

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 after 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, whereat the major version number indicates changes of the considered subset and the minor version number indicates adoptions and corrections.

The considered UML state machine subset includes:

1. Hierarchically and orthogonally structured states.
2. Multi-level transitions.
3. Use and manipulation of an associated data space.
4. Events comprising complex structured data.
5. Complex guards to control the firing of transitions.
6. Complex actions to update the data space and to generate new events.

Contents

1	Introduction	4
2	Abstract Syntax	7
2.1	States and Regions	7
2.2	Events	9
2.3	Data Space	10
2.4	Guards	10
2.5	Actions	11
2.6	Transitions	11
2.7	State Machine	11
3	Semantics	12
3.1	Active State Configuration	12
3.2	Event Store	12
3.3	Semantic State — Status	13
3.4	Transition Selection	13
3.5	Semantic Step	16
4	Summary	17
5	Sample ML Implementation	17

List of Definitions

Definition1	Node Set	7
Definition2	Type Function	7
Definition3	Attribute Mapping	7
Definition4	Subnodes Sets	8
Definition5	Well-formed Node Hierarchy	8
Definition6	Root Region	8
Definition7	Container Function	9
Definition8	Default States	9
Definition9	Event Names	9
Definition10	Event Instances	9
Definition11	Event Set	9
Definition12	Data Space	10
Definition13	Guards	10
Definition14	Actions	11
Definition15	Transitions	11
Definition16	State Machine	11
Definition17	Active State Configuration	12
Definition18	Event Store	12
Definition19	Status	13
Definition20	Enabled Transition	13
Definition21	Transition Scope	13
Definition22	Main Source and Main Target	14
Definition23	Exit Set	14
Definition24	Enter Set	14
Definition25	Conflict Relation	15
Definition26	Priority Relation	15
Definition27	Firing Transition Set	15
Definition28	Semantical Step	16

1 Introduction

The Unified Modeling Language (UML) comprises thirteen diagram types to specify structure and behavior of a system or a system component [22]. 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.

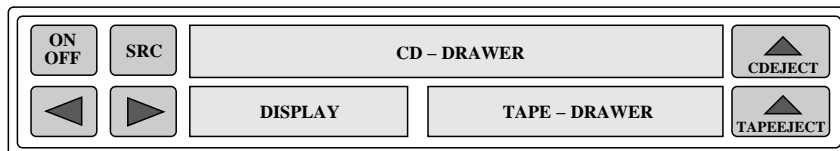


Figure 1: User interface of the Car Audio System.

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 these 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.

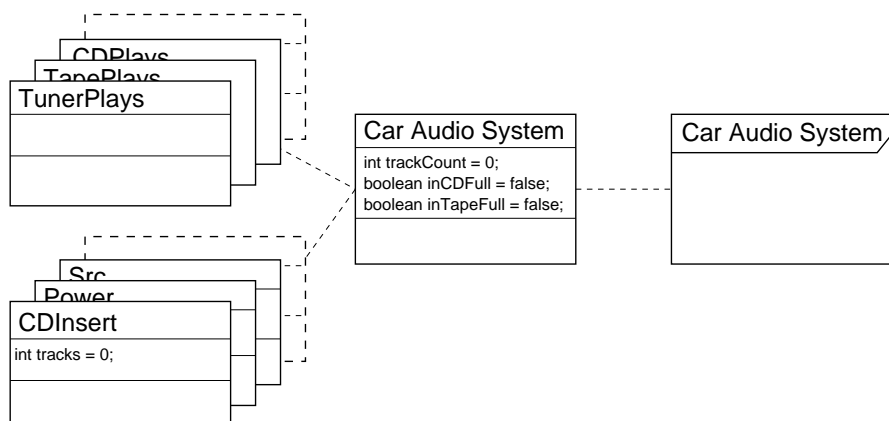


Figure 2: Class diagram for the Car Audio System.

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 `CDPlayer` and `TapePlayer` model if a tape or a compact disc are present in the system. The more complex region `AudioPlayer` 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 `TunerMode` 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 `AudioPlayer` 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 `CDEmpty` to state `CDFull` in region `CDPlayer` 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 indicates 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). □

