A Virtual Machine Supporting Multiple Statechart Extensions

Thomas Huining Feng
Modelling, Simulation and Design Lab
McGill University, Canada
`http://msdl.cs.mcgill.ca/`

Keywords: Statechart Extensions, Virtual Machine, Model Execution, Modularity, Model Reuse.

ABSTRACT

Because of its power in describing large systems, statechart is widely used to model both software systems and physical systems. As the distinction between the design phase and implementation phase is becoming more and more insignificant, it would be very nice if a model described in statechart can be executed before it is finished. Moreover, early model analysis, verification and code generation are also important.

Based on the implementation of the Statechart Virtual Machine, this article briefly describes the execution of statechart models, and discusses three important extensions to the statechart formalism in detail: submodel importation, tunable transition priority to solve conflicts and parameterized model templates.

INTRODUCTION

Statechart is a powerful tool to describe both software systems and physical systems. It is possible for an automatic system to take in a (system-dependent) statechart model and execute it, though the statechart semantics is not precisely defined. Early model analysis, verification and code generation are also desirable, because the boundary of design phase and implementation phase in the development process is becoming more and more obscure in the current trend. If a prototype can be executed and tested before the model if finished, design errors can be discovered earlier, and the cost to fix them is much less.

The SVM (Statechart Virtual Machine) is an attempt to the execution of statechart models. It is also an experimental environment to test and analyze semantic elements in the statechart formalism. A statechart model is written in a text file, which defines all its the states and transitions. The SVM accepts this self-contained (if it does not explicitly import other models) textual model as input, and simulates its execution.

Since this article is not aimed at describing the general semantics of statechart, the following sections will mostly focus on certain semantic extensions, which are found to be very useful in specifying complex systems. For general definition of the statechart semantics, the readers are referred to these articles: [1], [2] and [3].

As mentioned, the dependence on system and programming language still exists, though efforts are spent to minimize it. More information about a rigorous system-independent semantics for the UML statechart can be found in [4], [5] and [6].

MODEL EXECUTION

The goal is to build a generalized simulator capable of executing statechart models. There are two considerations: the common properties of all the statechart models should be figured out and the simulator must support as many of them as possible; enough flexibility is given to designers to build virtually all kinds of models for the use in different fields. The second consideration implies extensions must be made if the original statechart formalism is not powerful enough.

- There must be some way to specify a statechart model, either in a graphical form or in a textual form or both. The simulator must be able to take in this description and understand it. The language must be expressive enough to support all well-defined semantic elements. Every model written in this language must have a unique precise meaning. The simulator is in this sense a *virtual machine* capable of interpreting this language and executing the model (or the code generated from the model) without ambiguity.
- A good virtual machine should be portable to various systems itself. Otherwise, because currently there is no standard on the statechart semantics or file format, there is no hope for another virtual machine on another system to fully support its language, so there is no hope for the models designed in this language to be portable either.
- It is admitted that the statechart semantics exhibits its weakness in some cases. I.e., when the state space grows to an unlimited size (which is very natural in both physical systems and software systems), infinite number of states must be created. If no extension is made to the original statechart formalism, the use of the virtual machine is too restricted; if extensions are made, they must not violate the basic rules like modularity, substitutability, and in addition, reusability.
- The performance of the virtual machine is also important, because the domains where it is used are unpredictable. Some of them may have critical requirement on performance.
- Other than merely simulating the execution of models, the abilities of analyzing model performance, verifying

their correctness, testing (or debugging) them and generating code from them (if the virtual machine is not based on an interpretive language) are also necessary.

SVM supports common semantic elements in statechart including states and their properties (default, concurrent/orthogonal, final, enter actions and exit actions), transitions and their properties (events, sources, destinations, guards and output), state hierarchy, history (`H`) and deep history (`H*`), and so on. Their semantics is described in other articles (some of which are among the references) and will not be repeated here.

There are still many cases where it is difficult to design a complex and precise model with these basic semantic elements. Extensions to them are discussed in detail in the following sections.

SVM enables testing models in an early execution, and debugging them in the run-time debugger. Testers are allowed to look inside the model and execution environment. If they know the Python script language, they can modify variables or states.

IMPORTATION AND MODEL REUSE

The complexity of a statechart model exhibits itself even in solving small problems. For larger systems the state dimension grows to such an extent that the model cannot be written in a single file (or component). Furthermore, as the model becomes larger and larger, it is more and more difficult to keep it modular and reusable.

