Master's Thesis

Integrated Formal Verification of Dynamic UML Models

by

Gergely Varró

May 16th, 2003

Supervisor
Dr. András Pataricza
associate professor

Budapest University of Technology and Economics
Department of Measurement and Information Systems

Abstract. UML is widely used in recent software development projects. It is a standard graphical language for specifying system models on a high abstraction level. Despite its popularity, it has several shortcomings that are obstacles to model the system in a mathematically precise way. First of all, a precise semantics for behavioral diagrams is missing. Moreover, it does not specify how to obtain an integrated and consistent system model from individual UML diagrams.

Formal methods are mathematics-based techniques offering a rigorous and effective way to model and analyze computer systems. These techniques can be used for verification, but they require precise inputs and operate on a low abstraction level. Model checking tools are especially popular for formal verification due to the high-level of automation they provide.

In case of safety critical systems, UML based system models have to be formally verified against its requirements. Unfortunately, there is a huge abstraction gap between UML models and mathematical models, therefore automated transformations are frequently required to bridge that gap. Existing transformation based solutions typically handle different UML diagrams separately, which does not guarantee the overall correctness of the system under design.

The current thesis first overviews existing transformation based formal verification approaches to UML statecharts, and several model checking tools.

Then it proposes a complex two phase transformation for the formal analysis of the behavior of UML system models designed by an integration of UML Statecharts and Action Semantics expressions. The initial phase creates an intermediate formal representation of behavioral UML models based upon a combination of transition systems and Petri nets. Then the second phase derives a Promela output that is accepted by the SPIN model checker in order to carry out the verification. However, due to the intermediate representation, other model checker tools can easily be integrated without re-designing the initial phase.

As a result, we obtain an integrated and verification tool independent framework simultaneously handling the semantics of different behavioral UML diagrams.

Contents

List of Figures

Chapter 1

Introduction

1.1 System Modeling Overview

Large enterprise applications that surround us in nearly every field of life must be more than just a bunch of code modules. They must be structured in a way that enables scalability, security, and robust execution under stressful conditions, and their structure must be defined clearly enough that maintenance programmers can quickly find and fix a bug that shows up long after the original authors have moved on to other projects.

That is, these programs must be designed to work perfectly in many areas, and business functionality is not the only one (although it certainly is the essential core). Of course a well-designed architecture benefits any program.

Model-Driven Architecture. When a large application is being developed, interoperability and portability become a key question. OMG highlighted the same keywords [21] when formulating the primary goals of Model-Driven Architecture (MDA). MDA is an approach to IT system development that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform. These two specifications are called Platform-Independent Model (PIM) and Platform-Specific Model (PSM).

Platform-Independent Model (PIM) represents business functionality and behavior very precisely but does not include technical aspects. From the PIM, MDA-enabled development tools follow OMG-standardized mappings to produce one or more Platform-Specific Models (PSMs), one for each target platform that the developer chooses.

The promise of MDA is to allow definition of machine-readable application and data models, which allow long-term flexibility of:

- **implementation**: new implementation infrastructures can be integrated or targeted by existing designs
- **integration**: since not only the implementation but the design exists at time of integration, we can automate the production of data integration bridges and the connection to new integration infrastructures
- **maintenance**: the availability of the design in a machine-readable form gives developers direct access to the specification of the system, making maintenance much simpler

- **testing and simulation**: since the developed models can be used to generate code, *they can equally be validated against requirements, tested against various infrastructures and can be used to directly simulate the behavior of the system being designed.*

Unified Modeling Language. OMG created general modeling standards such as MOF and UML that can be used to model our system. Meta-Object Facility (MOF) is an OMG standard, closely related to UML, that enables meta-data management and language definition. The Unified Modeling Language (UML) (for an overview [4], for a complete reference [26]) is a *standard language for specifying the structure and behavior of a variety of systems* from small control systems to large and complex open systems. It uses a graphical notation on a high abstraction level, which makes easier to survey complex systems in several phases of design. The usage of UML is not limited to softwares only. It can also be used for business processes, communication protocols and lots of other systems.

The UML notation has some shortcomings.

- One major problem is that *the notation lacks precise semantics.*
- The other deficiency is that *it does not tell how to obtain an integrated and consistent system model from individual UML diagrams.*

Action Semantics. However, two advanced features were added to UML to extend its expressiveness. Object Constraint Language (OCL) has been part of UML since its beginning, while the Action Semantics extension is a recent addition. OCL will not be further discussed, since it does not have any role in defining dynamic behavior.

Action Semantics makes possible to describe dynamic behavior in a low abstraction level and to express the functionality of methods and activities attached to state machine transitions.

The main advantage of the Action Semantics is to *enable automatic code generation*, since it specifies methods precisely even in the design phase.

Formal verification. ***Formal methods*** *are mathematics-based techniques offering a rigorous and effective way to model, design and analyze computer systems.* They have been a topic of research for many years with valuable results but the industrial utilization have appeared at only specialized development sites, however, their necessity is vital due to the complexity of IT products and increasing requirements for dependability and Quality of Service (QoS).

Formal verification methods based on ***model checking*** *are applied on a finite state representation of system behavior.* Verification is usually carried out by using model checking algorithms to demonstrate the satisfiability of certain properties formalized as logical formulae over the model of the system. For example, safety and liveness requirements can be expressed as temporal logic formulae and can be checked on the model of the system.

Verification tools may give two typical answers. The first and more simple case, when all statements describing the required behavior have been verified and found to be correct. In this case the designer can go one step further in the design process. The other possibility is when an error is found and a configuration trace leading into the faulty state is retrieved. As can be expected this trace is also a low-level description, namely, an enumeration of the consecutive states.

Traditional testing methods and simulation could be carried out after the implementation step. As it was mentioned before, MDA allows definition of machine-readable models, which allow testing and simulation even in an early phase of system design. *Thus, verification can be performed without actually implementing the system and this is useful, since this procedure reduces costs.*

Several verification tools [7, 16, 22] already exist, but they all use precise inputs and operate on a low abstraction level. As one can immediately notice, *there is a gap in the level of abstraction between verification tool inputs and UML models.*

The problem that is investigated in this thesis is found at the junction of UML based modeling and early phase verification. The main goal is to present a transformation that receives UML models as inputs, and produces verification tool input.

1.2 Related Work

Since the original formalism of Harel [6], the theory of statecharts has been under an extensive research and many different semantic approaches evolved from the academic world (a comparison of different approaches can be found in e.g. [34]). However, as the industrial interest is rather limited to the Statemate and UML variants, therefore, the majority of recent approaches for statecharts semantics have typically focused on the formalization of those variants. In this section, we restrict our attention to compare only proposals for the UML dialect with a stress on the support of formal verification.

Extended Hierarchical Automata were introduced in [17] for Statemate and in [13] for UML. In a second phase, both approaches transform their models into Promela code and verify them by the model checker SPIN [7]. A major stress is put on formal verification in [29] where UML statecharts are encoded into a PVS [22] specification enabling the access to automated theorem proving of UML design, while in [12], the model checking of UML statecharts is aimed.

An entire verification round-trip is reported in [2] and [23] where the results of model checking are represented visually in the original UML models. The semantic core of statecharts is formalized in those papers by means of abstract state machines and state term graphs, respectively.

Despite their success from a verification point of view, the use of precise, formal mathematics is also the common weakness of all these approaches: they fail to provide a high level of abstraction that can be properly understood (and implemented) by systems engineers. Previous proposals in the field of graph transformation (e.g., [5, 11]) have tried to tackle this problem by providing a *visual specification* of statechart semantics.

In the sequel, we discuss in more details the two most related approaches to the current thesis.

The first approach (for a report see [13]) was prepared by CNUCE and the Department of Measurement and Information Systems. They defined a transformation from UML State Machines to PROMELA code, so it formed the antecedents of my work. Because of this stronger relationship, the framework that was used in [13] is overviewed in Section 3.1.

UML State Machines were represented as Extended Hierarchical Automata (EHA) and this representation was translated to PROMELA code in a mathematically precise way.

As another approach, a visual operational semantics was presented in [32]. It used EHA again as the underlying static structure and model transition systems for defining operational semantics.

We survey the strength and deficiencies of the EHA based approach from our point of view.

+ The construct of hierarchical automata gives a precise mathematical basis even for our input model and it eases the verification of the transformation itself.

– It does not deal with activities that are more complex than just sending a signal. Furthermore, activities inside states (e.g., entry and exit activities) are also missing. (These requirements were not included in their project goals.)

– They only discussed UML State Machines, which is only one possible way to describe dynamic behavior.

The other similar approach was presented in [3]. The authors defined a formal execution semantics for *UML activity diagrams* that is appropriate for workflow modeling. Their approach is based on the STATEMATE semantics of statecharts, but is also extended with some transactional properties to deal with data manipulation.

We summarize again the distinctive features of the approach.

- The authors proposed an execution semantics for UML activity diagrams and also formalized it in terms of labelled transition systems. That is the same method how several verification tools represent their semantics, so it is a *mathematically precise approach* again.
- *The report does not mention among its goal to generate verification tool input.*
- The semantics was definitely defined for *workflow models*. In such an environment, an activity is an uninterruptible amount of work that is performed in a non-zero span of time by an actor. A sample activity can be to receive an order, which is a correct definition for a workflow model, but it lacks the precision required to define an action. Action Semantics of OMG necessitates a low abstraction level and, thus, detailed description, since an action may only access or transform a single piece of data unit.
- *The approach does not deal with deferred event handling and swimlanes.* They disregard swimlanes, since these do not impact the execution semantics.
- UML also defines *object nodes and data flow connections*, but this approach omits these constructs. It is a drawback, since e.g., data flow relationship is a basic constraint in programs when the sequence of statements is wanted to be determined.
- At the time when the report was published, UML 2.0 did not exist. *Several constructs related to activity diagrams and action semantics were reorganized in UML 2.0.*

1.3 Research Objectives

My research objectives are:

1. to design a transformation that simultaneously handles several UML diagram types describing dynamic behavior (State Machines, Activity Diagrams) and Action Semantics on the input side,
2. to specify a precise framework that
   - enables us to have a verification tool independent description,

  - makes possible to merge different semantics into a common notation,

3. to investigate and to compare several verification tools in order to select the one that provides the best fit to our previous objectives.

1.4 The Structure of the Thesis

- Chapter 2 gives an overview on the UML notation. Since there are several more detailed sources in relation with UML (see [26] for example), only those language elements are discussed that are relevant and tightly connected to the topic.

  Class diagrams are used for describing the static aspect. They also provide a natural way to partition the system in order to enable performing a compositional verification.

  There is not only one widely accepted method for representing the dynamic behavior of the system, so the concepts of *actions*, *activities* and *state machines* should also be covered. These concepts are not independent of each other.

- Chapter 3 presents frameworks that can be used as the basis of a verification tool independent description.

- Chapter 4 surveys and compares two verification tools.

- Chapter 5 introduces the first phase of the transformation.

  The first step uses UML models as inputs and generates a tool independent description. This kind of independence is necessary, since tools may define different syntax for the same concept.

  The idea stems from a former transformation, but it is extended in two different ways. First of all, some new elements to be handled (e.g., entry and exit activities) are taken into consideration, and thus involved in the process of transformation. The other extension points towards the generalization in the mathematical background.

- Chapter 6 covers the second phase of the transformation, and has to generate a verification tool input from a low level compact description.

- Chapter 7 concludes and evaluates my work. Finally, it presents some research directions for the future.

Figure 1.1 shows the main information flow in the thesis. Dotted lines represent the weakest relation. Dashed lines with equally long black and white segments mark special relations. These lines are provided with labels denoting the speciality of the relation. Dashed lines with having longer black segments express dependency. Finally, solid lines represent data flows.

Note that in the figure those sections are denoted by gray rectangles, which are in a dominating way the results of my work.

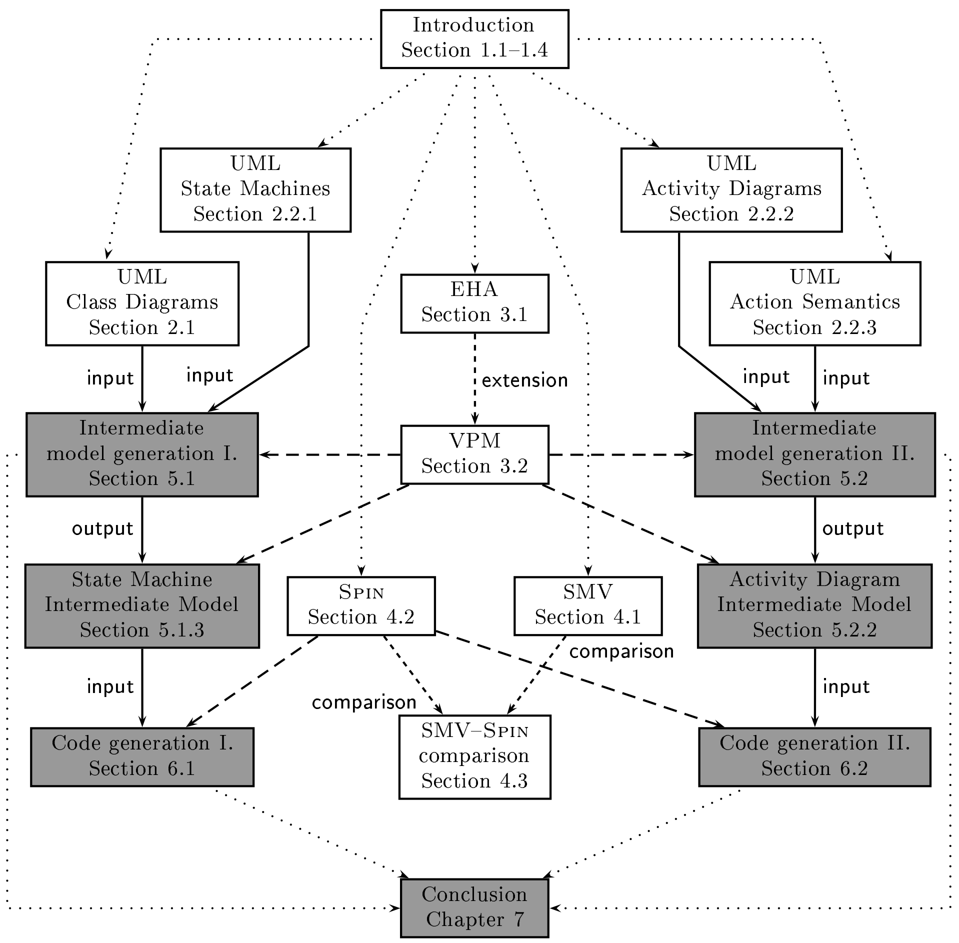


Figure 1.1: The structure of the thesis

Chapter 2

The Unified Modeling Language

This introduction is based on [20]. Modeling is the designing of software applications before coding. Modeling is an essential part of large software projects, and helpful to medium and even small projects as well. Using a model, those responsible for a software development project's success can assure themselves that business functionality is complete and correct, end-user needs are met, and program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, before implementation in code renders changes difficult and expensive to make. Surveys show that large software projects have a huge probability of failure — in fact, it's more likely that a large software application will fail to meet all of its requirements on time and on budget than that it will succeed.

Modeling is the only way to visualize your design and check it against requirements before programmers start to code.

The OMG's Unified Modeling Language (UML) (for an overview see [4], a complete reference book is [26]) helps to specify, visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements. (UML can be used for business modeling and modeling of other non-software systems too.)

The designer can model just about any type of application, running on any type and combination of hardware, operating system, programming language, and network, in UML. UML Profiles (that is, subsets of UML tailored for specific purposes) help modeling transactional, real-time, and fault-tolerant systems in a natural way.

UML defines twelve types of diagrams, divided into three categories: Four diagram types represent static application structure; five represent different aspects of dynamic behavior; and three represent ways you can organize and manage your application modules.

- Structural Diagrams include the Class Diagram, Object Diagram, Component Diagram, and Deployment Diagram.
- Behavior Diagrams include the Use Case Diagram (used by some methodologies during requirements gathering); Sequence Diagram, Activity Diagram, Collaboration Diagram, and Statechart Diagram.
- Model Management Diagrams include Packages, Subsystems, and Models.

Since our aim is to verify dynamic properties of UML models, we do not deal with Model Management Diagrams at all. This thesis only covers a subset of Structural and Behavior Diagrams. A short introduction about Class Diagrams will be given in Section 2.1.

Section 2.2.1 presents Statechart Diagrams (or State Machines). And finally, Section 2.2.2 is an overview of the notation that is used in Activity Diagrams.

We mentioned two advanced features in Section 1.1 that were added to UML to extend its expressiveness. One of the was the Object Constraint Language (OCL) and the other one is Action Semantics.

OCL will not be further discussed, since it does not have any role in defining dynamic behavior.

Action Semantics let the designer express actions as UML objects. An Action object may take a set of inputs and transform it into a set of outputs (although one or both sets may be empty), or may change the state of the system, or both. Actions may be chained, with one Action's outputs being another Action's inputs. Actions are assumed to occur independently. This concurrency model is a natural fit to the distributed execution environment of modern enterprise and Internet applications.

2.1 Static Structure Diagrams

This summary is based on [28]. A Class diagram gives an overview of a system by showing its classes and the relationships among them. Class diagrams are static – they display what interacts but not what happens when they do interact.

The class diagram below models a customer order from a retail catalog. The central class is the Order. Associated with it are the Customer making the purchase and the Payment. A Payment is one of three kinds: Cash, Check, or Credit. The order contains OrderDetails (line items), each with its associated Item.

*UML **class** notation is a rectangle divided into three parts: class name, attributes, and operations.* Names of abstract classes, such as Payment, are in italics. ***Relationships** between classes are the connecting links.*

The class diagram of Figure 2.1 has three kinds of relationships.

1. ***Association** is a relationship between instances of the two classes.* There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.

2. ***Aggregation** is an association in which one class belongs to a collection.* An aggregation has a diamond end pointing to the part containing the whole. In Figure 2.1, Order has a collection of OrderDetails.

3. ***Generalization** is an inheritance link indicating one class is a superclass of the other.* A generalization has a triangle pointing to the superclass. Payment is a superclass of Cash, Check, and Credit.