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,

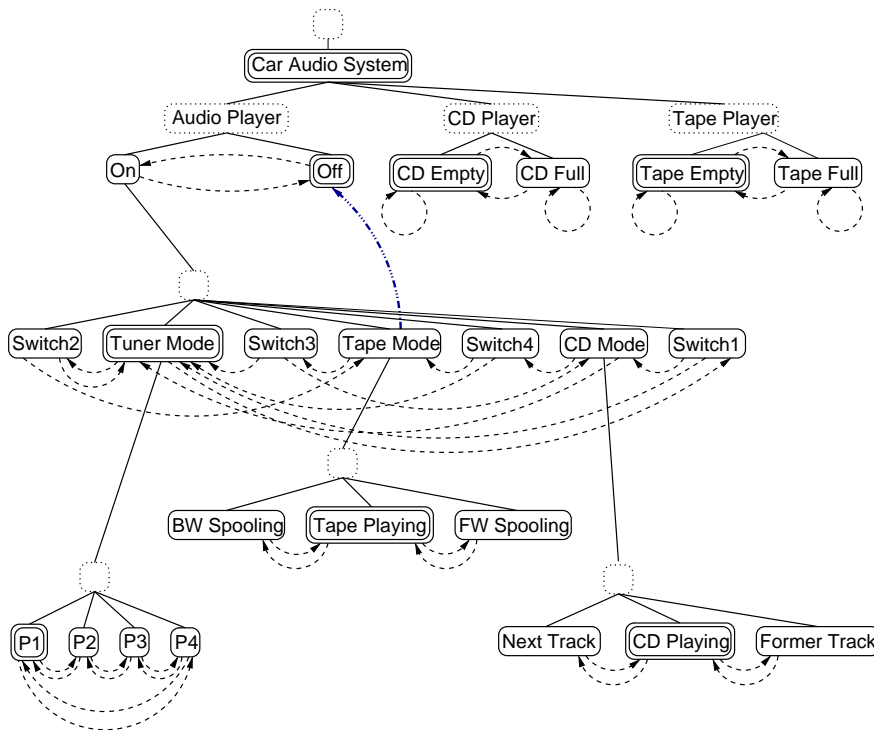
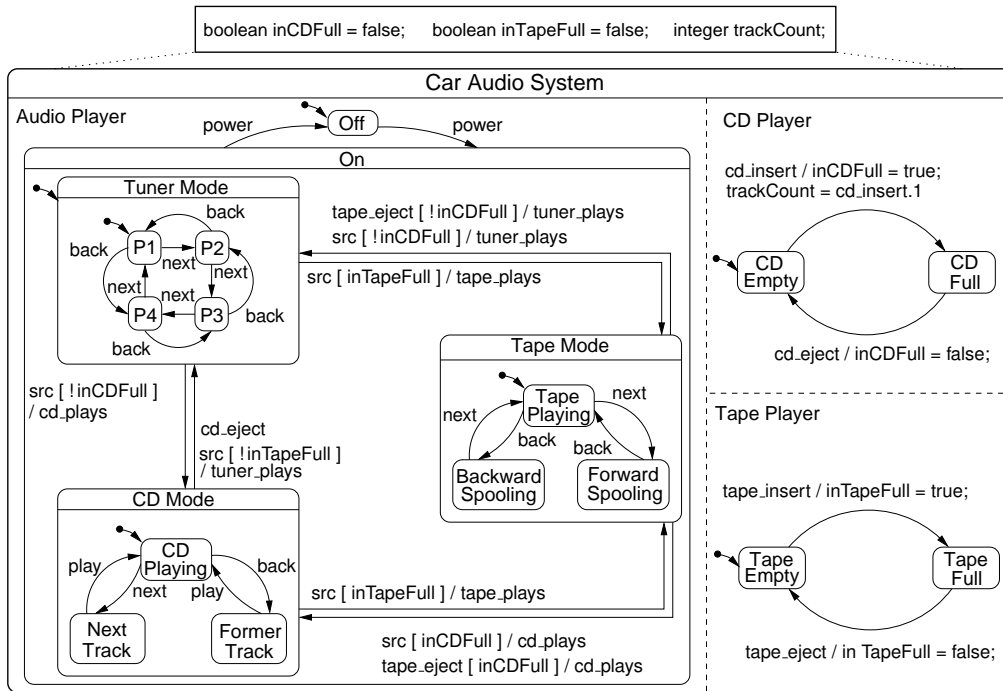


Figure 3: State machine model of the Car Audio System and the associated tree structure.

simple states at the leaves and alternating composite states and regions in between. Figure 3 shows this structure for the state machine `CarAudioSystem`. Regions are depicted as dotted nodes and have an optional name. States are depicted as simple framed nodes, whereas 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 `TapeMode` 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 [20, 23, 9, 17]. With respect to a tuple $p == (c_1, \dots, c_n)$, we denote with $p.1 = c_1, \dots, p.n = c_n$ the corresponding component of p .

2 Abstract Syntax

2.1 States and Regions

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

Definition 1 (Node Set)

Let N denote the given non-empty, finite set of states and regions. □

We distinguish among the states in N four types, namely *region*, *simple*, *simple composite* and *orthogonal*.

Definition 2 (Type Function)

Let N be the set of nodes. The function *type* takes a node as input and yields the type of the node.

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

Remark: Regions and states of a state machine are represented in the UML standard as separate classes. Hence, they are distinguished by their type. The different state types are distinguished based on three 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)

The following relation exists between the used node types and the UML class attributes.

$$\begin{aligned} simple &\Leftrightarrow isSimple \\ simple\ composite &\Leftrightarrow \neg isSimple \wedge isComposite \wedge \neg isOrthogonal \\ orthogonal &\Leftrightarrow \neg isSimple \wedge isComposite \wedge isOrthogonal \end{aligned}$$
□

Remark: 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 states contain exactly one region in which exactly one state is active (i.e., there is a exclusive choice among the states: *xor*). Orthogonal states contain at least two regions. Consequently there are at least two states simultaneously active at a time (i.e., exactly one state in every region: *and*).

The node hierarchy of a state machine is given by a tree (N, H) , where N denotes the node set and $H : \mathbb{P}(N \times N)$ denotes the non-empty subnode relation. A tuple $n \mapsto n' : H$ indicates, that node n is refined by node n' (i.e., n' is a subnode of n). Based on the subnode relation we define relative to a node three different nodes sets.

