transformation of an UML Call Operation Action to IOSTS

- *Send Signal Action*

  Send signal actions are treated the same way as call operation actions. The only difference is that the property that contains the signal is called *signal*, not *operation*.

  The transition in Figure 7.8 is transformed to the IOSTS depicted in Figure 7.10.

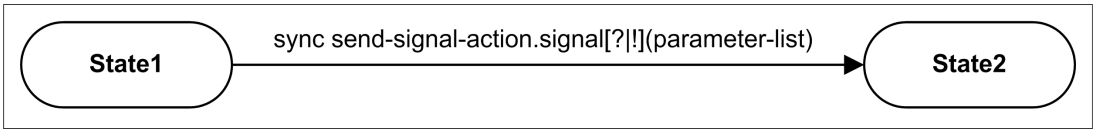


Figure 7.10: Transformation of an UML Send Signal Action to IOSTS

- *Value Specification Action*

  A value specification action contains a value that is a subtype of the class *ValueSpecification* (see Figure 6.10). In this application, only opaque expressions are allowed. The opaque expression contains a body attribute that consists of a list of expressions. Each of these expressions is added to the assignment in the context of IOSTS. If there is only one value specification action contained in an activity/effect of a transition in UML and the transition does not have a trigger, the action of the transition in IOSTS becomes the internal *tau* transition.

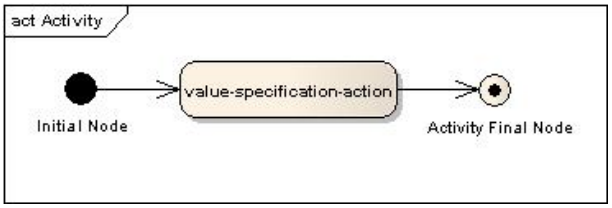


Figure 7.11: Value Specification Action

The transition of Figure 7.11 is transformed to the IOSTS depicted in Figure 7.12.

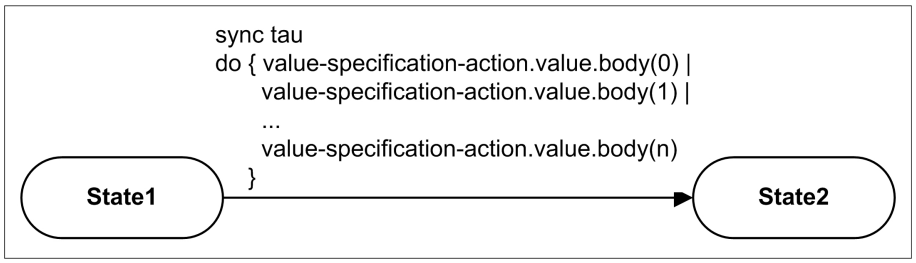


Figure 7.12: Transformation of a Value Specification Action to IOSTS

- *Multiple Actions*

  In UML it is possible to define the effect of a transition in terms of an *Activity Diagram*. Of course even the very simple *Activity Diagram* allowed in this application may have several actions. If there is more than one call operation or send signal action assigned to a transition, it has to be split in IOSTS since a transition in an IOSTS may only have one action assigned to it. The value of the value specification action is added to the transition that contains the preceding call operation or send signal action (if there is none, they are added to the first transition). If the transition has a guard, it is added to the first transition in IOSTS.

  Consider the *Activity Diagram* of Figure 7.13 describes the effect of a transition of a *State Machine*. The transition has no trigger and no guard associated with itself. The source of the transition is state *State1*, the target is state *State2*. It is transformed to the IOSTS depicted in Figure 7.14.

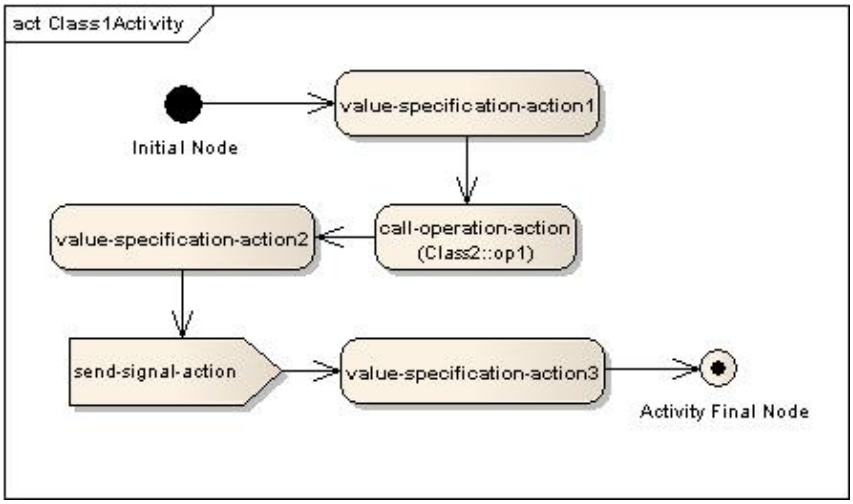


Figure 7.13: Multiple Actions

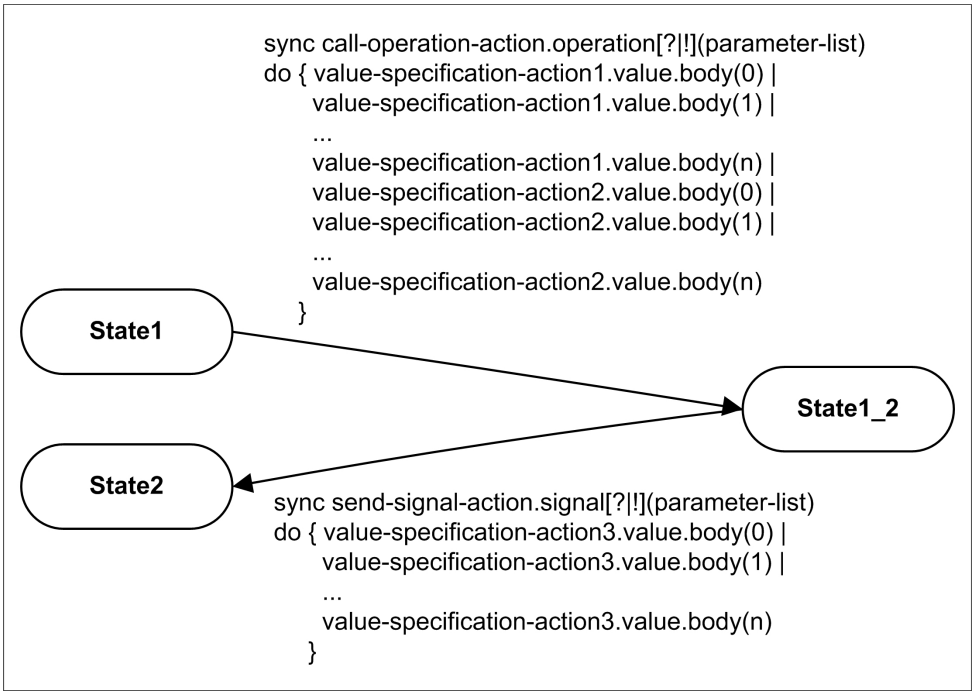


Figure 7.14: Transformation of Multiple Actions of an UML Effect to IOSTS

Trigger and Effect

Of course it is possible to have both, trigger and effect assigned to a transition in an UML *State Machine*. In that case the transition has to be split, except the effect of the transition is an activity that contains only one value specification action. Otherwise, the effect of the transition is an activity that contains any number of the three types of actions allowed in this application (see Chapter 6.4.3), but at least one call operation or send signal action. The guard of the transition in UML is always added to the transition in IOSTS that contains the event of the trigger as either input, output or internal action

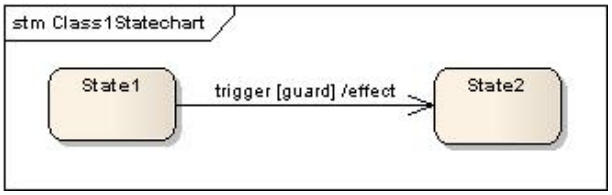


Figure 7.15: Trigger and Effect

according to the stereotype of the operation/signal assigned to the event of the trigger. The value of a value specification action is added to the transition that contains the preceding event or action.

Considering the effect of the transition in Figure 7.15 is defined by the *Activity Diagram* in Figure 7.13, the equivalent IOSTS representation looks like the IOSTS depicted in Figure 7.16:

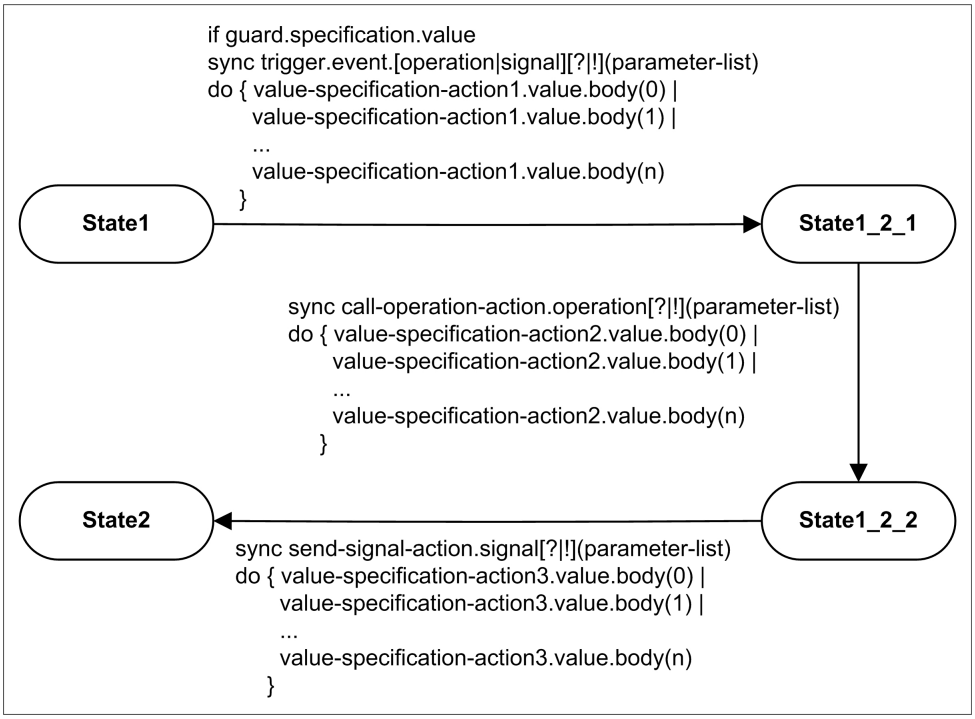


Figure 7.16: Transformation of an UML Transition with Trigger and Effect to IOSTS

7.4 Pseudo Code of the Algorithm

This section presents a pseudo code implementation of the transformation described in the previous chapter. As a matter of form, all UML elements are are named with the suffix *UML*, whereas all IOSTS elements carry the suffix *IOSTS*.

7.4.1 Function createModel(modelUML)

Function 1 *createModel(modelUML)* takes as argument an UML model *modelUML*. The model element is always the root element in an UML description, as it is in the IOSTS model. The function first sets the name of the IOSTS model element to the name used in the UML representation (Line 1). Then it iterates through all classes of the UML model. If there is a *State Machine* associated to a class (Line 3), it calls Function 2 *createSystem(classUML)* for the given class (Line 4).

Algorithm 1: createModel(modelUML)

input: UML model $modelUML$

```
1 name ← modelUML.name;
2 foreach Class classUML in modelUML do
3 |   if number of modelUML.class.stateMachine > 0 then
4 |   |   createSystem (classUML);
5 |   end
6 end
```

7.4.2 Function createSystem(classUML)

Function 2 *createSystem(classUML)* constructs an IOSTS system described in chapter 4.2. It first assigns an *id* to the overall system which is the name of the UML class. Then it initializes an empty set of gates because the gates of the system are determined later in the algorithm. After that, it calls Function 3 *createProcess(stateMachineUML)* for each *State Machine* the class contains.