An association has two ends. An end may have a role name to clarify the nature of the association. For example, an OrderDetail is a line item of each Order.

*A **navigability** arrow on an association shows which direction the association can be traversed or queried.* An OrderDetail can be queried about its Item, but not the other way around. The arrow also lets you know who "owns" the association's implementation; in this case, OrderDetail has an Item. Associations with no navigability arrows are bi-directional.

*The **multiplicity** of an association end is the number of possible instances of the class associated with a single instance of the other end.* Multiplicities are single numbers or

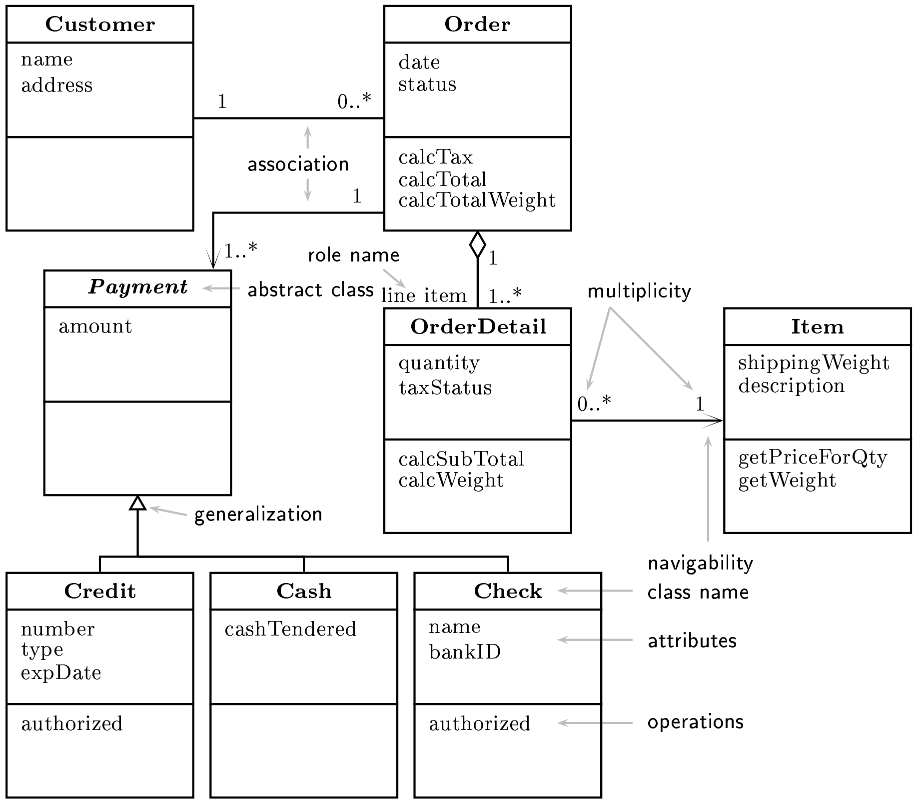


Figure 2.1: A sample class diagram

ranges of numbers. In the example, there can be only one Customer for each Order, but a Customer can have any number of Orders.

Table 2.1 gives the most common multiplicities.

Multiplicities	Meaning
0..1	Zero or one instance.
$n..m$	n to m instances.
0..*	No limit on the number of instances (including none).
*	
1	Exactly one instance
1..*	At least one instance

Table 2.1: The most common multiplicities

Every class diagram has classes, associations, and multiplicities. Navigability and roles are optional items placed in a diagram to provide clarity.

2.2 Dynamic Structure Diagrams

This section uses the same notational framework that has been described in [30] and [31]. However, the emphasis on different topics may be shifted depending on the degree of necessity they take part in the transformation. If an element is not used at all, it will be entirely excluded.

2.2.1 State Machines

State machines *can be used for modeling discrete behavior through finite state-transition systems.* In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage of protocol of part of a system. Depending on their role, they are referred as behavioral state machines and protocol state machines, respectively.

This thesis is focusing only on **behavioral state machines**. They can be used to *specify behavior of various model elements.* This kind of state machine is an object-based variant of Harel statecharts (for further details see [6]).

State Machine Syntax

Let us summarize the basic components of behavioral state machines.

A ***state*** *models a situation during which some invariant condition holds.* The invariant may represent a static situation such as an object is waiting for an external event to occur. (This representation corresponds to the traditional state concept.) On the other hand, the invariant may also model dynamic conditions such as the process of performing some activity.

A state is in general shown as *a rectangle with rounded corners*, with the state name shown inside the rectangle.

UML standard defines three kinds of states (namely **simple states**, **composite states** and **submachine states**) depending on their internal structure.

A ***submachine state*** *specifies the insertion of the specification of a submachine state machine.* A submachine state is semantically equivalent to a composite state and as such in the future they will be handled as if they were composite states. So during the rest of the thesis only the terms of simple and composite states will be used.

- *A state is a* ***simple state****, if it does not have any substates.*
- *A state is a* ***composite state****, if it contains at least one region.*

A ***region*** *is an orthogonal part of a composite state.* Each region has a (nonempty) set of mutually exclusive disjoint subvertices and a set of transitions. As a consequence, a composite state contains at least one embedded state that will be referred as a **substate** in the future.

A ***transition*** *is a directed relationship between a source vertex and a target vertex.* It may be a part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to a particular event.

A transition is shown as a *solid line* originating from the source vertex and terminated by an arrow on the target vertex.

A vertex can be either a state or a pseudostate. *A* ***pseudostate*** *is an abstraction that encompasses different types of transient vertices in the state machine graph.* Pseudostates are used to connect multiple transitions into more complex state transition paths. The different kinds of pseudostates and their graphical notations are summarized in Table 2.2.

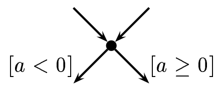
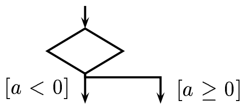
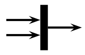
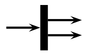
Initial	●
Shallow History	(H)
Deep History	(H*)
Entry Point	○
Exit Point	⊗
Junction	
Choice	
Join	
Fork	

Table 2.2: Pseudostate summary

For making the notation clear a state machine is presented in Figure 2.2.

State Machine Semantics

Now it is high time to go forward and to survey the semantics of state machines.

A state can be *active* or *inactive* during execution. In a hierarchical state machine more than one state can be active at the same time.

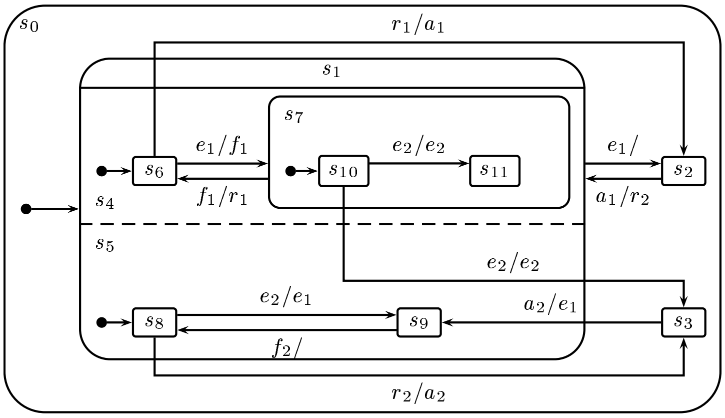


Figure 2.2: A sample state machine

- If a composite state is active and not orthogonal (i.e. has only one region), exactly one of its direct substates is active.
- If a composite state is active and orthogonal (i.e. has more than one region), then all of its regions are active. According to the previous rule, each active region has exactly one active direct substate.

As a consequence of the rules mentioned above, active states can be organized into a tree structure and are called *state configurations*. The root of the tree is always the state representing the top level state machine. Internal nodes and leaves correspond to active composite and simple states, respectively. The tree has further restrictions related to its structure. Only the orthogonal composite states can have more than one child. These child nodes correspond to the regions directly contained by the parent composite state.

Trees representing state configurations can be found in Figure 2.3. A notational guide is given in Figure 2.3(a). Simple states are marked by white circles. Light gray circles denote composite states. Striped circles indicate regions.

One can define entry and exit activities for each state. Whenever a state is entered, it executes its **entry activity** before any other activities, and conversely, whenever a state is exited, it executes its **exit activity** as the final step prior to leaving the state.

- Events are dispatched and processed by the state machine, one at a time. The order of dequeueing is not defined, leaving open the possibility of different priority-based schemes. So it can be supposed that the event queue is not part of the object itself, but belongs to the environment.
- The semantics of event processing is based on the *run-to-completion* assumption. It means that an event cannot be dequeued and processed until processing the transition caused by the previous event has not been finished completely.

  This means that activities to be executed in one run-to-completion step are not subject to concurrency issues, or in other words, these activities form an atomic process. Event processing is postponed until the next stable state configuration is reached. Consequently, if an activity invokes a method synchronously, then this delay may be fairly long, since the return has to be waited.

These assumptions make the separation of objects from the other parts of the system easier, since the state machine has to consider only the last event arrived from the event queue.

A transition is *enabled*, if and only if its source state is in the current configuration, its trigger is offered by the external environment and the guard is satisfied.

When an event is dispatched, it may result in one or more transitions being enabled for firing. If no transition is enabled, the event is discarded and the run-to-completion step is completed.

In case of orthogonal regions it is possible to fire multiple transitions as a result of the same event — as many as one transition in each region in the current state configuration. In case where one or more transitions are enabled, the state machine selects a subset and fires them. The transition selection algorithm determines which of the enabled transitions fire. The order, in which selected transitions fire is not defined.

Two transitions are said to *conflict* if the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously.

In situations where there are conflicting transitions, the selection of transitions to fire is based on an implicit priority. This priority scheme is based on the relative position of conflicting transitions in the state hierarchy. By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.

The set of transitions that will fire is a maximal set of transitions that satisfies the following conditions:

- All transitions in the set are enabled.
- There are no conflicting transitions within the set.
- There is no transition outside the set that has higher priority than a transition in the set.

The case of conflicting transitions have to be further examined and specified.

When an event is dispatched, the transitions whose label contains the specified event as a trigger have to be searched for. Furthermore, guard conditions on the label have also to be checked. After performing these steps, source states of selected transitions are to be assigned.

Any transition corresponding to a source state that is not contained in the active state configuration is not enabled and can be filtered out.

Now a set of marked states are selected in the tree. If a marked node is an ancestor of any other marked nodes, then the firing priority rule applies, so the ancestor node and the corresponding transition are thrown away.

There are still two possibilities remained.

1. In the first case, a state is marked at least twice because of the several enabled transitions leading from exactly the same state.
2. The other case is when marked states are on different branches. Each branch of a state tree corresponds to a region, enabling both transitions to fire simultaneously.

Example 1

A small example is presented for this purpose. Let the state machine of Figure 2.2 be given. As mentioned earlier, active state configuration has a tree structure. Figure 2.3(b) shows the initial configuration.

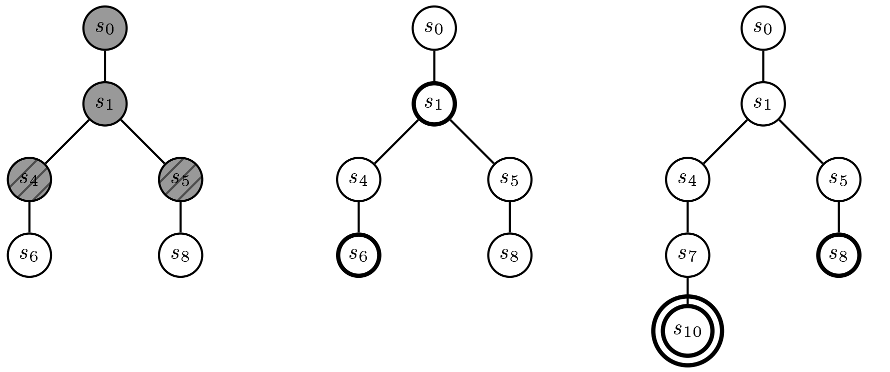
Let us suppose that an event e_1 *arrives. There are only two transitions (namely, that lead from* s_1 *to* s_2 *and from* s_6 *to* s_7*) triggered by* e_1*. These transitions do not specify any guards, so can be regarded as having been satisfied. Finally, the corresponding source states (*s_1 *and* s_6*) are to be selected. These are marked by thick circles in Figure 2.3(b).*

Despite the fact that both s_1 *and* s_6 *are in the initial state configuration, only the transition starting from* s_6 *will fire, since the other transition is filtered out by the firing priority rule. (As one can see in Figure 2.3(b)* s_1 *is an ancestor of* s_6*.)*

After having fired the selected transition the state configuration shown in Figure 2.3(c) will be active.

If the following event to be processed is e_2*, then both special cases occur.* s_{10} *is the source state of two different transitions marked by* e_2 *and this covers the first situation. The other case is represented in the relationship between states* s_8 *and* s_{10}*.*

As an overall result two transitions will fire. One that originates from s_8 *and another one starting from* s_{10}*.*



(a) Notational guide (b) Initial state configuration (c) Another state

Figure 2.3: State configuration trees

It is worth to survey the steps being executed when a transition fires. Since source and target states are not necessarily in the same depth of the configuration tree, their least common ancestor (LCA) has to be determined. After that the following steps are carried out in order:

1. The source state is properly exited. This means that one has to execute exit activities of every state that can be found along the path going from the source state to LCA. (The path includes the source state, but excludes LCA.)

2. Activities are executed in sequence following their linear order along the segments of the transition. The closer the activity to the source state, the earlier it is executed.

3. If a choice point is encountered, the guards following that choice point are evaluated dynamically and a path whose guards are true is selected.

4. The target state is properly entered. This means that the sequence of entry activity execution is determined according to the path in the configuration tree that conducts from LCA to the target state. (The path includes the target state, but excludes LCA.)

2.2.2 Activities

Activities usually appear in state machine transitions as embedded entities, but may also act independently as a possible method to describe dynamic behavior.

Activity modeling emphasizes the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors. These are commonly called control flow and object flow models. The actions coordinated by activity models can be initiated because other actions finish executing, because objects and data become available, or because events occur external to the flow. Topics in relation with actions are discussed in Section 2.2.3.

As a comparison, one may say that actions need access to data and they may perform some atomic transformations on data, but they also need sequencing. Activity models provide this framework for actions. So actions only define really low-level operations, while their temporal relationship and dependencies are determined in activities.

There are several complexity levels on which activities can be introduced.

- The **basic** level only supports modeling of sequential flow charts including control sequencing, but forks and joins are not supported at this level. However, decisions and merges are included.

- The **intermediate** level makes classic activity graph modeling possible. It includes concurrent control and data flow. It also requires the basic level.

- The **complete** level adds constructs that enhance the lower level models, such as edge weights and streaming. These enhancements are of no significant importance, especially when intended to implement the basic functionalities, so they are not further discussed.

- The **structured** level supports modeling of traditional structured programming constructs, such as loops and conditionals in addition to the basic nonstructured sequencing. It is based on the basic level and independent from intermediate and complete levels, so they can be used together.

- The **structured complete** level adds support for data flow output pins of conditionals and loops.

- The **extra structure** level supports exception handling as found in traditional programming languages and invocation of behaviors on sets of values. It requires the structured level.

The scope of this thesis does not allow us to cover everything, so our research is restricted only to basic and intermediate levels. They seem to be enough for the first time, since they contain all the basic elements that are needed for the most systems.

Activity Syntax

Let us survey the syntax of activity diagrams.

The most basic construct is the activity. *An* ***activity*** *specifies the coordination of executions of subordinate behavior, using control and data flow model.* The subordinate behaviors coordinated by activity diagrams may be initiated because other behaviors in the model finish executing, because objects and data become available, or because events occur external to the flow.

Since activities define behavior using control and data flow model, it should not be surprising that these models are graph based and as such they consist of activity nodes and activity edges. *An* ***activity node*** *is an abstract class for the steps of an activity.* It covers invocation (or executable) nodes, control nodes, and object nodes.

- *An* ***executable node*** *is an abstract class for activity nodes that may be executed.* Actions as basic (non decomposable) units performing data transformation or accessing data (for further details see Section 2.2.3) are executable nodes. Structured activity nodes such as conditionals and loops also belong to this type of activity nodes.

  Action nodes are denoted by rectangles with rounded corners.

- *A* ***control node*** *is an abstract activity node that coordinates flows in an activity.* The notations for control nodes are summarized in Table 2.3. However, they are quite similar to the pseudostates presented in Section 2.2.1, their syntax slightly differ, and thus they are discussed independently (see Table 2.3).

Initial node	●
Activity final node	◉
Flow final node	⊗
Decision node, Merge node	◇
Fork node, Join node	❙

Table 2.3: Control nodes

- *An* ***object node*** *is an abstract activity node that is part of defining object flow in an activity.* It indicates an instance of a particular classifier. For example, input pins of actions requiring objects to be produced by the previous action can be considered as object nodes.

  Rectangles are used for presenting object nodes.

An ***activity edge*** *is for directed connections between two activity nodes.* They can be divided into two categories. Both edges are denoted by arrowed lines.

- *A* ***control flow*** *is an edge that starts an activity node after the previous one is finished.* Thus it defines temporal dependencies on activity nodes.

- *An* ***object flow*** *is an activity edge that can have objects or data passing along it.*

As a rule of thumb, one may say that if at least one end of the edge is an object node then this specific edge is an object flow. If both ends are control nodes or executable nodes then the edge is a control flow. Since both edges express some kind of dependencies that affect execution order of activities, they can be handled similarly in the future.

An activity diagram is presented in Figure 2.4 in order to summarize the syntax.

Example 2
It represents the execution order of actions that are needed to be performed to increase an integer attribute. Since the attribute is embedded into an object, a reference pointing to the object (called a self reference) has to be determined first. Then the execution branches into three parts. Some steps on these paths are independent, however, others are restricted by control flows. It is trivial that the value of the attribute should be read first, and then can be destroyed or removed, and finally the new value can be inserted. (This constraint chain is emphasized in Figure 2.4 by thick lines.) There are neither direct nor transitive dependencies between Remove Structural Feature Action and Apply Function Action, so they can be executed concurrently. The only reason for using multiple Unit object nodes is to generate three copies of self reference instead of the one that is produced by the Read Self Action. Each emphasized action needs one copy of self reference to work.