Definition 4 (Subnodes Sets)

Let (N, H) be the node hierarchy, and let H^+ denote the transitive closure of H . The functions $subnodes : N \rightarrow \mathbb{P}N$ takes a node $n : N$ as input and yields a set of direct subnodes of n . The functions $subnodes^+$ and $subnodes^*$ yield the transitive and transitive-reflexive closures, respectively.

$$\begin{aligned} subnodes(n) &== \{n' : N \mid n \mapsto n' \in H\} \\ subnodes^+(n) &== \{n' : N \mid n \mapsto n' \in H^+\} \\ subnodes^*(n) &== \{n\} \cup subnodes^+(n) \end{aligned}$$

□

Is $n_2 \in subnodes^*(n_1)$ we say that n_1 is a *supernode* of n_2 , and that n_2 is a *subnode* of n_1 . Due to the reflexivity of $subnodes^*$ n is supernode as well as subnode of oneself. Is $n_2 \in subnodes^+(n_1)$ we say that n_1 is a *strict* supernode of n_2 , and that n_2 is a *strict* subnode of n_1 .

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 simple states.

Definition 5 (Well-formed Node Hierarchy)

A well-formed node hierarchy (N, H) must preserve the following constraints.

$$\exists_1 n : N \bullet n \notin \text{ran}H \wedge n \in \text{dom}H \wedge \forall n' : N \setminus \{n\} \bullet \exists_1 n'' : N \bullet n'' \mapsto n' \in H \quad (1)$$

$$\forall n : N \mid \text{type}(n) = \text{region} \bullet (\forall n' : subnodes(n) \bullet \text{type}(n') \neq \text{region}) \quad (2)$$

$$\forall n : N \mid \text{type}(n) = \text{simple} \bullet subnodes(n) = \emptyset \quad (3)$$

$$\forall n : N \mid \text{type}(n) = \text{simple composite} \bullet (\forall n' : subnodes(n) \bullet \text{type}(n') = \text{region}) \wedge (\#subnodes(n) = 1) \quad (4)$$

$$\forall n : N \mid \text{type}(n) = \text{orthogonal} \bullet (\forall n' : subnodes(n) \bullet \text{type}(n') = \text{region}) \wedge (\#subnodes(n) > 1) \quad (5)$$

□

Constraint (1) requires in a well-formed node hierarchy a tree structure. That means, that there exists a distinguishable root node, all remaining nodes have exactly one supernode, and that there exists no loops in the structure. From the UML standard it follows that the root node must be a region (cf. Definition 6). Constraint (2) to (5) characterize the different refinement variants for states and regions. They include, that the leaves of the tree structure are simple states ($\forall n : N \mid \nexists n' : N \bullet n \mapsto n' : H \bullet \text{type}(n) = \text{simple}$).

Definition 6 (Root Region)

Let (N, H) be the node hierarchy. *root* denotes the root region of the inherent tree structure in H .

$$\begin{aligned} \text{root} &== \mu n : N \mid n \notin \text{ran}H \\ \text{type}(\text{root}) &= \text{region} \end{aligned}$$

□

The substate relation H is injective and, except for the root region, surjective. The inverse relation H^{-1} is a partial function which yields the container of a node.

Definition 7 (Container Function)

Let (N, H) be the node hierarchy. The partial function $container : N \rightarrow N$ takes a node $n : N \setminus \{root\}$ as input and yields the direct supernode $n' : N$ of n . We call n' container of n .

$$container(n) == \mu n' : N \mid n' \mapsto n \in H$$

□

We call the state which will be activated if one of its supernodes 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 states are initially active (e.g., if an enclosing orthogonal state is not initially active).

Definition 8 (Default States)

Let (N, H) be the node hierarchy. The set $\bar{N} \subseteq N$ comprises all default states.

$$\forall n : N \mid type(n) = region \bullet \exists_1 d : N \mid d \in subnodes(n) \bullet d \in \bar{N}$$

□

2.2 Events

Events are used to trigger transitions. They can be parameterized to exchange detailed information. An event comprises of a 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.

Definition 9 (Event Names)

Let \mathcal{E} denote 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_v denotes the set of all event instances to an event name $v : \mathcal{E}$.

$$E_v == v \langle \langle P_1 \times \dots \times P_n \rangle \rangle$$

□

Definition 11 (Event Set)

Let \mathcal{E} be the set of event names. The set E denotes the set of all event instances with respect to \mathcal{E} .

$$E == \bigcup_{v:\mathcal{E}} E_v$$

□

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 $v(\dots)$. To demonstrate the construction of an event set we give a small example.

Example 1 (Set of Event Instances)

Let $\mathcal{E}_1 = \{a, b\}$ be a 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_1 denotes the set of all event instances with respect to \mathcal{E} .