Algorithm 2: createSystem(classUML)

input: Class $classUML$

```
1 id ← classUML.name;
2 Set of Gates gatesIOSTS ← {};
3 foreach State Machine stateMachineUML in classUML do
4 |   createProcess (stateMachineUML);
5 end
```

7.4.3 Function createProcess(stateMachineUML)

Function 3 *createProcess(stateMachineUML)* first assigns the name of the *State Machine* to the *id* attribute of the process. After that, a couple of sets are initialized according to the definition of the process in the IOSTS meta model introduced in section 8.5. The set of variables *varIOSTS* is initialized with the attributes of the class that are not constant (in UML, the *isReadOnly* attribute is set to false; the *owner* attribute of a *State Machine* is the *Class* that contains it). The set of parameters *parIOSTS* is initialized with the constant attributes of the class. The set of internal actions *internalIOSTS* is initialized with the default value *tau*. All other sets are yet empty, since their values are determined during the algorithm. After the initialization, Function 4 *createState(stateUML)* is called for each state of the State Machine (Lines 9-11) and Function 5 *createTransition(transitionUML)* is invoked for each transition (Lines 12-14).

Algorithm 3: createProcess(stateMachineUML)

input: StateMachine $stateMachineUML$

```
1  id ← stateMachineUML.name;
2  Set of Variables varIOSTS ← stateMachineUML.owner.attributes;
3  Set of Parameters parIOSTS ← stateMachineUML.owner.constantAttributes;
4  Set of InputActions inputIOSTS ← {};
5  Set of OutputActions outputIOSTS ← {};
6  Set of InternalActions internalIOSTS ← {tau};
7  Set of Transitions transitionsIOSTS ← {};
8  Set of States statesIOSTS ← {};
9  InitialState initStateIOSTS ← null;
10 foreach State stateUML in stateMachineUML do
11 |  createState (stateUML);
12 end
13 foreach Transition transitionUML in stateMachineUML do
14 |  createTransition (transitionUML);
15 end
```

7.4.4 Function createState(stateUML)

Function 4 *createState(stateUML)* takes as argument a state of the UML State Machine. First it checks the kind of the state. If the state is a choice pseudostate, no state in the IOSTS is created. Remember a choice pseudostate has to be transformed to at least two transitions, one for each outgoing transition of the choice pseudostate (it does not make sense to have a choice pseudostate with only one outgoing transition, since a normal transition can be used it that case). Each guard and effect of the outgoing transition is combined with the trigger of the incoming one. This is done in Function 10 *combineTransition(sourceUML, targetUML)* that is called in Line 3. The returned transition *tempT* is then processed in Function 5 *createTransition(transitionUML)* (Line 4). If the state is an initial pseudostate, an init state is created in IOSTS(Lines 6-9).

The name of the initial pseudostate in UML is assigned to the *id* of the init state of the IOSTS. At last the initial state variable *initStateIOSTS* of the process is set to the state *stateIOSTS*. Simple and final states are treated the same way as initial states, except a normal state in the IOSTS is created, no init state (Lines 10-14) and all of those states are added to the set *statesIOSTS* of states of the process.

Algorithm 4: createState(stateUML)

```
   input: State stateUML
 1 if stateUML is kind of ChoiceState then
 2 |   foreach outgoing Transition oT in stateUML do
 3 |   |   Transition tempT ← combineTransition (stateUML.incoming, oT);
 4 |   |   createTransition (tempT);
 5 |   end
 6 else if stateUML is kind of InitialState then
 7 |   State stateIOSTS ← new init State;
 8 |   stateIOSTS.id ← stateUML.name;
 9 |   initStateIOSTS ← stateIOSTS;
10 else
11 |   State stateIOSTS ← new State;
12 |   stateIOSTS.id ← stateUML.name;
13 |   statesIOSTS.add (stateIOSTS);
14 end
```

7.4.5 Function createTransition(transitionUML)

Function 5 *createTransition(transitionUML)* takes as argument a transition of the UML model and processes it in order to generate transitions in the IOSTS. First it checks whether the transition's source or target is a choice state (Line 1). These transitions are treated in Function 4 *createState(stateUML)*. After that, the transition is passed to Function 6 *splitTransition(transitionUML)* which returns a set of transitions (Line 2). Recall that a transition of the UML model may result in multiple transitions in the IOSTS. Then, all returned transitions are added to the set *transitionsIOSTS* of transitions of the process (Lines 3-5).

7.4.6 Function splitTransition(transitionUML)

Function 6 and 7 *splitTransition(transitionUML)* describe the most complex function of the algorithm. It takes as argument an UML transition and creates one or more IOSTS transitions.

Lines 1, 2 and 3 initialize three variables, one that contains the set of created IOSTS transitions (*setTransIOSTS*), another that comprises the information about the current IOSTS transition being processed (*tempT*) and one boolean variable, that decides whether the transition has to be split or not.

Algorithm 5: createTransition(transitionUML)

input: Transition *transitionUML*

```
1 if transitionUML.source is not kind of ChoiceState and transitionUML.target is
  not kind of CoiceState then
2 |  Set of Transitions setTransIOSTS ← splitTransition (transitionUML);
3 |  foreach Transition t in setTransIOSTS do
4 |  |  transitionsIOSTS.add (t);
5 |  end
6 end
```

After that (Line 4) an empty set of actions is introduced (*setActions*). A transition in UML may have an effect. An effect is a behavior. This application only allows simple *Activities* (see Section 6.4.3). In Line 5 all actions of the effect are appended to the set *setActions* of actions (they are ordered the same way as they appear in the activity from the initial to the activity final node). Function 8 *processBehavior(activityUML)* is called for the effect, which takes as argument an activity and returns a set of actions.

Variable *tempT* is a placeholder for an IOSTS transition. Here is the definition of a transition given in Appendix A:

```
<transition> := from <state-id>
                  [<guard>]
                  [<action>]
                  [<statement>]
                to <state-id>;
```

There are five properties an IOSTS transition may contain: *fromState* and *toState* are mandatory, *guard*, *action* and *statement* are optional. First of all, the action of the transition *tempT* is determined. As mentioned in Section 7.3, an IOSTS action can arise either from call/signal events associated to a trigger or from call operation/send signal actions contained in an effect that is an *Activity* in this application. Since an IOSTS transition may only contain one action, the UML transition has to be split if there is more than one of the elements mentioned above assigned to it. The expressions of the body of the value of a value specification action are added to the statement of the IOSTS transition.

The first *if* instruction (Line 6) checks whether there is a trigger assigned to the transition. If not, the algorithm checks if there is an effect (Line 7). Line 8 and 9 handle the case that neither a trigger, nor an effect is contained in the UML transition. Thus, the action of the IOSTS transition is the internal *tau* transition which is saved to variable *a*.

If there is an effect (no trigger, Line 10) the set *setActions* that contains all actions of the activity is not empty, it must contain at least one call operation, send signal or value specification action. So the first call operation or send signal action is saved in variable *a* (Line 11). Line 12 checks whether *a* is not null. Function 9 *processStereotype(behavorialFeatureUML)* is called in Line 13 and passes the operation or signal of the

Algorithm 6: splitTransition(transitionUML)

```
   input: Transition transitionUML
   output: Set of Transitions setTransIOSTS
 1 Set of Transitions setTransIOSTS ← {};
 2 Transition tempT ← new IOSTS Transition;
 3 split ← false;
 4 Set of Actions setActions ← {};
 5 setActions.append (processBehavior (transitionUML.effect));
 6 if transitionUML.trigger == null then
 7 |   if transitionUML.effect == null then
 8 |   |   Action a ← tau;
 9 |   |   tempT.action ← a;
10 |   else
11 |   |   Action a ← first Action not kind of Value Specification in setActions;
12 |   |   if a != null then
13 |   |   |   processStereotype (a.[operation|signal]);
14 |   |   |   tempT.action ← a.[operation|signal];
15 |   |   |   setActions.remove (a);
16 |   |   else
17 |   |   |   Action a ← tau;
18 |   |   |   tempT.action ← tau;
19 |   |   end
20 |   end
21 else
22 |   Event a ← transitionUML.trigger.event;
23 |   processStereotype (a.[operation|signal]);
24 |   tempT.action ← a.[operation|signal];
25 end
26 if number of Actions not kind of Value Specification in setActions >= 1 then
27 |   split ← true;
28 end
29 gatesIOSTS.add (a.[operation|signal]);
30 tempT.fromState ← transitionUML.source;
31 tempT.guard ← transitionUML.guard.specification.value;
32 ...
```

action (operations and signals are *behavorialFeatures* in UML). It just determines if the operation/signal is transformed to an input, output or internal gate of the IOSTS. After that (Line 14), the action attribute of variable *tempT* is set to the operation/signal of the action. Action *a* can be removed from the set *setActions* (Line 15) since it has already been processed. Now it is possible, that *setActions* only contains value specifications actions. Since, in that case, the transition does not have to be split (there is no trigger,

only one call operation or send signal action), these actions are treated in Lines 23-27 in Function 7.

Lines 17 and 18 handle the case, the transition has no trigger, but an effect that only consists of one value specification action (if variable *a* in Line 11 is null, there is no call operation or send signal action contained in the activity and this application does not allow two consecutive value specification actions, as described in chapter 6.4.3). This value specification action is also treated in Lines 23-27 in Function 7. The action of *tempT* is the *tau* transition, since there is neither a trigger, nor a call operation or send signal action contained in the effect.

If the transition has a trigger (Line 21), the event of the trigger is saved in variable *a*. The operation/signal (the event can be a call event containing an operation or a signal event containing a signal) of the event is processed in Function 9 *processStereotype(behavorialFeatureUML)*, just like the operation/signal of the call operation/send signal action in Line 13. Then (Line 24) the attribute action of *tempT* is set to the operation/signal of the event. Again it is possible that there is an effect that only contains one value specification action, but since the transition does not have to be split in that case, this is treated as well in Lines 23-27 in Function 7.

Now *tempT* has either an input, output or internal (*tau*) action assigned to it. If the set *setActions* contains another action that is not kind of a value specification action, the transition has to be split since it can only have one action assigned to it. This is shown in Lines 26-28.

The operation/signal of the action/event *a* that has been assigned to *tempT* must also be added to the set *gatesIOSTS* of gates of the system (Line 29). Then, two properties of *tempT* are assigned, the source of the transition and the guard. The function is continued in Function 7 *splitTransition(transitionUML) ...continue*.

If the transition has to be split (Line 2), all remaining actions of the set *setActions* are checked successively (Line 3). If the current action is a value specification action, all expressions of the body of the value of the value specification action are added to the statement property of *tempT* (Lines 4-7). All other kinds of action (and there is at least one) start the splitting (Lines 8-18). First, a new intermediate state *iState* is introduced. The property *toState* of *tempT* is set to it. Now *tempT* is complete, i.e. all needed properties are set, so it can be added to the set *setTransIOSTS* of transitions (remember this set is the return value of the function). In order to save the values of the next transition in the variable *tempT* it has to be reset (Line 12). Function 9 *processStereotype(behavorialFeatureUML)* is called to determine if the action is added to the input, output or internal actions of the process (Line 13). Then, the action of *tempT* is set to the operation/signal of the action. It also has to be added to the set *gatesIOSTS* of gates of the system (Line 15). The origin (attribute *fromState* of *tempT*) is set to the previously created intermediate state *iState* which is also added to the set *statesIOSTS* of states of the process (Lines 16-17). This is done as long as there are actions left in the set *setActions*. If there are no more actions left, the destination of *tempT* is set to the destination of the original UML transition and *tempT* is added to the set *setTransIOSTS* of transitions (Lines 20-21).

