An Executable Formal Semantics for a UML State Machine Kernel Considering Complex Structured Data

Dirk Seifert

An Executable Formal Semantics for a UML State Machine Kernel Considering Complex Structured Data

Dirk Seifert

LORIA — Université Nancy 2
Campus scientifique, BP 239
54506 Vandœuvre lès Nancy cedex
France
dirk.seifert@loria.fr

Technische Universität Berlin
Software Engineering Research Group
Franklinstraße 28/29, 10587 Berlin
Germany
seifert@cs.tu-berlin.de

April 2008 — Version 1.1

Abstract

We present a comprehensive formal semantics for a UML state machine kernel which also considers the use and manipulation of complex structured data. We refer to the UML standard Version 2.1.1 which was published in year 2007. There has been no work that completely integrates complex structured data into a UML state machine semantics. We follow a "semantics-first" approach (in opposite to a "complete-notation-first" approach) in which we consider a sound basic kernel of the UML state machine notation, and extend this kernel only ater a thorough investigation of the impacts. We define an operational semantics which is intended to be implemented in a standard programming language. Currently we use such an implementation to automatically generate test cases out of a state machine specification. This document is intended to be adapted if necessary. We will indicate that by the version number given above.

Contents

1 Introduction **2**

2 Abstract Syntax **6**

2.1 States and Regions 6
2.2 Events 8
2.3 Data Space 9
2.4 Guards 9
2.5 Actions 10
2.6 Transitions 10
2.7 State Machine 11

3 Semantics **11**

3.1 Semantical States 11
3.2 Transition Selection 12
3.3 Semantical Step — *Run-To-Completion* 16

4 Summary 17

5 Sample ML Implementation 17

1 Introduction

The Unified Modeling Language (UML) comprises thirteen diagram types to specify structure and behavior of a system or a system component [23]. The included state machines are used to either describe the discrete reactive behavior of a system (behavioral state machines) or to describe the usage protocol of a system (protocol state machines). We focus on behavioral state machines and use them to specify the states a system can take and actions it can execute during its lifetime in response to internal and external events. The discrete reactive character of state machines and the possibility to completely specify the behavior of a system make state machines appropriate to model reactive systems.

State machines are an object-oriented extension of the classical Harel Statecharts [4]. They are mathematical models with a graphical representation: the nodes depict simple or composed states of the system and the labeled edges depict transitions between these states. Composite states are used to hierarchically and orthogonally structure the model, thus reducing its graphical complexity. Simple composite states contain exactly one region and orthogonal states contain at least two regions. In every region only one state must be active at a time. The state which is entered by default if the enclosing region is entered, is marked by a transition emanating from a filled circle (called the default transition). Labels express conditions under which transitions can be taken and the actions that will be executed when the transitions are taken. Events are used as triggers to activate transitions and can be parameterized to exchange data. Optional, every state machine has a data space that can be read and manipulated by the state machine during its execution. More precisely, it is possible to read data values to describe fine-grained conditions when a transition can be taken, or to manipulate data values and exchange information within the actions. A transition comprises a source state, a trigger event, an optional guard, an optional effect (which consists of a sequence of actions), and a target state. A guard describes with a possible reference to the state machines data space a fine-grained condition that must evaluate to true to enable the transition. Hence, the activation of the source state, the available trigger event and the fulfilled guard condition constitute the precondition of the transition. An action can either be a statement manipulating the data space or the generation of new events. Hence, the action sequence and the subsequently reached target state constitute the postcondition of the transition. In opposite to the classical Statecharts, the event processing takes place in a so-called run-to-completion step. This asynchronous event processing demands the processing of the previous step to be completely finished before the next step can be executed. In the following we introduce the state machine notation by means of an simple example, namely a Car Audio System. The principle user interface of this system is shown in Figure 1. The textual requirements for the Car Audio System could be as follows:

> It should be possible to turn the Car Audio System on and off. When turned on, it should play one of three different audio sources, namely radio, tape or compact disc, respecting the presence of a tape or a compact disc. It should be possible to change between available sources. Furthermore, it should be possible to switch between four radio stations, to spool a tape backward or forward, and to select the previous or the next track of a compact disc.

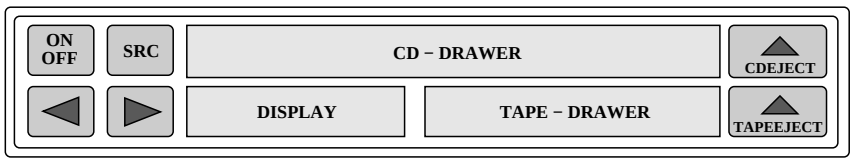


Figure 1: User interface of the Car Audio Sysem.

Based on the textual requirements we introduce the following events to model the required behavior: `power`, `src` (to switch between the different sources), `next`, `back` and `play`. We also introduce events signaling the insertion and the ejection of a tape or a compact disc as well as events to signal system reactions (`cd_insert`, `cd_eject`, `tape_insert` and `tape_eject`). Furthermore, we use data variables to store detailed information about the current state. We use an integer variable `trackCount` to store the number of titles of an inserted compact disc, and the two boolean variables `inCDFull` and `inTapeFull` to store if a compact disc and a tape are inserted into the Car Audio System. We will use this variables to control the switching between the different sources. In most applications a state machine is assigned to a class diagram describing the behavior of the class instances. In this setting, the class attributes constitute the data space of the state machine. The events of a state machine will also be represented as classes having their own attributes. We differentiate between events which are referenced by the state machine (i.e., to send events to other system components) and events which will be processed by the state machine. Figure 2 shows the class diagram for the Car Audio System and the related state machine model.

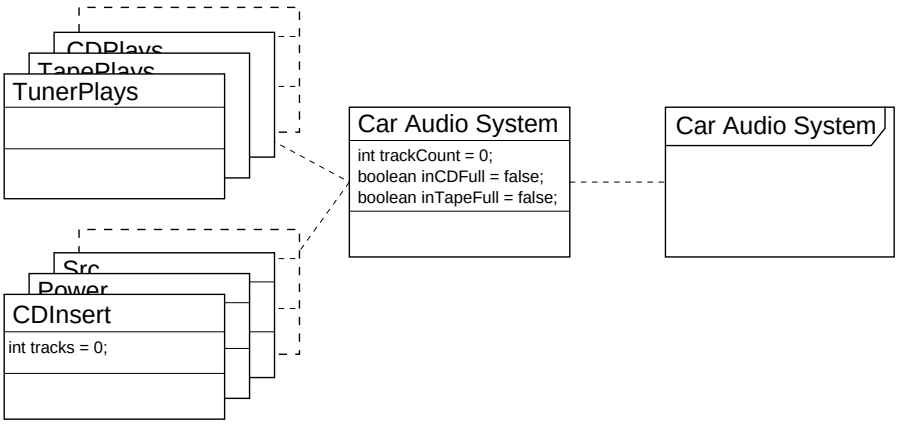
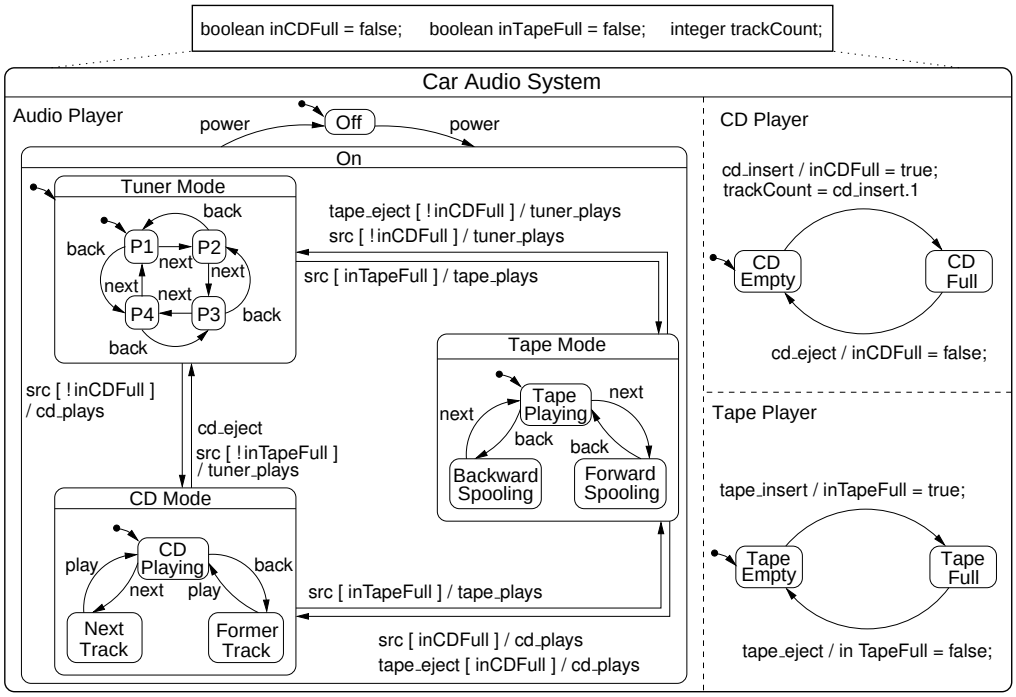