Adding hierarchy to state diagram partly solves this problem. Hierarchy itself has no definition on model reuse, i.e., how to specify the reusing model, the reused model, and the behavior of this reuse.

To design a good reuse scheme, the following two points must be considered:

- Key behavior is guaranteed to remain intact in the reused model for modularity.
- At the same time the reusing model is given enough flexibility to adjust the behavior of the reused model.

SVM presents a general idea of model importation. An imported model is a full-function model in its own right. This means, instead of being imported only, it can also be executed and tested as a separate component. When imported, all its states and transitions are preserved. These states are certainly not scattered about the importing model. Otherwise there is no means to guarantee the well-formedness of both the two models. So these imported states must be substates contained in a single state of the importing model.

In SVM, for a model to import another, one of its states must have the `IMPORTATION` property specifying (by name) which model is to be imported. Then, it can be imagine that all the states and transitions of the imported model are included in this state. For example, because SVM uses a dot-notation to denote the paths of states, we may have a state in model `M1` like `A.B[import M2]`. Also suppose there is a model `M2`

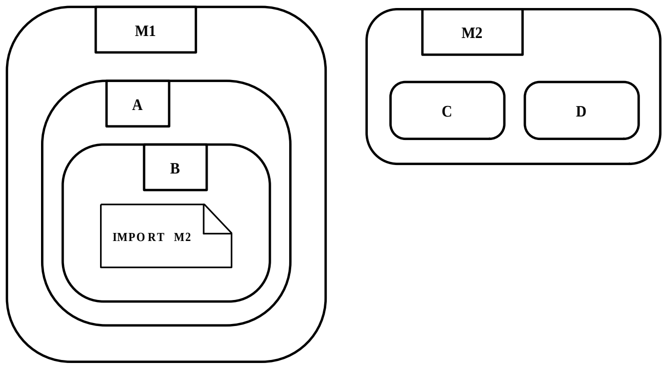

Figure 1: An example of model importation

with states `C` and `D` (they may have substates). Then after the importation is actually done at run-time, `C` and `D` are substates of `A.B` in the context of `M1` (`A.B.C` and `A.B.D`). Properties of `C` and `D` are preserved, like `Default`, `Concurrent` (or `Orthogonal`) and even their `IMPORTATION` properties if any. All the transitions in `M2` are also copied with the name of source states and destination states accordingly changed. However, after the importation is done, the run-time system no longer has any knowledge about model `M2`. Its name is completely forgotten.

Importing a model, though specified statically in the model description file, is done only when necessary. In the example where `A.B` imports model `M2`, only when a transition requires a substate of `A.B` is the submodel loaded and its states are combined with those of `M1`. This, though decreases the run-time performance, makes it possible for a model to directly or indirectly import itself. This recursive state hierarchy with infinite number of levels usually simplifies complex systems. A concrete example is given in the latter part.

When designing importation, there are two rules to follow:

- For the importing model, the submodel is a black box with its own behavior. It is possible to specify parameters for it (described later), but the importing model is not allowed to modify the behavior in the black box in other ways.
- The imported model is not allowed to modify the behavior of the importing model either. It only *adds* functionality to the importing model, or in another word, specializes one of its states.

The statechart simulator must conform to these rules to keep modularity and substitutability.

Additionally, if a state specified to import also has its own substates, i.e., state `A.B` in `M1` imports `M2` and it also has a substate `E`, then a warning is given at load-time. This is because if states `C` and `D` defined in `M2` are orthogonal states, while `A.B.E` in `M1` is not, then the three substates of `A.B` are incompatible.

There is no limit on how many times a model is allowed to import or be imported. Any of its states can be associated with an `IMPORTATION` property, and some or all of them may import the same submodel. As long as the above well-formedness rules are satisfied, the decision is left to the designer.

UNTANGLING TRANSITION CONFLICTS

Whenever two or more transitions are enabled by the same event there is a *run-time* conflict. All those transitions are capable of handling the event, and their guards are all satisfied. To randomly pick one of them and fire it is not a bad idea. However most of the simulators choose the transition in an implementation-dependent way, so the model may run very well in a system while misbehaves in another.

Two possible kinds of transition conflicts are described in [2].

- At least two transitions are enabled by the same event, one of them is from a state at a lower level; the other is from a higher level in the hierarchy.
- At least two transitions are enabled by the same event, they have the same source state.