If the transition does not have to be split (Lines 22-30), there are only value specification actions left in the set *setActions*; it is also possible that the set *setActions* is empty,

Algorithm 7: splitTransition(transitionUML) ...continue

```
1  ...
2  if split then
3      foreach Action act in setActions do
4          if act is kind of Value Specification then
5              foreach String s in vsAction.value.body do
6                  tempT.statement.add (s);
7              end
8          else
9              Create new Intermediate State iState;
10             tempT.toState ← iState;
11             setTransIOSTS.add (tempT);
12             tempT ← new IOSTS Transition;
13             processStereotype (act.[operation|signal]);
14             tempT.action ← act.[operation|signal];
15             gatesIOSTS.add (act);
16             tempT.fromState ← iState;
17             statesIOSTS.add (iState);
18         end
19     end
20     tempT.toState ← transitionUML.target;
21     setTransIOSTS.add (tempT);
22 else
23     foreach Value Specification Actions vsAction left in setActions do
24         foreach String s in vsAction.value.body do
25             tempT.statement.add (s);
26         end
27     end
28     tempT.toState ← transitionUML.target;
29     setTransIOSTS.add (tempT);
30 end
31 return setTransIOSTS;
```

if the UML transition does not have an effect. For each value specification action left in *setActions* (there may be at most two: this is the case when the transition has no trigger and the sequence of actions in the effect looks like: value specification action → call operation/send signal action → value specification action; if so, the call operation/send signal action is the action of the IOSTS transition and is then removed in Line 15 of Function 6. The two value specification actions remain in the set *setActions*), all expressions of the body of the value of the value specification actions are added to the statement property of *tempT* (Lines 23-27). After that, the target of *tempT* is set to the target of the UML transition (Line 28). Now, all required values of *tempT* are set, so it can be added to the set *setTransIOSTS* (in that case, the set *setTrans* contains only one transition). The set

setTransIOSTS is then returned in Line 29.

7.4.7 Function processBehavior(activityUML)

Function 8 *processBehavior(activityUML)* is called by Function 6 *splitTransition (transitionUML)*. It takes as argument an activity *activityUML* and returns a set of actions. First it introduces an empty set of actions *setActions* (Line 1). Then it iterates sequential from the initial to the activity final node (Lines 2-7) of *activityUML*. Remember all nodes (except initial and final) must have exactly one incoming and one outgoing edge. So, all nodes (nodes are actions) between the initial and the activity final one are added to the set *setActions* of actions. This set is then returned.

Algorithm 8: processBehavior(activityUML)

input: UML Behavior $activityUML$
output: Set of Actions $setActions$

```
1 Set of Actions setActions ← {};
2 ActivityNode an = activityUML.initialNode;
3 while an.outgoing.target != ActivityFinalNode do
4 |   an ← an.outgoing.target;
5 |   setActions.add (an);
6 end
7 return setActions;
```

7.4.8 Function processStereotype(behavorialFeatureUML)

Function 9 *processStereotype(behavorialFeatureUML)* takes as argument a behavorial feature of UML. This may either be an operation or a signal. Recall Section 7.2, each operation and signal must have a stereotype assigned to it. So this function simply reads the stereotype, that may either be *input*, *output* or *internal* and then adds the operation/signal to the set of input *inputIOSTS*, output *outputIOSTS* or internal *internalIOSTS* actions of the process.

Algorithm 9: processStereotype(behavorialFeatureUML))

input: UML Behavorial Feature (Operation or Signal) $bfUML$

```
1 if bfUML.IO_Stereotype.equals ("input") then
2 |   inputIOSTS.add (bfUML);
3 else if bfUML.IO_Stereotype.equals ("output") then
4 |   outputIOSTS.add (bfUML);
5 else
6 |   internalIOSTS.add (bfUML);
7 end
```

7.4.9 Function combineTransition(incomingUML, outgoingUML)

Function 10 *combineTransition(incomingUML, outgoingUML)* is called by Function 4 *createState(stateUML)* in order to combine the incoming and outgoing transitions of a choice pseudostate. The new transition has the source of the incoming transition, as well as its trigger. The target, guard and effect are taken from the outgoing transition.

Algorithm 10: combineTransition(incomingUML, outgoingUML)

input: UML Transition $incomingUML$, UML Transition $outgoingUML$
output: UML Transition *combined*

```
1 Transition combined ← new UML Transition;
2 combined.source ← incomingUML.source;
3 combined.trigger ← incomingUML.trigger;
4 combined.target ← outgoingUML.source;
5 combined.guard ← outgoingUML.guard;
6 combined.effect ← outgoingUML.effect;
7 return combined;
```

Chapter 8

Implementation

Contents

8.1 Introduction

This chapter describes details about the implementation of the transformation. Section 8.2 describes the platform on which the transformation is implemented, i.e. the eclipse platform. There is a short description of two eclipse plugins used in the transformation process from UML to IOSTS, the eclipse modeling framework and the openArchitectureWare plugin. Section 8.3 illustrates the overall test case generation workflow from generating UML models to obtaining executable test cases using the STG tool. The workflow of the IOSTS generation step in the test case generation workflow is described in detail in Section 8.4. Section 8.5 presents the meta model of the IOSTS models used by this application. Finally, Section 8.6 comprises some details about the implementation, some issues and assumptions that have to be made in this application.

8.2 Eclipse Environment

The transformation is implemented in the eclipse environment. Reasons for this are the good model engineering tool support as well as the framework used in this application is open source. Following plugins are needed:

- *Eclipse Modeling Framework (EMF)* [Eclipse, 2008a]: this is the basic plugin for model engineering with eclipse
- *Eclipse Model Developing Tools (MDT)* [Eclipse, 2008c]: this plugin contains the UML2 and the OCL plugin
- *openArchitectureWare (oAW)* [oAW, 2008]: this plugin contains capabilities for model to model and model to text transformation

8.2.1 Eclipse Modeling Framework

The application is implemented with the help of the eclipse modeling framework. This framework offers the basic components for model engineering. One essential possibility is the definition of meta models by means of Ecore. All meta models in the eclipse modeling framework are defined by Ecore which itself is based on EMOF (Essential Meta Object Facility). EMOF is basically the meta meta model that defines elements that may be used by meta models. EMOF itself is a subset of MOF [OMG, 2004]. Actually the EMOF and Ecore specification are not exactly the same, there are a few small differences (mostly naming differences). Ecore is very simple, there are basically only classes allowed which may contain attributes and operations. There are three types of links between classes, generalization, association and aggregation. Generalization is the same concept as in almost every object oriented programming language. Association is a reference to another class. Aggregation is stronger than association, the referenced object must not live without the object containing the aggregation. Packages and enumerations are also part of the Ecore specification. Figure 8.1 shows the class hierarchy of the elements that may be used in Ecore (gray boxes are abstract classes).

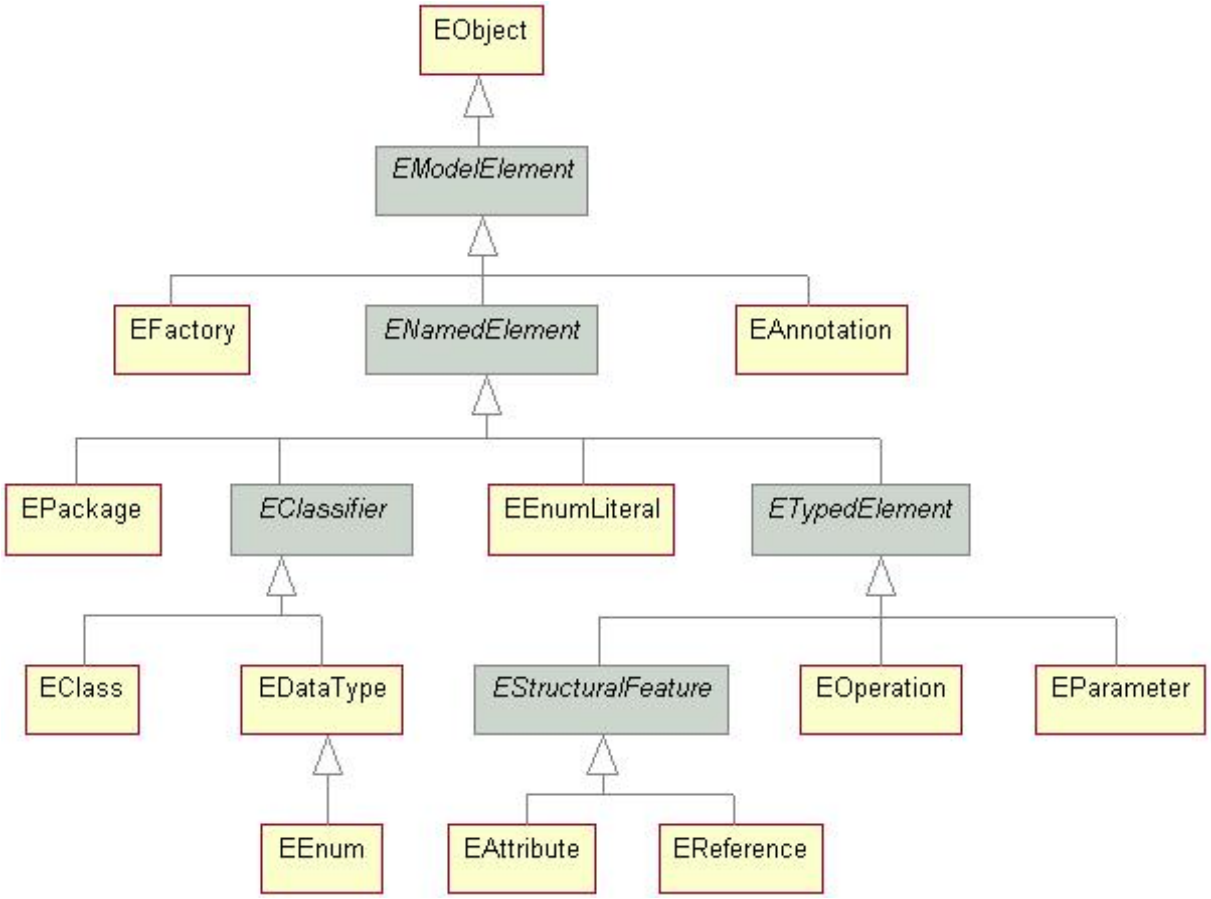


Figure 8.1: Ecore Elements

8.2.2 openArchitectureWare (oAW)

On top of the eclipse modeling framework, the openArchitectureWare plugin provides a whole model engineering framework, consisting of facilities for model to model and model to text transformation. It also offers possibilities for constraint checking on models. The UML2 implementation of the eclipse framework is also totally supported.

oAW introduces three languages, *Check*, *Xtend*, *Xpand2*. Each of these languages is based on the same type system and expression language, though they are used for different purposes:

- *Check*: this language is used for model validation. It is possible to define constraints that are checked against an arbitrary model. *Check* is feasible to define constraints on UML2 and Ecore models. *Check* files have the file extension *.chk*.
- *Xtend*: this language can be used for model to model transformations. It is also possible to define arbitrary operations based on oAW expressions. Another feature of *Xtend* is the possibility to call static methods written in *JAVA*. Thus it is possible to define *JAVA* extensions and call them directly from *Xtend*. *Xtend* files have the file extension *.ext*.