Parameter sets:	$P_1^a = \mathbb{N} \wedge P_2^a = \mathbb{B}$
	$P_1^b = \mathbb{B}$
Sets of event instances:	$E_a = a \langle \langle \mathbb{N}, \mathbb{B} \rangle \rangle = \{a(0, true), a(0, false), a(1, true), a(1, false), \dots\}$
	$E_b = b \langle \langle \mathbb{B} \rangle \rangle = \{b(true), b(false)\}$
Set of all event instances:	$E_1 = E_a \mid E_b = E_a \uplus E_b = \{a(0, true), a(0, false), \dots, b(true), b(false)\}$

□

The transitions of a state machine SM can only be triggered by a subset of the event instances in E . In anticipation of the following definitions 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 subsets in E_{SM} . Third, a subset which comprises all event instances which can only be 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 these subsets the following constraints apply: $E_{SM}^{prv} \uplus E_{SM}^{pub} = E_{SM}$ and $E_{SM} \uplus E_{ENV} = E$. Figure 4 illustrates the partitioning into the different subsets.

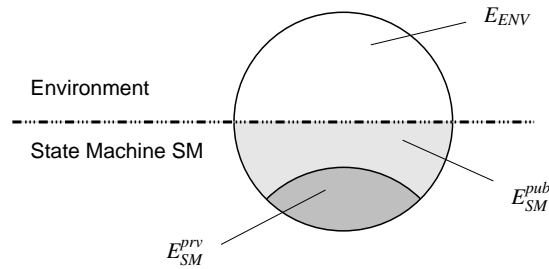


Figure 4: Partitioning of the set of event instances E .

2.3 Data Space

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

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 denotes the set of all data space assignments.

$$D == P_1 \times \dots \times P_n$$

□

2.4 Guards

A guard is a condition that provides a fine-grained control over the enabling of a transition. It is a function which with takes an event instance and a data space assignment as inputs and yields a boolean value indicating, if the specified condition is fulfilled. The expression language for guards is defined by the used action language (cf. Semantic Variation Point 1). From the UML standard it is only required that guards should be pure expressions without any side effects. Guards 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 15.

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 a 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 event instance. Updating the data space means that a new data space assignment will be selected. Generating a new event instance means that an event instance will be selected. In practice, the effect correspond to the execution of the action code and the generation of new event class instances.

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.

$$A == event\langle\langle D \mapsto E \rangle\rangle \mid update\langle\langle D \mapsto D \rangle\rangle \quad (6)$$

□

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. We denote with $n_1 \xrightarrow{e[g]/A} n_2$ a transition graphically, where n_1 denotes the source state, e denotes the triggering event instance, g denotes the guard, A denotes a sequence of actions, and n_2 denotes the target state of the transition.

Definition 15 (Transitions)

Let (N, H) be the node hierarchy, let \mathcal{E} be the set of event names, 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 $T : \mathbb{P}(N \times (E \mapsto G \times \text{seq}A) \times N)$ denotes the set of all transitions. A well-formed transition set must preserves the following constraints.

$$\forall t : T \bullet \exists_1 v : \mathcal{E} \bullet E_v \subseteq E_{SM} \wedge \text{dom}t.2 = E_v \quad (7)$$

$$\forall t : T; e : E; d : D \bullet e \in \text{dom}t.2 \wedge t.2(e).1(d) \Rightarrow \forall i : \text{dom}t.2(e).2 \bullet (\text{let } a == t.2(e).2(i) \bullet$$

$$(\forall f : D \mapsto E \mid a = event(f) \bullet d \in \text{dom}f) \wedge (\forall f : D \mapsto D \mid a = update(f) \bullet d \in \text{dom}f)) \quad (8)$$

□

Constraint 7 requires that a label function must be defined for all event instances to an event name, whereat the set of possible event instances is restricted to that of the state machine. Constraint 8 requires that if the label function is applied to an event instance and the guard function applied to a data space assignment evaluates to true, all action functions must be defined as well.

2.7 State Machine

Definition 16 (State Machine)

Let (N, H) be the node hierarchy, let \bar{N} 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.

$$SM == ((N, H), \bar{N}, E, D, G, A, T) \quad (9)$$

□

A state machine is well-formed if Definition 5 and Definition 15 apply.

3 Semantics

The semantics of state machines is defined by means of executing a hypothetical machine (i.e., to processes events and to executes transitions). This comprises the execution of semantic steps which describe the transition from one semantic state to the subsequent semantic state.

3.1 Active State Configuration

A state machine can be in more than one state at a time due to its hierarchical and orthogonal structure. We call a set of simultaneously active states an *active state configuration* or just configuration.

Definition 17 (Active State Configuration)

Let $SM = ((N, H), \bar{N}, E, D, G, A, T)$ be a state machine, and let *root* be the root region of SM . The set $C : \mathbb{P}\mathbb{P}N$ denotes the set of all active state configurations of SM .

$$C ::= \{c : \mathbb{P}N \mid (\forall s : c \bullet \text{type}(s) \neq \text{region} \wedge \\ \exists_1 s' : \text{subnodes}(\text{root}) \bullet s' \in c) \wedge \\ (\forall s : c \mid \text{type}(s) \neq \text{simple} \bullet \forall r : \text{subnodes}(s) \bullet \exists_1 s'' : \text{subnodes}(r) \bullet s'' \in c')\}$$

□

An active state configuration comprises no regions. It is required that exactly one direct substate of the root region is contained in an active state configuration. For all states in an active state configuration which are composite states it is required that for every region refining this state, exactly one direct substate is also contained in the active state configuration. An active state configuration can be visualized as a complete subtree of the state machine's tree structure. We call an active state configuration c *start configuration* if it comprises only default states $c \subseteq \bar{N}$.