Solutions to the first kind of conflicts are found in both the STATEMATE semantics [2] and the UML [7]. Unfortunately, the solutions from the two sources are contrary: in UML, if the source state of a transition is a substate of the source state of the other, it gets higher priority; however, in the STATEMATE semantics, it gets lower priority.

It is possible to customize the priority of transitions by setting a global option in an SVM model: `InnerTransitionFirst`. If it is true, the transition from an inner state always has higher priority than the one from an outer state; and vice versa.

This does not solve the problem completely, taking into account model importation. What if an inner-transition-first model is imported into an outer-transition-first one (or the contrary)? The rules of importation must be conformed to, so neither the option of the submodel nor that of the supermodel can be changed. In another word, a single global option is not enough. The option must be allowed to vary in the state hierarchy.

In SVM, there are two ways to modify this option:

- The first way to change transition priority is by importing a submodel. The submodel has its own global setting, either it is `Inner Transition First=1` or `Inner Transition First=0` (Default). This setting is preserved when it is imported, and thus the priority of its transitions is preserved.
- The idea is carried on by identifying the fact that a submodel is just a set of substates and transitions within a state of the supermodel. Importing a model (ignoring possible parameters) is identical with writing all its contents in the appropriate places, with the paths of all its states accordingly modified. In this view, it is also reasonable to customize the priority of transitions within the scope of any state.

  This extension to the statechart is introduced in SVM. Any state can have one of the following properties: `ITF`, `OTF` and `RTO`. The `ITF` property means within the scope of this state, inner transitions get higher priority than outer ones. `OTF` takes up the contrary meaning. `RTO` (Reverse Transition Ordering) means the priority of transitions within the state is different from its outer context. A substate can override this setting of its parent within its scope (constituted by itself and all its substates). If it does not override this setting, it is subject to the setting of its parent.

The term *transition ordering* is used in SVM because in its implementation, transitions are sorted in a list in the decreasing order of their priority. This is from the consideration of run-time performance. Whenever a conflict occurs in the execution, the simulator blindly takes the first enabled transition from the list, and this must be the one with highest priority. So the sorting, though very complex due to the tunable transition priority and importation, is carried out only when a model is loaded by the simulator or imported.

Since a substate can have its setting which overrides its parent's, one may ask whether the substitution of a substate affects the behavior of its parent, hence breaking modularity. Obviously, this cannot happen, because the change of its property is restricted in its scope.

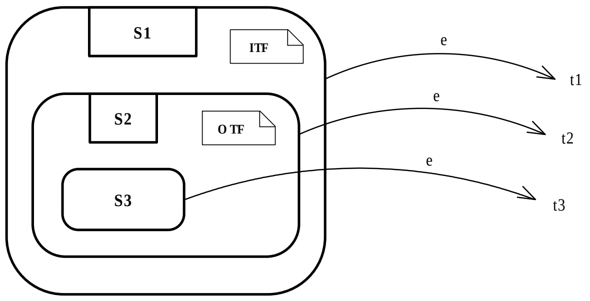

Figure 2: Customizing the priority of transitions

Suppose there are three transitions `t1`, `t2` and `t3` as illustrated in Figure 2. When event `e` occurs, they are all enabled, so there is a conflict of the first kind. To understand the priority of these transitions, one must first consider the outermost state and then step inward. Because `S1` is specified to be `ITF`, the priority of `t1` must be lower than both `t2` and `t3`. Since `S2` is `OTF`, `t2` has a higher priority than `t3`. So the ordering by priority is `t2`, `t3`, `t1`.

Now suppose state `S1` is `OTF` and `S2` is `ITF`. Also consider the outermost state `S1` first. Because `S1` is `OTF`, `t1` has a higher priority than `t2` and `t3`; on the other hand, `S2` is `ITF`, meaning `t3` has a higher priority than `t2`. So the order becomes `t1`, `t3`, `t2`.

The consideration always starts from the outer states. It is obvious that transitions in the inner states get higher priority only when the outer states grant the priority to them by an `ITF`. Conversely, `OTF` gives the transitions from outer states higher priority regardless of the settings of their substates. When the outer states are themselves substates, they are also subject to the setting of their parents. This maintains the well-formed behavior of the importing model.

On the other hand, because submodels always have their global option `InnerTransitionFirst`, the priority inside it is deterministic. The importing model is not capable of changing the priority inside the black box either.