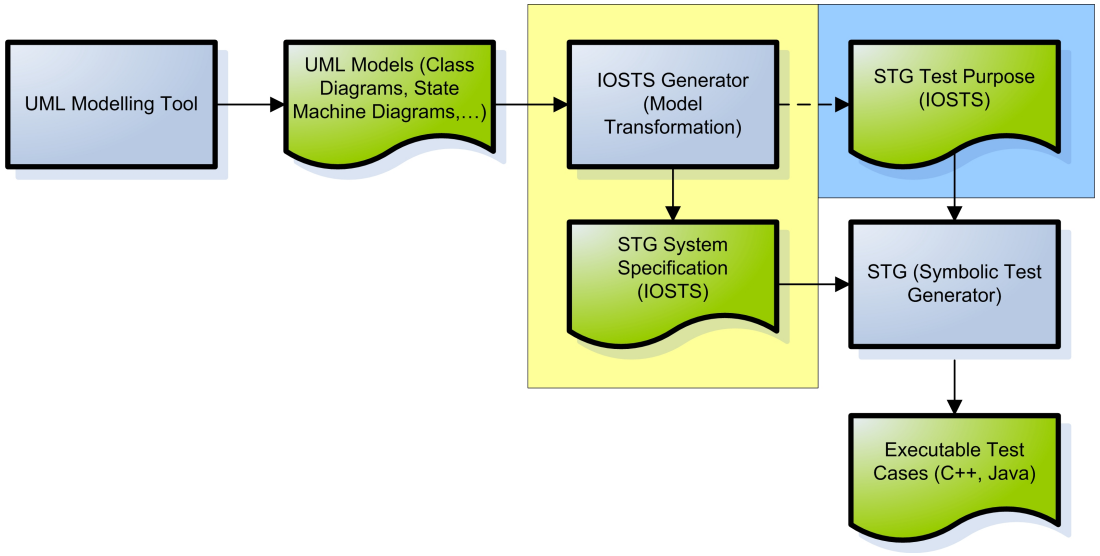


Figure 8.2: Test Case Generation Workflow

- *Xpand2*: this language is used for model to text transformations. *Xpand2* files have the file extension *.xpt*.

The oAW framework offers a lot more facilities, such as aspect oriented templating and text to model transformation but these components are not used in this application.

8.3 Test Case Generation Workflow

Figure 8.2 shows the overall test case generation process for generating executable test cases from UML models. First, an UML model containing *Class* and *State Machine Diagrams* has to be constructed with an UML modeling tool. This implementation uses the UML2 plugin of the eclipse platform for constructing these UML models. The generation of UML models with this plugin is not very user-friendly. It is used because the whole implementation of the transformation is based on the eclipse modeling framework. Of course the model has to conform to the constraints described in Chapter 6. After that, the model is passed to the IOSTS Generator. There the actual transformation takes place. The generator takes as argument an UML model and produces one or more IOSTS system specifications that conform to the STG language specification described in Chapter 5.4. This specification is then passed to the STG tool. Together with an IOSTS test purpose the STG tool is able to generate executable test cases. Actually the STG tool does not generate the executable test cases directly. It first generates symbolic test cases that are then transformed to executable test cases using a test driver. Generating the test purpose is not scope of the application, only the transformation from UML to an IOSTS system specification is taken into account (yellow frame in Figure 8.2). Generating the test purpose as well as defining an UML representation for it is not scope of this application, this transformation could be topic of some future work (blue frame in Figure 8.2).

8.4 IOSTS Generator

This chapter takes a closer look at the IOSTS generator where the actual transformation takes place. Figure 8.3 shows the workflow of the generator. The whole transformation is implemented with the openArchitectureWare plugin described in Section 8.2.2. The transformation consists of two major steps:

- the transformation from UML to IOSTS (model to model transformation)
- the transformation from IOSTS to STG (model to text transformation)

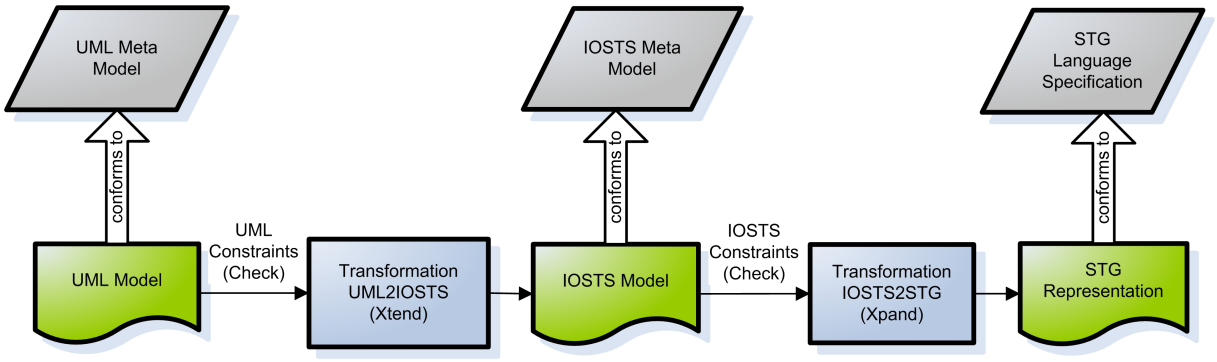


Figure 8.3: Transformation Workflow

First of all an UML model is needed. This model has to conform to the eclipse implementation of the UML2 meta model and to the restrictions described in Chapter 6. Basically the model can be created by any arbitrary UML tool, it just needs to be exported to the eclipse UML2 representation. Then, the UML model is passed to the UML2IOSTS transformator. This transformation is implemented in *Xtend*.

The UML2IOSTS transformator produces an IOSTS model that conforms to the IOSTS meta model explained in Section 8.5. Defining this meta model is also part of this thesis. After that, the IOSTS model is passed to the IOSTS2STG transformator. This transformation is the model to text transformation implemented in *Xpand2* that produces one or more IOSTS that conform to the STG specification language shown in Chapter 5.4. The top level element of the IOSTS model is *Model*, as it is in an UML model. Since an IOSTS model can have more than one system and each system becomes an IOSTS specification in STG, it is possible to obtain more than one IOSTS specification from only one UML/IOSTS model.

As mentioned in Section 8.2.2, the oAW framework offers possibilities to validate both, UML and Ecore models. This is done in *Check* components. In this application, there are two validation components that are called before each of the two transformations takes place. The UML constraint check validates the UML model that has to be transformed. Remember that there are a couple of restrictions introduced in Chapter 6. There are almost hundred constraints defined which check whether the passed UML model fulfills these restrictions. There are also some IOSTS constraints, but not so many. There are basically only checks, if the types of the parameters or variables are either *int* or *bool*,

since these are the only datatypes the STG tool can handle. The transformation can only take place if all constraints validate to true.

The whole workflow depicted in Figure 8.3 is contained in an openArchitectureWare workflow file (extension *.oaw*). This is an *XML* based workflow description which makes it very easy to modify the workflow to e.g. change the UML model or add remove constraint files etc.

8.5 IOSTS Meta Model

As depicted in Figure 8.3, the transformation from UML to IOSTS needs an IOSTS meta model in order to perform the model to model transformation. It would also be possible to skip the model to model transformation and define the model to text transformation directly on the UML model, but having the model to model transformation step in between has some benefits:

- *extensibility*: the IOSTS meta model does not depend on the tool that uses the IOSTS model such as STG in this case. Thus, it is possible to define another model to text transformation to transform the IOSTS model into a language another tool can understand. The model to model transformation remains the same, only the model to text transformation has to be changed (and of course the constraints that rely on the underlying tool, the STG tool in this application).
- *restrictions*: The STG tool has a lot of restrictions, e.g. the type of a variable or parameter may only be *int* or *bool*. The IOSTS meta model does not have these restrictions. It is better to define the constraints that have to be made according to the tool that uses the IOSTS model as late as possible, in this case before the model to text transformation.

The IOSTS meta model is depicted in Figure 8.4. There are a couple of datatypes defined in the meta model. A *Type* has three attributes: *type*, *upper* and *lower*. The *type* attribute consists of a *TypeValue*, which is an enumeration of the possible datatypes allowed in an IOSTS model that conforms to this meta model. These are: *int*, *bool*, *float*, *string* and *char*. As mentioned before, the STG tool can only handle *int* and *bool* datatypes, but since the IOSTS meta model is more general, most of the basic datatypes are taken into account. The *Type* class also has an upper and lower attribute. These are integer values, that define the *upper* and *lower* bound of the *Type*. These attributes are used to define arrays or sets of *Types* (of course the lower bound has to be smaller than the upper bound), sets of infinite length are defined using the value *-1* for the *upper* attribute. There are two subclasses of *Type*, *UnnamedType* and *NamedType*. A *NamedType* is also generalized from *NamedElement*, so it also has an attribute called *name*. *Type* and *NamedType* are abstract.

The IOSTS meta model contains the elements *Model*, *System* and *Process*. These elements are not defined in the IOSTS definition (see Section 4.2), but they are needed because an UML model can produce more that one IOSTS specification (see Section 7.4). These specifications should all be contained in one model (as it is in UML), so there have to be some container elements that comprise these IOSTS. The *Process* is the actual

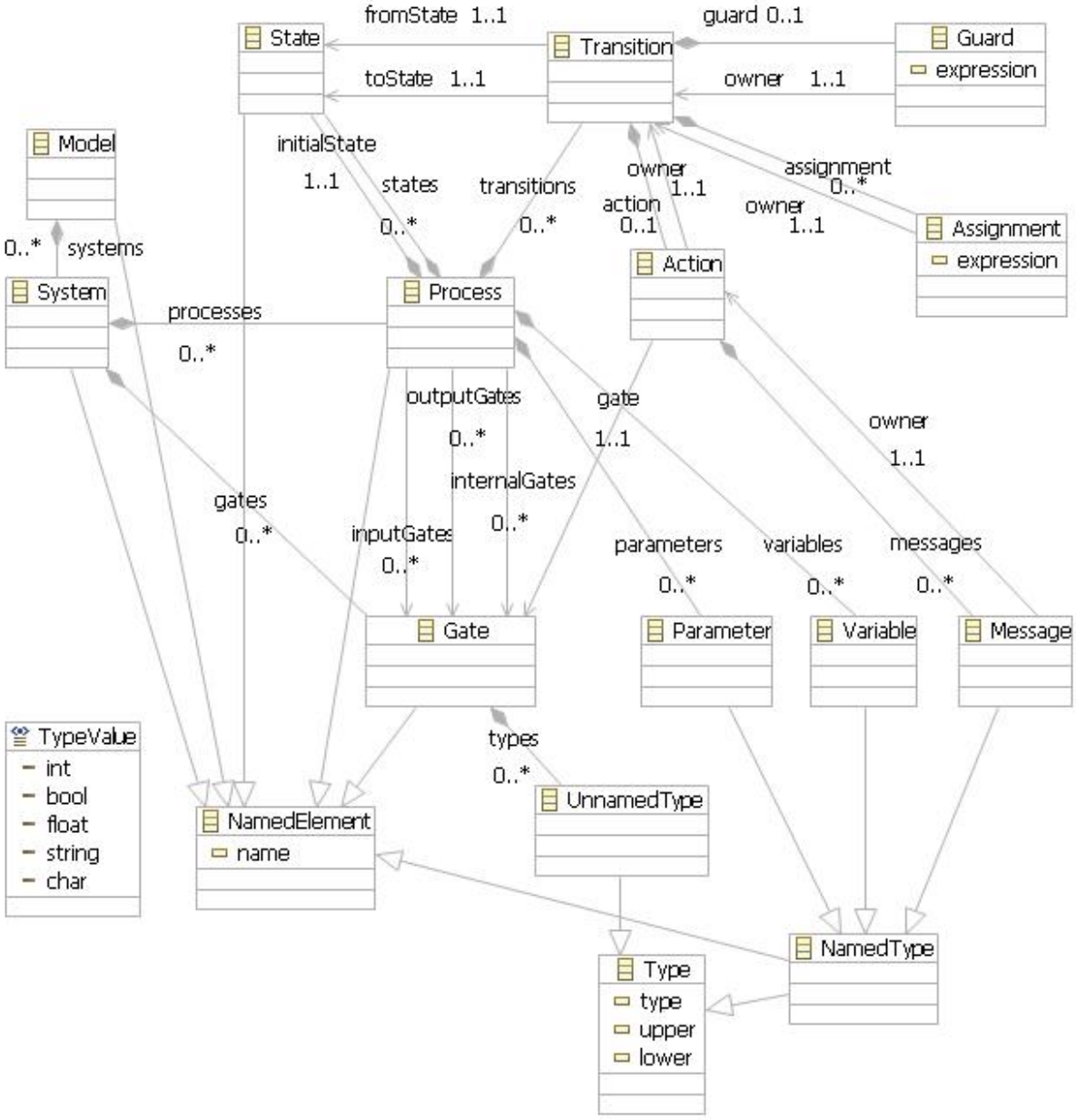


Figure 8.4: IOSTS Meta Model

IOSTS, IOSTS specifications that belong together are maintained in a *System* (e.g. an IOSTS system specification and an IOSTS test purpose).