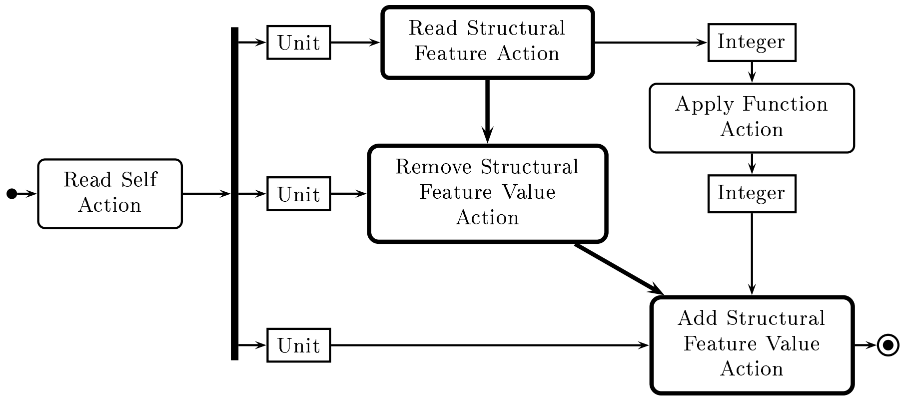


Figure 2.4: A sample activity diagram

Activity Semantics

The semantics of activities is based on token flow. By **flow**, we mean that the execution of one node affects and is affected by the execution of other nodes. These dependencies are defined by the above-mentioned edges. The term **token** refers to a single object, or locus of control that is present in the activity graph at a particular node.

When a node begins execution, tokens are accepted from some of its input edges and a token is placed on the node. When a node completes execution, a token is removed from the node and tokens are offered to some of its output edges.

All restrictions on the relative execution order of two or more actions are explicitly constrained by flow relationships. If two actions are not directly or indirectly ordered by flow relationships, they may execute concurrently.

Activities may appear in different environments. They can be embedded into an other activity, or they can turn up as methods. They may also be attached to transitions, or can sometimes appear independently. If it is not the case, then the classifier is referred to as the **context** of the activity. The activity has access to the attributes and operations of its context object and any objects linked to the context object, transitively. So navigation through public accessible links is supported.

Nodes and edges have token flow rules. Nodes control when tokens enter and leave them. Edges have rules about when a token may be taken from the source node and moved to the target node. A token traverses when it satisfies the rules for source node, edge, and target node all at once. This means that a source node can only offer tokens to the outgoing edges rather than force them along the edge, since tokens may be rejected by the edge or the target node on the other side. Since multiple edges can leave the same node, token flow semantics is subject to timing issues and race conditions.

Tokens cannot rest at control nodes. Control nodes act as traffic switches managing tokens as they make their way between object nodes and actions, which are the nodes where tokens can rest for a period of time. Initial nodes are excepted from this rule.

A data token without any value is called the **null token**. Null tokens satisfy all the type restrictions of object nodes.

After this general overview let us survey the semantics of different activity nodes.

- **Object nodes** contain data tokens at runtime. The term object is not precise, since its meaning can be extended in at least two ways. On one hand the object may also be a data value, which may be treated as an instance of a built-in class. (Data types are automatically generated in XMI descriptions by the design tools.) On the other hand, tokens are rather references to the corresponding objects and do not represent the objects themselves.

  So basicly these references are stored in object nodes. It is quite natural that an object node may have several tokens that have the same value.

  Data tokens must conform to the types of object nodes. (Or in other words, there is a type restriction on the data tokens.)

- **Initial node** is a starting point for invoking an activity. A control token is placed at the initial node when the activity starts. Tokens in an initial node are offered to all outgoing edges. If an activity has more than one initial node, then invoking the activity starts multiple flows, one at each initial node. Initial nodes are the only exception to the rule that control nodes cannot hold tokens if they are blocked for moving downstream.

- All tokens offered on incoming edges of **final nodes** are accepted.

  - **Flow final node** destroys tokens flowing in.
  - A token reaching an **activity final node** aborts all flows in the containing activity, that is the activity is terminated and the token is destroyed. Any object nodes declared as outputs are passed out of the containing activity. If there is more than one final node in an activity, the first that has been reached terminates the activity, including the flow going towards the other activity final.

- Each token arriving at a **decision node** can traverse only one outgoing edge. Tokens are not duplicated in contrast to fork nodes. Each token offered by the incoming edge is offered to the outgoing edges.

Guards of the outgoing edges are evaluated to determine which edge should be traversed. The order in which guards are evaluated is not defined. The modeler should arrange that each token only be chosen to traverse one outgoing edge, otherwise there will be race conditions among the outgoing edges. For decision points, a predefined guard 'else' may be defined for at most one outgoing edge. This guard succeeds for a token only if the token is not accepted by all other edges outgoing from the decision point.

- All tokens offered on incoming edges of a **merge node** are also offered to the outgoing edge. There is no synchronization of flows or joining of tokens.

- Tokens arriving at a **fork node** are duplicated across the outgoing edges. Tokens offered by incoming edges are all offered to outgoing edges. A token must be accepted on all outgoing edges at the same time. When it is the case the offered token is duplicated and one copy traverses each outgoing edge.

- If there is a token offered on all incoming edges of a **join node**, then tokens are offered on the outgoing edge according to the following rules:

  1. If all incoming tokens are control tokens then one control token is offered on the outgoing edge.
  2. If at least one incoming token is a data token then only the data tokens are offered on the outgoing edge.

- Semantics of **actions** is discussed in Section 2.2.3.

- For the sake of completeness semantic rules for **activity edges** are repeated. A token may traverse the edge if all rules of source nodes, edges (e.g. guards), and target nodes are satisfied at the same time.

  Since control nodes cannot hold tokens, the scope of rules to be satisfied simultaneously is extended through all the incoming and outgoing edges, until an executable or an object node is reached. Thus, a subtree whose leaves cannot be control nodes and whose internal nodes are control nodes is needed. This subtree indicates nodes and edges for which conditions must hold at the same time.

  Tokens in the intermediate level of activities can only pass along the edge individually at different times. The guard must evaluate to true for each token.

2.2.3 Actions

An ***action*** *is the fundamental unit of behavior specification.* It takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. In addition, some actions modify the state of the system in which the action executes. The values that are the inputs to an action may be obtained from the results of other actions (using activity diagrams as described in Section 2.2.2), or they may be described by instance specifications. The outputs of the action may be provided as inputs to other actions in a similar manner.

Actions are contained in activities, which provide their context. Activities provide control and data sequencing constraints among actions as well as nested structuring mechanisms for control and scope.

Actions are executable nodes in the activity diagram. They may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. An action does not begin execution until all of its input conditions are satisfied. When the action terminates, it may enable the execution of a set of successor nodes including other actions. The output of the specific action may be used by its successors.

Inputs and outputs are traversed through constructs that are called pins. They can be regarded as object nodes. This approach means that type restrictions also apply to them. Pins are generally marked by squares attached to the rectangle that corresponds to the action. However, squares may be missing, when the surroundings determine dependencies unambiguously. E.g. if there is an object node that have a predecessor and a successor action, then it is clear that the predecessor should produce an output and the successor should be served an input that conforms to the type defined by the object node. In this case there is no need for indicating input and output pins explicitly (see Figure 2.4).

Action Semantics

The steps of executing an action is as follows:

1. All the object flow and control flow prerequisites have to be satisfied. A flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions. It can be considered as an implicit join operation.

2. It consumes the input control and object tokens. Object tokens are moved to the input pins of the action. The execution may eventually begin.

3. The action continues execution. Most of them operate only on their inputs, but they can also have access to a wider context, such as variables, links, or the self object.

4. When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges, thus performs an implicit fork operation.

The actions are grouped and summarized in the following.

Invocation Actions

Call Behavior Call Behavior Action is a call action that invokes a behavior (e.g. an activity) directly. The argument values of the action are available to the execution of the invoked behavior. The execution of the call behavior action waits until the invoked behavior terminates and a result is returned on its output pin.

Call Operation Call Operation Action is an action that transmits an operation call request to the target object, where it may cause the invocation of associated behavior. If the action is synchronous, then the reply message has to be waited in order to allow the action to terminate. In this case any value that has been arrived with the reply message are put on the output pin. If the call is asynchronous, then the call operation immediately terminates, when the call request has been sent.

Broadcast Signal Broadcast Signal Action transmits a signal instance to all the potential target objects in the system, which may cause a state machine to fire or activities to be executed.

Send Object Send Object Action is quite similar to the Broadcast Signal Action. But Send Object Action sends *an object*. The target of the transmission is only a *single* object.

Send Signal Send Signal Action operates like Send Object Action, but it transmits a *signal* instead of an object. The target object is a single object again.

Structural Feature Actions

Structural feature actions operate on statically specified structural feature (e.g. an attribute) of some classifier. The action requires an object on which to act, provided at runtime through an input pin. The structural features of an object may change over time due to dynamic classification.

Clear Structural Feature Clear Structural Feature Action is a structural feature action that removes all values of a structural feature.

Read Structural Feature Read Structural Feature Action retrieves the values of a structural feature. The object whose structural feature should be read is placed on the input pin, while the values in demand are put on the output pin. The type and ordering of the output pin are the same as the specified structural feature.

Add Structural Feature Value Add Structural Feature Value Action is for adding values to a structural feature. Thus, it is for setting the value of an attribute.

Remove Structural Feature Value Remove Structural Feature Value Action removes values from structural features.

Variable Actions

Clear Variable Clear Variable Action removes all values of a variable. Variables generally appear in methods and have local scope meaning that they are destroyed, when the method completes.

Read Variable Read Variable Action retrieves values of a variable. The same rules that were presented for structural features hold for variables, too.

Add Variable Value Add Variable Value Action is for adding values to a variable.

Remove Variable Value It could not be very surprising that Remove Variable Value Action removes values from variables.

Link Actions

Links are instances of associations that have been defined between two (or more) classes.

Create Link Create Link Action is for creating links. An additional input pin is required for specifying the insertion point of the link. This action has no output pins, because links are not necessarily values that can be passed to and from actions.

Destroy Link Destroy Link Action destroys links.

Read Link Read Link Action navigates across associations to retrieve objects on one end. Navigation requires the source end of the link to be specified. There must be at least one end that should not be specified.

Other Actions

Apply Function The function is not defined in the framework of UML. Apply Function Action invokes a primitive predefined function that computes output values based only on the input values and the function. The execution does not have access to object memory or to any objects. The execution of the primitive function has no side effects on any other object.

Clear Association Clear Association Action destroys all links of an association in which a particular object participates. This action has a statically specified association end. It has an input pin for a runtime object that must be of the same type as at least one of the association ends of the association.

Create Object Create Object Action creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime.

Destroy Object Destroy Object Action not surprisingly destroys objects that have to be placed on the input pin.

Read Self Read Self Action retrieves the host object of an action. Every action is a part of some activity, which is attached in some way to the specification of a classifier — for example as the body of a method or as a part of a state machine. This action does not have any inputs, however, the host instance is placed on the output pin, whose type is the classifier to which the activity is statically associated.

Test Identity Test Identity Action determines whether two values are identical objects.

Chapter 3

A Mathematical Modeling Framework

Verification tools require a mathematical description of systems to be investigated. As it was stated earlier in Section 1.1, UML lacks mathematics, or more precisely it does not provide a clear semantics formulated in the language of mathematics.

So a mathematically precise framework is needed, to which UML diagrams presenting dynamic behavior can be mapped. There already exist such descriptions (see [18] for example), each having their own benefits and deficiencies. Some of these frameworks are discussed in the present chapter. Moreover, several approaches [13,19] exist that transform UML State Machines to PROMELA code.

The concept of extended hierarchical automata (EHA) is presented first in this chapter as being one possible framework. Then the subject is changed to a recently published visual and precise metamodeling (VPM) framework that

- is again **mathematically precise**,
- contains **more general concepts** than EHA,
- makes possible **to merge several semantics to be represented in a unique framework** and, thus,
- is **a possible applicant to form the base of a verification tool independent description**.

3.1 Extended Hierarchical Automaton

The notation of this section follows the notation that was used in [13]. Syntax and semantics of UML State Machines were discussed in Section 2.2.1. As it is presented in [13] state machines were transformed into EHA in the first step.

Hierarchical automata are an abstract syntax for state machines in the sense that they abstract from the purely syntactical details and describe only the essential aspects of the state machine. They are composed of simple sequential automata related by a refinement function. A state is mapped via the refinement function into the set of automata which refine it.

The sample state machine of Figure 3.1 is mapped into the hierarchical automaton of Figure 3.2. There is a clear correspondence between the states of the two structures. Initial

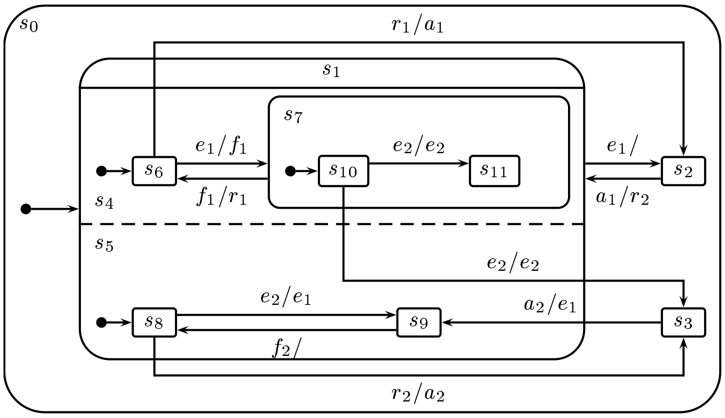


Figure 3.1: The state machine of Figure 2.2

states are denoted by thick boxes. The refinement of a state into one or more substates is represented by a refinement function ϱ. $\varrho(s_1) = \{ A_1, A_2 \}$, $\varrho(s_7) = \{ A_3 \}$ and $\varrho(s) = \emptyset$ for any other state s. State refinements are marked by dashed arrows.

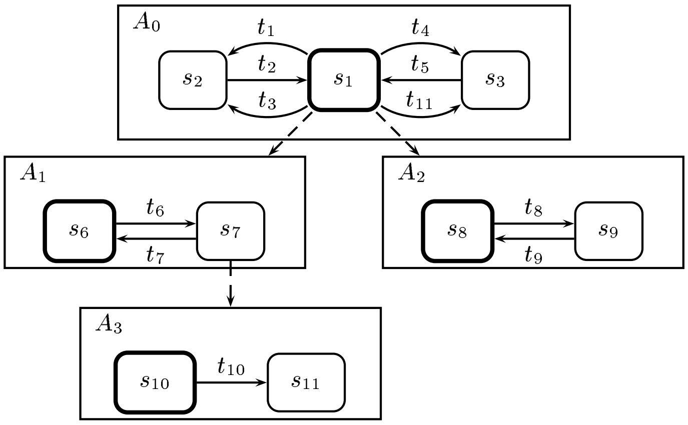


Figure 3.2: A sample hierarchical automaton

Transitions whose source and target nodes are in the same embedding depth are called non-inter-level transitions. (In other words one may say that a transition is a non-inter-level transition if its graphical representation (the corresponding arrow) does not cross any state border.) $t_2, t_3, t_6, t_7, t_8, t_9$ and t_{10} are such transitions. They are represented in the natural way.

A label is attached to each transition. They contain information about the triggering event, the guard condition and the activities to be executed. (EHA restricted the structure of activities in such a way that they could only send signals.)

Inter-level transitions are handled in a somewhat more difficult way, since the above-mentioned labels are extended. They also specify source restrictions and target determinators, which are sets of states. Arrows that correspond to inter-level transitions are drawn into the automaton of the least common ancestor (LCA) of the source and target states. All source and target configuration states below the LCA are used as source restrictions and target determinators, respectively.

If the target state of the transition is a compound state, then its initial state in the embedded automaton has to be entered somehow. This is also solved by placing the initial state(s) into the target determinator set.

Example 3
Let us have a look at an example. Consider first the non-inter-level transition t_2 going from s_2 to s_1. (The transition is a non-inter-level one, since s_1, s_2 and t_2 are directly *(and not transitively) inside the top-level automaton (s_0).) As a consequence, the source restriction set will be empty. The triggering event is e_1, there is neither a guard, nor an activity on the transition. Since the target state s_1 is a compound state, the initial states of embedded state machines (A_1 and A_2) have also to be entered. As a solution, the target determinator set that is chosen to be $\{ s_6, s_8 \}$.*

Let us select the inter-level transition t_1 now. It goes from s_6 to s_2, thus the corresponding arrow crosses the border of s_1. It is obvious that the trigger is r_1, the activity is to send a_1 and there is still no guard condition. The LCA of s_6 and s_2 is s_0 (or the corresponding A_0 in the hierarchical automaton), so the arrow should go in A_0 between s_1 and s_2. But having only s_1 as a source state would cause loss of information, so basic state s_6 has to appear in the set of source restriction. Since target state s_2 is a simple state the target determinator will be empty.

Conclusion

+ The construct of hierarchical automata gives a precise mathematical basis from which a transformation may generate the input of a verification tool.

– It does not deal with activities that are more complex than just sending a signal. Furthermore, activities inside states (e.g. entry and exit activities) are also missing.

– The use of refinement function is a straightforward solution for handling states only, but becomes a bit clumsy, when it is wanted to be extended for other structures.

3.2 A Visual and Precise Metamodeling Framework

It seems to be self explanatory that if the advantages of a description are preserved but its deficiencies are also eliminated then a better solution is resulted. Thus our aim is to preserve precision, while extending the notation towards activities.

The authors of [33] proposed a visual and mathematically precise metamodeling framework (VPM) that seems to be useful in specifying verification tool independent models (see Section 5.1.3 and 5.2.2). These models will be referred as intermediate models (IM), since they are situated between the UML input and the verification tool output. The mathematical basis of IMs are built on the VPM framework, so a short introduction cited from [33] is given about the topic.

A naming convention, which simultaneously refers to UML and mathematical elements, is used in VPM. A **model element** in VPM may be either an **entity**, a **connection**, or a **mapping** (see the MOF metamodel of this approach in Fig. 3.3). **A unique identifier** (accessed by a *.id* postfix in the sequel) *and* **a set** including the *identifier of the model element* and the *identifiers of all the (intended) refinements of the element* (accessed by a *.set* postfix) are related to each of this constructs. For mathematical reasons, the set associated to a model element should also contain the identifier of the element.