Figure 2: Class diagram for the Car Audio System.

Figure 3 shows a state machine model of the Car Audio System and the related data space. At the highest level of abstraction the state machine `CarAudioSystem` consists of an orthogonal state which comprises three regions. The two regions `CD Player` and `Tape Player` model if a tape or a compact disc are present in the system. The more complex region `Audio Player` models the control unit of the Car Audio System. It is refined by two states, namely `Off` and `On`. By default the system is assumed to be switched off. If an event occurrence *power* is processed the system is switched on and starts to play the radio (due to the default transition). The composite state `On` is refined into states modeling the three signal sources. The transitions between these states describe the changes between the sources as reaction to an event occurrence *src*. For example, if the system is in `Tuner Mode` and a tape and a compact disc are inserted into the system (i.e., the boolean variables `inCDFull` and `inTapeFull` are true) and an event occurrence *src* is processed, the system can either switch to the tape mode or switch to the compact disc mode since both transitions are enabled. All three substates of `Audio Player` are further refined to describe the particular behavior in reaction to the events `next`, `back` and `play` in each state. If a transition is taken (the transition is said to fire) the associated action sequence is executed. That includes the generation of new event occurrences and the manipulation of the data space. For example, if the transition from state `CD Empty` to state `CD Full` in region `CD Player` fires, the attribute assignments changes such that `inCDFull` is set to `true` and that `trackCount` is set to the value of the first parameter of the triggering event occurrence `cd_insert`.

Semantic Variation Point 1 (Expression Language)
The UML standard indicate that the expression language which can be used to specify guards and actions depends on the chosen action language for the state machine (usually the target programming language).

□



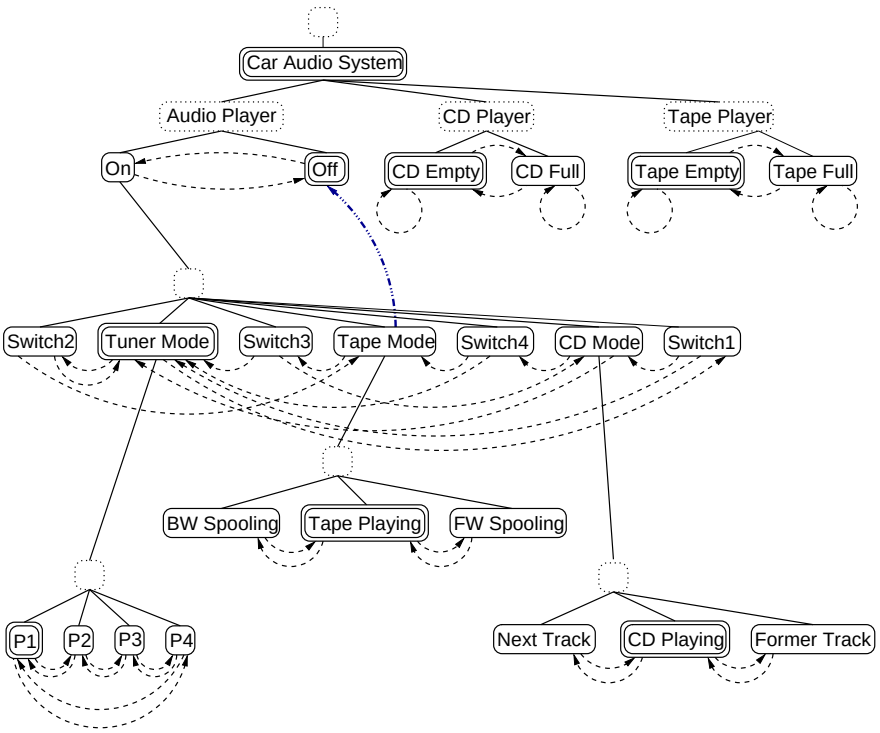


Figure 3: State machine model of the Car Audio System and the associated tree structure.

This allows a wide application domain for state machines. But it considerably complicates formal reasoning about state machine models. Currently, we use a subset of the JAVA programming language [10] to specify guard and action expressions. For the future we propose to define an independent expression language which can be mapped to an action language but which allows formal reasoning.

The hierarchical alignment of state machine states forms a tree structure with a region as the root node, simple states at the leaves and alternating composite states and regions in between. Figure 3 shows this structure for the state machine `Car Audio System`. Regions are depicted as dotted nodes and have an optional name. State are depicted as simple framed nodes, whereat default states are depicted as double framed nodes. The transitions are depicted as dashed arrows. In the tree structure it is easily identifiable which state will be left and which state will be entered if a transition is taken. Affected by the transition is the subtree which contains both the source state and the target state. Most transitions are within one hierarchy level. But it is also possible that a transition crosses multiple hierarchy levels. Such transitions are called *multi level transitions*. For example, the transition leaving state `Tape Mode` and entering state `Off` is a multi level transition. The transition could model, for example, that if the tape reaches its end the Car Audio System is turned off. The affected subtree can be also easily identified.

We use for all definitions as the basic notation the Z Formal Specification Notation. An introduction to Z and its formal semantics can be found in [21, 24, 9, 18].