The top level element of an IOSTS is *Model*. It is generalized from *NamedElement*. *NamedElements* have an attribute called *name*. So each IOSTS model has a name. *NamedElement* is abstract. A *Model* has null or more *Systems* aggregated to it. A *System* is also generalized from *NamedElement*. The name of the *System* is the *system-id*. A *System* consists of *Gates* and *Processes*.

A *Gate* is basically an operation or signal in UML. Each *Gate* has null or more *UnnamedTypes*. These are the types of the parameters of the operation and the types of the attributes of the signal respectively. A *Gate* is also generalized from *NamedElement*, so

it contains a name which is in principle the name of the operation/signal of the transformed UML model.

Null or more *Processes* are part of a *System*. A *Process* consists of a disjoint set of *internalGates*, *inputGates* and *outputGates*. The *Process* only references the *Gates* (association); it may only reference *Gates* that are contained in the owning *System*. A *Process* also contains a couple of *Variables* and *Parameters*. They are generalized from *NamedType* and have no more attributes.

Each *State Machine* of an UML model becomes a *Process* in an IOSTS model, so the *Process* also consists of *States* and *Transitions*. A *State* is generalized from *NamedElement*, so it also has a name. A *Process* has exactly one *initialState* and null or more *states*.

A *Transition* has two associations to the class *State*. It has exactly one *fromState*, the origin of the *Transition* and exactly one *toState*, the target of the *Transition*.

A *Transition* may also have a *Guard*. A *Guard* has an *expression* attribute which is a string value containing the boolean expression of the *Guard*.

There are also null or more *Assignments* contained in a *Transition*. An *Assignment*, like a *Guard*, consists of an attribute called *expression*, but unlike the *Guard*, this *expression* is not a boolean value but an update mapping, an assignment of a *Variable* of the underlying *Process*.

A Transition has at most one *Action* as well. An *Action* is always associated to exactly one *Gate*. This *Gate* must be contained in the *System* that contains *Action*. The *Action* also comprises null or more *Messages*. A *Message* is generalized from *NamedElement*. The number of *Messages* and their type, as well as their *upper* and *lower* value must be exactly the same as defined in the set of *UnnamedTypes* of the associated *Gate*.

The classes *Guard*, *Assignment* ans *Action* also have an association to their owning *Transition* called *owner*. This is due to implementation issues (see Section 8.6.3).

8.6 Implementation Details

This section gives a closer look at some implementation details and issues.

8.6.1 Parameters

In an IOSTS, the action of a transition relates to an operation or signal in UML. In a *State Machine*, the operation/signal is referenced by a call/signal event of a trigger. In the UML Superstructure, the notation of a call/signal event is denoted as (source [OMG, 2007b], page 425 and 450):

```
<call|signal-event> :: <name> ['(' [<assignment-specification>] ')']
<assignment-specification> ::= <attr-name> [',' <attr-name>]*
```

The *assignment-specification* contains the names of the parameters passed to the operation defined in the *name*. The *assignment-specification* is optional, even if the associated operation has parameters. Unfortunately there is no possibility to define the *assignment-specification* with the UML2 plugin of eclipse. Hence the application needs an *assignment-specification* since the action in an IOSTS contains messages that are *NamedTypes* (see

Section 8.5). So the names of the parameters are passed in the property name of the call/signal event. The name of an event must be an arbitrary event name, followed by a comma separated list of parameter names enclosed in parenthesis. For example:

```
callEvent(param1, param2)
```

The UML check component of the transformation workflow (see Figure 8.3) checks the correctness of the name of the events. It first checks if the number of parameters passed in the name is the same as the number of parameters of the associated operation and the same as the number of attributes of the associated signal respectively. Then it checks the correctness of the name by comparing it to a regular expression. This is implemented with the *java.util.regex* package [Sun, 2008]. The regular expression is denoted by:

```
\w+\((\w+[,\w+]*)?\)
```

The \w+ denotes an arbitrary sequence of letters and/or numbers (+ means that there must be at least one). This defines the name of the event. The parameters passed by the event are enclosed in parenthesis '\(...\)'. Inside the parenthesis reside the parameters '\w+', if there are more, they are separated by a *,* (see [,\w+] - the square brackets denote that these are optional). The *** after the closing square bracket indicates that there may be more that one ore none. The *?* defines that the whole parameter-list is optional.

The operation/signal that is transformed to an action in an IOSTS is not only referenced by events. It can also derive from call operation/send signal actions in an activity of an effect of a transition in UML. In that case it is easy to determine the parameters. In UML, call operation/send signal actions may have input pins as arguments. An input pin has properties like name, type, upper and lower, just the same as the message in an IOSTS. So each input pin is directly transformed to a message. The UML check component verifies whether the number of input pins is the same as the number of parameters of the associated operation and the same as the number of attributes of the associated signal respectively. It also checks the type values of the input pins against the type values of parameters of the operation and attributes of the signal respectively.

Return parameters also have to be considered. Actions of an IOSTS must not have return parameters. Return parameters defined in operations of the *Class Diagram* in UML are added to the action of the IOSTS to be passed by reference, thus a return parameter is translated to a message of an action in an IOSTS with the name *returnValue*.

8.6.2 Guards and Assignments

The guards in UML in this application are constraints that contain a literal string. The value specification actions that become assignments in an IOSTS contain an opaque expression whose body attribute consists of a list of strings that define the context updates of the variables. There was no validation of these strings so far.

The UML check component contains an OCL parser that checks if the passed strings are valid OCL expressions. The context of the OCL expression (the context is basically the reference of *self* in an OCL expression) is set to the underlying class of the *State Machine* that contains the guard and value specification action respectively. So each

guard and body string of a value specification action is an OCL expression. In an IOSTS, the guard is a boolean expression on the may contain variables, parameters and messages of an action (see 4.2). So each expression, no matter if guard or assignment must also be able to call the parameters passed by a message. Since the context of an OCL expression is set to the underlying class, it cannot access the passed parameters. So every time the OCL parser checks an expression, the passed parameters (no matter if they come from an event or an action) are also added to the context classifier (it is possible to add user defined attributes to the context classifier).

However, STG does not understand the whole OCL syntax. As mentioned in chapter 5.4, guards are simple boolean expressions and statements are very simple instructions. Only these types of expressions can be translated to STG. The = operator also has a different meaning in guards and value specification actions. In the context of a guard, it denotes a comparison and is therefore translated to a = in STG. In a value specification action, the = means an assignment. The assignment operator in STG is denoted by := (see Appendix A). That is why an = in a value specification becomes a := in context of STG.

8.6.3 Owner Association

As mentioned in Section 8.5, the classes *Guard*, *Assignment* and *Action* of the meta model have an association to their owning transition. This is due to some implementation issues that emerged using the openArchitectureWare framework. A *State Machine* that consists of two or more transitions, some of them contain a guard and two or more guards have the same value in their literal string is not transformed correctly. Only the last guard of those that have the same value is transformed properly, the other guards simply get lost. By saving the owner of the guard as property, each guard becomes unique and the problem disappears. The same issue appeared on assignments and actions.

Chapter 9

Example: Conference Protocol

Contents

9.1 Introduction

This chapter illustrates the example used in this application; the Conference Protocol. First, the Conference Protocol is explained in Section 9.2. After that, the UML specification of the Conference Protocol is described in Section 9.3. Section 9.4 depicts the resulting IOSTS produced by the transformation,.

9.2 General

The example application used for the transformation is the Conference Protocol [ConfProt, 2008]. This protocol is used as a case study within the Côte de Resyste project. The goal of this project is the development of methods/techniques for automatic test case generation. The development of tools for this purpose is also scope of this project. One of the main achievements of the project is the *TorX* tool [Tretmans and Brinksma, 2003] (the tool can be obtained here [TorX, 2008]), a tool for automated test case generation and execution from formal transition-based specifications.

The conference protocol is a simple UDP based chat protocol. There exist 28 different implementations in C, so it is very useful to test the quality of an automated test case generation process. There are also a lot of specifications available in the formal languages *LOTOS*, *Promela*, *SDL* and *FSM/EFSM*.

The conference protocol is a service that provides the capabilities for a simple multicast chat between a couple of users participating in a conference. The service consists of the following primitives:

- *join(nickname, confID)*: This service primitive is called when the user wants to join a conference with a given ID. The user can later be identified by the nickname. Initially, the user is only allowed to perform a *join*. Afterwards, the user may perform any of the other three service primitives.
- *leave()*: This service primitive is called when the user wants to leave the conference she has joined before. The user must participate in a conference to be able to perform a *leave*.
- *datareq(message)*: This service primitive is called when the user wants to send a message to all participants of the conference which the user has joined before. A *datareq* performed by one user causes a *dataind* by all other users of the same conference.
- *dataind(nickname, message)*: This service primitive is called when the user receives a message from any participant of the conference which the user has joined before.

The messages that are exchanged by the service are called PDUs (Protocol Data Units). A PDU that is sent to the underlying network is always delivered to all participants of the conference (except answerPDUs). There are four types of PDUs available:

- *joinPDU*: used to indicate that a user wants to join the conference (contains nickname of the sender and conference id)

- *answerPDU*: used to indicate that the sender of this PDU participates in the same conference as the receiver (contains nickname of the sender and conference id)
- *dataPDU*: contains the actual message that has to be delivered to all participants (contains the nickname of the sender and the actual message)
- *leavePDU*: used to indicate that a user wants to leave the conference (contains nickname of the sender and conference id)

The basic element of the conference protocol is a CPE (Conference Protocol Entity). A CPE is responsible for implementing the conference service primitives and so for sending and receiving PDUs to/from the underlying network. The CPE is also responsible to keep track of the conference participants; it holds a set of potential conference participants and a set of actual participants. A CPE that performs a *join* sends *joinPDUs* to all potential conference partners contained in the set of potential participants. After receiving a *joinPDU*, the CPE sends an *answerPDU* back to the sender (the nickname of the sender is included in the *answerPDU*). Then it adds the sender to the set of actual participants if the conference id of the PDU is the same as of the conference the CPU is joined. A CPE that receives an *answerPDU* adds the sender to the set of actual conference participants if the conference id of the *answerPDU* is the same as the conference id of the CPE. By performing a *datareq*, the CPE sends a *dataPDU* to all actual participants of the conference. A CPE that receives a *dataPDU* delivers the message to the user by performing a *dataind*. A CPE that performs a *leave* sends *leavePDUs* to all participants contained in the set of actual participants. Then it clears this set. Receiving a *leavePDU* results in removing the sender from the set of actual participants.

Figure 9.1 shows the interfaces that are provided by the CPE. The CSAP interface provides the communication between the user interface and the CPE. The service primitives communicate through this interface. The USAP interface provides the communication between the CSAP and the underlying (network) layer. The PDUs are sent and received by this interface.

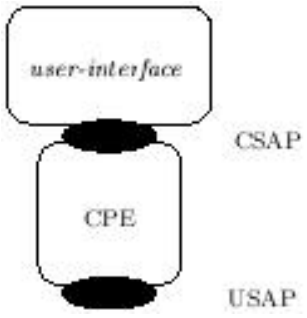


Figure 9.1: CPE Interfaces (source: [ConfProt, 2008])

9.3 UML Specification

This section explains the UML specification for the conference protocol described in Section 9.2. It consists of a *Class* and *State Machine Diagram*. It also summarizes the differences that had to be made to the original protocol in the UML specification.