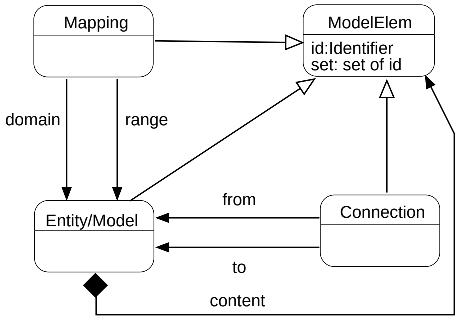


Figure 3.3: The metamodel of VPM

- An **entity** E is a set (called as **basic entity** in this case) or a tuple (denoted as **compound entity** or **model**) consisting of sets, relations, functions and tuples (a collection of entities, connections, and mappings, respectively). Entities will be represented visually either by UML classes or UML packages while the notion of containment will be captured by graphical containment (e.g., classes inside a package) or aggregations (leading from entities to both entities, connections and mappings) depending on the context to provide the better match with the conventional notation.

- A **connection** R between two entities is a binary relation between the associated sets or tuples. Connections are depicted as (directed) associations.

- A **mapping** F from entity E_1 to entity E_2 is a function with the domain of (the set of) E_1 and range of E_2. Mappings can be denoted visually by an attribute assigned to the entity of its domain with an attribute type corresponding to its range.

Distinct notions of entities and models are merged into a single entity construct, which idea stems from the fact that a one-dimensional tuple (consisting of only a single set) can be regarded as a set, thus a certain redundancy is eliminated from the underlying mathematical framework of this approach. In an object-oriented term, a uniform class concept is used for both classes and packages (models).

A refinement calculus for inheritance and instantiation

The static semantics of the VPM framework is based upon a unifying refinement calculus, which captures the notion of inheritance and type-instance relationship (depicted by UML generalization and instance-of relations, respectively) between arbitrary metalevels *without actually defining the notion of metalevels.*

Definition 1 (Comparison of elements)
A model element P (i.e., either entity, connection or mapping) is less than (or equal to) a model element Q (denoted as $P \leq Q$) iff $P.set \subseteq Q.set \wedge P.id \in Q.set$ thus, if the related set of P is a subset of the corresponding set of Q and the identifier of P is contained by the set of Q, which is an ordering relation combining the subset and set containment relations.

For the notational convention of Def. 2, let $E^{(n)}$ denote an entity consisting of exactly n subcomponents (in case of $n = 1$, the entity is regarded to be *basic*, otherwise *compound*) where $E[i]$ accesses the ith component (argument) of entity E. $R(A, B)$ refers to a connection R between entities A and B, while $F(A, B)$ denotes a mapping with the domain of entity A and range of entity B.

The general refinement relation between model elements Sub and $Super$ will be denoted as $Sub \sqsubseteq Super$ symbol, which means that Sub is a refinement of $Super$ (for the use of the terminology, see Table 3.1). Refinement is a unified relation that is either an inheritance $\Mapsto$ or an instantiation $\mapsto$ (or both).

	sub is ... *super*	*super* is ... *sub*
$\sqsubseteq$: refinement = inheritance + instantiation	refinement of	abstraction of
$\Mapsto$: inheritance (subtype)	inherited from (subtype of)	generalization of (supertype of)
$\mapsto$: instantiation (type-instance relation)	instance of	type of

Table 3.1: Notation guide for refinement, inheritance and instantiation

Definition 2 (Refinement calculus)
The refinement ($\sqsubseteq$) rules of the VPM framework (that simultaneously handle inheritance $\Mapsto$ and instantiation $\mapsto$) are as follows.

1. **Basic entity refinement**: $E^{(n)}_{sub} \sqsubseteq E^{(1)}_{super} \stackrel{\text{def}}{=} E^{(n)}_{sub} \leq E^{(1)}_{super}$, thus if E_{super} is a simple entity (one-dimensional tuple consisting only of a set) then refinement is defined as the "less-than" relation. Informally, E_{super} is either a generalization or a type of E_{sub} (which is either a class or a package).

2. **Connection refinement**: $R_{sub}(A_{sub}, B_{sub}) \sqsubseteq R_{super}(A_{super}, B_{super}) \stackrel{\text{def}}{=} R_{sub} \leq R_{super} \wedge A_{sub} \sqsubseteq A_{super} \wedge B_{sub} \sqsubseteq B_{super}$ (where all A_i and B_i are entities). Connection inheritance expresses the fact that associations can also be refined during the evolution of metamodels in addition to the refinement of classes.

3. **Mapping refinement**: $F_{sub}(A_{sub}) : B_{sub} \sqsubseteq F_{super}(A_{super}) : B_{super} \stackrel{\text{def}}{=} F_{sub} \leq F_{super} \wedge A_{sub} \sqsubseteq A_{super} \wedge B_{sub} \sqsubseteq B_{super}$ From a practical point of view, the refinement of attributes is also handled in the VPM framework (similarly to classes and associations).

4. **Mapping is a connection**: $F(A_{sub}) : B_{sub} \sqsubseteq R(A_{super}, B_{super}) \stackrel{\text{def}}{=} F \leq R \wedge A_{sub} \sqsubseteq A_{super} \wedge B_{sub} \sqsubseteq B_{super}$, i.e., functions can be interpreted as special relations.

5. **Compound (model) refinement**: the refinement of compound entities (or models/packages) is explicitly split into the inheritance and instantiation case, thus $E^{(n)}_{sub} \sqsubseteq E^{(k)}_{super} \stackrel{\text{def}}{=} E^{(n)}_{sub} \Mapsto E^{(k)}_{super} \vee E^{(n)}_{sub} \mapsto E^{(k)}_{super}$

   (a) **Compound entity (model) inheritance** $E^{(n)}_{sub} \Mapsto E^{(k)}_{super} \stackrel{\text{def}}{=} E_{sub} \leq E_{super} \wedge \forall i \exists j : E_{sub}[j] \Mapsto E_{super}[i]$. Informally, there exists a subtype relation for each argument of E_{super} in a corresponding argument of E_{sub}.

(b) **Compound entity (model) instantiation** $E_{sub}^{(n)} \mapsto E_{super}^{(k)} \stackrel{\text{def}}{=} E_{sub}.set \subseteq E_{super}.set \wedge \forall i \exists j : E_{sub}[i] \mapsto E_{super}[j]$. Informally, there exists a *type* element for each component of E_{sub} in a corresponding component of E_{super}.

The most crucial consequence of these definitions is that the handling of refinement (inheritance and instance-of) relations is identical for basic entities, connections and mappings (while there is a certain orthogonality for compound entities/models). As a result, *two model elements can simultaneously be in subtype and instance-of relations.*

In order to obtain a mathematically complete definition of the refinement calculus (that encapsulates the metametamodel of Fig. 3.3 within the framework of VPM), a top section of the inheritance and containment hierarchy is introduced as follows.

Definition 3
The model space of VPM framework will always consists of (at least) the following elements.

- *The abstract model element Universe is greater than all the other model elements (entities, connections and mappings), thus being the root of both the inheritance and the containment hierarchy.*
- *The Entity is contained by (and refined from) Universe.*
- *The Connection is leading from and to Entity.*
- *The Mapping is leading from and to Entity.*

Thereafter, any well-formed model space has to fulfill the following axioms.

Property 1 (Inheritance and instantiation is partial order)
Each element in the model space (except for Universe) has at least one supertype, and both refinement relations (inheritance and instantiation) are reflexive, transitive *and* antisymmetric.

Informally, multiple inheritance and instantiation are allowed but circularities are therefore forbidden in the type hierarchy. Note that the definition formally permits that an element is inherited from (alternatively, instance of) itself, which only eases the mathematical treatment of the framework without foregoing consequences on the intuitive meaning.

Property 2 (All model elements are contained)
Each element in the model space (except for Universe) is contained by at least one element, moreover, the containment relation is transitive.

As a consequence, multiple and circular containment are thus allowed by this axiom, and each element should be reachable from the top element by navigating containment relations.

Conclusion

As it was discussed earlier in Section 1.4, the transformation (that will be presented in Chapters 5 and 6) consists of two separate steps. Between these steps an intermediate model (IM) is generated. The attribute intermediate refers to its position, since IM is situated between the UML input and the verification tool output.

The VPM framework that has just been presented will give the mathematical basis of intermediate models. ***The main feature of VPM that is wanted to be exploited is its ability to capture several metamodeling levels.*** The usage of VPM framework is discussed in exhaustive details, when the transformation steps are described in a formal way (see Sections 5.1.3 and 5.2.2).

Chapter 4

An Overview of Model Checking Tools

As it was written in [1], model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model.

It is quite straightforward, that the mission of model checking tools is to automatically verify, whether $M \models \phi$ holds, where M is a (finite-state) model of a system and property ϕ is stated in some formal notation.

Verification tools (like e.g., PVS [22], SMV [16] or Spin [7]) help the designer to notice flaws that get into the system model in the design phase. These flaws may consist of deadlocks, livelocks (or starvation), underspecification, overspecification, violation of constraints, and wrong assumptions about the speed.

Two of such tools will be presented. Section 4.1 will give an overview about the SMV tool, while Section 4.2 presents Spin. The reason for covering several tools is mainly historical. My previous attempts used SMV, while the recent concept comprises the use of Spin. As a consequence it gives an opportunity to compare these tools.

4.1 The SMV Model Checker

SMV was originally designed for hardware verification as it can be seen from its syntax covering every detail. So its input has to be specified in a low abstraction level. For a tutorial see [15]. A complete reference is given by [16].

Some advantages of SMV are enumerated below.

- Requirements can be stated as *temporal logic* expressions. It makes possible to formulate the state combinations to reach or to avoid compactly. For more information on temporal logic have a look at [25].

- *Compositional verification* is also supported. It is a kind of verification, when the results of the verification executed on several subsystems are reused when proving properties in a larger and more complex system.

- SMV also observes the *symmetry* of the system and reduces the state space on the basis of symmetry. It takes care not to lose any meaningful state, of course.

- Finally, SMV may use *inductional proof* for verifying properties.

Syntax of SMV There are three simple types in SMV.

- *Boolean variables* (`boolean`) may have true or false as values.
- The members of *sets* are to be listed between braces.
  `{elem1,elem2,elem3}`
- The last simple type is the *interval* (denoted by `7..0`).

So integers, characters and strings that are well-known types in many programming languages are missing from SMV resulting in a clumsy way of handling such variables in the verification phase.

It is possible to use either one dimensional, or multi dimensional *arrays*. They are compound types, since they can be put together from simple types.

- Each member in the array may have the same type as in the traditional approach. For example an array of three Boolean variables indexed from 0 to 2 can be defined as `array 0..2 of boolean`.
- But SMV also supports the concept of *generic arrays*. The types of members may differ from each other in a generic array. For example one may define an array of three members whose types are the three simple types.

```
a : array 0..2;
a[0] : boolean;
a[1] : {elem1,elem2,elem3};
a[2] : 7..0;
```

The input of SMV can be built up in a modular way. *Modules* are a collection of definitions (type declarations and assignments) that can be reused later. Module definitions may also have parameters. When a copy of a module is instantiated, then signals declared as parameters are linked into the other parts of the program, so only references are used in different modules when the same variable is wanted to be accessed.

Values have to be assigned to outputs and to state variables as a function of inputs and previous state variable values in each module just as in the whole program.

Modules can be embedded into each other. The embedded module can be regarded as a compound type afterwards.

The inputs of an embedded module are the outputs of embedding module at the same time, and vice versa, the output of the embedded (inner) module forms input to the embedding module on the other hand.

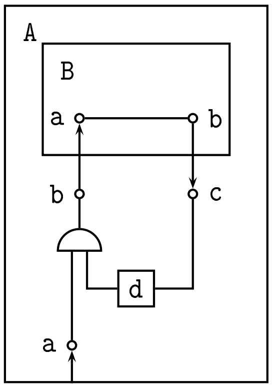
An example for the concept is presented in Table 4.1.

Inputs and outputs should be marked in the type declaration part of modules by `INPUT` and `OUTPUT` keywords, respectively. Variables declared without keywords form the group of state variables.

The `:=` operator can be used for assignments, while the following value of a variable `var` can be accessed as `next(var)`. The latter can obviously appear only on the left side of assignments. The initial value of a state variable `var` can be accessed with the `init(var)` notation.

Expressions appearing on the right side of assignments may contain

- integer and symbolic constants,




```
module B (a,b) {
   INPUT a : boolean;
   OUTPUT b : boolean;

   b := a;
}

module A (a) {
   INPUT a : boolean;
   b : boolean
   c : boolean;
   d : boolean;
   mod : B(b,c);

   b := a & d;
   init(d) := 0;
   next(d) := c;
}
```

Table 4.1: Merging modules and the corresponding SMV code

- logic operators (`&`, `|`, `~`, `^`, `->`, `<->`),
- conditional operators, (`?:`, `if`, `switch`, `case`),
- arithmetic operators (`+`, `-`, `*`, `/`, `mod`, `«`, `»`),
- comparison operators (`=`, `~=`, `<`, `>`, `<=`, `>=`) and
- operators related to set operations
  (`in`, `{e1,e2}`, `{f(i) : i = 1..5}`).

Integer constants are in a bit vector representation in SMV.

4.2 The Model Checker SPIN

During this section a lot of things has been cited from [7]. SPIN is a generic verification system that supports the design and verification of asynchronous process systems [8, 9]. SPIN verification models are focused on proving the correctness of *process interactions*, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these. In focusing on asynchronous control in software systems, rather than synchronous control in hardware systems, SPIN distinguishes itself from other well-known approaches to model checking, e.g., [1, 10, 14].

As a formal methods tool, SPIN aims to provide:

1. an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail, and

2. a powerful, concise notation for expressing general correctness requirements.

In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanically by the tool. SPIN accepts design specification written in the verification language PROMELA (a Process Meta Language) [8], and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [25].

There are no general decision procedures for unbounded systems, and one could well question the soundness of a design that would assume unbounded growth. Models that can be specified in PROMELA are, therefore, always required to be bounded, and have only countably many distinct behaviors. This means that all correctness properties automatically become formally decidable, within the constraints that are set by problem size and the computational resources that are available to the model checker to render the proofs. All verification systems, of course, do have physical limitations that are set by problem size, machine memory size, and the maximum runtime that the user is willing, or able, to endure. These constraints are an often neglected issue in formal verification.

This part of the section is a summary that is based on [27]. Let us summarize the features of the PROMELA (Process Meta Language).

- It allows for the *dynamic creation of concurrent processes.*
- It supports *both synchronous and asynchronous communication* via message channels.
- It resembles the programming language C.
- It is worth to note that PROMELA is a specification language to model *finite-state systems.*

The consequence of the last point is that data structure of infinite size cannot be specified and one cannot have an infinite number of resources.

A PROMELA model consists of:

- type declarations
- variable declarations
- channel declarations
- process declarations, and
- the `init` process.

Types. PROMELA has five different (integer) basic types (`bit`, `bool`, `byte`, `short`, `int`). Variables of these basic types have a default initial value set to 0.

Compound types can also be defined in several ways.

1. If several variables of the same type are put together, the well-known **array** construct is resulted. (E.g., `byte a[2]`)
2. The other possibility is to construct **records** using the `typedef` keyword.

As a basic feature, PROMELA detects type conflicts at runtime.

Variables. Variables should be declared first. Then they can be given a value by assignment (denoted by `=`), argument passing, or message passing. Assignment and argument passing operate exactly like in programming languages. The only special thing is the message passing, since it is related to communication between processes.

Variables can be used in expressions. Most arithmetic, relational and logical operators of C and Java are supported by PROMELA.

Channels. Processes communicate each other via **channels**. Channels may also be considered as queues or buffers.

The syntax of channel declaration is:

```
chan <name> = [<dim>] of <t1>, <t2>, ... <tn>;
```

<t$_i$> denotes the type of the ith component of the messages that can pass through the channel called `<name>`.

As mentioned earlier PROMELA supports both synchronous and asynchronous communication. Message passing realizes asynchronous communication. In order to implement it, a channel with at least one element (`<dim>`≥ 1) have to be declared.

Message sending can be initiated with a statement of the following form:

```
ch ! <expr1>, <expr2>, ... <exprn>;
```

The values of <expr$_i$> should correspond to the types of the channel declaration. The send statement is executable if the channel is not full.

Message receiving may have the following forms:

```
ch ? <var1>, <var2>, ... <varn>;
```

In this case, if the channel is not empty, the message is fetched from the channel and the individual parts of the messages are stored in the variables.

The other case corresponds to **message testing**.

```
ch ? <const1>, <const2>, ... <constn>;
```

If the channel is not empty and the message at the front of the channel evaluates to the individual <const$_i$>, the statement is executable and the message is removed from the channel.

The synchronous communication is realized as a rendez-vous synchronization. In this case the number of elements in the channel is zero (`<dim>`$= 0$).

If a send statement is enabled and if there is a corresponding receive that can be executed simultaneously and the constants match, then both statements are enabled and take the transition together.

Processes. A **process** is defined by a `proctype` definition, which consists of:

- a name
- a list of formal parameters
- local variable declarations and
- a body.

Processes are defined in a way that is similar to function declaration in programming languages, but their semantics differ from functions. Processes execute concurrently with all other processes, independent of their speed of behavior. They may also communicate with each other using either global (shared) variables or channels.

Processes can be created using the `run` statement at any point in the execution. The `run` statement returns the process identifier. Statements of processes can be executed after the `run` statement. It is quite straightforward that there may be several processes of the same type.

Each process has its own local state. The local state consists of the process counter and local variables.

Statements. The process body consists of a sequence of **statements**, which are separated by a semi-colon (`;`). A statement is

- **executable**, if it can be executed immediately, and
- **blocked**, if it cannot be executed.

The following basic rules hold for statements.