List of Definitions

Definition1 States Set . . . 6
Definition2 Type Function . . . 6
Definition3 Attribute Mapping . . . 6
Definition4 Substates Sets . . . 7
Definition5 Well-formed State Hierarchy . . . 7
Definition6 Root . . . 7
Definition7 Container Function . . . 8
Definition8 Default States . . . 8
Definition9 Event Names . . . 8
Definition10 Event Instances . . . 8
Definition11 Event Set . . . 8
Definition12 Data Space . . . 9
Definition13 Guards . . . 10
Definition14 Actions . . . 10
Definition15 Transitions . . . 10
Definition16 Well-formed Transitions . . . 11
Definition17 State Machine Structure . . . 11
Definition18 Configuration set . . . 11
Definition19 Event Store . . . 12
Definition20 Status set . . . 12
Definition21 Enabled Transitions . . . 13

Definition22 Scope of a Transition . . . 13
Definition23 Main Source and Main Target of a Transition . . . 13
Definition24 Exit Set of a Transition . . . 14
Definition25 Conflict Relation . . . 14
Definition26 Priority Relation . . . 14
Definition27 Firing Transition Set . . . 15
Definition28 Enter Set of a Transition . . . 15
Definition29 Semantical Step . . . 16

2 Abstract Syntax

2.1 States and Regions

A state machine is composed of hierarchically and orthogonal structured states. The states set of a state machines comprises the states of a state machine as well as the regions of a state machine which are used to group refining states.

Definition 1 (States Set)
Let S be the given set of states and regions.

□

States, which are not further refined, are called *simple states*. States, which are further refined, are called *composite states*. Composite states which are refined by exactly one regions are called *simple composite states*. Composite states which are refined by at least two regions are called *orthogonal states*. Hence, we distinguish in S four types of states, namely *region*, *simple*, *simple composite* and *orthogonal*.

Definition 2 (Type Function)
Let S be the set of states. The function *type* yields for every state in S its type.

$$type : S \rightarrow \{region, simple, simple\,composite, orthogonal\}$$

□

Remark: In the UML standard regions and states of a state machine are represented as different classes. Thus, they are distinguished by the class name. The different state types are distinguished on the basis of class attributes. The boolean attribute *isSimple* indicates if the state is a simple state. The boolean attribute *isComposite* indicates if the state is further refined. Finally, the boolean Attribute *isOrthogonal* indicates if the state is an orthogonal state. Using a type function avoids unnecessary redundancy since invalid attribute combinations are avoided. The relation between these class attributes and the function *type* is as follows.

Definition 3 (Attribute Mapping)
Let S be the set of states. For every state $s : S \mid type(s) \neq region$ and its equivalent UML representation $s' : State$ the following relation holds.

$$type(s) = \begin{cases} simple & \text{iff} \quad s'.isSimple \\ simple\,composite & \text{iff} \neg s'.isSimple \wedge s'.isComposite \wedge \neg s'.isOrthogonal \\ orthogonal & \text{iff} \neg s'.isSimple \wedge s'.isComposite \wedge \quad s'.isOrthogonal \end{cases}$$

□

Remark: Sometimes simple composite states are also called *xor*-states and orthogonal states are also called *and*-states. This is motivated by the number of simultaneously active direct substates. Refined states contain regions. Within a region exactly one state must be active at a time. Simple composite state contain exactly one region in which exactly one states is active (i.e., there is a choice among the states: *xor*). Orthogonal states contain at least two regions. Consequently there are at least two states active at a time (i.e., exactly one in every region: *and*).

The state refinement of a well-formed state machine forms a tree structure, denoted (S,H). We call (S,H) *state hierarchy* and $H : \mathbb{P}(S \times S)$ *substate relation*. The substate relation is nonempty and gives the relative ordering of all state machine states. A tuple $s \mapsto s' : H$ indicates, that state s is refined by state s' (i.e., state s' is a substate of state s). Based on the substate relation we can define several state sets with respect to a state.

Definition 4 (Substates Sets)
Let (S,H) be the state hierarchy. The functions $substates : S \rightarrow \mathbb{P}S$ yields for every state $s : S$ the set of direct substates of s. The functions $substates^+$ and $substates^*$ yield the transitive and transitive-reflexive closures, respectively.

$$substates\,(s) == \{s' : S \mid s \mapsto s' \in H\} \tag{1}$$
$$substates^+\,(s) == \{s' : S \mid s \mapsto s' \in H^+\} \tag{2}$$
$$substates^*\,(s) == \{s\} \cup substates^+\,(s) \tag{3}$$

Where H^+ denotes the transitive closure of H.

□

Is $s_2 \in substates^*\,(s_1)$ we say that s_1 is a *superstate* of s_2, and that s_2 is a *substate* of s_1. Due to the reflexivity of $substates^*$ s is superstate as well as substate of oneself. Is $s_2 \in substates^+\,(s_1)$ we say that s_1 is a *strict* superstate of s_2, and that s_2 is a *strict* substate of s_1.

In a well-formed state machine it is required that a region is refined by at least one state, a simple composite state is refined by exactly one region, and an orthogonal state is refined by at least two regions. This implies that regions and state alternate in the tree structure and that the leaves are composed of simple states.

Definition 5 (Well-formed State Hierarchy)
A a state hierarchy (S,H) which preserves the following constraints is well-formed.

$$\exists_1 s : S \bullet s \notin \mathsf{ran}\,H \;\wedge\; s \in \mathsf{dom}\,H \;\wedge\; \forall s' : S \setminus \{s\} \bullet \exists_1 s'' : S \bullet s'' \mapsto s' \in H \tag{4}$$
$$\forall s : S \mid type\,(s) = region \bullet (\forall s' : substates\,(s) \bullet type\,(s') \neq region) \tag{5}$$
$$\forall s : S \mid type\,(s) = simple \bullet substates\,(s) = \varnothing \tag{6}$$
$$\forall s : S \mid type\,(s) = simple\,composite \bullet (\forall s' : substates\,(s) \bullet type\,(s') = region) \wedge (\#substates\,(s) = 1) \tag{7}$$
$$\forall s : S \mid type\,(s) = orthogonal \bullet (\forall s' : substates\,(s) \bullet type\,(s') = region) \wedge (\#substates\,(s) > 1) \tag{8}$$

□

Constraint (4) requires the tree structure of the hierarchy relation for a well-formed state hierarchy. That means, that there exists a distinguishable root state, all remaining states have exactly one superstate, and that there exists no loops in the structure. From the UML standard it follows that the root state must be a region. Constraint (5) to (8) characterize the different refinement variants for states and regions. They include, that the leaves of the tree structure are composed of simple states ($\forall s : S \mid \nexists s' : S \bullet s \mapsto s' : H \bullet type\,(s) = simple$).

Definition 6 (Root)
Let (S,H) be the state hierarchy. *root* denotes the root of the tree structure induced by H.

$$root == \mu\, n : N \mid n \notin \mathsf{ran}\,H$$
$$type\,(root) = region$$

□

The hierarchy relation is injective and, except for the root state, surjective. The inverse H^{-1} is a partial function, which yields for every state, except for the root state, its direct superstate.

Definition 7 (Container Function)
Let (S,H) be the state hierarchy. The partial function $container : S \nrightarrow S$ yields for every state $s : S \setminus \{root\}$ its direct superstate $s' : S$. We call s' container of s.

$$container(s) == \mu\, s' : S \mid s' \mapsto s \in H$$

□

We call the state which will be activated if one of its superstates is activated *default state*. This is in opposite to the common literature. There, such states are called initial states. We use a different term since not all default state are initially active (e.g., if an enclosing orthogonal state is not initially active).

Definition 8 (Default States)
Let (S,H) be the state hierarchy. The set $\mathscr{D} \subseteq S$ comprises the default states.

$$\forall s : S \mid type(s) = region \bullet \exists_1 d : S \mid d \in substates(s) \bullet d \in \mathscr{D}$$

□

2.2 Events

Events are used to trigger transition. They can be parameterized to exchange detailed information. Hence, an event comprises of an name and a finite number of data partitions. In practice, the event name corresponds to the class name and the particular data partitions correspond to the attributes of this class. Consequently, the sets of event instances correspond to the parameter domains.