The use of these properties to customize transition ordering gives extra power to statechart: the designer is able to explicitly specify which transition within a hierarchy gets the highest or lowest priority. Such a transition always has a turning point as its source state. The transition ordering in the outer context of the turning point is different from the context inside it; namely, this state has a property that changes the transition ordering, like `S2` in the example. Of course, the highest or lowest priority is a relative concept only applicable in a certain scope.

To solve the second kind of conflicts, it is desirable to guarantee mutual exclusiveness of the transitions with the same source state. But this is usually difficult, because transitions have boolean expressions as guards, which only have their value at run-time.

In SVM, each transition is associated with an integer priority number. Whenever there is a conflict which cannot be solved by the above scheme, the transition with the smallest priority number is fired. By default, each transition has a priority of 0.

If conflicts are still found even if these two mechanisms are used, the result is implementation-dependent.

PARAMETERIZED MODEL TEMPLATES

When reusing a model by importation, it is possible to specify one or more parameters. In SVM, the parameterized model is called a *template*, and a parameter is called a *macro redefinition*.

Macros are defined in the `MACRO` part of the statechart text file. Occurrence of the macro left-hand side is literally substituted by the right-hand side before any processing of the model. When a macro has parameters in parentheses following it, it is much like a single-line function definition.

It is recommended that macros be widely used in the model, because they reduces language-dependence (e.g., in the guards and in the output part). As an extreme case, all these parts are specified with macros, and then there would be no programming language dependency in the model. Changing programming language only results in the change of the whole macro set. If the macros are defined in the OCL or action semantic syntax, then the model is universally executable (in theory). It is possible to reach this point, because only boolean expressions are accepted as guards, and only sequential commands are allowed for output.[1]

Macros can be redefined at load-time. When a model is first loaded in the SVM environment, the redefined macros are specified as command line parameters. When a model is imported, the importing model may specify parameters for it. These parameters (redefined macros) override the original macro definitions in it. For example, the `DUMP` macro is defined as `DUMP(msg)=print '[msg]'` to print out debugging messages in a reusable model. When it is imported as a black box, the dump message is not important for the importing model. The importing model thus gives a parameter `DUMP(msg)=` to disable all debugging output from the submodel.

The outside world is able to modify the behavior of a model only by parameters, which is defined in the model with its consent. There is no way to modify its hard-coded parts.

A CONCRETE EXAMPLE

As a concrete example to show the expressiveness of SVM, a model representing a queue with a capacity of 9 is given here. When the model is executed, it accepts 13 events at any time. Event `arrive` means an object arrives. The queue length is increased by 1. Event `depart` means an object departs and the queue length decreases. Events `0` to `9` change the queue length accordingly (useful for initializing or resetting the queue). Event `get` retrieves the current queue length.

If it is modelled in a straightforward manner, at least 10 states must be explicitly created and also $13 \times 10 (= 130)$ transitions. Of course these similar states and transitions can be created with a script running in the modelling environment, but this means the statechart meta-model must be extended to accept a scripting language, and this makes models less portable.

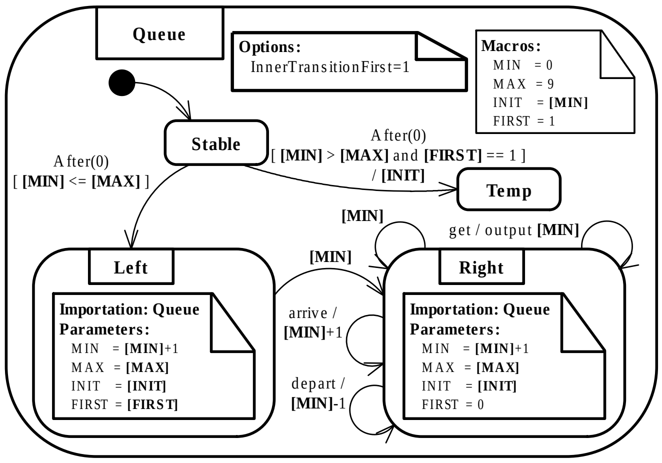

Figure 3: A statechart example: a queue with a capacity of 9

With parameterized importation, only 4 states and 7 transitions must be *explicitly* created, no matter it is a queue with a capacity of 9, or a character cell representing `a` to `z`, or even an integer cell representing 1 to 1000. The idea is to structure all the possibilities in a bi-tree. Valid states in a model execution include `LEFT.LEFT.RIGHT...STABLE` and `RIGHT.LEFT.LEFT.RIGHT.LEFT.RIGHT...STABLE`. (There are 11 levels in total, with the last one named `STABLE`.) The rightmost `RIGHT` represents the current queue length. Suppose the names in "`...`" are all `LEFT`, then the first snapshot represents a queue length of 2, and the second represents 5.