3.2 Event Store

Information exchange is realized by sending and receiving of events to and from the environment of a state machine. Events for internal communication and events which will be received from the environment will be buffered until they are processed.

Semantic Variation Point 2 (Event Store)

Events will be buffered in a state machine until they are processed. The nature of this event store is not specified by the UML standard.

□

It is left open to the user of state machines to specify the behavior of the event store. To respect this variation point we define the inductive data type *event store*. An event store is either empty or it is composed of an event instance and an event store.

Definition 18 (Event Store)

Let E be the set of event instances. Q denotes the data type of an event store.

$$Q ::= \langle \rangle \mid \text{add}\langle\langle Q \times E \rangle\rangle \quad (10)$$

□

We use the function $\oplus : Q \times \text{seq}E \rightarrow Q$ to add a sequence of event instances to an event store and we use the partial function $\ominus : Q \rightarrow Q \times E$ to remove an event instance from a non-empty event store. With it we abstract from a special order of event instances in an event store. In most practical applications, the type Q is instantiated by a FIFO-queue.

3.3 Semantic State — Status

A semantic state of a state machine, called a status, comprises an active state configuration, an event store and a data space assignment.

Definition 19 (Status)

Let $SM = ((N, H), \bar{N}, E, D, G, A, T)$ be a state machine, let C be the set of configurations, and let Q be the set of event stores. The set Z denotes the set of all status of SM .

$$Z == C \times Q \times D \quad (11)$$

□

We denote with $\llbracket c, q, d \rrbracket$ a member of Z . An initial status comprises the start configuration.

3.4 Transition Selection

During a semantic step the state machine moves from one status to the subsequent status. If an event instance is selected for processing one or more transitions can be activated for firing. If no transition is activated, the event instance will be discarded. Please note, currently we do not use the notion of deferring events.

Semantic Variation Point 3 (Discarding Events)

Although the UML standard defines that an event instance which does not enable a transition and which is not in the deferred event list will be discarded, we note that in practice this behavior could be unwanted. For this reason we mark the discarding of event instances as a semantic variation point.

□

A transition is called enabled, if the source state of the transition is included in the current configuration, the event name is equal to the trigger name and the guard evaluates to true.

Definition 20 (Enabled Transition)

Let $SM = ((N, H), \bar{N}, E, D, G, A, T)$ be a state machine, let $\llbracket c, q, d \rrbracket : Z$ be a status, and let $(q', e) == \ominus q$ be the result from removing an event instance e from the event store q . The function $enabled : T \times C \times E \times D \rightarrow \mathbb{B}$ takes a transition $t : T$, a configuration c , an event instance e , and a data space assignment $d : D$ as input and yields a boolean value indicating if the transition is enabled.

$$enabled(t, c, e, d) == \begin{cases} true & \text{if } (t.1 \in c) \wedge (e \in \text{dom } t.1) \wedge (t.2(e).1(d)) \\ false & \text{otherwise} \end{cases} \quad (12)$$

□

Transitions are allowed to cross multiple level in the node hierarchy. We call such transitions *multi-level transitions*. To correctly deal with multi-level transitions we need to know the scope of a transition. The scope allows to identify the set of states which will be exited and entered if a transition is fired. This information is needed to identify conflicting transitions among the set of enabled transitions. The scope of a transition is identified by the *least common ancestor* of the source and target state. It is of type region or orthogonal.

Definition 21 (Transition Scope)

Let $SM = ((N, H), \bar{N}, E, D, G, A, T)$ be a state machine. The function $lca : T \rightarrow N$ takes a transition $t : T$ as input and yields the least common ancestor of the source and target state.

$$lca(t) == \mu n : N \mid (t.1 \in \text{subnodes}^+(n)) \wedge (t.3 \in \text{subnodes}^+(n)) \wedge (\nexists n' : N \mid n' \in \text{subnodes}^+(n) \bullet t.1 \in \text{subnodes}^+(n') \wedge t.3 \in \text{subnodes}^+(n')) \quad (13)$$

□

Based on the scope we can identify the *main source* and the *main target* of a transition. These two states represent the root nodes of the subtrees which are affected by a transition.

Definition 22 (Main Source and Main Target)

Let $SM = ((N, H), \bar{N}, E, D, G, A, T)$ be a state machine. The function $mainSource : T \rightarrow N$ and the function $mainTarget : T \rightarrow N$ take a transition $t : T$ as input and yield the main source and the main target of t , respectively.

$$mainSource(t) == \begin{cases} \mu s : subnodes(lca(t)) \mid t.1 \in subnodes^*(s) & \text{if } type(lca(t)) = region \\ lca(t) & \text{if } type(lca(t)) = orthogonal \end{cases} \quad (14)$$

$$mainTarget(t) == \begin{cases} \mu s : subnodes(lca(t)) \mid t.3 \in subnodes^*(s) & \text{if } type(lca(t)) = region \\ lca(t) & \text{if } type(lca(t)) = orthogonal \end{cases} \quad (15)$$

□

The set of states which will be exited if a transition fires contains all substates of the main source that are included in a configuration.