9.3.1 Class Diagram

Figure 9.2 shows the class diagram of the conference protocol specification. It is very simple, it just contains one class called CPE. This class has three attributes: *num* holds the number of actual participants, *confID* holds the id of the joined conference (it has the value 0 if the CPE is not connected to a conference) and *userID* holds the actual id of the user (0 if the CPE is not connected, i.e. there is no user specified). Due to the fact that the STG tool only understands *int* and *bool* parameters, the *userID* is an integer value and not a string. The service primitives described in Section 9.2 are mapped to the four public operations:

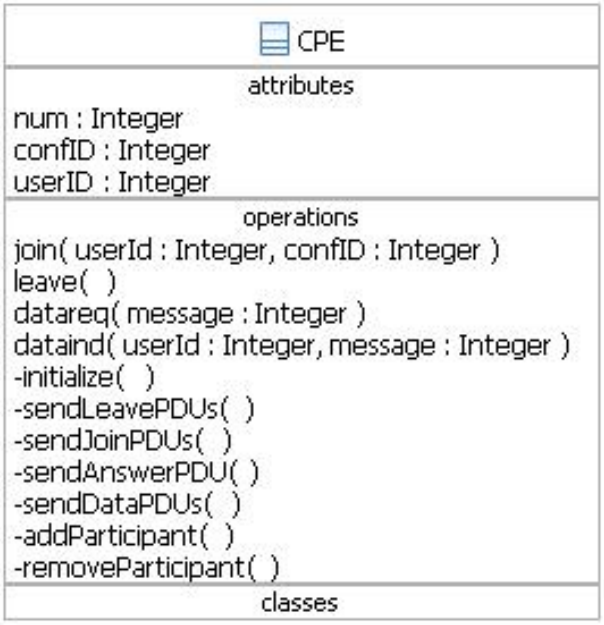


Figure 9.2: Class Diagram of the Conference Protocol

- *«input»join(userID : Integer, confID : Integer)*
- *«input»leave()*
- *«input»datareq(message : Integer)*
- *«output»dataind(userID : Integer, message : Integer)*

Operations *sendLeavePDUs()*, *sendJoinPDUs()*, *sendAnswerPDUs()* and *sendDataPDUs()* are stereotyped with «output», whereas *initialize()*, *addParticipant()* and *removeParticipant()* are «internal». Because of the limitation of the STG tool, the type

of the *message* parameter of the *datareq* operation is integer, not string. The conference protocol also keeps track of the actual participants by saving them in a set. This is not possible in STG, it cannot handle sets or arrays. Therefore the operations *«internal»addParticipant()* and *«internal»removeParticipant()* are introduced to simulate this, as well as the attribute *num*. The four PDUs described in Chapter 9.2 are implemented as signals (they are not depicted in Figure 9.2). There are four signals:

- *«input»joinPDU(uID : Integer, cID : Integer)*
- *«input»answerPDU(uID : Integer, cID : Integer)*
- *«input»dataPDU(uID : Integer, message : Integer)*
- *«input»leavePDU(uID : Integer, cID : Integer)*

9.3.2 StateMachineDiagram

Figure 9.3 shows the *State Machine Diagram* that specifies the behavior described in Section 9.2. As described in Chapter 6.4.2, each transition consists of a trigger, a guard and an effect, all of them are optional. This *State Machine Diagram* shows the name of the event that is associated with a trigger. It also shows the name of the effect of the transition. Remember, an effect of a transition is always an *Activity*.

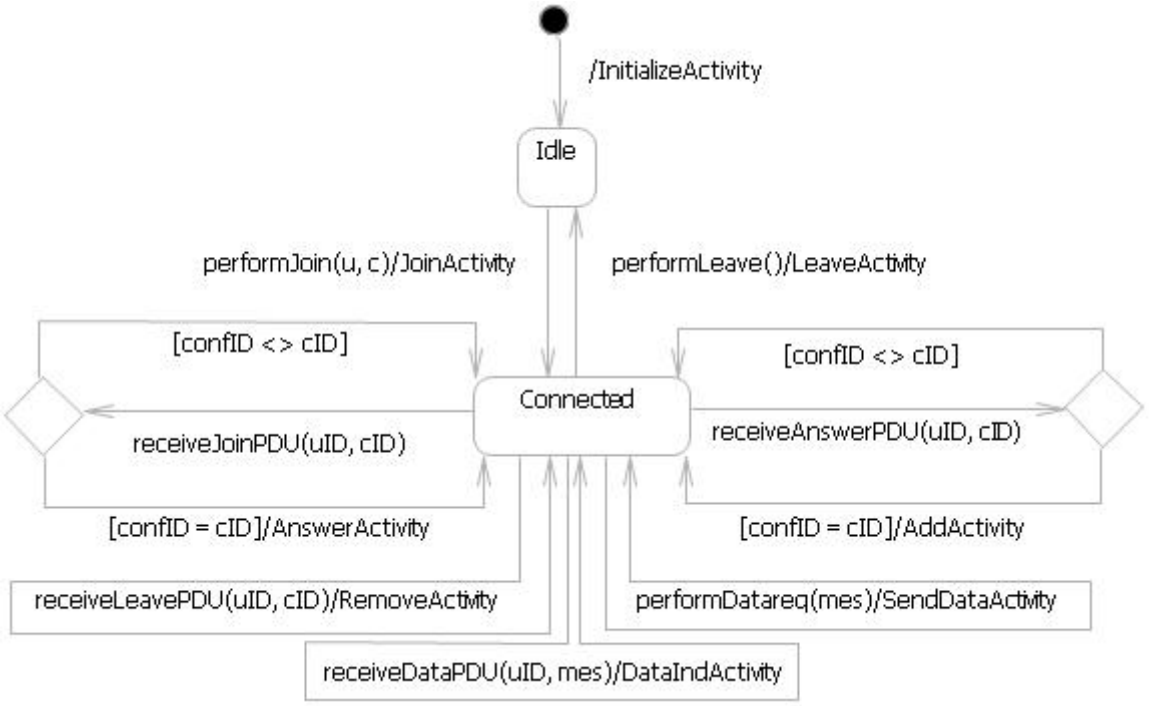


Figure 9.3: State Machine Diagram of the Conference Protocol

Event Type	Event	Called Operation/Signal
CallEvent	performJoin(u, c)	«input»join(u, c)
CallEvent	performLeave()	«input»leave()
CallEvent	performDataReq(mes)	«input»datareq(mes)
SignalEvent	receiveJoinPDU(uID, cID)	«input»joinPDU(uID, cID)
SignalEvent	receiveAnswerPDU(uID, cID)	«input»answerPDU(uID, cID)
SignalEvent	receiveLeavePDU(uID, cID)	«input»leavePDU(uID, cID)
SignalEvent	receiveDataPDU(uID, mes)	«input»dataPDU(uID, mes)

Table 9.1: Event - Operation

Table 9.1 shows a mapping between trigger (actually the name of the called event is depicted in Figure 9.3) and the associated operation/signal. The type of the event is also displayed.

Table 9.2 gives a closer look at the *Activities* used in the *State Machine Diagram* of Figure 9.3. Since an *Activity* in this application is a linear sequence of actions without branching, the sequence of actions is shown consecutively numbered in column two. Column three shows the called operation or signal, as well as the assignments of a value specification action.

Activity	Actions	Operation/Signal/Assignments
InitializeActivity	1. CallOperationAction	«internal»initialize()
	2. ValueSpecificationAction	num = 0 userID = 0 confID = 0
JoinActivity	1. ValueSpecificationAction	num = num + 1 userID = u confID = c
	2. CallOperationAction	«output»sendJoinPDUs()
LeaveActivity	1. ValueSpecificationAction	num = 0 userID = 0 confID = 0
	2. CallOperationAction	«output»sendLeavePDUs()
AnswerActivity	1. CallOperationAction	«internal»addParticipant()
	2. ValueSpecificationAction	num = num + 1
	3. CallOperationActio	«output»sendAnswerPDUs()
AddActivity	1. CallOperationAction	«internal»addParticipant()
	2. ValueSpecificationAction	num = num + 1
RemoveActivity	1. CallOperationAction	«internal»removeParticipant()
	2. ValueSpecificationAction	num = num - 1
DataIndActivity	CallOperationAction	«output»dataind(uID, mes)
SendDataActivity	CallOperationAction	«output»sendDataPDUs()

Table 9.2: Activities

9.3.3 Differences to the Original Protocol

One main difference is the restriction to *int* and *bool* datatypes in order to be consistent with the possibilities of the STG tool. The *message* and the *userID* are simple integer values. This does not really change anything in the functionality, it just had to changed due to the limitations.

Another difference is the lack of array support in the STG tool. The list of participants is not provided in the UML specification. Adding and removing a participant is simulated using the *addparticipant()* and *removeParticipant()* operations. The size of the array is simulated with the *num* attribute.

9.4 Conference Protocol IOSTS

This section shows the resulting IOSTS that has been created by the transformation. The IOSTS produced by the STG tool is depicted in Figure 9.4. Its definition in the STG language can be found in Appendix B.

States are depicted by round shapes, the properties of a transition are situated in rectangles between the transition. The first line describes the guard, if there is none, *true* is displayed ([bool] just denotes the type of the expression, it is always bool on guards). The *sync* keyword denotes an action, *?* specifies an input action, *!*labels an output action respectively. The *do* {...} block denotes the assignments of the transition, the type of an assigned variable of an assignment is surrounded by square brackets. The state *Init* is the initial state of the *State Machine Diagram* of Figure 9.3, states *Idle* and *Connected* are its simple states. All other states are a result of a transition that had to be split during the transformation algorithm. For example the transition *receiveLeavePDU(uID, cID)/RemoveActivity* of the *State Machine* is split into two transitions (upper right corner of Figure 9.4), one from the state *Connected* to *RemoveActivityCall1* and another from *RemoveActivityCall1* to *Connected*. Consider Table 9.1, the event *receiveLeavePDU(uID, cID)* is associated with the signal ≪input≫leavePDU(uID, cID). According to Table 9.2, *Activity RemoveActivity* contains a *CallOperationAction* that is associated to the operation ≪*internal*≫*removeParticipants*. Recall Sections 7.3 and 7.4, this transition has to be split, as can be seen in Figure 9.4. The other Transition work the same way, they are not discussed here.