[1]This is to say, it is not allowed to control the execution sequence with `if ... then ... else ...` clauses and `for`, `while` loops and so on. This does not give a limit to the expressiveness, because the statechart structure itself is always able to model these structures.

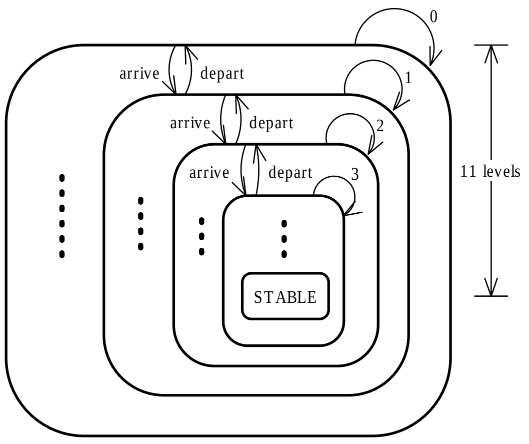

Figure 4: 27 nested levels are created by recursion (for simplicity, LEFT states and RIGHT states are not distinguished)

The design is shown in Figure 3. Macros are shown in bold font so that they can be easily distinguished from the guards.

The parameter [MIN] is the queue length for a certain level. The [MIN] value of the next level is [MIN]+1, and then [MIN]+2, until [MAX]. [INIT] is a constant for a model execution. It is the initial queue length, which is 0 by default.

When initiated, the model must nest deep enough so that the transitions in the first level are duplicated (with only the event names changed) 10 times (Figure 4). When the innermost STABLE state is reached, events in all these states are accepted. For the get event to return the queue length from the deepest RIGHT state, the transitions in this model must be *inner-first* ordered.

As illustrated in Figure 3, both the LEFT and RIGHT states import the Queue model itself, whose default state is STABLE. However, if the level is not deep enough (namely, [MIN]<=[MAX]), the imported model changes to its LEFT state immediately, which in its turn imports Queue and nests down to the next lower level. So when the model is stable, the depth of this nesting always equals to [MAX]+1.

Having nested deep enough ([MIN]>[MAX]) and the model is being initialized ([FIRST]=1), the state changes to TEMP — a dummy state, and at the same time an event [INIT] is raised. Then the queue immediately changes to the initial length. Whenever the first RIGHT state is entered, the model is no longer being initialized and is able to accept events from the user (possibly input from the SVM graphical interface). The [FIRST] is set to 0 from then on.

When event 0 to 9 (the [MIN] value) is raised, the state in the appropriate level changes to RIGHT. If it is already in RIGHT, a self-transition back to this RIGHT state is fired. This transition, though seems to be superfluous, is actually important. It is to eliminate the RIGHT states in all the lower levels, so it becomes the deepest RIGHT state.

When the arrive event is raised, the transition from the deepest RIGHT state is triggered. This transition does not change the state but raises another event [MIN]+1. The lower state (if any) receives this event and changes the state of the model. It is similar for the depart event.

```
MACRO:
  MIN   = 0
  MAX   = 9
  INIT  = [MIN]
  FIRST = 1
IMPORTATION:
  myself = Queue.des
OPTIONS:
  InnerTransitionFirst = 1
STATECHART:
  STABLE [DS]
  TEMP
  LEFT   [myself] [MIN = [EVAL([MIN]+1)]] [INIT = [INIT]]
         [FIRST = [FIRST]] [MAX = [MAX]]
  RIGHT  [myself] [MIN = [EVAL([MIN]+1)]] [INIT = [INIT]]
         [FIRST = 0] [MAX = [MAX]]
TRANSITION:
  S:STABLE
  T:0
  C:[MIN] <= [MAX]
  N:LEFT
TRANSITION:
  S:STABLE
  T:0
  C:[MIN] > [MAX] and [FIRST]==1
  N:TEMP
  O:[EVENT('[INIT]')]
TRANSITION:
  S:LEFT
  E:[MIN]
  N:RIGHT
  O:[DUMP('Current length is [MIN].')]
TRANSITION:
  S:RIGHT
  E:[MIN]
  N:RIGHT
  O:[DUMP('Current length is [MIN].')]
TRANSITION:
  S:RIGHT
  E:get
  N:RIGHT
  O:[EVENT('[MIN]')]
TRANSITION:
  S:RIGHT
  E:arrive
  N:RIGHT
  O:[EVENT(str([MIN]+1))]
TRANSITION:
  S:RIGHT
  E:depart
  N:RIGHT
  O:[EVENT(str([MIN]-1))]
```