- An assignment is always executable.
- An expression (that is also a statement) is executable, if it evaluates to non-zero.
- The `skip` statement is always executable. It is the no operation statement in PROMELA.
- The `run` statement is only executable if a new process can be created.
- A `printf` statement is always executable (but is not evaluated during verification).
- The `assert(expr)` statement is always executable. If `expr` evaluates to zero, then SPIN exits with an error, as the expression has been violated. This statement is often used to check whether certain properties are valid in a state.
- The `if` statement has the following syntax:

  $$\begin{array}{l}\texttt{if}\\ \texttt{::}\ \texttt{choice}_1\ \ \texttt{->}\ \ \texttt{stat}_{1,1}\texttt{;}\ \texttt{stat}_{1,2}\texttt{;}\ \texttt{stat}_{1,3}\texttt{;}\ldots\\ \texttt{::}\ \texttt{choice}_2\ \ \texttt{->}\ \ \texttt{stat}_{2,1}\texttt{;}\ \texttt{stat}_{2,2}\texttt{;}\ \texttt{stat}_{2,3}\texttt{;}\ldots\\ \texttt{::}\ \ \ldots\\ \texttt{::}\ \texttt{choice}_n\ \ \texttt{->}\ \ \texttt{stat}_{n,1}\texttt{;}\ \texttt{stat}_{n,2}\texttt{;}\ \texttt{stat}_{n,3}\texttt{;}\ldots\\ \texttt{fi}\end{array}$$

  If there is at least one $\texttt{choice}_i$ (guard) executable, then the `if` statement is executable and SPIN non-deterministically chooses one of the executable choices. If no $\texttt{choice}_i$ is executable, the `if` statement is blocked.

  The operator `->` is equivalent to the semi-colon (`;`) operator. By convention, it is used within `if` statements to separate the guards from the statements that follow the guards.

  There is a special guard that is called `else`, which becomes executable if none of the other guards is executable.

- The `do` statement has a syntax that is similar to the `if` statement.

```
do
:: choice1  ->  stat1,1; stat1,2; stat1,3;...
:: choice2  ->  stat2,1; stat2,2; stat2,3;...
::  ...
:: choicen  ->  statn,1; statn,2; statn,3;...
od
```

  With respect to the choices, a `do` statement behaves in the same way as an `if` statement. However, instead of ending the statement at the end of the chosen list of statements, a `do` statement repeats the choice selection.

- The `break` statement is always executable and exits a `do` loop statement and transfers control to the end of the loop.

As it was mentioned earlier PROMELA processes execute concurrently. Processes are scheduled in a non-deterministic manner and they are interleaved meaning that statements of different processes do not occur at the same time.

All statements are atomic, so each statement is executed without interleaving with other processes.

Each process may have several different possible actions enabled at each point of execution. In this case only one choice is made randomly (in a non-deterministic way).

Some more sophisticated statements.

- The `atomic` statement can be used to group statements into an atomic sequence. In this case all statements are executed in a single step.

```
atomic { stat1; stat2; ... statn }
```

  The `atomic` statement is executable if $\mathtt{stat}_1$ is executable. If a $\mathtt{stat}_\mathtt{i}$ (with $i > 1$) is blocked, the atomicity token is temporarily lost and other processes may do a step.

- The `d_step` statement is a more efficient version of `atomic`, since no intermediate states are generated and stored.

```
d_step { stat1; stat2; ... statn }
```

  The `d_step` block may only contain deterministic steps, so it is a perfect solution for performing intermediate computations in a single transition. It is a run-time error, if $\mathtt{stat}_\mathtt{i}$ ($i > 1$) blocks.

  Both `atomic` and `d_step` can be used to decrease the number of states of the model.

- PROMELA does not have real-time features, so one can only specify functional behavior. But it is sometimes required to be able to define timeouts e.g., to avoid deadlocks. SPIN also has this feature, but in a somewhat special form.

  The `timeout` statement becomes executable if there is no other process in the system which is executable, so it models a global timeout. Thus `timeout` provides an escape from deadlock states.

- The `goto` statement transfers execution to a label that can be defined, of course, for each PROMELA statement.

`Init` process. The whole procedure of model checking begins with the `init` process. Its role is to initialize global variables and to start other processes.

Verification using Spin. Since Spin is a model checking tool, one should be able to verify properties of the model that has been constructed so far. With Spin one may check the following type of properties:

- **Deadlocks.** In case of proving deadlock freedom, the user wants to be sure that the system never reaches a state where no further actions are possible.

  Spin tries to find a trace leading to a deadlock situation. If there is not such a trace, then the system is deadlock free.

- **Assertions.** Assertions are expressions that have to be true, when the execution reaches the `assert` statement.

- **Unreachable code.** It is also a safety property like deadlock freedom. Verification of unreachable code may be performed with a technique that has already been presented related to deadlocks.

- **LTL formulae.** They are composed of propositional logic expressions and temporal operators. Spin supports the temporal operators always, eventually and until (denoted by `[]P`, `<>P`, and `P U Q`, respectively).

- **Liveness properties.** Liveness properties are somewhat different. In this case one has to prove that something good (e.g., termination of the execution) will eventually happen.

  Spin tries to find an infinite loop in which the specified thing does not happen. If there is not such a loop, the liveness property is satisfied.

The above-mentioned property types can be formulated in several ways. For further information see [27] for example. Optimization algorithms of Spin are not covered either by this thesis. For a detailed introduction to this topic see [27] again.

4.3 Tool Comparison

Now the comparison of the two model checking tools is presented.

- SMV was designed for hardware verification and as such it requires a very detailed input model description. Some constructs are missing from the SMV model description language, however they are well-known from the software design. One such item is the *FIFO queue*, whose behavior has to be specified manually as a module, while Spin allows the user to use FIFO queues. (The communication channel can be regarded as a FIFO.) The other deficiency in SMV is the missing *integer variable*. (The only possibility to handle integer variables is to represent them as bit vectors, which is not pretty comfortable.) Contrary to SMV, Spin supports both FIFO queues and integer variables and as such fits better for modeling softwares.

- Different approaches are used for handling *transitions*. In SMV, the following values of state variables and outputs have to be specified based on the present values of state variables and inputs. In case of the Spin model checker, transitions are defined in the form of processes that can be executed concurrently. However, processes also operate on state variables that consist of variables and channel contents.

- The third major difference turns up related to *non-determinism*. SMV only supports to make a non-deterministic choice from a set when specifying the next state. Even the `if` and `do` statements allow non-determinism in SPIN. Furthermore, the following process that will step forward is selected randomly from the group of processes that stand just before an executable statement.

Among similarities one may mention that both tools

- support temporal logic formulae to draw up properties,
- support compositional verification, and
- use several techniques to decrease the state space to work efficiently.

On the grounds of the previous properties it seems straightforward that I have switched to SPIN, since the model structure of its input fits better for modeling softwares.

Chapter 5

An Integrated Formal Representation of Behavioral UML Models

The transformation can be divided into two parts. The input of the whole process is the UML model, while the output is a program that can be used as an input to a model verification tool (e.g., SMV). There is an intermediate model (IM) between the two steps. (IMs were introduced in Section 3.2.) Its role is to provide a verification tool independent description in a mathematically precise way.

The concept of building first the intermediate model facilitates code generation for different verification tools, since only the second part of the transformation has to be implemented again. Furthermore, the intermediate model is compact enough to be able to represent the traces of counterexamples generated by the verification tool.

5.1 Intermediate Model Generation from State Machines

Since this section is highly based on the notation presented in Section 2.2.1, it may be useful to refresh our memory and to survey the notation again.

The basic idea intended to use in the first part of the transformation is to map every State Machine construct to a node, which will be further transformed into a state in the second step. It has several advantages.

- States and activities can be handled in **a uniform way.**
- This uniform handling corresponds to the state based semantics of SPIN.
- Statements and properties can also be formulated for **dangerous activities**, and not only for states and regions.

5.1.1 A Summary of Node Types

It seems to be quite natural that *each state of a state machine should be mapped to an individual node* in the intermediate model. This was basically the only mapping, that existed in the transformation to EHA described in [13].

Each transition should be executed in one uninterruptible (the so-called run-to-completion) step. As mentioned earlier in Section 2.2.1, a transition may have a triggering event, a guard condition and an activity attached. Triggering events may be dispatched and processed, when a system is in a stable state configuration, and guards should be checked before the system leaves the source state of the transition. Furthermore, all the implicitly defined remaining tasks such as selecting enabled transitions or determining firing priorities are to be done in a stable configuration.

So activities are the only elements that have to be executed, while the system is between two stable configurations. Entry and exit activities may also exist in UML State Machines. The main drawback of UML State Machines published by OMG is that the semantics is given in a textual (and thus informal) way. As a consequence, the order of activities belonging to a single run-to-completion step is expressed in a textual form again instead of having a formal representation. So the first part of the transformation should take care of sequencing activities in order to result in a precise intermediate model. In order to ease sequencing *each activity is mapped to a new node* in the intermediate model, and a new node representing activity in the metamodel level is also introduced. Or more precisely, since an activity may appear in several run-to-completion steps, the same amount of new nodes have to be generated in the IM. The intermediate model has to be able to distinguish between activities appearing in different run-to-completion steps.

Activities inside states can be handled in the same way as it was just presented. The only difference is that the corresponding nodes in the IM have to be placed into a separate layer. The layer structure of IMs is further discussed in Section 5.1.2.

Since pseudostates are a subtype of states in the context of state machines, they form a node type in the intermediate model. Moreover, *each pseudostate is mapped to a node* in the IM. (The subtypes of pseudostates may also appear as separate node types if needed.)

The last major node type that can be distinguished is the region. They behave like compound states. But if several regions are embedded into the same state, then each region belongs to the active configuration until the embedding state is active. (In order to emphasize the contrast, if normal states are embedded into a compound state, then **only one** embedded state can be active at the same time.) In other words they provide the only way of having concurrent behaviour in state machines. When performing the first transformation step, *each region is mapped to an individual node.*

State Machine construct	Intermediate model entity
State	Node
Region	Node
Activity	Node
Pseudostate	Node

Table 5.1: Summary for the first part of the transformation

Table 5.1 summarizes the one-to-one mappings that make up the first part of the transformation. Note that transformation mappings are formulated in the metamodelling level.

The four node types form two groups depending on whether they can process events. *States that allow event processing are called* ***active states***. Regions, simple, and compound states are active states. It is worth to extend the term active to state configurations. *A* ***state configuration*** *is* ***active****, if all the states belonging to the configuration are active.*

In contrary, *states that cannot process events are said to be* ***passive***. So pseudostates and activities are passive. In a similar manner, one may say that *a* ***state configuration*** *is* ***passive****, if at least one state in the configuration is passive.*

Thus, the starting point and the endpoint of run-to-completion steps are marked by active state configurations, while passive configurations can be found along a run-to-completion step.

Let us summarize the node types of intermediate models in Table 5.2.

Main group	Node type	Comment
Active	State	Both simple and compound states belong to this group.
	Region	Regions directly correspond to their UML state machine equivalent.
Passive	Activity	Their notational extension is presented in Section 5.2.
	Pseudostate	They can be further divided into smaller groups, if needed. (E.g., initial nodes, history nodes, etc.)

Table 5.2: Major node types

5.1.2 The Layer Structure of Intermediate Models

A multilayer structure is built up for the intermediate model. Layers are embedded into each other just like states in state machines. If a state B is embedded into A in a state machine, then the corresponding nodes of the intermediate model should be in a similar relationship (i.e. node B should be below node A).

Let us number the layers in a top-down order according to their level of embedding. The top-level state machine appears in the first layer. The second layer contains entry and exit activities of the top-level state machine. The states that are directly embedded into the top-level state machine are put to the third layer, while the corresponding entry and exit activities appear in the fourth layer. The concept is shown in Figure 5.1 on a sample state machine. (Note that Figure 5.1 does not tell anything about the exact intermediate model. It only demonstrates how the layer structure is constructed.)

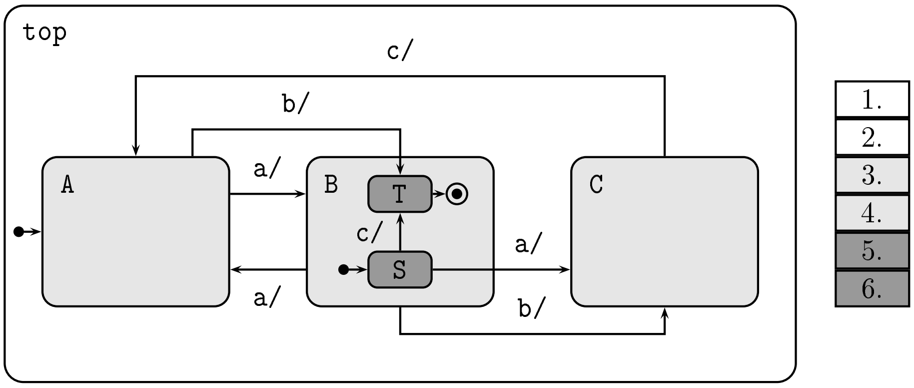


Figure 5.1: A demonstration example

Since triggers and guards only formulate restrictions on when to leave active state configurations, they do not have to be mapped into nodes in the IM. They are simply transformed into labels (or constraints) of arrows leading from an active state.

Now the reflections can be generalized.

- Even-numbered layers consist of nodes that correspond to entry and exit activities of the states represented in the layer that is one level higher.

  Let us suppose that we have a hypothetical state S in our state machine. If in-state activities are defined for S, then S is transformed into a structure that is presented in Figure 5.2. So if S was in the embedding level k in the original state machine, then its image will go into the $(2k+1)$. layer, and all its in-state activities will be mapped into nodes in the $(2k+2)$. layer.

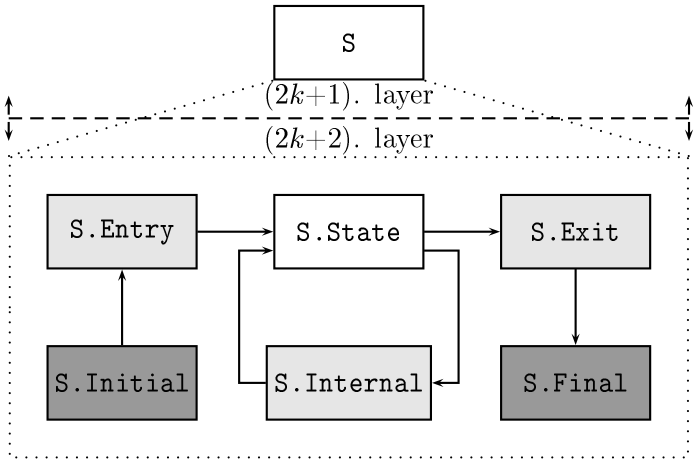


Figure 5.2: The output of the first transformation step in the even layers

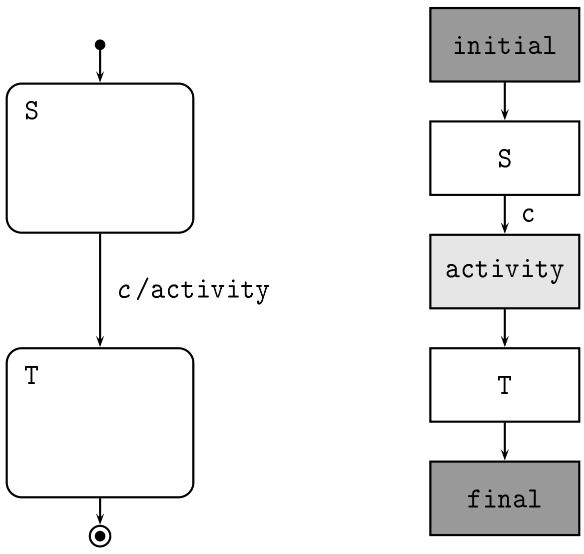
- Odd-numbered layers contain simple and compound states. States of embedding level k go to the $(2k+1)$. layer. As a consequence, LCA of source and target states of a transition is also in an odd layer. In this case, the node that corresponds to the transition goes to the next odd layer that is below the LCA. (Thus, the difference of the layers of LCA and the transition is always two.)

  Let us pick out the state machine of the 3rd embedding level of Figure 5.1. This state machine is repeated in Figure 5.3(a) for the sake of clarity. Note that neither the transition from A to S, nor the one from T to C is presented on this level, since the LCA of both transitions is the top-level state machine, and as a consequence they have to be handled together with the second embedding level.

  Figure 5.3(b) shows the output of the first transformation step. Simple states are mapped into the corresponding nodes, the activity into a separate activity node, and the two pseudostates into the two dark grey nodes, respectively. The whole structure of Figure 5.3(b) will go into the $(2k+1)$. layer of the IM.

It is worth to say some words about the notation of Figures 5.2 and 5.3, which are still not complete intermediate model representations. White rectangles denote active states regardless of their exact node types. Grey rectangles mark passive states. In this case pseudostates and activities are distinguished by using dark grey and light grey rectangles, respectively.

Up to this point I have supposed that entry and exit activities have been defined for every state and thus even-numbered layers have always been generated in the intermediate



(a) UML State Machine

(b) A Simplified Representation of the Intermediate Model

Figure 5.3: The transformation of odd layers

model. It does not seem to be a pretty correct assumption, so the idea that has been used need to be slightly refined.

If there are no in-state activities defined for a simple state `x`, then all we have to do is just to drop all the children of `x` from the (even-numbered) layer.

If one has a compound state (or a region), then its children are located in an even layer in the IM. There has to exist a child that is called `x.State`, where `x` denotes the parent node. Since `x` is a compound state (or a region), a subtree has to be hanging from `x.State`. If the children of `x` are dropped, then the whole subtree originally hanging from `x.State` have to be moved upward to its parent `x`.

It is still possible that not all, but only some of the in-state activities are defined. In this case, all the nodes that correspond to undefined activities may be omitted, however, nodes for defined activities still remain in the even layer.

5.1.3 Formal Representation of the Intermediate Model I.

I use a formal representation that is based on the VPM framework [33], which has been shortly summarized in 3.2.

Definition 4
*Let the **intermediate model** $IM = (V, E)$ be a graph, where V and E denote the set of nodes and edges, respectively. In the future, nodes are refer as **entities** and edges will be called as **connections** (or relations).*

In Section 5.1.1 I have defined a couple of so-called node types. Now each node type is represented as an entity, thus belongs to the set V. Each entity is identified in an unambiguous way, e.g., by an integer identifier (for further details see [33] again). So the

IM at a bare minimum consists of the entities listed in Table 5.3. Note that the table does not show the so called Universe model element.