Definition 9 (Event Names)
Let $\mathscr{E}$ be the given set of event names.

□

Definition 10 (Event Instances)
Let K be an index set with n elements, and let $(P_i)_{i:K}$ a family of sets P_i. The set E_ν denotes the set of all event instances to event name $\nu : \mathscr{E}$.

The index set can be empty for particular event names.

$$E_\nu == \nu\, \langle\langle P_1 \times \ldots \times P_n \rangle\rangle$$

□

Definition 11 (Event Set)
Let $\mathscr{E}$ the set of event names. The set E denotes the set of all event instances with respect to $\mathscr{E}$.

$$E == \mid_{\nu:\mathscr{E}} E_\nu$$

□

The set E is the disjunctive union of all particular set of event instances. We call a member of the event set *event instance* and denote it $\nu(\ldots)$. If it is clear from the context we use the term event name and event synonymously. To demonstrate the construction of an event set we give a small example.

Example 1 (Set of Event Instances)
Let $\mathscr{E} = \{a, b\}$ be the set of event names, and let a have two parameters of type $\mathbb{N}$ and *bool*, and let b have one parameter of type $\mathbb{B}$. The set E denotes the set of all event instances with respect to $\mathscr{E}$.

Parameter sets:	$P_1^a = \mathbb{N} \wedge P_2^a = \mathbb{B}$
	$P_1^b = \mathbb{B}$
Sets of event instances:	$E_a = a\,\langle\langle \mathbb{N}, \mathbb{B} \rangle\rangle = \{a(0, true),\ a(0, false),\ a(1, true),\ a(1, false),\ \ldots\}$
	$E_b = b\,\langle\langle \mathbb{B} \rangle\rangle = \{b(true),\ b(false)\}$
Set of all event instances:	$E = E_a \mid E_b = E_a \uplus E_b = \{a(0, true),\ a(0, false),\ \ldots\ ,\ b(true),\ b(false)\}$

□

A transition of a state machin *SM* can only be triggered by a subset of the event instances in E. In anticipation of the following sections we distinguish four subsets of E. First, a subset which comprises all event instances which can trigger transitions of the state machine: $E_{SM} \subseteq E$. Second, a subset which comprises all event instances which are sent by the state machine to other system components: $E_{ENV} \subseteq E$. We further distinguish two subset in E_{SM}. Third, a subset which comprises all event instances which can only by used by the state machine itself: $E_{SM}^{prv} \subseteq E_{SM}$. Forth, a subset which comprises all event instances which can also be used by the environment $E_{SM}^{pub} \subseteq E_{SM}$. This set represents the public interface of the state machine *SM*. With respect to this subset of event instances the following constraints apply: $E_{SM}^{prv} \uplus E_{SM}^{pub} = E_{SM}$ and $E_{SM} \uplus E_{ENV} = E$. Figure 4 illustrates this partition into subsets.

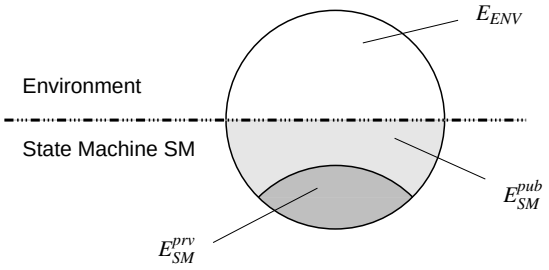


Figure 4: Partitioning of the event instances in E.

2.3 Data Space

A data space is associated to every state machine comprising of a finite number of data partitions. These partitions can be read and manipulated during the execution of state machines. In practice, these partitions correspond to the class attributes.

Definition 12 (Data Space)
Let K be an index set with n elements, and let $(P_i)_{i:K}$ a family of sets P_i. The set D comprises all data space assignments.

The index set can be empty for particular event names.

$$D == P_1 \times \ldots \times P_n$$

□

2.4 Guards

A guard of a transition is a constraint, with respect to the trigger of the transition and with respect to the data space of the state machine, that provides a fine-grained control over the enabling of the transition. It is a function which yields for an event instance and a data space assignment a boolean value indicating if the constraint is fulfilled or not. The expression language for guards is defined by the used action language (cf.

Semantic Variation Point 1). From the UML standard it is required that guards should be pure expressions without side effects. Guard expressions with side effects are ill formed. Please note that the event instance is not part of Definition 13 and 14. It will be added in Definition 16 of transitions.

Definition 13 (Guards)
Let D be the data space. The set G denotes the set of all guard functions.

$$G == D \rightarrow \mathbb{B}$$

□

2.5 Actions

The effect of a transition is sequence of actions which will be executed if the transition fires. An action can either be a statement updating the data space or the generation of a new events instance. Updating the data space means that, with respect to the data space of the state machine, a new data space assignment will be selected. Generating a new event instance means that, with respect to the trigger of the transition and with respect to the data space of the state machine, a new event instance will be selected. In practice it corresponds to the execution of the action code and the generation of a new event object.

Definition 14 (Actions)
Let D be the data space and let E be the set of all event instances. The set A denotes the set of all action functions.

$$A == event\langle\langle D \nrightarrow E \rangle\rangle \mid update\langle\langle D \nrightarrow D \rangle\rangle \tag{9}$$

□

2.6 Transitions

Transitions describe the state changes in state machines. Graphically, they are directed labeled edges between the source state and the target state. A label comprises a trigger, an optional guard and an optional sequence of actions.

Definition 15 (Transitions)
Let (S,H) be the state hierarchy, let E be the set of all event instances, let G be the set of all guards, and let A be the set of all actions. The set T denotes the set of all transitions.

$$T \subseteq S \times (E \nrightarrow G \times \mathrm{seq} A) \times S$$

□

We use the following projections to access the particular parts of a transition: *source*, *label* and *target*. Let $t : T$ be a transition. The projection $source == t.1$ yields the source state, the projection $label == t.2$ yields the label, and the projection $target == t.3$ yields the target state of t. The label of a transition is a function whose parameter is the triggering event instance and whose result is an ordered pair comprising a guard and a sequence of actions. Let p be such an ordered pair. The projection $guard == p.1$ yields the guard and the projection $effect == p.2$ yields the actions sequence of p.

With $s_1 \xrightarrow{e\,[g]\,/\,A} s_2$ we denote a transition graphically, whereat s_1 denotes the source state, e denotes the triggering event instance, g denotes the guard, A denotes a sequence of actions, and s_2 denotes the target state of the transition.

Definition 16 (Well-formed Transitions)
Let T be the set of transitions, let $\mathscr{E}$ be the set of event names, let E be the set of event instances and let G be the set of guards. A set $T \subseteq S \times (E \nrightarrow G \times \mathrm{seq} A) \times S$ of transitions which preserves the following constraints is well-formed.

$$\forall t : T \bullet \exists_1 v : \mathscr{E} \bullet E_v \subseteq E_{SM} \wedge \mathrm{dom}\, label\,(t) = E_v \tag{10}$$

$$\forall t : T;\ e : E;\ d : D \bullet e \in \mathrm{dom}\, label\,(t) \wedge guard\,(label\,(t)(e))(d) \Rightarrow \tag{11}$$
$$\forall i : \mathrm{dom}\, actions\,(label\,(t)(e)) \bullet (\textbf{let}\ a == actions\,(label\,(t)(e))(i) \bullet$$
$$(\forall f : D \nrightarrow E \mid a = event(f) \bullet d \in \mathrm{dom} f) \wedge (\forall f : D \nrightarrow D \mid a = update(f) \bullet d \in \mathrm{dom} f))$$

□

Constraint 10 requires that a transition (i.e., precisely a label function) must be defined for all event instances of an event name, whereat the set of possible event instances is restricted to that associated to the state machine. Constraint 11 requires that if the label function is applied to a event instance and the guard function applied to a data space assignment evaluates to true, all action functions of the effect must be defined with respect to this data space assignment.