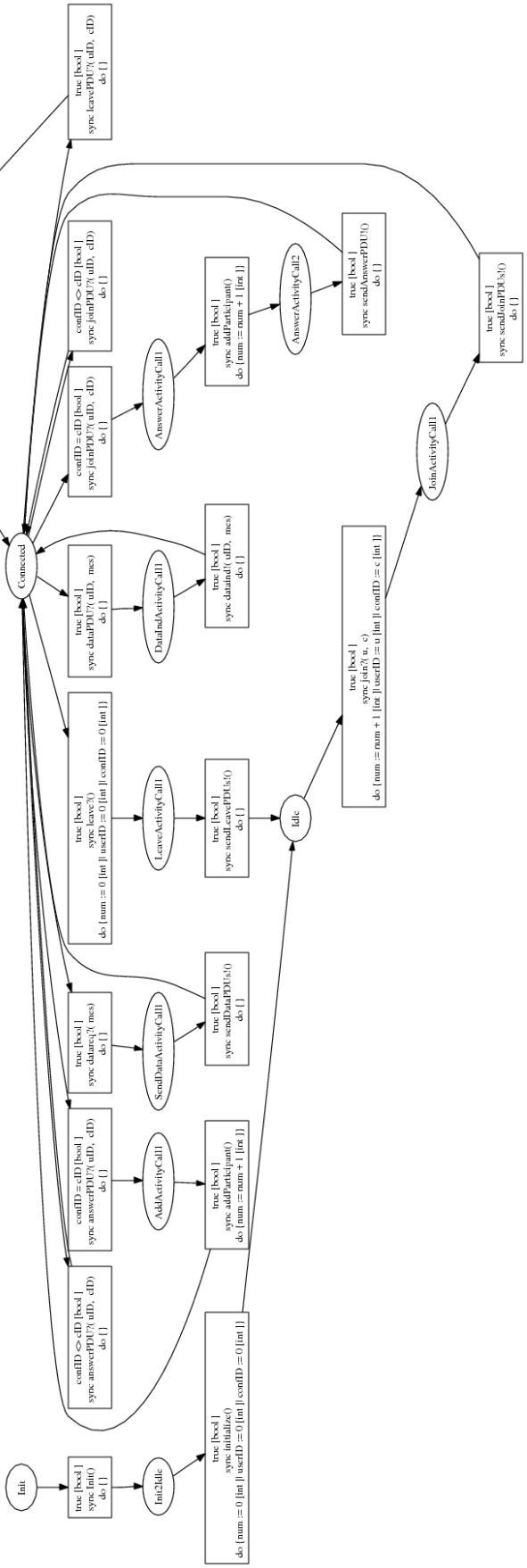


Figure 9.4: IOSTS CPE Specification

Chapter 10

Future Work

This chapter contains some ideas about how the transformation could be improved or enhanced in some further work

- *Concurrency*: by now, all elements in UML that imply concurrency are not considered in this application, like fork and join nodes in an *Activity* or different regions in a *State Machine*. As mentioned in Chapter 6.4.2, an IOSTS does not really support concurrency. Though there is a possibility to simulate concurrency in an IOSTS. Consider the *Activity Diagram* of Figure 10.1 and imagine action A and B are placeholders for any kind of action sequence. It is possible to transform this into an IOSTS with two paths, one that first contains all actions of A followed by B and another path that consists of the actions of B followed by the actions of A. [Pickin et al., 2002] also propose this method in their work for handling concurrency. This can be applied to every possible kind of concurrent element of UML. Of course, if there are a lot of concurrent paths in an UML model this would lead to a huge amount of paths in an IOSTS since n concurrent paths are transformed to $n!$ paths in an IOSTS.

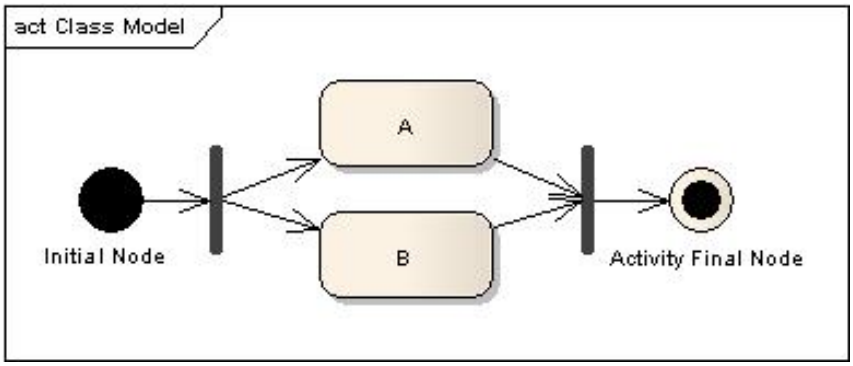


Figure 10.1: Concurrency

- *Reducing UML restrictions*: As explained in Chapter 6, there are a lot of restrictions that have to be applied to the UML models in order to perform the transformation. Reducing these restrictions would be a topic for future work. Some proposals are given below:

  - *State Machine*: By now, only simple states are allowed. As mentioned in Section 6.4.2, states may have entry and exit actions which are also behaviors. Taking these actions into account would be an improvement to the overall transformation.

    Another point concerning *State Machines* is the restriction that an effect of a transition must be an *Activity*. Regarding Section 6.4.1, an effect may be an *OpaqueBehavior*, *FunctionBehavior*, *State Machine*, *ProtocolStateMachine*, *Activity* or *Interaction*. Taking some of these elements into account would also improve the UML modelling facilities. Incorporating *FunctionBeahviors* should not be very difficult since a *FunctionBehavior* references only an operation of the underlying UML model. It should also be possible to allow *Interactions* (that are actually *Sequence Diagrams*) although this is not so easy since *Interactions* and *Activities* are quite different.

    By now, the assignments of the IOSTS are modelled by means of value specification actions in the *Activity Diagrams*. It should also be possible to define context updates in terms of postconditions on transitions.

    This application only allows call and signal events. Regarding Figure 6.5, UML defines more than these events (change and signal events). Allowing more of these events would improve the modelling possibilities and thus enhance the whole transformation. Especially change events should not be difficult to include (basically a change event is an event that is not associated with an operation or signal). Time events are problematic since an IOSTS does not support timing. It should be possible to use timed automata [Alur and Dill, 1994] or a symbolic version of *tioco* [Krichen and Tripakis, 2004] instead.

  - *Activity*: This application uses *Activities* to describe the effect of a transition in a *State Machine*. These activities may only contain three types of actions: call operation actions, send signal actions and value specification actions. Incorporating more types of actions would improve the transformation.

    Instead of the value specification actions that contain the context updates of the variables, it should be possible to add pre- and postconditions to the actions. Postcondition can comprise the context updates, whereas preconditions define boolean conditions that could be transformed to guards in an IOSTS.

    By now, *Activities* are only straight sequences of actions without branching since branching mostly denotes concurrency. This is not always the case. It is possible to add decision nodes to *Activity Diagrams* in order to split the activity path. The concept of decision nodes is the same as of choice pseudostates in *State Machines* (see Section 6.4.2).

- *Front-End*: By now, the implementation of the transformation is only in a prove of concept phase, so it is not very user friendly. Building a front-end (graphical user interface) on the top of the transformation algorithm would be a major enhancement of the overall project.

- *Using other UML modelling tools*: The transformation uses the eclipse implementation of UML 2. The generation of UML models in the eclipse environment is

not very user friendly. There exist other (non-)commercial tools that are quite a lot more handy. Defining the UML model in an external tool and importing it in the eclipse environment was not tested during this work. Though it should not be difficult since UML models can be exchanged between different tools [OMG, 2006a].

- *Eclipse Plugins*: Some improvements to the implementation could be achieved by using some of the facilities offered by the eclipse framework:
  - *Graphical Modeling Framework (GMF)*: GMF [Eclipse, 2008b] is based on the eclipse modeling framework (EMF). It offers facilities to generate a graphical editor to meta models defined in EMF (Ecore). Thus it is possible to define a graphical editor for the IOSTS models that conform to the meta model defined in Section 8.5. This offers possibilities for editing easily any kind of IOSTS model. It also offers great presentation benefits because transformed UML models could be displayed in a much more intuitively manner as it allows a textual representation.
  - *Xtext*: Xtext is part of the openArchitectureWare framework [Efftinge et al., 2006]. It provides mechanisms for defining a textual editor in the eclipse environment for any kind of language, actually it supports the definition of a domain-specific language. This mechanism is also based on ecore models. It should be possible to enhance the transformation project with a textual editor for the STG language specification with all the benefits a textual editor provides, such as syntax highlighting, code completion, code folding, a configurable outline view and static error checking for the given syntax.
- *Other Transformations*: The transformation is split into two parts, the model to model and the model to text transformation. This makes it easy to generate other IOSTS representations, not only the STG language specification. Another topic would be the transformation to another target language, refer to [Tretmans, 2008] for more information of tools that are available. The model to model transformation must possibly also be adapted in that case.
- *Test Purpose*: The test case generation process described in Section 4.4 takes as argument an IOSTS system specification and an IOSTS test purpose. Defining the test purpose is not part of this application. This could be the task of some future work, i.e. how the test purpose can be modelled in UML and then transformed into an IOSTS test purpose. It should be possible to model the test purpose in terms of a *State Machine Diagram* with special states stereotyped with *Accept* and *Reject* to determine these states which are needed by the test purpose (see Section 4.3).

Chapter 11

Conclusion

UML is the de facto standard language for system specification purposes. There exist proper tools that are able to generate test cases like the STG tool, but the system specification used by this tools is not based on UML. There is a lack of generating test cases directly from UML models (there are approaches to generate test cases from UML, though not from UML2 and not with respect to *sioco*). This thesis fills this gap by providing a transformation from UML to IOSTS, a specification language used amongst others by the STG tool that is able to generate executable test cases from IOSTS system specifications. Defining a meta model for IOSTS specifications is also comprised in this thesis.

The UML specification is very extensive, not all diagrams are appropriate for such a transformation. One goal of this thesis is to show that the transformation is possible, thus only a subset of UML is used for the transformation, i.e. *Class*, *State Machine* and *Activity Diagrams*. There are also a lot of restrictions on these diagrams that have to be made. This of course limits the practical use of the application. Some future work has to be done in order to reduce these restrictions to improve the whole transformation process.

The transformation is implemented in the eclipse environment. There are some powerful plugins and frameworks for model engineering, especially the openArchitectureWare framework offers effective components for model to model and model to text transformation, as well as possibilities for constraint checking on models. The UML models are generated using the eclipse UML2 plugin. This is not very user-friendly, since the UML models have to be constructed with a tree editor. There is also a plugin called UML2Tools that offers a graphical editor for UML2 models, but it is not possible to modify all elements of an UML model in this editor.

It is important to mention that the main focus of the thesis is the transformation from UML to IOSTS, so a generalized IOSTS meta model is introduced. The transformation is split into two parts, the model to model and the model to text transformation. The IOSTS model obtained after the model to model transformation can be used as input for any arbitrary tool that can handle IOSTS models (there is no restriction to the STG tool). Only the model to text transformation, that is tailored to the STG tool in this application, has to be changed.

The Conference Protocol example shows a practical example of the transformation algorithm presented in this thesis, although it is not easy to model an UML specification with all the restrictions that have to be made to the UML model (and the limitations

the STG tool provides). Reducing these limitations in some future work will improve the modelling capabilities and so the practical value of the whole test case generation workflow.

The application developed in this thesis fills the gap between UML modelling and test case generation tools based on Input/Output Symbolic Transition Systems with respect to the *sioco* conformance relation. Thus, it is a step of generating test cases directly from UML models.

Appendix A

STG Language Specification

```
<system> :=           system <system-id>;
                      [const <constant>]
                      [type <type>]
                      gate <gate>
                      <process>

<gate> :=             <gate-id> [(<type-id>{, <type-id>})];

<process> :=          process <process-id>;
                      input <gate-id>
                      output <gate-id>
                      internal <gate-id> [tau]
                      [parameters <params>]
                      [variables <vars>]
                      state <state>
                      transition <transition>

<params> :=           <ident_name> : <type>;

<vars> :=             <ident_name> : <type>;

<state> :=            init : <state-id>; | <state-id>;

<transition> :=       from <state-id>
                        [<guard>]
                        [<action>]
                        [<statement>]
                      to <state-id>;

<guard> :=            if <expression>

<action> :=           sync <action_type>
```

```
<action_type> :=      <internal_action> | <other_action>

<internal_action> :=  <gate> | tau

<other_action> :=     <gate> {?|!} (<io_list>)

<io_list> :=          <mess-id>{, <mess-id>}

<mess-id> :=          <ident> | <constant-id>

<statement> :=        do {<inst>{| <inst>}}

<inst> :=             <expression> := <expression>

<expression> :=       <constant_value> |
                      <name> |
                      + <expression> |
                      - <expression> |
                      not <expression> |
                      <expression> [<expression>] |
                      <expression> . <expression> |
                      <expression> -> <expression> |
                      (<expression>) |
                      <expression> <binop> <expression>

<binop> :=            or | and | = | <> | < | <= | >= | > | + | - |
                      * | / | %

<name> ::=            _IDENT_
```