Id	Node type	Id	Node type
e_1	Entity	e_{10}	Guard
e_2	SM State	e_{11}	Initial
e_3	Active	e_{12}	Shallow History
e_4	Passive	e_{13}	Deep History
e_5	State	e_{14}	Junction
e_6	Region	e_{15}	Choice
e_7	Activity	e_{16}	Join
e_8	Pseudostate	e_{17}	Fork
e_9	Event	e_{18}	Final

Table 5.3: Basic entities I.

The notation used for entities slightly differs from the notation of VPM. Entities are denoted in graphical environment by circles holding their unique identifiers inside. Entity identifiers appear in subscripts in textual contexts (just like in Table 5.3).

As stated in Section 5.1.1 there was some kind of **inheritance** (denoted by $\Rrightarrow$ in texts and by arrows with hollow triangle arrowheads in figures) between node types. E.g., every region is active, and this can be described as $e_6 \Rrightarrow e_3$. In other words one may say that the concept of active state is a more general concept than the region. Figure 5.4 summarizes the knowledge having been gathered so far.

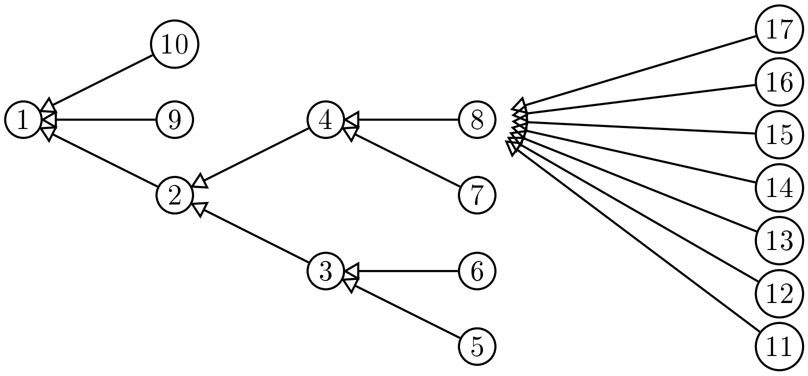


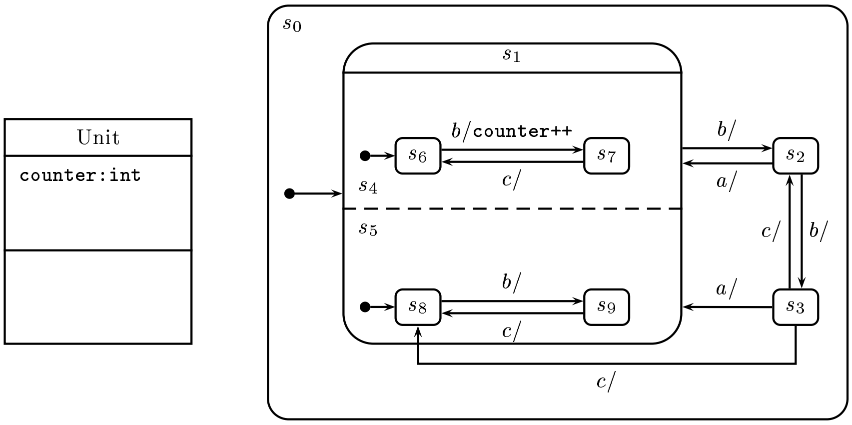
Figure 5.4: Inheritance in the IM

It is worth to note that the inheritance relation is transitive, so e.g., $(e_6 \Rrightarrow e_3 \wedge e_3 \Rrightarrow e_2) \implies e_6 \Rrightarrow e_2$ resulting in a relation between region and state of a state machine meaning that every region is a state in a state machine at the same time.

The framework of [33] also mentions a binary **instantiation** relation (denoted by $\mapsto$ in textual environment and by arrows with dashed lines in graphical representation) on the set of entities.

Example 4

As a working example let us consider the UML model presented in Figure 5.5. Let s_1 be



(a) Class diagram (b) State machine

Figure 5.5: The Unit class

the only state that has an entry and an exit activity. These activities are not depicted in the figure.

One may pick a lot of new entities. E.g., s_0 is mapped to e_{19}, s_1 to e_{20} and so forth. Since e_{19} is a state instance, the instantiation relation $e_{19} \mapsto e_5$ holds. e_{23} corresponds to region s_4, so $e_{23} \mapsto e_6$ also holds.

The VPM of [33] allows us to have user-defined connections. For this purpose an **inclusion** (denoted by a line that has a rhombus at the container edge) is defined. Inclusion relation expresses a situation, in which a state is completely inside an other state. E.g., state s_6 is in region s_4, so e_{23} includes e_{25}, where the entities correspond to the original constructs of the state machine.

The way how inclusion and instantiation is represented in the intermediate model is depicted in Figure 5.6.

The other user-defined relation is the **transition** relation (denoted by arrows with labels marking the serial number of parameters). It represents an atomic step in the state machine. Note that a single run-to-completion step may consists of several atomic steps. As it was discussed earlier, triggering events and guards may be attached to transitions. Because of this, the transition relation have four parameters, namely a source state, an event, a guard and a target state. Or one may say it in a formal way as $T \subseteq (Q \times Ev \times G \times Q)$, where

- $\forall v \in Q, \exists x \in V : v \mapsto x \overset{*}{\Rightarrow} e_2$
- $\forall v \in Ev, \exists x \in V : v \mapsto x \overset{*}{\Rightarrow} e_9$
- $\forall v \in G, \exists x \in V : v \mapsto x \overset{*}{\Rightarrow} e_{10}$

In other words, Q, Ev, and G contain those nodes that have been instantiated from a class that is inherited from state machine states (e_2), events (e_9), and guards (e_{10}), respectively.

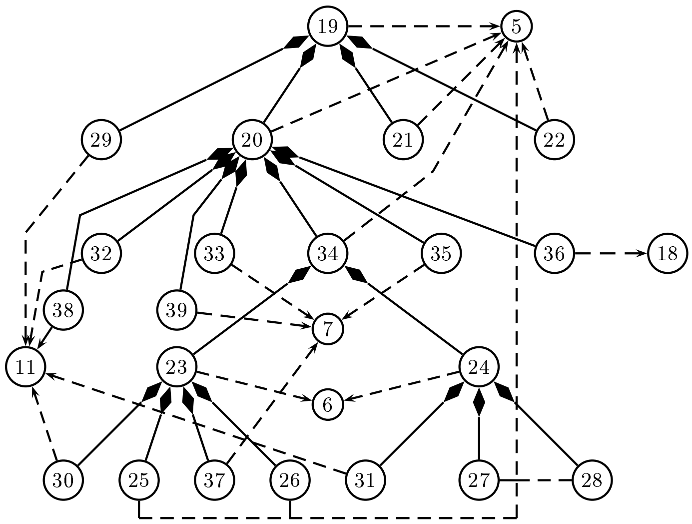


Figure 5.6: Inclusion and instantiation in the IM

The original study only defines binary relations, but also provides a way to extend connections having more than two parameters.

The second and the third parameter of the transition relation may be empty. A further constraint can be formulated for the event parameter. It has to be empty, if the source state is passive. (It is not allowed to process an event in a passive state configuration.)

In order to avoid a chaotic figure the following simplifications are applied. Since there are no guard conditions in the state machine of Figure 5.5(b), they will be completely omitted. In contrast with the standard notation, I will use arrows from the source node to the target node labelled by the event to arrive (see Figure 5.7).

(a) Standard notation

(b) Simplified notation

Figure 5.7: Simplifying the notation of transition relation

Now the transition relation is presented in Figure 5.8.

Figures depicting the intermediate model need some further comments. The most important is that the union of Figures 5.4, 5.6 and 5.8 form the precise intermediate model. However, some obvious instantiations are still missing, because of the simplified transition notation used in Figure 5.8. If one had not used the simplified notation, one would have had separate nodes for each event and some further edges denoting instantiation between the new nodes and e_9 and marking that events are instances of the Event class.

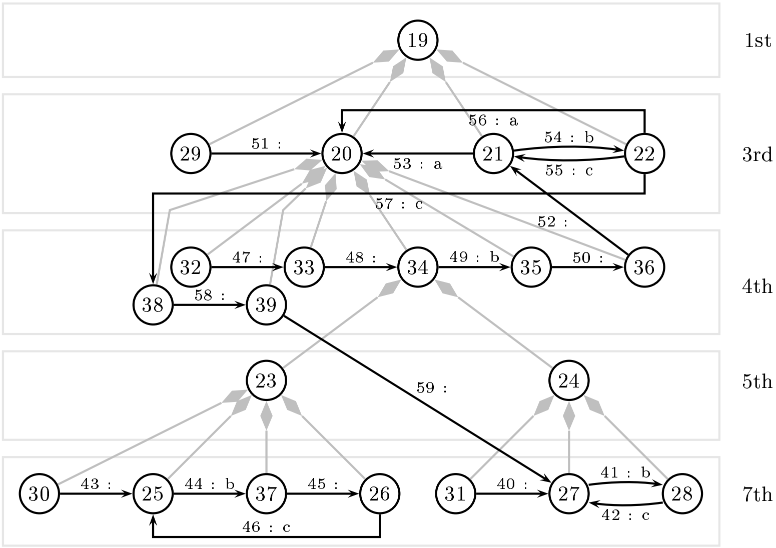


Figure 5.8: Transition and inclusion in the IM

The careful Reader may notice that the path $e_{22}, e_{38}, e_{39}, e_{27}$ representing the step leading from s_3 to s_8 seems to be an irregular notation. On one hand, entering region e_{23} and initial state e_{30} is an implicit consequence of the run-to-completion step ending at e_{27}. Since s_4 and s_5 are concurrent regions embedded into state s_1, they should be entered at the same time, even if it is not stated explicitly.

On the other hand, nodes e_{33} and e_{39} correspond to the same construct (namely to the entry activity of state s_1) in the original UML model. (Similarly, e_{32} and e_{38} correspond to $s_1.Initial$.) The only reason for having several nodes mapped to the same construct is to distinguish between paths on which the same activities have to be executed.

Initial states of compound nodes are not denoted explicitly, they have three distinctive features.

1. They are included in the compound node.

2. They are instances of Initial State class (e_{11}).

3. They are not the target states of any transition relation.

Finally, I would like to point out that all unnecessary nodes were dropped before Figures 5.4, 5.6 and 5.8 have been prepared. As a consequence, there are layers that are not represented at all in Figure 5.8. (E.g., there is no 2nd layer, since state s_0 does not have any in-state activities defined.) In order to have a well arranged figure, nodes of different layers are encircled by light gray rectangles.

State configuration trees (depicted in Figure 2.3) also appear in the VPM based intermediate models. They are designated by the inclusion relation.

5.2 Intermediate Model Generation from Activity Diagrams

I would like to refer to Section 2.2.2 in which the role of activities was discussed. As it has already been stated activities usually appear in state machine transitions as embedded entities, but may also act independently as a possible method to describe dynamic behavior. It is obvious that the first case is more complicated.

Our aim is to construct a precise description for activity diagrams that can be attached to a state machine transition as an atomic step. If generating such a description is succeeded, it can also be used in cases when activities appear as independent dynamic behaviors.

5.2.1 Petri Net Representation of Intermediate Models

Activity semantics is based on token flow, so I have chosen a Petri net like structure to describe dynamic behavior in activities. Further details on Petri nets can be found in [24] for example.

Definition 5
*The basic **Petri net structure** is a 4-tuple $N = (P, T, E, W)$ where P is a set of places, T is a set of transitions, E is the set of edges and can be defined as $E \subseteq (P \times T) \cup (T \times P)$. Finally, $W : E \mapsto N^+$ denotes the weight function. A further constraint on sets P and T that they have to be disjoint ($P \cap T = \emptyset$).*

In other words the Petri net structure is a directed bipartite graph with weights on its edges. Sets P and T form the two groups of nodes needed to define a bipartite graph.

Places may hold tokens. The role of tokens is to define the states of the corresponding automaton. This is done by counting the number of tokens in each place.

Definition 6
*The **Petri net** is a pair $PN = (N, M_0)$, where N is a Petri net structure. $M_0 : P \mapsto N^+$ denotes the initial marking. The M_0 function defines the number of tokens that can be found in the places in the beginning.*

So let us survey the basic constructs of the intermediate model to be generated.

- The concept of **places** is preserved in my model, but also slightly extend it. Several types of places are needed. The reason of this will be discussed later in this section. (There are classes of Petri nets that also make this step, i.e. allow places to hold tokens with identities.)

- The concept of **transitions** also appears in the IM. Constraints are needed to be formulated when a transition is not allowed to fire. These constraints may contain statements about pre-places and post-places.

- **Edges** of IMs are exactly the same as in Petri nets.

- **Weight function** assigns 1 to each edge by default. Any special cases will be pointed out explicitly.

- Two types of **tokens** are used. One that has identity and an other one that does not.

Object nodes. The mapping of object nodes into constructs in the IM is quite simple. Object nodes are transform to places that may hold tokens with identity. Since object nodes may have type restrictions attached, these restrictions also have to appear in the intermediate model. (Tokens inside such places must conform to the corresponding type.)

Figure 5.9: Transformation step by step I. (Object nodes)

Actions. Actions are embedded into activities. There are some special requirements for such activities.

1. An implicit join has to be executed before starting an action. This operation is mapped into a transition in the intermediate model. Each incoming edge of the action is transformed to an edge that ends at the new transition. The inputs of the action will correspond to places holding tokens with identity. These places can be reached on outgoing edges starting from the new transition. (If the action has no inputs, then a place without identity is created.)

   The newly-introduced transition has a further restriction. It cannot fire until there is a token inside the subnet representing action execution.

2. An implicit fork has to be executed when an action has been finished. This operation is also mapped into a transition in the intermediate model. This new transition will be a part of the action specific subnet. The outputs of the action will correspond to places holding tokens with identity. These places can be reached on outgoing edges starting from the new transition. Furthermore, a new place is generated for each control flow edge leaving the action. These new places are also connected to the new transition as it is shown in Figure 5.10.

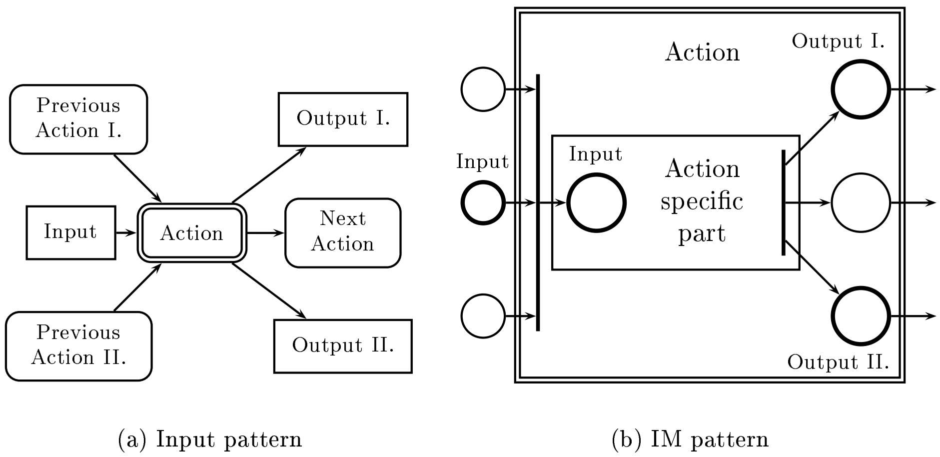
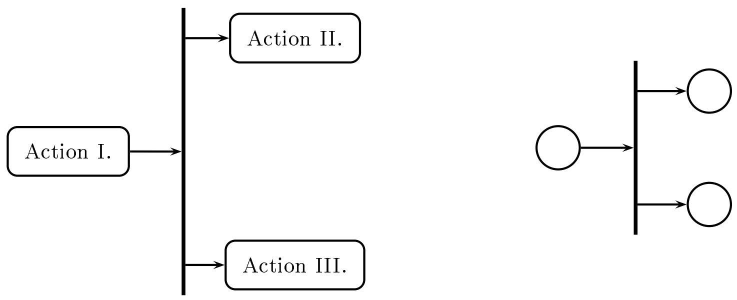


Figure 5.10: Transformation step by step II. (Actions)

Fork nodes. Fork nodes are transformed to transitions (see Figure 5.11). These transitions should have at least as many outgoing edges as incoming edges.

- If the incoming edge starts from an activity node that is not an object node, then all outgoing edges must end in a similar (not object) node.

- If the incoming edge starts from an object node, then all outgoing edges must lead to an object node of a conforming type.



(a) Input pattern

(b) IM pattern

Figure 5.11: Transformation step by step III. (Fork nodes)

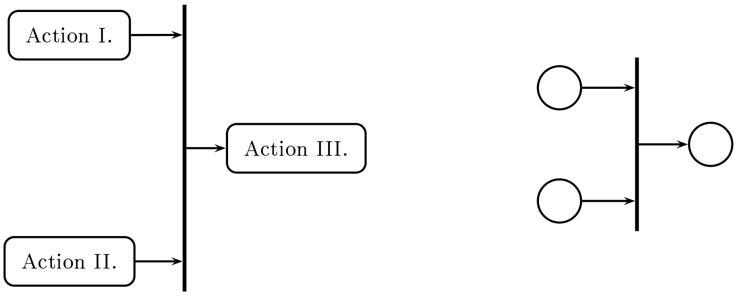
Join nodes. Join nodes are transformed to transitions. These transitions should have at least as many incoming edges as outgoing edges.

- If none of the incoming edges is an object node (see Figure 5.12), then the outgoing edge should lead to an executable node. (Places that one gets after the transformation should also meet this requirement. The outgoing edge of the new transition may end in a place holding unidentified tokens, if all the incoming edges start from the same place type.)

- If at least one incoming edge of the new transformation starts from a place that holds tokens with identity (see Figure 5.13) and, thus, whose origin is an object node in the Activity Diagram, then the outgoing edge must end in a similar place, whose type conforms to all preplaces with identity.