2.7 State Machine

Definition 17 (State Machine Structure)
Let (S,H) be the state hierarchy, let $\mathscr{D}$ be the set of default states, let E be the set of event instances, let D be the data space, let G be the set of guards, let A be a set of actions, and let T be a set of transitions. The tuple SM denotes a state machine.

$$SM == ((S,H), \mathscr{D}, E, D, G, A, T) \tag{12}$$

□

A state machine is well-formed if Definition 5 and Definition 16 apply.

3 Semantics

The behavior of a state machine is defined by means of traversing the state graph from the node hierarchy and the transitions. In that, the state transitions are triggered by event instances which are received and stored for further processing by the state machine. During this traversing a set of transitions will be activated for a state transition. If a transition fires, all actions from the effect will executed. In the following we describe a operational semantics for state machines. Based on the semantical state, a status, and a set of activated transitions we define how the state machines moves from one well-formed status to the next well-formed status. We call such a mode semantical step.

3.1 Semantical States

During the execution of a state machine a state can be active or inactive. Due to the hierarchical and orthogonal structuring of states it is possible for state machines that a set of states is active at a time. We call the set of simultaneously active states *active state configuration* or in short *configuration*.

Definition 18 (Configuration set)
Given a state machine $((N,H),I,E,T,D)$ and the root node *root* of the node hierarchy (N,H). The set of all well-formed configurations $C : \mathbb{P}\mathbb{P}N$ is defined as follows.

$$\begin{aligned} C == \{c : \mathbb{P}N \mid (\exists_1 s' : substates\,(root) \bullet s' \in c) \wedge \\ (\forall s : c \mid type\,(s) \neq simple \bullet \forall r : substates\,(s) \bullet \exists_1 s'' : substates\,(r)\ \bullet\ s'' \in c)\} \end{aligned} \tag{13}$$

□

A configuration comprises only states. Since the root node is a region, which is always active, *root* is not contained in a configuration. But it is required, that exactly one direct substate of the root node is contained in a configuration. This state is the anchor of the definition. For all states in a configuration which are not simple states it holds, that for every region refining this state exactly one state must also be contained in the region. Thus all active states are contained in a configuration and only one state of every region is contained in a configuration. A configuration, plus the required regions, is a sub-tree of the node-hierarchy. We call a configuration c_{start} a start configuration if all states in the configuration are default states ($c_{start} \subseteq I$).

Communication within the state machine and among different system components takes place via event instances, which will be generated during the execution of transitions or received from the environment. Event instances will be stored in the state machine for further processing. The UML standard does not specify the nature of this event store. It must only be possible to add and to remove events from this event store. It is left to the user to specify a special policy.

Semantic Variation Point 2 (Event Store)
Generated internal events and events received from the environment will be stored in the state machine until they are processed. The nature of the event store is not determined. Different realizations are possible.

□

For this reason, we inductively define an abstract set of event stores, which do not restrict possible realizations. An event store is either empty or it is composed of an event instance and an event store.

Definition 19 (Event Store)
Given a set E of event instances. The set Q of *event stores* to the set of event instances E is defined as follows.

$$Q == \langle\rangle \mid add\langle\langle Q \times E\rangle\rangle \tag{14}$$

□

We use the function $\oplus : Q \times \operatorname{seq} E \rightarrow Q$ to add a sequence of events to an event store and we use the partial function $\ominus : Q \nrightarrow Q \times E$ to remove an event instance from a non-empty event store. Thus we abstract from a special order of event instances in an event store. Here we remark, that in most practical applications of UML state machines a priorized FIFO-queue is used to store event instances.

Based on configurations, event stores and data space assignments we define the set of semantical states.

Definition 20 (Status set)
Given a state machine $((N,H),I,E,T,D)$, a set C of configurations, and a set Q of event stores. The set of semantical states Z is the Cartesian product of the configurations, events stores and data space assignments.

$$Z == C \times Q \times D \tag{15}$$

We call a semantical state $z : Z$, denoted $z = [\![c,q,d]\!]$, *status*. An *initial status* $z_{initial} = [\![c_{initial},q,d]\!]$ comprises of an initial configuration, an initial event store and an initial data space assignment.

□

In the following we describe how the set of transitions is selected which will be executed when moving from one status to the next status.

3.2 Transition Selection

In state machines event instances are processed one after another in a so-called *run-to-completion* step. This implies, that the processing of the next event instance does not start until the processing of the previous

event is not completely finished. During a semantical step the state machine moves from one status to a next status. If an event instance is selected for processing one or more transitions can be activated for firing. If no transition could be activated the event instance will be discarded. In the literature this is often denoted as *stuttering* [11].

Semantic Variation Point 3 (Event Processing)
The UML standard describes that event instances which do not activate any transition will be discarded. We believe that in some practical applications this behavior is not desired and should be exchanged. Thus, we mark this as a semantic variation point although we use the interpretation of discarding events in the following.

□

If transitions could be enabled by the event instance a subset of these transitions fires during the semantical step. In the following we describe how the set of simultaneously firing transitions is selected. We call a transition enabled, if the source state of the transition is included in the current configuration, the current event instance fits to the trigger of the transition and the guard evaluates to true.

Definition 21 (Enabled Transitions)
Given a state machine $((N,H),I,E,T,D)$ and a status set Z. The boolean function $enabled : T \times C \times E \times D \rightarrow \mathbb{B}$ identifies, if a transition $t : T$ is enabled with respect to a status $[\![c,q,d]\!] : Z$ and an event instance $e : E$.

$$enabled\,(t,c,e,d) \Leftrightarrow (source\,(t) \in c) \wedge (e \in \mathsf{dom}\, label\,(t)) \wedge (guard\,(label\,(t)(e))(d)) \tag{16}$$

□

Execution or firing a transition is defined in three steps. First, the source state and all effected states will be exited. Second, the actions of the transition will be executed. Third, the target state and all effected states will be entered. If more than one transition is enabled within a semantical step it is possible that one or more transition are in conflict with each other. Such a conflict appears, if two transition exit same states in the node hierarchy. Firing both transitions would lead to an ill-formed status.

Transition are allowed to cross multiple level in the node hierarchy. We call such transitions *multi-level transitions*. To respect this fact in the following we need to know the scope of a transition. The scope enables to identify the set of states which will exited or entered if a transition is fired. The scope of a transition is identified by the *least common ancestor* of the source state and the target state of a transition. It is of type region or of type orthogonal.