Definition 23 (Exit Set)

Let $SM = ((N, H), \bar{N}, E, D, G, A, T)$ be a state machine, and let Z be the set of status. The function $exits : T \rightarrow \mathbb{P}N$ takes a transition $t : T$ and a status $[[c, q, d]] : Z$ as input and yields the set of states which will be exited by transition t .

$$exits(t, [[c, q, d]]) == subnodes^*(mainSource(t)) \cap c \quad (16)$$

□

The set of states which will be entered if a transition fires contains all substates of the main target which contain the target state or which are implicitly entered by the transition (default states).

Definition 24 (Enter Set)

Let $SM = ((N, H), \bar{N}, E, D, G, A, T)$ be a state machine. The recursive function $enters : N \times N \rightarrow \mathbb{P}N$ takes two nodes $n_1, n_2 : N$ as input and yields with respect to n_1 the set of states which will be entered if n_2 is entered.

$$enters(n_1, n_2) == \begin{cases} \{n_1\} & \text{if } type(n_1) = simple \\ \{n_1\} \cup \bigcup_{n' \in subnodes(n_1)} enters(n', n_2) & \text{if } type(n_1) = simple\ composite \vee orthogonal \\ enters(n'', n_2) & \text{if } type(n_1) = region \end{cases} \quad (17)$$

Where $n'' = \mu n''' : subnodes(n_1) \mid (n_2 \in subnodes^*(n''')) \vee (n''' \in \bar{N} \wedge n_2 \notin subnodes^+(n_1))$.

□

If parameter n_1 is of type simple, just this state will be entered. Is parameter n_1 is of type simple composite or orthogonal, this state and its contained regions will be entered ($subnodes(n_1)$). If parameter n_1 is of type region, we differentiate two alternatives. Either, the direct substate n''' will be entered which contains the target state — $target \in subnodes^*(n''')$ 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 are moving down to the leaves of the state hierarchy (i.e., the target state is not a substate of n_1) — $n''' \in \bar{N} \wedge target \notin subnodes^+(n_1)$. 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.

The function *enters* initially applied to the main target (initial parameter n_1) and target of a transition (parameter n_2) yields all states which will be entered by the transition. We use the abbreviation *enters*(t) instead of *enters*(*mainTarget*(t), t .3).

It is possible that more than one transition is enabled by an event instance. If more than one transition is enabled it may be the case that one or more transitions are in conflict with each other. Such a conflict appears, if two transitions exit same states. Firing both transitions would lead to an ill-formed status. A set of non-conflicting transitions must be selected for firing.

Definition 25 (Conflict Relation)

Let $SM = ((N,H),\bar{N},E,D,G,A,T)$ be a state machine. The relation $\# : T \times T$ relates two transitions $t_1, t_2 : T$ if they are in conflict with each other.

$$\# == \{(t_1, t_2) : T \times T \mid exits(t_1) \cap exits(t_2) \neq \emptyset\} \quad (18)$$

□

We call two transitions $t_1, t_2 : T$ conflict-free, denoted $t_1 \parallel t_2$, if they are not in conflict ($\parallel == \neg \#$).

Conflict resolution is carried out in two steps. In the first step, transitions with the highest priority are selected. The used priority scheme is defined on the relative positions of the source states in the node hierarchy. A transition, whose source state is a strict substate of the source state of another transition has priority over this transition.

Definition 26 (Priority Relation)

Let $SM = ((N,H),\bar{N},E,D,G,A,T)$ be a state machine. The relations $\prec : T \times T$ relates two transitions $t_1, t_2 : T$ if t_1 has priority over t_2 , denoted $t_1 \prec t_2$.

$$\prec == \{(t_1, t_2) : T \times T \mid t_1.1 \in subnodes^+(t_2.1)\} \quad (19)$$

□

If there are still conflicts among selected transitions, we identify sets of transitions in a second step, which are maximal with respect to their cardinality and in which all transitions are pairwise conflict-free. Altogether, a so-called *transition selection algorithm* selects conflict-free subsets of enabled transitions using the given priority scheme.

Definition 27 (Firing Transition Set)

Let $SM = ((N,H),\bar{N},E,D,G,A,T)$ be a state machine, let Z be the set of status, let Q be the set event stores, let $q : Q$ be an events store and $e : E$ be an events instance such that $(q', e) = \ominus(q)$, and let $\llbracket c, q, d \rrbracket : Z$ be a status of SM . A firing transition set $T_{\parallel} \subseteq T$ must fulfill the following constraints.

$$\forall t : T_{\parallel} \bullet enabled(t, c, e, d) \quad (20)$$

$$\forall t_1, t_2 : T_{\parallel} \mid t_1 \neq t_2 \bullet t_1 \parallel t_2 \quad (21)$$

$$\nexists t' : T \setminus T_{\parallel} \mid enabled(t', c, e, d) \bullet \forall t : T_{\parallel} \bullet t \parallel t' \vee t' \prec t \quad (22)$$

□

Constraint 20 requires all transitions in a firing transition set to be enabled. Constraint 21 requires all transitions in a firing transition set to be pairwise conflict-free. Constraint 22 requires that there exists no transition outside the set which is enabled and conflict free with respect to all transitions in the set, or which is enabled and has priority over a transition in the set.

Semantic Variation Point 4 (Firing Transition Set Selection)

It is possible that there exists more than one valid firing transition set at a time. The UML standard does not specify which set to choose for the next semantic step.