Appendix B

Conference Protocol in STG Language

```
system CPE;

gate
  initialize();
  join(int,int);
  sendJoinPDUs();
  leave();
  sendLeavePDUs();
  leavePDU(int,int);
  removeParticipant();
  dataPDU(int,int);
  dataind(int,int);
  datareq(int);
  sendDataPDUs();
  answerPDU(int,int);
  addParticipant();
  joinPDU(int,int);
  sendAnswerPDU();

process CPEBehavior;

input
  join, leave, leavePDU, dataPDU, datareq, answerPDU, joinPDU;

output
  sendJoinPDUs, sendLeavePDUs, dataind, sendDataPDUs, sendAnswerPDU;

internal
  initialize, removeParticipant, addParticipant;
```

```
variables
  num : int;
  confID : int;
  userID : int;

state
  init : Init;
  Idle;
  Connected;
  JoinActivityCall1;
  LeaveActivityCall1;
  RemoveActivityCall1;
  DataIndActivityCall1;
  SendDataActivityCall1;
  AddActivityCall1;
  AnswerActivityCall1;
  AnswerActivityCall2;
  Init2Idle;

transition

  from Init
    if(true)
  to Init2Idle;

  from Init2Idle
    sync initialize
    do {
      num := 0 |
      userID := 0 |
      confID := 0
      }
  to Idle;

  from Idle
    sync join?(u,c)
    do {
      num := num + 1 |
      userID := u |
      confID := c
      }
  to JoinActivityCall1;

  from JoinActivityCall1
    sync sendJoinPDUs!()
```

```
to Connected;

from Connected
  sync leave?()
  do {
    num := 0 |
    userID := 0 |
    confID := 0
    }
to LeaveActivityCall1;

from LeaveActivityCall1
  sync sendLeavePDUs!()
to Idle;

from Connected
  if (confID = cID)
  sync joinPDU?(uID,cID)
to AnswerActivityCall1;

from AnswerActivityCall1
  sync addParticipant
  do {
    num := num + 1
    }
to AnswerActivityCall2;

from AnswerActivityCall2
  sync sendAnswerPDU!()
to Connected;

from Connected
  if (confID <> cID)
  sync joinPDU?(uID,cID)
to Connected;

from Connected
  if (confID = cID)
  sync answerPDU?(uID,cID)
to AddActivityCall1;

from AddActivityCall1
  sync addParticipant
  do {
    num := num + 1
```

```
    }
to Connected;

from Connected
  if (confID <> cID)
  sync answerPDU?(uID,cID)
to Connected;

from Connected
  sync leavePDU?(uID,cID)
to RemoveActivityCall1;

from RemoveActivityCall1
  sync removeParticipant
  do {
    num := num - 1
    }
to Connected;

from Connected
  sync dataPDU?(uID,mes)
to DataIndActivityCall1;

from DataIndActivityCall1
  sync dataind!(uID,mes)
to Connected;

from Connected
  sync datareq?(mes)
to SendDataActivityCall1;

from SendDataActivityCall1
  sync sendDataPDUs!()
to Connected;
```

Bibliography

[Alur and Dill, 1994] Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

[Bozga et al., 2002] Bozga, M., Graf, S., and Mounier, L. (2002). If-2.0: A validation environment for component-based real-time systems. In *Computer Aided Verification*, volume 2404/2002 of *Lecture Notes in Computer Science*, pages 630–640, London, UK. Springer-Verlag.

[Bozga et al., 2004] Bozga, M., Graf, S., Ober, I., Ober, I., and Sifakis, J. (2004). The IF Toolset. In *Formal Methods for the Design of Real-Time Systems*, volume 3185/2004 of *Lecture Notes in Computer Science*, pages 237–267, Berlin, Heidelberg. Springer-Verlag.

[Bozga and Olvovsky, 2002] Bozga, M. and Olvovsky, S. (2002). Intermediate Language 2.0 with Test Directives Specification. AGEDIS Project Website: http://www.agedis.de. Last visited: 09.2008.

[Cavarra and Davies, 2001] Cavarra, A. and Davies, J. (2001). AGEDIS: Language Specification. AGEDIS Project Website: http://www.agedis.de. Last visited: 09.2008.

[Clarke et al., 2001a] Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2001a). Automated Test and Oracle Generation for Smart-Card Applications. In *Smart Card Programming and Security*, volume 2140/2001 of *Lecture Notes in Computer Science*, pages 58–70, Berlin, Heidelberg. Springer-Verlag.

[Clarke et al., 2001b] Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2001b). STG: A Tool for Generating Symbolic Test Programs and Oracles from Operational Specifications. *ACM SIGSOFT Software Engineering Notes*, 26(5):301–302.

[Clarke et al., 2002] Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2002). STG: A Symbolic Test Generation Tool. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280/2002 of *Lecture Notes in Computer Science*, Berlin, Heidelberg. Springer-Verlag.

[ConfProt, 2008] ConfProt (2008). The Conference Protocol Case Study. Conference Protocol Website: http://fmt.cs.utwente.nl/ConfCase. Last visited: 09.2008.

[Constant et al., 2007] Constant, C., Jéron, T., Marchand, H., and Rusu, V. (2007). Integrating Formal Verification and Conformance Testing for Reactive Systems. *IEEE Transactions on Software Engineering*, 33(8):558–574.

[Eclipse, 2008a] Eclipse (2008a). Eclipse Modeling Framework Project (EMF). EMF Website: http://www.eclipse.org/modeling/emf. Last visited: 09.2008.

[Eclipse, 2008b] Eclipse (2008b). Graphical Modeling Framework Website (GMF). GMF Website: http://eclipse.org/gmf. Last visited: 09.2008.

[Eclipse, 2008c] Eclipse (2008c). Model Development Tools (MDT). MDT Website: http://www.eclipse.org/modeling/mdt. Last visited: 09.2008.

[Efftinge et al., 2006] Efftinge, S., Graf, S., and Völter, M. (2006). oAW xText - A framework for textual DSLs. Eclipse Summit 2006, Workshop: Modeling Symposium.

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

[Frantzen et al., 2005] Frantzen, L., Tretmans, J., and Willemse, T. A. C. (2005). Test Generation Based on Symbolic Specifications. In *Formal Approaches to Software Testing*, volume 3395/2005 of *Lecture Notes in Computer Science*, pages 1–15, Berlin, Heidelberg. Springer-Verlag.

[Frantzen et al., 2006] Frantzen, L., Tretmans, J., and Willemse, T. A. C. (2006). A Symbolic Framework for Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262/2006 of *Lecture Notes in Computer Science*, pages 40–54, Berlin, Heidelberg. Springer-Verlag.

[Graf et al., 2006] Graf, S., Ober, I., and Ober, I. (2006). A real-time profile for UML. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(2):113–127.

[Hartman, 2004] Hartman, A. (2004). Final Report. AGEDIS Project Website: http://www.agedis.de. Last visited: 09.2008.

[Hartman and Nagin, 2004] Hartman, A. and Nagin, K. (2004). The AGEDIS Tools for Model Based Testing. *ACM SIGSOFT Software Engineering Notes*, 29(4):129–132.

[Ho et al., 1999] Ho, W. M., Jézéquel, J.-M., Guennec, A. L., and Pennaneac'h, F. (1999). UMLAUT: An Extendible UML Transformation Framework. In *ASE '99: Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, page 275, Washington, DC, USA. IEEE Computer Society.

[IRISA, 2008a] IRISA (2008a). The STG Tool Page. STG Website: http://www.irisa.fr/prive/ployette/stg-doc/stg-web.html. Last visited: 09.2008.

[IRISA, 2008b] IRISA (2008b). UMLAUT: Unified Modeling Language All pUrposes Transformer. UMLAUT Website: http://www.irisa.fr/UMLAUT. Last visited: 09.2008.

[Jard and Jéron, 2005] Jard, C. and Jéron, T. (2005). TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315.

[Jeannet, 2008] Jeannet, B. (2008). The NBAC verification/slicing tool. NBAC Website: http://pop-art.inrialpes.fr/people/bjeannet/nbac. Last visited: 09.2008.

[Jéron et al., 2006] Jéron, T., Marchand, H., and Rusu, V. (2006). Symbolic Determinisation of Extended Automata. In *Fourth IFIP International Conference on Theoretical Computer Science- TCS 2006*, volume 209/2006 of *IFIP International Federation for Information Processing*, pages 197–212, Boston, MA, USA. Springer-Verlag.

[Krichen and Tripakis, 2004] Krichen, M. and Tripakis, S. (2004). Black-Box Conformance Testing for Real-Time Systems. In *Model Checking Software*, volume 2989/2004 of *Lecture Notes in Computer Science*, pages 109–126, Berlin, Heidelberg. Springer-Verlag.

[oAW, 2008] oAW (2008). openArchitectureWare. oAW Website: http://www.openarchitectureware.org. Last visited: 09.2008.

[Ober et al., 2006] Ober, I., Graf, S., and Ober, I. (2006). Validating timed UML models by simulation and verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(2):128–145.

[OMEGA, 2008] OMEGA (2008). IST OMEGA Project. OMEGA Project Website: http://www-omega.imag.fr. Last visited: 09.2008.

[OMG, 2004] OMG (2004). Meta Object Facility (MOF) 2.0 Core Specification. Technical report, The Object Management Group.

[OMG, 2006a] OMG (2006a). Diagram Interchange, V1.0. Technical report, The Object Management Group.

[OMG, 2006b] OMG (2006b). Object Constraint Language, V2.0. Technical report, The Object Management Group.

[OMG, 2007a] OMG (2007a). OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Technical report, The Object Management Group.

[OMG, 2007b] OMG (2007b). OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. Technical report, The Object Management Group.

[Pender, 2003] Pender, T. (2003). *UML Bible*. John Wiley & Sons, Inc., New York, NY, USA.

[Pickin et al., 2001] Pickin, S., Jard, C., Heuillard, T., Jézéquel, J.-M., and Desfray, P. (2001). A UML-integrated Test Description Language for Component Testing. In *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001 October 1st, 2001 in Toronto, Canada*, pages 208–223. German Informatics Society.

[Pickin et al., 2002] Pickin, S., Jard, C., Traon, Y. L., Jéron, T., Jézéquel, J.-M., and Guennec, A. L. (2002). System Test Synthesis from UML Models of Distributed Software. In *Formal Techniques for Networked and Distributed Systems - FORTE 2002*,

volume 2529/2002 of *Lecture Notes in Computer Science*, pages 97–113, Berlin, Heidelberg. Springer-Verlag.

[Rusu et al., 2000] Rusu, V., du Bousquet, L., and Jéron, T. (2000). An Approach to Symbolic Test Generation. In *Integrated Formal Methods: Second International Conference, IFM 2000, Dagstuhl Castle, Germany, November 2000. Proceedings*, volume 1945/2000 of *Lecture Notes in Computer Science*, pages 338–357, Berlin, Heidelberg. Springer-Verlag.

[Sun, 2008] Sun (2008). The java.util.regex Package. API Documentation: http://java.sun.com/javase/6/docs/api/java/util/regex/package-summary.html. Last visited: 09.2008.

[TorX, 2008] TorX (2008). TorX Test Tool. TorX Website: http://fmt.cs.utwente.nl/tools/torx/introduction.html. Last visited: 09.2008.

[Tretmans, 1994] Tretmans, J. (1994). A Formal Approach to Conformance Testing. In *Protocol Test Systems*, volume C-19 of *IFIP Transactions*, pages 257–276, Amsterdam, The Netherlands. North-Holland Publishing Co.

[Tretmans, 1996] Tretmans, J. (1996). Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120.

[Tretmans, 2008] Tretmans, J. (2008). Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949/2008 of *Lecture Notes in Computer Science*, pages 1–38, Berlin, Heidelberg. Springer.

[Tretmans and Brinksma, 2003] Tretmans, J. and Brinksma, E. (2003). TorX : Automated Model Based Testing. In *Hartman, A., Dussa-Zieger, K. eds. Proceedings of the First European Conference on Model-Driven Software Engineering*, Nürnberg, Germany. Imbuss.

[Verimag, 2008] Verimag (2008). Lucky: A (target) language for describing and simulating stochastic reactive systems. Lucky Website: http://www-verimag.imag.fr/~synchron/index.php?page=lurette/lucky. Last visited: 09.2008.