Definition 22 (Scope of a Transition)
Given a state machine $((N,H),I,E,T,D)$. The function $lca : T \rightarrow N$ yields for every transition $t : T$ the least common ancestor of the source state and the target state of t.

$$\begin{aligned} lca\,(t) == \mu\, n : N \mid\ & (source\,(t) \in substates^{+}\,(n)) \wedge \\ & (target\,(t) \in substates^{+}\,(n)) \wedge \\ & (\nexists n' : N \mid n' \in substates^{+}\,(n) \bullet \\ & \quad source\,(t) \in substates^{+}\,(n') \wedge target\,(t) \in substates^{+}\,(n')) \end{aligned} \tag{17}$$

□

Based on the scope of a transition we can identify the *main source* and the emphmain target of a transition. These two states denote the root nodes of the sub-trees which are affected when firing the transition.

Definition 23 (Main Source and Main Target of a Transition)
Given a state machine $((N,H),I,E,T,D)$. The functions $mainSource : T \rightarrow N$ and $mainTarget : T \rightarrow N$ are

defined as follows.

$$mainSource(t) ==$$
$$\begin{cases} \mu\, s : substates(lca(t)) \mid source(t) \in substates^*(s) & \text{iff } type(lca(t)) = region \\ lca(t) & \text{iff } type(lca(t)) = orthogonal \end{cases} \tag{18}$$

$$mainTarget(t) ==$$
$$\begin{cases} \mu\, s : substates(lca(t)) \mid target(t) \in substates^*(s) & \text{iff } type(lca(t)) = region \\ lca(t) & \text{iff } type(lca(t)) = orthogonal \end{cases} \tag{19}$$

In which it holds, that $\forall n : \mathsf{ran}\, lca \bullet type(n) = region \vee type(n) = orthogonal$.

□

Now we define the set of states which will be exited when firing a transition. This set contains all substates of the main source which are included in the current configuration.

Definition 24 (Exit Set of a Transition)
Given a state machine $((N,H),I,E,T,D)$ and a status set Z. The function $exits : T \rightarrow \mathbb{P}\, S$ yields with respect to a status $[\![c,q,d]\!] : Z$ the set of states, which will be exited by the transition $t : T$.

$$exits(t) == substates^*(mainSource(t)) \cap c \tag{20}$$

□

The definition of conflicts is based on the exit sets of transitions. Two transitions are said to be in conflict with each other if they exit common states.

Definition 25 (Conflict Relation)
Given a state machine $((N,H),I,E,T,D)$. The relation $\nparallel : T \times T$ defines, if to transitions $t_1, t_2 : T$ are in conflict with each other, denoted $t_1 \nparallel t_2$.

$$\nparallel == \{(t_1,t_2) : T \times T \mid exits(t_1) \cap exits(t_2) \neq \varnothing\} \tag{21}$$

Additionally, we use the relation $\parallel == \neg \nparallel$ and call two transitions conflict-free if they or not in conflict, denoted $t_1 \parallel t_2$.

□

Conflict resolution is carried out in two steps. First, transition with the highest priority are selected. If there are still conflicts there will be sets of transitions identified in the second step, which are maximal with respect to its cardinality and in which all transitions are pairwise conflict-free.

The applied priority scheme is defined on the relative position of the source state in the node hierarchy. A transition, whose source state is a strict substate of the source state of another transition has priory over this transition.

Definition 26 (Priority Relation)
Given a state machine $((N,H),I,E,T,D)$. The relation $\prec : T \times T$ defines for two transitions $t_1, t_2 : T$, if t_1 *has priority over* t_2, denoted $t_1 \prec t_2$.

$$\prec == \{(t_1,t_2) : T \times T \mid source(t_1) \in substates^+(source(t_2))\} \tag{22}$$

□

As we already mentioned, not all conflicts can be resolved based on the priority relation $\prec$. For example, none of two transition $t_1, t_2 : T$ emanating from the same state has priority over the other ($t1 \not\prec t2 \wedge t_2 \not\prec t_1$). Thus we can fire only one of these transitions to reach a well-formed status. In general, we identify subsets of the enabled transition which fulfill the following constraints.

Definition 27 (Firing Transition Set)
Given a state machine $((N,H),I,E,T,D)$, an event store $q : Q$, an event instance $e : E$ of the event store q ($(q',e) = \ominus(q)$), and $[\![c,q,d]\!] : Z$ a status. A set of simultaneously firing transitions $T_{\|} \subseteq T$ fulfills the following constraints.

$$\forall t : T_{\|} \bullet enabled\,(t,c,e,d) \tag{23}$$

$$\forall t_1,t_2 : T_{\|} \mid t_1 \neq t_2 \bullet t_1 \parallel t_2 \tag{24}$$

$$\nexists t' : T \setminus T_{\|} \mid enabled\,(t',c,e,d) \bullet \forall t : T_{\|} \bullet\ t \parallel t' \vee t' \prec t \tag{25}$$

□

Definition 27 requires three constraints to be fulfilled. First, all transitions in a firing transition set must be enabled. Second, all transitions in a firing transition set must be pairwise conflict-free. Third, there must not exists a transition outside the set which is enabled and is conflict free with all transitions inside the set, or which is enabled and has priority over a transition inside the set. Consequently, a firing transition set contains all transitions which are enabled and have maximal priority. It is possible that there exists at a time more than one set fulfilling these constraints. This is a source of non-determinism in state machines. During the execution of a state machine on set is non-deterministically chosen.

Semantic Variation Point 4 (Firing Transition Set Selection)
In a semantical step it is possible to have more than one firing transition set. The policy to select a firing transition set among multiple possible firing transition sets is not determined by the UML standard. A specific selection policy can be defined.

□

The algorithm to calculate the firing transition set is called *transition selection algorithm*. If the chosen firing transition set contains more than one transition the order in which the transition are executed is not fixed. An arbitrary order can be chosen. The order of transition execution is another source of non-determinism in state machines.

Semantic Variation Point 5 (Firing Transition Set Order)
The order in which the transitions of a firing transition set will be executed is not determined by the UML Standard. A specific order can be defined.

□

Now we define based on the $mainTarget\,(t)$ the set of states, which will be entered when firing a transition. This set contains all substates of the main target which contain the target state or which are implicitly entered (default states).

Definition 28 (Enter Set of a Transition)
Given a state machine $((N,H),I,E,T,D)$. The recursive function $enters : N \times S \to \mathbb{P}\,S$ yields, applied to $mainTarget\,(t)$ and $target\,(t)$ of a transition $t : T$, the set of states, which will be entered by t.

$$enters\,(n,target) == \begin{cases} \{n\} & \text{if } type\,(n) = simple \\ \{n\} \cup \bigcup\limits_{n' \in substates\,(n)} enters\,(n',target) & \text{if } type\,(n) = simple\,composite \vee orthogonal \\ enters\,(n'',target) & \text{if } type\,(n) = region \end{cases} \tag{26}$$

In which $n'' = \mu\, n''' : substates\,(n) \mid\ (target \in substates^*\,(n''')) \vee (n''' \in I \wedge target \notin substates^+\,(n))$.

□

The function *enters* applied to the $mainTarget\,(t)$ (initial parameter n) yields all states which will be entered by t. We use the abbreviation $enters\,(t)$ instead of the recursive call $enters\,(mainTarget\,(t),target\,(t))$. If

parameter n is of type simple, just this state will be entered. Is parameter n is of type simple composite or orthogonal, this state and its contained regions will be entered ($substates(n)$). If parameter n is of type region, we differentiate two alternatives. Either, the direct substate n''' will be entered which contains the target state —$target \in substates^*(n''')$ or the direct substate n''' will be entered which is the default state. The former is the case, if we are moving down the node hierarchy to the target state. The latter is the case, if we already passed the target state and moving down to the leaves of the state hierarchy (i.e., the target state is not a substate of n) — $n''' \in I \ \wedge \ target \notin substates^+(n)$. This situation arises, if the target state is a refined state or we have to enter an orthogonal regions which is not directly entered by the transition.