□

the execution or firing of 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 the chosen firing transition set contains more than one transition the order in which the transition are executed is not fixed.

Semantic Variation Point 5 (Firing Transition Set Execution Order)

The UML standard does not specify an order in which transitions in a firing transition set will be executed.

□

3.5 Semantic Step

A semantic step describes the transition from one status to the subsequent status. We distinguish two cases. The first case covers the situation in which the event store does not contain an event instance. And the second case covers the situation in which an event instance can be removed from the event store. In the first case, the configuration and the data space assignment remain unchanged. Only event instances received from the environment are added to the event store. In the second case, a firing transition set is identified and the contained transitions are executed. This may comprise updating the data space and generating new event instances. Generated event instances which belong to the state machine are added to the event store. Finally, event instances received from the environment are added to the event store.

A semantic step is atomic and the next semantic step can only be executed, if the current semantic step is completely finished — a so-called *run-to-completion* semantics. We denote with $\llbracket c, q, d \rrbracket \xrightarrow{E_{in}, E_{out}} \llbracket c', q', d' \rrbracket$ a semantic step $(\llbracket c, q, d \rrbracket, E_{in}, E_{out}, \llbracket c', q', d' \rrbracket) : Z \times \text{seq} E_{SM}^{pub} \times \text{seq} E_{ENV} \times Z$.

Definition 28 (Semantical Step)

Let $SM = ((N, H), \bar{N}, E, D, G, A, T)$ be a state machine, let Z be the set of status, let $\llbracket c, q, d \rrbracket : Z$ be a status, let $T_{\parallel} \subseteq T$ be a valid firing transitions set, and let $E_{in} : \text{seq} E_{SM}^{pub}$ a sequence of received event instances. A semantic step $\llbracket c, q, d \rrbracket \xrightarrow{E_{in}, E_{out}} \llbracket c', q', d' \rrbracket : Z \times \text{seq} E_{SM}^{pub} \times \text{seq} E_{ENV} \times Z$ is defined as follows.

$$\begin{array}{c}
 q = \langle \rangle \\
 q' = \oplus(q, E_{in}) \\
 \hline
 \llbracket c, q, d \rrbracket \xrightarrow{E_{in}, \langle \rangle} \llbracket c, q', d \rrbracket
 \end{array}
 \quad (23)$$

$$\begin{array}{c}
 q \in \text{ran } add \\
 (q'', e) = \ominus(q) \\
 c' = (c \setminus \bigcup_{t \in T_{\parallel}} \text{exits}(t)) \cup \bigcup_{t \in T_{\parallel}} \text{enters}(t) \\
 A_{seq} \in \text{perm}(\{t : T_{\parallel} \bullet t.2(e).2\}) \\
 (d', E_{gen}) = \text{performAll}(\wedge / A_{seq})(d) \\
 (E_{int} = E_{gen} \upharpoonright E_{SM}) \wedge (E_{out} = E_{gen} \upharpoonright E_{ENV}) \\
 q' = (q'' \oplus E_{int}) \oplus E_{in} \\
 \hline
 \llbracket c, q, d \rrbracket \xrightarrow{E_{in}, E_{out}} \llbracket c', q', d' \rrbracket
 \end{array}
 \quad (24)$$

Where $E_{out} : \text{seq} E_{ENV}$ denotes the sequence of generated outgoing event instances and $E_{int} : \text{seq} E_{SM}$ denotes the sequence of generated internal event instances.

□

The event instances in E_{out} are sent by the state machine to the environment (i.e., to the particular system components). The function $\text{perm} : \mathbb{P} \text{seq} X \rightarrow \mathbb{P} \text{seq}(\text{seq} X)$ takes a set as input and yields the set of possible permutations. The function $\wedge / : \text{seq}(\text{seq} X) \rightarrow \text{seq} X$ flattens a sequence of sequences (i.e., the particular sequences will be concatenated to a single sequence). The function $\text{performAll} : \text{seq} A \rightarrow D \rightarrow (D, \text{seq} E)$ takes a sequence of actions and a data space assignment as input and yields an ordered pair comprising the new data space assignment and the sequence of generated events. performAll calculates the intrinsic effect

which results from the execution of all transitions in the firing transition set. To illustrate Definition 28 we list a sample ML implementation in Section 5.

Remark: From the literature related to the semantics of Statecharts (e.g., [7, 8]) it is well known that write conflicts to data variables can happen due to the parallel execution of transitions. This happens if two transition try to write to the same data variable. A lot of conflict resolution strategies have been proposed, for example, the so-called interleaving semantics, where conflicts are resolved in non-determinism. Such conflicts cannot happen during the execution of transitions in UML state machines. These conflicts are avoided by the sequential execution of transitions.

4 Summary

The UML state machines semantics is adapted from the STATEMATE semantics [7] to fit into the object-oriented paradigm [6]. The Statemate semantics itself is related to the classical Harel Statecharts [4, 16, 5].