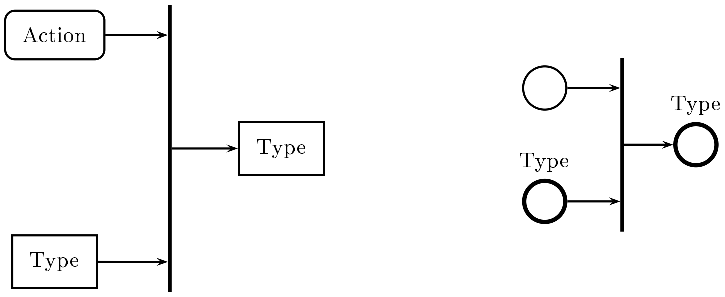
Initial nodes. Initial nodes are mapped to places holding tokens without identity (see Figure 5.14).

Merge nodes. Each merge node is transformed into a couple of transitions (see Figure 5.15). The number of incoming edges leading to the merge node has to be equal to the number of new transitions that have just been created.



(a) Input pattern (b) IM pattern

Figure 5.12: Transformation step by step IV. (Join nodes)

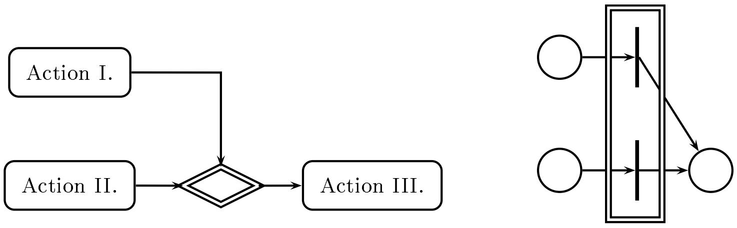


(a) Input pattern (b) IM pattern

Figure 5.13: Transformation step by step V. (Join nodes)

(a) Input pattern (b) IM pattern

Figure 5.14: Transformation step by step VI. (Initial nodes)



(a) Input pattern (b) IM pattern

Figure 5.15: Transformation step by step VII. (Merge nodes)

Decision nodes. Each decision node is transformed into a couple of transitions just as in the case of merge nodes (see Figure 5.16). The number of new transitions in the IM should be equal to the number of outgoing edges starting from the decision node. But the transformation of decision nodes is somewhat more difficult, since there are two cases that cannot be managed in the framework of activity diagrams.

1. Each new transition must have some kind of guard condition that expresses the same restriction that has been specified on the labels of outgoing edges of the decision node. So the intermediate model has to be able to handle guard condition attached to transitions. This requires some actions to be executed on tokens representing some variable without actually moving the token itself.

   So this is a situation that has to be handled with care when it is time to implement the model as a verification tool input.

   The common preplace of the newly introduced transitions will hold tokens with identity, since only this type of places allows us to access attributes and to navigate through links that is essential when guard conditions wanted to be checked.

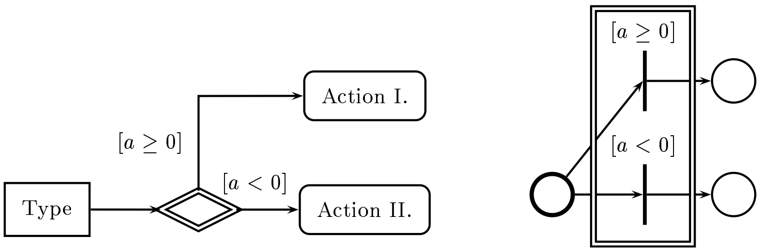
2. Activity diagrams allow users to specify activities that produce values that can be used in guards without explicitly presenting them in the diagram. These activities can be treated as a standard compound executable node during the transformation (see the following paragraph).

Compound executable nodes. *Executable nodes that are not actions are referred as* ***compound executable nodes****.* They give the opportunity to embed activities into each other and to construct complex activities. These nodes are transformed to a subnet while taking care to reach its initial node when entering the compound state, and to leave the compound state when the activity final has been reached.

5.2.2 Formal Representation of the Intermediate Model II.

The formal representation uses the same notation as I used in Section 5.1.3.

There are several constructs that have been mentioned so far. The notation is summarized in Table 5.4.



(a) Input pattern (b) IM pattern

Figure 5.16: Transformation step by step VIII. (Decision nodes)

`Id`	Node type	`Id`	Node type
e_1	Entity	e_{68}	Place for object tokens
e_7	Activity	e_{69}	Place for control tokens
e_{60}	Node	e_{70}	Place for tokens of Unit type
e_{61}	Place	e_{71}	Place for tokens of Integer type
e_{62}	Transition	e_{72}	Read Self
e_{63}	Edge	e_{73}	Apply Function
e_{64}	P2T Edge	e_{74}	Read Structural Feature
e_{65}	T2P Edge	e_{75}	Remove Structural Feature Value
e_{66}	Activity Final	e_{76}	Add Structural Feature Value
e_{67}	Guard	e_{77}	Action

Table 5.4: Basic entities II.

Note that only a subset of actions have been enumerated in Table 5.4. These are the nodes that will be used in the following example.

Nodes with identifiers less than 60 mark the same nodes that have been mentioned so far in Table 5.3.

Inheritance relation between the node types of activity diagrams is presented in Figure 5.17.

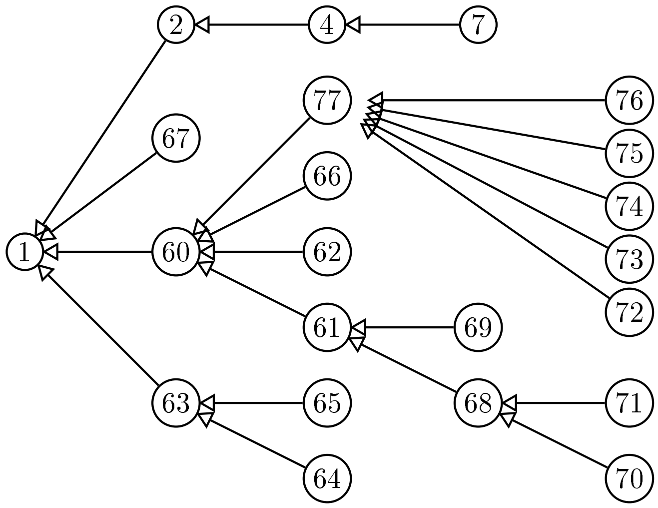


Figure 5.17: Entity refinements between activity diagram constructs

Example 5

Let us consider the sample activity diagram of Figure 2.4. Using the transformation steps presented in Section 5.2.1 a Petri net is resulted. This net is depicted in Figure 5.18.

One may also notice that the intermediate model can be further simplified, if it is required to decrease state space. E.g., places that can be reached from transitions with a single incoming and outgoing edge seem to be unnecessary. (Two such places exist in Figure 5.18.)

In order to get the precise intermediate model description of the example, one new node for each graphical element of Figure 5.18 is introduced. Thus I insert

- *4 nodes ($e_{78} \dots e_{81}$) for holding tokens of Integer type,*
- *4 nodes ($e_{82} \dots e_{85}$) for holding tokens of Unit type,*
- *4 nodes ($e_{86} \dots e_{89}$) for places holding control tokens,*
- *5 nodes ($e_{90} \dots e_{94}$) for representing actions,*
- *9 nodes ($e_{95} \dots e_{103}$) for transitions,*
- *one node (e_{104}) for representing the completion of the activity,*
- *12 nodes ($e_{105} \dots e_{116}$) for expressing edges that lead from places to transitions,*

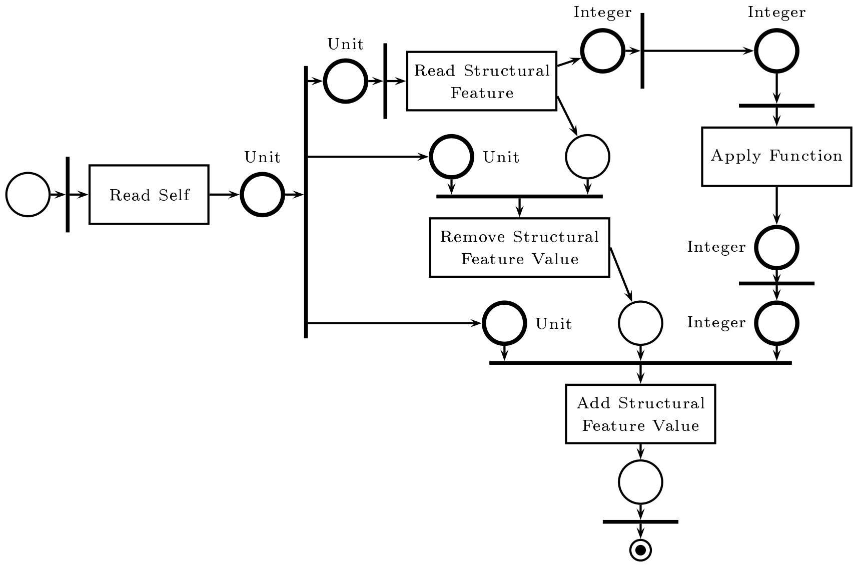


Figure 5.18: Petri net representation of the corresponding IM

- *17 nodes ($e_{117} \dots e_{133}$) for expressing edges that lead from transitions to places. It is worth to note that edges that are related to actions in any way belong to this group.*

Nodes of each group are instances of (or in instantiation relation with) node types defined in Table 5.4. E.g., $\forall x \in \{ e_{78}, \dots, e_{81} \} : x \mapsto e_{71}$, $\forall y \in \{ e_{82}, \dots, e_{85} \} : y \mapsto e_{70}$, and $\forall z \in \{ e_{86}, \dots, e_{89} \} : z \mapsto e_{69}$. The only exception is the group representing actions, since in this case each new node is an instance of a separate node type. E.g., $e_{90} \mapsto e_{72}$, and $e_{91} \mapsto e_{73}$.

It is obvious that the relationship could have also been depicted in a graphical and complete way, but it would have resulted in a confused figure without any benefit, so the illustration has been omitted.

Earlier the so-called inclusion relation was defined. In this context every new node is included into the embedding activity (e.g., e_{37}), so one should introduce one new pair for each new node. The first member would always be the new node and the other member is always the node that corresponds to the embedding activity.

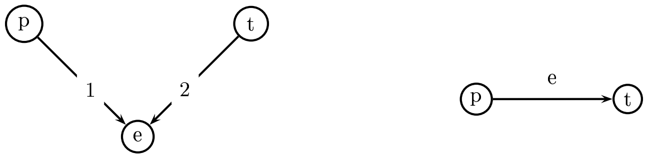
The graphical representation of the inclusion relation would not offer us any useful information, so it is omitted again.

As one could see the inclusion relation defined a structure with exactly two layers. One that contained all the new nodes and an other one containing only the embedding activity. If the activity has a compound executable node then the number of layers increases, since nodes embedded into this specific compound node go into a lower layer.

There are some new relations to be introduced to make our description complete.

1. First of all, **edges** (instances of e_{64} and e_{65}) connect places and transitions, so it may be regarded as a binary relation. In order to get a well arranged figure, the graphical notation is simplified again (see Figure 5.19).

   In the standard notation one would have one place node, one transition node, and one node for representing the edge that joins the nodes. Furthermore, there would be two labelled lines that specify the direction of the original edge.



(a) Standard notation (b) Simplified notation

Figure 5.19: Simplifying the notation of edge relation

The simplified notation only uses two nodes (one for representing the place and an other one for the transition), and an edge with an arrow showing the direction. The edge also has a label to mark its identifier.

2. One should be able to define guards and to attach them to transitions. This **guard attachment** is also a binary relation between a guard and a transition node.

Example 6
It is high time to present the remaining part (edge and guard attachment relations) of the precise intermediate model of the activity diagram. Since the original activity diagram did not contain any decision nodes, we do not have any guards and any guard attachments. It is shown in Figure 5.20.

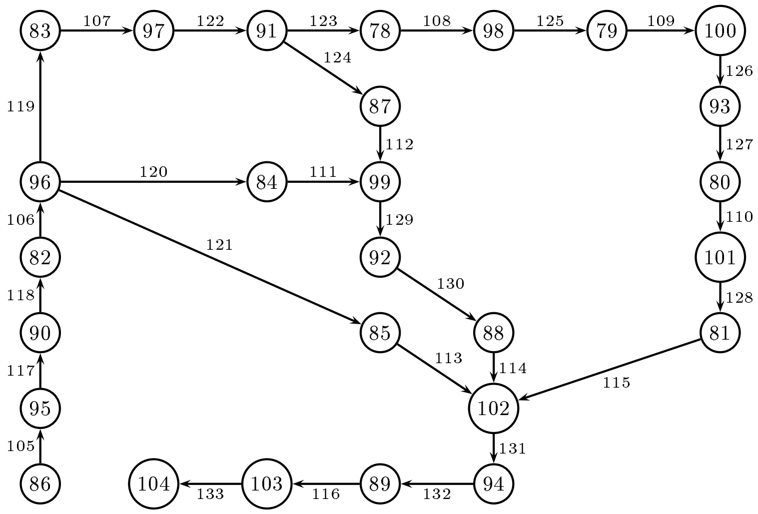


Figure 5.20: Edge relations in the IM

As a starting point we have chosen the activity presented in Figure 2.4. Its intermediate model representation was depicted through several figures including 5.17 and 5.20. There were some aspects (e.g., inclusion and instantiation) that were mentioned, but have not been presented with a graphical notation. The complete intermediate model that corresponds to the original activity consists of the union of all the aspects we have discussed.

But this was only a single activity. There can be, of course, many more activities in the UML model. We will generate new nodes for each activity when constructing the intermediate model. Note that all the activities share the nodes that were enumerated in Table 5.4, since in UML terms one would say that they are the part of the metamodel and not the model itself.

The intermediate model generated for activities (activity intermediate model (AIM)) should also appear in the intermediate model that was generated for state machines (state machine intermediate model (SMIM)) and was presented in Section 5.1.3. The link between the IMs is provided by the inclusion relation, since all the nodes in the activity intermediate model must be a descendant of the corresponding activity node in the SMIM.

As one could see in Figure 5.8, the SMIM had a tree structure, when a filter is used that only allows inclusion edges (grey lines in Figure 5.8) to appear. The root of the tree was the node that represented the top-level state machine. There can be found some leaves ($e_{33}, e_{35}, e_{37}, e_{39}$) that represent activities.

AIMs also generated a tree structure along the inclusion relation lines for each activity. The roots of these trees are the nodes that represent the whole activity. It is obvious that the root nodes of AIM trees should be merged together with the corresponding leaf nodes of the SMIM.

Chapter 6

Code Generation for the Spin Model Checker

At this point we have a complete and, thus, large intermediate model description covering class diagrams, state machines and activity diagrams of the UML notation. Our goal is now to generate verification tool input. This code generation has to be correct, and furthermore the correctness has to be proven.

After the features of two model checking tools (for the comparison of SMV and Spin see Section 4.3) had been compared I chose Spin and its input language called Promela.

6.1 Code Generation for the Intermediate Model of State Machines

State machines are attached to classes in the UML model, and this relationship is also preserved in the IM. If one has several instances of a certain class, then all such instances will have the same state machine inside.

So a state machine attached to a class appears as a process type in the Promela *code.*

```
proctype UnitSM (chan signal)
{
   ...
}
```

When one has several instances of a certain type, then just the same amount of processes should appear. Signals arrive to the state machine through a channel.

Each state in the state machine is mapped to a Boolean variable. Its value will be 1, if the state machine is in the state, and it will be 0, if the state machine is not in the state. These variables are organized into an array for the sake of simplicity.

Example 7
The IM generated for the state machine of the Unit class has 21 states as one can check e.g. in Figure 5.6, so the corresponding Promela *code would be the following:*

```
bit state[21];
```

The first few statements of the process should set the initial state configuration. Afterwards the main do loop comes. There can be some branches expressing behavior that is required to offer a uniform interface to the environment. Note that the UML model does not deal with the existence of the embedding environment at all. It just states that e.g. a signal has to be sent (i.e. Send Signal Action), but it does not specify how the signal reaches the destination. In order to get a useful system, one also has to generate the code for those parts that maintain relation with the environment. The `else` branch of the main `do` loop will handle the steps of the state machine.

```
state[0]=1;
state[10]=1;

do
::         /* Statements to handle requests
              from the environment */
:: else -> /* Statements describing one
              run-to-completion step of the
              state machine */
od
```

State machines execute run-to-completion steps. A run-to-completion step cannot be interrupted, but of course its basic steps can overlap with steps of other state machines.

Event dispatching and processing. During a run-to-completion step, events are dispatched and processed first. Events can only be processed in stable state configurations, so it must be determined when a system is in such configuration.

So a logic expression is formulated according to the following rules:

- If a node is a compound state or a region and its children are not regions, then the corresponding logic expression is:

  `state[i] && ((`$child_1$`) || (`$child_2$`) || ... || (`$child_n$`))`

  `state[i]` denotes the Boolean variable to which the compound state has been mapped, while $child_j$ marks the jth *child active state*. Note that these states have to be in inclusion relation with the parent state, and they have to be instances of active states (either states or regions). If activities or pseudostates appear as children, they are completely left out of consideration.

- If a node is a simple state, then the corresponding Boolean variable (`state[i]`) will be used in the logic expression.

- If a node is either a pseudostate or an activity, then nothing is created.

- If a node is a compound state or a region and its children are regions, then the logic expression is:

  `state[i] && (`$child_1$`) && (`$child_2$`) && ... && (`$child_n$`)`

If the expression generation is started from the node that corresponds to the top level state machine, then a correct expression is resulted when applying the above-mentioned rules.

The event processing part will appear in the following form as the first segment of a run-to-completion step:

```
if
:: /* Expression describing
      stable state configuration */ ->
   if
   :: (signal?[sig,_]) ->
      signal?_,msgtype;
   :: else -> skip
   fi
:: else -> skip
fi
```

In this code `signal` is a channel where signals may arrive to the state machine. `sig` is a constant denoting that an event has arrived, while `msgtype` is a variable that stores the type of the event.

Parameter processing. The following part is for processing additional parameters arriving together with the events. An `if` statement is used again for this purpose.

```
if
:: msgtype==Car_create ->
      d_step {
         signal?ref,a;
         signal?ref,b
      }
:: /* Other events */
:: else -> skip
fi
```

The code segment above expects a `Car_create` event with two additional parameters (`a` and `b`). These parameters are object references. It is worth to note that if not exactly the same amount of parameters arrive, then an error message is generated, since it means that the UML model is not well-formed. (Imagine a situation, when a function requires two inputs, but only one arrives.)