Now we can define in detail the transition of a state machine from one status to the next status.

3.3 Semantical Step — *Run-To-Completion*

A semantical step denotes the transition of a state machine from one well-formed semantical state to the next well-formed semantical state. Concerning such a transition we have to differentiate two situations. First, if the current event store does not contain an event instance, the current configuration and the current data space assignment remains unchanged. Only event instances received from the environment will be added to the current event store. Second, if an event instance can be removed from the current event store, we identify a firing transition set and execute the transitions contained in this set. Executing the transitions comprises updates of the current data space assignment and the generation of new event instances. Event instances which are related to the state machine will be added to the current event store and the remaining event instances are sent to the environment. Finally, all event instances received from the environment during this step are added to the current event store.

Semantic Variation Point 6 (Adding Events to a Event Store)
It is not determined in the UML standard whether the newly generated event instances are added before or after the received event instances. Both variants are possible. Since there is no relation to the time when event instances are generated in the environment, both result in the same overall behavior. If there would be a relation, the behavior could be different.

□

The next semantical step will only be started, if the last semantical step is completely finished — *run-to-completion*.

The definition of a semantical step comprises two cut-rules. The first defines a semantical step if the current event store is empty. The second rule defines a semantical step if the event store is not empty. To improve the readability we denote a semantical step $([\![c,q,d]\!],E_{in},E_{out},[\![c',q',d']\!]) : Z \times \mathrm{seq}\, E_{SM}^{pub} \times \mathrm{seq}\, E_{ENV} \times Z$ abbreviated $[\![c,q,d]\!] \xrightarrow{E_{in},E_{out}} [\![c',q',d']\!]$.

Definition 29 (Semantical Step)
Given a state machine $((N,H),I,E,T,D)$, a status $[\![c,q,d]\!] : Z$, a firing transition set $T_{\|} \subseteq T$, and a sequence of received event instances $E_{in} : \mathrm{seq}\, E_{SM}^{pub}$. A semantical step $[\![c,q,d]\!] \xrightarrow{E_{in},E_{out}} [\![c',q',d']\!] : Z \times \mathrm{seq}\, E_{SM}^{pub} \times \mathrm{seq}\, E_{ENV} \times Z$ is defined by the following two cut-rules.

$$\frac{q = <>}{[\![c,q,d]\!] \xrightarrow{E_{in},\langle\rangle} [\![c,q',d]\!]} \quad (27)$$

$$\frac{\begin{array}{c} q \in \mathsf{ran}\, add \\ (q'',e) = \ominus(q) \\ c' = (c \setminus \bigcup_{\forall t:T_{\|}} exits(t)) \cup \bigcup_{\forall t:T_{\|}} enters(t) \\ A_{seq} \in perm(\{t : T_{\|} \bullet effect(label(t)(e))\}) \\ (d', E_{gen}) = performAll(\frown/A_{seq})(d) \\ (E_{int} = E_{gen} \upharpoonright E_{SM}) \wedge (E_{out} = E_{gen} \upharpoonright E_{ENV}) \\ q' = (q'' \oplus E_{int}) \oplus E_{in} \end{array}}{[\![c,q,d]\!] \xrightarrow{E_{in},E_{out}} [\![c',q',d']\!]} \quad (28)$$

In which $E_{out} : \mathrm{seq}\, E_{ENV}$ denotes the sequence of generated outgoing external event instances and $E_{int} : \mathrm{seq}\, E_{SM}$ denotes the sequence of internal generated internal event instances.

□

The event instances in E_{out} will be sent by the state machine to the environment (i.e., to the particular system components). The function $perm : \mathbb{P}\,\mathrm{seq}\, X \to \mathbb{P}\,\mathrm{seq}(\mathrm{seq}\, X)$ yields, applied to a set, the set of possible permutations. The function $\frown/ : \mathrm{seq}(\mathrm{seq}\, X) \to \mathrm{seq}\, X$ flattens a sequence of sequences (i.e., the particular sequences will be concatenated to a single sequence). The function $performAll : \mathrm{seq}\, A \to D \to (D, \mathrm{seq}\, E)$ yields, applied to a sequence of actions and a data space assignment, a ordered pair comprising the new data space assignment and the sequence of all newly generated events. This function calculates the effect which results from the execution of the actions of all transitions in the firing transition set. At the end of this document we list a sample ML implementation of the function *performAll*.

Remark: From the literature [7, 8] to the semantics of Statecharts it is well known that due to the parallel execution of transition write conflicts to the data variables can happen (so-called *racing*). This happens if two transition write to the same variable. A lot of conflict resolution strategies have been proposed, for example, in the so-called interleaving semantics the conflict is resolved in a non-determinism. In our semantics such an conflict cannot happen. It is resolved by execution transitions in sequence and in the fact that more than one transition sequence is possible.

4 Summary

The semantics of UML state machines bases on the work on the semantics of Statecharts [4, 17, 5] and is adapted from the STATEMATE semantics [7] to fit into the object-oriented paradigm [6]. Our formalization of UML state machines is influenced by several works (e.g., [1, 19, 16, 2, 14, 15, 12, 13, 22, 3]). A review of state machine semantics can be found in [20]. But, most approaches focus on specialized applications with special notations, or the used subset differs from our required one. Moreover, there have been major changes in UML 2 that require a revision of previous works. We presented a *operational* semantics that is complete for the considered subset and includes all necessary definitions to implement the semantics. Moreover, it is the first formalization which includes all definitions related to complex structured data.

5 Sample ML Implementation

```
signature SM_STRUCTURE =
  sig
  type D; (* type of data space *)
  type E; (* type of events      *)
```

```
5  end;
7  signature STATEMACHINE =
     sig
9    type D; (* type of data space *)
     type E; (* type of events     *)
11
     (* type of guards: predicate over an event (from transition) and *)
13   (* a data space assignment                                       *)
     datatype G = guard of D -> bool;
15
     (* type of actions: function that creates an event or updates a  *)
17   (* data space wrt an event (from transition) and a data space    *)
     (* assignment                                                    *)
19   datatype A = event of D -> E | update of D -> D;

21   (* type of transitions (incomplete) *)
     datatype T = transition of E -> G * A list;
23
     (* example step implementation                          *)
25   (* not considering the configuration and the event pool *)
     val partialStep : T -> E * D -> D * E list;
27 end;

29 functor SM(structure sm_struc : SM_STRUCTURE) : STATEMACHINE =
     struct
31   type D = sm_struc.D;
     type E = sm_struc.E;
33
     datatype G = guard of D -> bool;
35   datatype A = event of D -> E | update of D -> D;
     datatype T = transition of E -> G * A list;
37
     (* perform one single action *)
39   (* input:  data space assignment and list of (generated) events  *)
     (* yields: new data space assignment and new event list          *)
41   (* perform single action: append a new generated event to the    *)
     (* event list or update the data space; function f holds the     *)
43   (* action                                                        *)
     fun performAction(event(f)) (d,es) = (d, es @ [f(d)])
45     | performAction(update(f))(d,es) = (f(d), es)

47   (* fold left: function f will be the composition function o and  *)
     (* the neutral element will be the id function                   *)
49   (* fold(o,[f, g, h])(n) = h o (g o ( f o n ))                    *)
     fun fold(f,n)([]) = n
51     | fold(f,n)(x::xs) = fold(f,f(x,n))(xs);

53   (* sequential composition of all actions                         *)
     (* perform all actions: apply perfomAction to every action and   *)
55   (* then (sequentially) compose all actions                       *)
     fun performAllActions(actions)(d) =
57         fold(o, fn x => x)(map(performAction)(actions))(d);

59   (* combination of guard checking and performing all actions      *)
     (* yields a new data space assignment and a list of all          *)
61   (* generated events if the guard is satisfied or the old data    *)
     (* space assignment and an empty list otherwise                  *)
```

```
63  fun partialStep(transition(t))(e,d) =
      let
65      val (guard(g), actions) = t(e);
      in
67      if g(d) then
          performAllActions(actions)(d,[])
69    else (d, [])
      end;
71 end;
```