Table 1: Source text for the Queue model (Queue.des)

Table 1 shows the source text in the model description file. Predefined macro EVAL is to evaluate a string. DUMP is to dump a message to the output. EVENT is to raise an event. Model-specific macros (which also serve as parameters) are defined in the MACRO part. Once defined, they can be cited anywhere in brackets. In a transition definition, S stands for source state, E stands for event, N stands for new state, O stands for output, C stands for condition, and T stands for timed transition (the transition to be triggered after a certain time from when the state is entered).[2]

[2] E and T are conflicting for a single transition defi nition.

If the model is run in the SVM environment with the parameter `[INIT]` explicitly given, the initial queue length is changed accordingly.

ASSESSMENT AND DISCUSSION

This example illustrates a way to divide complex problems into relatively simpler ones by reusing a model template. In this manner, a nice design is more easily achieved, and the work is less error-prone.

An assessment is made based on the requirements in section *Model Execution*.

- The textual format of a statechart model is easy to process by an interpreter. The semantics is a superset of the statechart formalism. It is also very readable, because different parts are clearly separated. The format is also intrinsically modular, since each model is written in a separate but reusable file.
- SVM is implemented in the Python language. It is system-independent.
- Extensions are made to the statechart formalism.
  Importation and parametrization are supported in SVM to facilitate model reuse. A good model maximizes its reusability by taking in parameters. In the previous example, by setting the `[MIN]` and `[MAX]` parameters from the command line, the model can be changed to have a different lower bound and a different capacity, without the necessity of modifying the model design.
  The transition priority is tunable, so engineering models (usually use the outer-first scheme) and software models (usually use the inner-first scheme) are compatible. They can even be written in a single statechart file.
  These extensions provide designers with great flexibility, and at the same time modularity and substitutability are guaranteed.
- Because SVM is a statechart interpreter on top of the Python interpreter, the performance is relatively low. An alternative is to build the system on a compilation language such as C++. However, flexibility is lost, which is important for a *simulation* environment.
- Since SVM is a virtual machine capable of immediately interpreting a model, compilation is not necessary. Model analysis will be the topic of future work.

Other than those discussed above, more extensions are introduced in SVM including the *initializer* and *finalizer* of models, since a generalized statechart simulator is incapable of performing model-specific initialization or finalization. The user interface of the execution environment can be specifically provided in the model. *Timed transitions* are automatically fired at a certain time after a state is entered; *repeated timed transitions* are used to simulate pulling an object (repeatedly testing a boolean expression or invoking a method, until a condition becomes *true*).

More statechart examples and the python source for SVM are available at `http://msdl.cs.mcgill.ca/people/tfeng/?research=svm` [8].

CONCLUSION

Extensions to the statechart formalism are presented in this article, including submodel importation, tunable transition priority to solve conflicts and parameterized model templates. These extensions add expressive power to statechart. Tunable transition priority allows designers to explicitly specify which transition is actually fired in case of a conflict; importation and parametrization together allow designers to build up more complex models from simpler ones or use the recursion idea to simplify problems. These features guarantee modularity and substitutability. A concrete example is given to illustrate their usefulness.

ACKNOWLEDGEMENT

Special thanks to Prof. Hans Vangheluwe for his excellent supervision and advice!

REFERENCES

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

[2] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[3] Michael von der Beeck. A structured operational semantics for uml-statecharts. *Software and Systems Modeling*, 1(2), 2002.

[4] Alcatel, I-Logix, Kennedy-Carter, Inc. Kabira Technologies, Inc. Project Technology, Rational Software Corporation, and Telelogic AB. *Action Semantics for the UML*. Document ad/2001-03-01. OMG, 2000.

[5] Gerson Sunyé, François Pennaneac'h, Wai-Ming Ho, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML action semantics for executable modeling and beyond. In *Advanced Information Systems Engineering. 13th International Conference, CAiSE 2001, Interlaken, Switzerland, June 4-8, 2001, Proceedings*, volume 2068 of *LNCS*, pages 433–447. Springer, 2001.

[6] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML action semantics for model execution and transformation. *Information Systems*, 27:445–457, 2002.

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

[8] Thomas Feng. Statechart virtual machine, 2003. MSDL, McGill.