Transition handling. Now I have arrived to the most interesting part, where the transitions have to be handled. As stated before, transitions may start from an active state configuration, so

1. all the situations in the IM, where *a transition relation leaves an active state* have to be searched for, then

2. these situations have to be arranged in a row depending on which layer the active starting state appears. This has to be done in a *reverse order* meaning that the lower the starting state is in the layer structure, the earlier it appears in the sequence.

3. `If` statements embedded into each other recursively are created. Embedded `if` statement appears in the `else` branch of the embedding statement.

```
if
:: /* First transition with starting state in the
      nth layer */
:: /* Second transition with starting state in the
      nth layer */
:: else -> /* (n-1)th layer */
   if
   ⋮ /* Transitions in the (n-1)th layer */
   :: else ->
      \
         if
         ⋮ /* Transitions in the 1st layer */
         :: else -> skip
         fi
      /
   fi
fi
```

I map each situation to a branch in the `if` statement. This mapping is done in the sequence that has just been defined in the previous step. Every situation that has a starting state in the lowest layer occupies a branch in the outermost `if` statement. Situations with starting states in a higher layer may go somewhere deeper into the `if` statement. Finally, situations defined in the topmost layer go into the innermost `if` statement.

4. The `else` branch of the innermost `if` statement is a single `skip` statement.

One may notice that each original UML transition is mapped to a branch in an `if` statement. Moreover, it is worth to emphasize that the recursively embedded `if` statements have exactly the same semantics needed for determining priority among enabled transitions.

Each branch has a condition part and an action part. Condition part contains the starting state of the transition, the examination of the triggering event and the guard condition. The last two components may be omitted.

```
:: state[i] && msgtype==Car_create && guard ->
   /* Action part */
```

In the action part the transition relation (TR) of the IM defines what to do.

- The target state of the TR should be entered.
- If a compound state node is entered, then several things can happen.
  - If the compound state has initial nodes as children, then the one without incoming transition relation is entered. (If there are more than one child initial nodes without incoming TR, the original UML model is ill-formed.)
  - If the compound state has no initial node among its children, then all its children have to be regions and they should also be entered. (If the children are not regions the model is ill-formed.)
- If a node of any kind is entered, then all its ancestors have to be active. If ancestors are not active at the time, when the specific node is entered, then ancestors have to be entered, too.

- If the source node of the TR is a compound state, then all its descendants should be exited immediately. (Descendants are determined by the inclusion relation.)
- If a final node is exited, then its parent node should also be exited.
- If an activity node is entered, then it has to be executed. *Each activity is mapped to a process.* The termination of embedded activities has to be waited for, before the actual node can be left. So one has to make some kind of synchronization, when the activity terminates.

```
d_step { state[0] = 0; state[1] = 1 };
run A1(child);
d_step { child?[1]; child?_ };
d_step { state[1] = 0; state[2] = 1 }
```

In the example code one can see that the first line corresponds to a transition relation that connects a state to an activity. The second line starts the execution of the activity. The role of the third line is to synchronize to the termination of the activity, while the fourth line is an other transition relation.

It seems practical if these statements are placed into an `atomic` block, since unnecessary overlapping can be avoided, and as a consequence the state space gets smaller.

6.2 Code Generation for the Intermediate Model of Activity Diagrams

As mentioned earlier, *each activity is mapped to a process.* If several activities share the same behavior, then these activities will be several copies of the same process type. But, of course, they will differ in their actual parameters.

It was also discussed earlier in Section 5.2.1 that activity diagrams have a semantics that is similar to Petri nets. When executing an activity, the number of places and transitions does not change at all, while the number of tokens may fluctuate dynamically. E.g., a Read Link Action may result in a lot of object tokens appearing on the output pin of the action.

Tokens and place nodes. Among the constructs of PROMELA processes are the only ones that can be created dynamically. (It is not possible to declare an array of variable size.) So it seems obvious that tokens are mapped to processes. If tokens appeared in a slot of an array, then the instances one might create should be limited, which would not be a good decision. *So I use a one-to-one mapping again between tokens and the corresponding processes.* One may say that the lifetime of the token is delimited by the process creation and the final synchronization command. (Unfortunately it is not sure that SPIN terminates a process immediately after the execution of its last statement.)

The basic structure of tokens would look like this:

```
proctype ObjToken(ChPair end)
{
   /* Synchronizing starting points of processes */
   /* Never claims to filter ill-formed models */
   end.st!value;
   end.ack?1
}
```

Note that tokens are consumed, when action begins execution, and they are produced, when action execution has been finished. So tokens are entities that may only exist in places, so some further comments are required.

One may notice that a channel pair argument has to be defined for each token process. SPIN allows us to have several processes sharing the same channel. If this channel is full, then processes that are waiting for the channel to become empty and to send message are being blocked. When the channel becomes empty, a nondeterministic choice is made and one process may put its token (its message) into the channel. This latter represents the token that will be consumed next time.

So places do not appear directly in the code, however the number of places of the IM and the number of new channels are the same. One would rather say that *the relation expressing that a token is in a certain place are mapped either to a process waiting for a certain channel to become empty or to a token being in the same channel.* The token in the channel represents the next token to be consumed, while the other processes correspond to the other tokens in the same place. (The channel may contain only one token.)

The `ChPair` type may be defined as follows:

```
typedef ChPair {
   chan st = [1] of { Token };
   chan ack = [0] of { bit }
}
```

The `st` channel is the place where token values (`value`) may be grabbed, while the `ack` channel is for synchronization. Actions (or transitions) may signal in the `ack` channel that the token in the `st` channel has been successfully consumed. Actions and transitions should also have similar statements in an `atomic` block to ensure the atomicity of statements consuming tokens from inputs.

Action nodes.

```
proctype ReadSelfAction(int parent; ChPair in,next)
{
   Token t;
   do
   ::         /* Statements to handle requests
                 from the environment */
   :: atomic { /* Negated conditions && */ in.st?[_]) ->
      in.st?_;
      in.ack!1;
      /* Action specific part */
      d_step {
         t.tokentype=ref;
         t.value=parent;
         run ObjToken(t,next)
      };
   }
   od;
   /* Some other statements that
      communicate with the environment. */
}
```

Let us have a look at how actions are handled in the PROMELA code. It should not be surprising that they will also be processes and the one-to-one mapping holds again. *So I map each action to a process. Each edge of the action (both incoming and outgoing) is mapped to a process parameter whose type is `ChPair`.* So e.g., two Read Self Actions may differ from each other, because they have different number of input control flow dependencies and, thus, the corresponding processes must be of different types. (E.g. it may happen that a process type with one channel expressing control flow dependency, and an other one with two such channels have to be defined.) But the majority of the code (especially the action specific part) is the same, if one models the same kind of action.

First, the structure that is shared within any action process types is presented. Our demonstration example will be the Read Self Action with one incoming edge arriving from a place that can hold control tokens. This process type has two channel pairs, one for grabbing (control) tokens and an other one for producing an object token (the one that represents the embedding object itself).

The main part of the code is a `do` loop whose first few branches may be used for communication with the environment. Its last branch deals with the effective functionality.

1. First of all, it examines whether there is a (control) token in its input place. (For practical purposes, it is wise to add the negated expression of all the conditions of previous branches. It makes sure that the action terminates when its parent activity orders this.)

2. If there is a token in the input place, the action grabs the token (`in.st?_`) and sends an acknowledgement (`in.ack!1`). (Note that the last statement of the control token process is when it waits for the acknowledgement, so when it arrives, the control token disappears.) These steps are executed inside an `atomic` block, so they cannot overlap with the steps of other processes.

3. Then the action specific part is executed. In this case, it creates a reference pointing to its parent object. (That is why the `parent` parameter is required for the process.)

4. Finally, output is generated with creating a new process representing a token in the output place. Its `next` parameter tells where the next action or transition accepts tokens as inputs.

Transition nodes. As it was discussed earlier in Section 5.2.1, action execution is prepended by an implicit join operation, and the last step of the action specific part is a fork operation (see Figure 5.10).

So the behavior of transition nodes is basically the same as the semantics of action nodes. However, it is obvious that transitions do not have any action specific statements.

The example presents a transition with one preplace holding an object token and an other one holding a control token (see Figure 5.13). In this case the postplace will receive the object token that has been grabbed from the first preplace.

Finally, I would like to mention that *each transition node is transformed to an own process.*

```
proctype Transition2_1(ChPair in1,in2,out)
{
   Token t;
   do
   ::        /* Statements to handle requests
                from the environment */
   :: atomic { /* Negated conditions && */
         in1.st?[_] && in2.st?[_]) ->
      in1.st?t;
      in2.st?_;
      in1.ack!1;
      in2.ack!1;
      run ObjToken(t,out)
   }
   od;
   /* Some other statements that
      communicate with the environment. */
}
```

Activities.

```
proctype A1(int final)
{
   ChPair pl[12];
   int id[9];
   Token t;
   atomic {
      id[0]=run ReadSelfA(pl[0],pl[1]);
      id[1]=run ReadStructuralFeatureA_out1(pl[2],pl[5],pl[6]);
      id[2]=run RemoveStructuralFeatureValueA_in1(pl[3],pl[6],
                                                  pl[7]);
      id[3]=run AddStructuralFeatureValueA_in1(pl[4],pl[10],
                                               pl[7],pl[11]);
      id[4]=run ApplyFunctionA(pl[8],pl[9]);
      id[5]=run Transition1_3(pl[1],pl[2],pl[3],pl[4]);
      id[6]=run Transition1_1(pl[5],pl[8]);
      id[7]=run Transition1_1(pl[9],pl[10]);
      id[8]=run FinalNode(pl[11],_pid);
      d_step {
         t.tokentype=noid;
         t.value=0;
         run CtlToken(t,pl[0])
      };
   };
   do
   :: /* When the final node has been reached */ ->
      break
   od;
   /* Forcing to finish processes */
   final!1
}
```

After having gone through all the basic elements, we survey how a certain activity is to be handled. *It is a one-to-one mapping again, since each activity is transformed to a* *process.*

Let us consider the Petri net representation (see Figure 5.18) of the activity diagram of Figure 2.4. It contains 12 place nodes, so our code requires an array of 12 channel pairs. There are 9 transitions in the model. Five of them represent actions, one corresponds to the final node, and there exist three traditional transitions. The process IDs of all the transitions should be stored, so an array containing 9 integers is required.

In the `atomic` block processes are run first. (Characters behind the underscore character in action process names denote the number of extra (control) places required.) This basically generates the structure where tokens may roam. Then the `d_step` block creates the first token and puts it into the node that represents the initial node (`run CtlToken(t,pl[0])`).

Then a `do` loop comes, which is blocked until a token gets into the final node. This notification arrives through the environment.

When the `do` loop is finished, the activity forces its children to terminate, and finally it gives a signal to its parent (generally to the embedding state machine) that its work has been completed.

Environment. During object construction memory is allocated for the object (and for its attributes). This is an aspect that does not appear at all in UML modeling, and as a consequence neither appears in the SPIN code. However, memory shortage is also a real world problem, and verification should be sensible in this field, too.

Moreover, SPIN limits the number of processes to ensure a finite state space during verification. These observations resulted in the following concept.

I set up two global arrays to track the objects and links in the system. These arrays will only be manipulated from object and link related actions (e.g., Read Link Action, Destroy Object Action) and from the `init` process. Since objects are mapped to processes in a one-to-one fashion again, it is enough to store process identifiers in the array of objects. The array of links contains pairs of process identifiers. The structure of members can be easily extended in case of non-binary associations towards n-tuples.

```
/* Global limits */
#define MAX_OBJ 2
#define MAX_LINK 2
/* Global arrays */
Object object[MAX_OBJ];
Link link[MAX_LINK];
chan cdest = [1] of { int, chan };
```

Type definitions for objects and links can be the following at a bare minimum:

```
typedef Link {
   byte class;
   byte rel1;
   byte rel2;
}
typedef Object {
   byte class;
   Token id;
   chan signal = [5] of { byte, int };
}
```

The `class` record is for storing the class (association) from which the object (link) is instantiated. The `signal` variable denotes the channel where events may arrive to the state machine of the object.

As one could notice an additional channel (`cdest`) has also been declared as a global variable. All the processes must have access to this channel, and must behave in exactly the same way. Namely, if they get their process identifier as the first parameter, they have to terminate and must also send an acknowledgement through the channel whose identifier arrives as the second part of the message.

Now, one may resurvey the code where processes communicated with the environment. Let us consider e.g., the transition process again.

```
proctype Transition2_1(ChPair in1,in2,out)
{
   chan f = [0] of  bit ;
   do
   :: atomic {
      cdest?[eval(_pid),_] ->
      cdest?eval(_pid),f;
      break
   }
   :: !(cdest?[eval(_pid),_]) &&
            /* Transition specific part */
   od;
   f!1
}
```

So an additional local channel (`f`) is needed where the final synchronization message can be placed. A code has been introduced that checks the `cdest` global channel and is executed when the own identifier of the process arrives. The negated expressions in the transition specific branches ensure that they cannot run when the process is to terminate, so in other words the first branch has higher priority.

Chapter 7

Evaluation and Conclusion

7.1 Result Assessment

In the final chapter, a short summary, evaluation and conclusion is presented, together with a brief overview focusing on directions of future work, in addition.

7.1.1 Main Conclusion

First of all, I would like to refer to Figure 1.1, which directly shows my results.

As a summary, my results are expressed in the form of main conclusion statements.

Main Conclusion 1
I overviewed some already existing semantics defined for behavioral UML diagrams and some transformations.

Main Conclusion 2
I proposed a mathematically precise semantic framework that simultaneously handles different behavioral UML diagrams including State Machines and Activity Diagrams.

- *State Machines were described as transition systems.*
- *Activity Diagrams and Action Semantics got a Petri net based semantics.*

Main Conclusion 3
I designed a transformation that constructs the model in the unified framework.

Main Conclusion 4
I elaborated a transformation that receives a description in the semantic framework and provides SPIN *code as output.*

7.2 Future Work

My transformation still lacks several functionalities that have not or just partially been considered in the current thesis.

At least the following should be mentioned (listed randomly):

- A graph transformation or XSLT based implementation resulting in an automated transformation.

- An industrial size benchmark application.
- A database implementation of the VPM supporting the reverse transformation.
- A possible extension to the VPM framework.
- Integrating some other semantics under the same notational framework.

7.3 Acknowledgement

I would would like to express my gratitude to all the people who helped me in some way preparing my work during the last months. Special thanks to

- András Pataricza, who undertook the supervision of my thesis. The many hours we spent on discussing the problems arisen were very helpful to create this work.
- Dániel Varró, for his fruitful cooperation and his useful comments.
- István Majzik, for introducing me to the concept of extended hierarchical automata.
- the whole team of the Fault Tolerant Systems Research Group, for providing such friendly conditions.
- Last but not least, special thanks to my parents for being supportive all the time despite the fact that many times they were the sufferers of my work.

Bibliography

[1] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71, Yorktown Heights, New York, May 1981. Springer.

[2] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An Automatic Verification Tool for UML. Technical Report CSE-TR-423-00, 2000.

[3] R. Eshuis and R. Wieringa. A formal semantics for UML Activity Diagrams – Formalising workflow models. Technical Report CTIT Technical Report 01-04, University of Twente, February 2001.

[4] M. Fowler and K. Scott. *UML Distilled. Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[5] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

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

[7] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991.

[9] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992. `citeseer.nj.nec.com/holzmann92coverage.html`.

[10] R. P. Kurhsan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[11] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.

[12] G. Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.

[13] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[14] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[15] K. L. McMillan. *Getting started with SMV*, March 1999. `http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial.ps`.

[16] K. L. McMillan. *The SMV language*, March 1999. `http://www-cad.eecs.berkeley.edu/~kenmcmil/language.ps`.

[17] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *ASIAN'97 Third Asian Computing Conference. Advances in Computer Science*, volume 1345 of *LNCS*, pages 181–196. Springer-Verlag, 1997.

[18] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *LNCS*. Springer-Verlag, December 1997.

[19] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in Promela/SPIN. In *Proceedings of WIFT98*, 1998.

[20] Object Management Group. *Introduction to OMG's Unified Modeling Language*, August 2002. `http://www.omg.org/gettingstarted/what_is_uml.htm`.

[21] Object Management Group. *MDA Guide Version 1.0*, May 2003. `http://doc.omg.org/omg/03-05-01.pdf`.

[22] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS language reference, Version 2.3. Technical report, SRI International, September 1999.

[23] I. Paltor and J. Lilius. vUML: A tool for verifying UML models. In R. J. Hall and E. Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[24] J. L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, Englewood Cliffs, N.J., April 1981.

[25] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, pages 46–57, Providence, R.I., 1977.

[26] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[27] T. C. Ruys. *SPIN Beginners' Tutorial.* Grenoble, France, April 2002. `http://spinroot.com/spin/Doc/SpinTutorial.pdf`.

[28] TogetherSoft, Inc. *Practical UML A Hands-On Introduction for Developers*, August 2002. `http://www.togethersoft.com/services/practical_guides/umlonlinecourse/`.

[29] I. Traoré. An outline of PVS semantics for UML Statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108, Nov. 2000.

[30] U2-Partners. *UML: Infrastructure v. 2.0 (Third revised proposal)*, January 2003. `http://www.u2-partners.org/artifacts.htm`.

[31] U2-Partners. *UML: Superstructure v. 2.0 (Third revised proposal)*, January 2003. `http://www.u2-partners.org/artifacts.htm`.

[32] D. Varró. A formal semantics of UML Statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. ICGT 2002: 1st International Conference on Graph Transformation*, volume 2505 of *LNCS*, pages 378–392, Barcelona, Spain, October 7–12 2002. Springer-Verlag.

[33] D. Varró and A. Pataricza. Metamodeling mathematics: A precise and visual framework for describing semantics domains of UML models. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, volume 2460 of *LNCS*, pages 18–33, Dresden, Germany, September 30 – October 4 2002. Springer-Verlag.

[34] M. von der Beeck. A comparison of statecharts variants. In *Proc. Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *LNCS*, pages 128–148, 1994.