Listing 1: Sample Implementation of a Semantical Step

$$\textbf{partialStep}(\textbf{t1})\big(\textbf{a}(\textbf{3},\textbf{true}),(\textbf{3},\textbf{true})\big) \Rightarrow \big((\textbf{5},\textbf{true}),[\textbf{a}(\textbf{4},\textbf{true}),\textbf{a}(\textbf{8},\textbf{true}),\textbf{b}(\textbf{false})]\big)$$

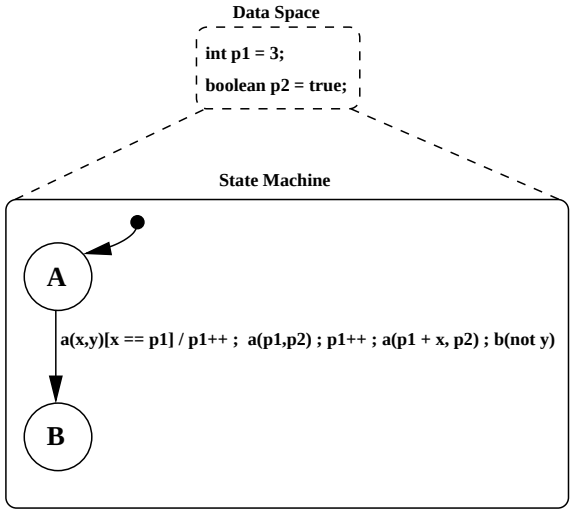


Figure 5: State Machin for the Listing 2

```
use "statemachine.ml";

(* example structure of a state machine *)
structure sm1_struc =
  struct

  (* the data space is constituted of an int value and a boolean   *)
  (* value; for example: dataspace(11, true)                       *)
  datatype D = dataspace of int * bool;

  (* event a carries an int and a boolean value and event b        *)
  (* carries a boolean value; for example: a(5,false) and b(true)  *)
  datatype E = a of int * bool | b of bool;
end;

(* instantiate state machine 1 *)
structure sm1 = SM(structure sm_struc = sm1_struc);

(* transition t1: *)
(* a(x,y)[x == p1] / p1++; a(p1, p2); p1++; a(p1+x, p2); b(not y)  *)
val t1 = sm1.transition(fn sm1_struc.a(x,y) =>
  (* guard *)
  (sm1.guard(fn sm1_struc.dataspace(p1,p2) => x = p1),

   (* effect for (dataspace(p1, p2) and a(x,y)) *)
   [

    sm1.update(fn sm1_struc.dataspace(p1, p2) =>
      sm1_struc.dataspace(p1+1, p2)),                    (* p1++ *)

    sm1.event (fn sm1_struc.dataspace(p1,p2)  =>
      sm1_struc.a(p1, p2)),                          (* a(p1, p2) *)

    sm1.update(fn sm1_struc.dataspace(p1, p2) =>
       sm1_struc.dataspace(p1+1, p2)),                   (* p1++ *)

    sm1.event (fn sm1_struc.dataspace(p1,p2)  =>
      sm1_struc.a(p1 + x, p2)),                   (* a(p1 + x, p2) *)
```

```
42
    sml.event (fn sml_struc.dataspace(p1,p2)  =>
44      sml_struc.b(not y))                              (* b(not y) *)
    ]
46  ));

48 sml.partialStep(t1)(sml_struc.a(3, true),
                       sml_struc.dataspace(3, true));
50
(* partialStep applied to t1, a(3,true) and (3,true)                *)
52 (* yields:                                                         *)
(* (dataspace (5, true), [a (4, true), a (8, true), b false])       *)
```

Listing 2: Beispiel f374r die Ausf374hrung eines semantischen Schritts

References

[1] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In *Formal Engineering Methods(ICFEM)*, LNCS. Springer, 2004. 17

[2] Rik Eshuis and Roel Wieringa. Requirements Level Semantics for UML Statecharts. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems*. Kluwer Academic Publishers, 2000. 17

[3] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P.de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In Kung-Kiu Lau and Richard Banach, editors, *Formal Engineering Methods, ICFEM05*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer-Verlag, 2005. 17

[4] David Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 1987. 2, 17

[5] David Harel. Some Thoughts on Statecharts, 13 Years Later. In *International Conference on Computer Aided Verification (GAV'97)*, 1997. 17

[6] David Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, 1997. 17

[7] David Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996. 17

[8] Steffen Helke and Florian Kammüller. Verification of Statecharts Including Data Spaces. In David Basin and Burkhard Wolff, editors, *TPHOLs03: Emerging Trends Proceedings*, Technischer Report 189, pages 177–190. Albert-Ludwigs-Universität Freiburg, 2003. 17

[9] International Organisation for Standardization. *Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics*, 2000. Reference number: ISO/IEC 13568:2002. 5

[10] Java SE Development Kit 6. Sun Microsystems, 2007. `java.sun.com`. 5

[11] Leslie Lamport. "Sometimes" is sometimes "not never": On the temporal Logic of Programs. In *popl80*, pages 164–185, 1980. 13

[12] 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*, 1999. 17

[13] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, page 465. Kluwer, 1999. 17

[14] Johan Lilius and Ivan Porres Paltor. Formalising UML State Machines for Model Checking. In *The Unified Modeling Language (UML)*, LNCS. Springer, 1999. 17

[15] Johan Lilius and Ivan Porres Paltor. The Semantics of UML State Machines. Technical Report TUCS Tecnical Report No 273, Turku Centre for Computer Science, Finland, 1999. 17

[16] Ivan Porres Paltor. *Modeling and Analyzing Software Behavior in UML.* PhD thesis, 305bo Akademi University, Department of Computer Science, 2001. 17

[17] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *Theoretical Aspects of Computer Software (TACS'91)*, pages 244–264. Springer-Verlag, 1991. 17

[18] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice-Hall, 1991. 5

[19] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001. 17

[20] Software Technology Laboratory. STL: UML Semantics Project. School of Computing, Queen's University, 2007. http://www.cs.queensu.ca/-stl/internal/uml2/ bibtex/ref_umlstatemachines.htm. 17

[21] M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992. 5

[22] Andreas Thums, Gerhard Schellhorn, Frank Ortmeier, and Wolfgang Reif. Interactive verification of statecharts. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 355–373. Springer Verlag, 2004. 17

[23] UML2. Unified Modeling Language: Infrastructure and Superstructure. Object Management Group, 2007. Version 2.1.1, formal/07-02-03, www.uml.org/uml. 2

[24] J. Woodcock and J. Davies. *Using Z. Specification, Refinement, and Proof.* Prentice-Hall, 1996. 5