We presented an operational semantics which is complete for the considered subset and which includes all necessary definitions to implement it. Moreover, it is the first formalization which includes all definitions related to the use of complex structured data in state machines. The presented semantics partly benefits from previous works (e.g., [1, 18, 15, 2, 13, 14, 11, 12, 21, 3]). A review of several semantics can be found in [19]. Most approaches focus on specialized applications and use specialized notations, or the used subset differs from ours. Moreover, there have been major changes in UML 2 that require a deep revision of previous works.

5 Sample ML Implementation

```
1 signature SM_STRUCTURE =
  sig
3   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_struct : SM_STRUCTURE) : STATEMACHINE =
  struct
31   type D = sm_struct.D;
   type E = sm_struct.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)])
     | performAction(update(f))(d,es) = (f(d), es)
45
   (* fold left: function f will be the composition function o and *)
   (* the neutral element will be the id function                  *)
47   (* fold(o,[f, g, h])(n) = h o (g o (f o n))                    *)
   fun fold(f,n)([]) = n
     | fold(f,n)(x::xs) = fold(f,f(x,n))(xs);
49
   (* sequential composition of all actions                        *)
   (* perform all actions: apply performAction to every action and *)
51   (* then (sequentially) compose all actions                    *)
   fun performAllActions(actions)(d) =
     fold(o, fn x => x)(map(performAction)(actions))(d);
53
   (* combination of guard checking and performing all actions    *)
   (* yields a new data space assignment and a list of all        *)
55   (* generated events if the guard is satisfied or the old data  *)
   (* space assignment and an empty list otherwise                *)
61   fun partialStep(transition(t))(e,d) =
     let
63       val (guard(g), actions) = t(e);
     in
65       if g(d) then
67         performAllActions(actions)(d,[])
69       else (d, [])
     end;
71 end;

```

Listing 1: Sample Implementation of a Semantical Step

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

```

4 use "statemachine.ml";
6 (* example structure of a state machine *)
  structure sml_struct =
8   struct
10   (* the data space is constituted of an int value and a boolean *)

```

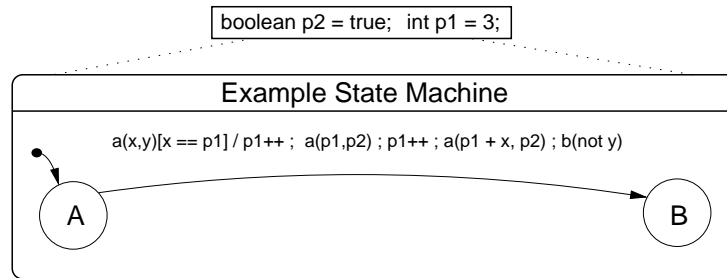


Figure 5: State Machin for the Listing 2

```

12 (* value; for example: dataspace(11, true) *)
datatype D = dataspace of int * bool;

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

18 (* instantiate state machine 1 *)
20 structure sm1 = SM(structure sm_struct = sm1_struct);

22 (* transition t1: *)
(* a(x,y)[x == p1] / p1++; a(p1, p2); p1++; a(p1+x, p2); b(not y) *)
24 val t1 = sm1.transition(fn sm1_struct.a(x,y) =>
    (* guard *)
26 (sm1.guard(fn sm1_struct.dataspace(p1,p2) => x = p1),

28 (* effect for (dataspace(p1, p2) and a(x,y)) *)
[
30
32 sm1.update(fn sm1_struct.dataspace(p1, p2) =>
    sm1_struct.dataspace(p1+1, p2)), (* p1++ *)

34 sm1.event (fn sm1_struct.dataspace(p1,p2) =>
    sm1_struct.a(p1, p2)), (* a(p1, p2) *)

36 sm1.update(fn sm1_struct.dataspace(p1, p2) =>
38 sm1_struct.dataspace(p1+1, p2)), (* p1++ *)

40 sm1.event (fn sm1_struct.dataspace(p1,p2) =>
    sm1_struct.a(p1 + x, p2)), (* a(p1 + x, p2) *)

42 sm1.event (fn sm1_struct.dataspace(p1,p2) =>
44 sm1_struct.b(not y)) (* b(not y) *)
]
46 ));

48 sm1.partialStep(t1)(sm1_struct.a(3, true),
    sm1_struct.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 Balsler, 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. 4, 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. 7
- [10] Java SE Development Kit 6. Sun Microsystems, 2007. java.sun.com. 5
- [11] 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
- [12] 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
- [13] Johan Lilius and Ivan Porres Paltor. Formalising UML State Machines for Model Checking. In *The Unified Modeling Language (UML)*, LNCS. Springer, 1999. 17
- [14] Johan Lilius and Ivan Porres Paltor. The Semantics of UML State Machines. Technical Report TUCS Technical Report No 273, Turku Centre for Computer Science, Finland, 1999. 17
- [15] Ivan Porres Paltor. *Modeling and Analyzing Software Behavior in UML*. PhD thesis, 305bo Akademi University, Department of Computer Science, 2001. 17
- [16] 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
- [17] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice-Hall, 1991. 7
- [18] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001. 17

- [19] 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
- [20] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992. 7
- [21] 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
- [22] UML2. Unified Modeling Language: Infrastructure and Superstructure. Object Management Group, 2007. Version 2.1.1, formal/07-02-03, www.uml.org/uml. 4
- [23] J. Woodcock and J. Davies. *Using Z. Specification, Refinement, and Proof*. Prentice-Hall, 1996. 7