Unifying the Semantics of UML 2 State, Activity and Interaction Diagrams

Jens Kohlmeyer and Walter Guttmann

Universität Ulm, 89069 Ulm, Germany
jens.kohlmeyer@uni-ulm.de · walter.guttmann@uni-ulm.de

Abstract. We define a formal semantics of the combined use of UML 2 state machines, activities and interactions using Abstract State Machines. The behaviour of software models can henceforth be specified by composing these diagrams, choosing the most adequate formalism at each level of abstraction. We present several reasonable ways to link different kinds of diagrams and illustrate them by examples. We also give a formal semantics of communication between these diagrams. The resulting rules reveal unclear parts of the UML specification and serve as a basis for tool support.

1 Introduction

Ideally, software development proceeds continuously from requirements through specification to implementation, using an integrated formalism, method and tool set. The state-of-the-art proposal aiming at such an integrated approach is to use Model Driven Development as the method and the Unified Modelling Language (UML) [17] as the formalism.

In this paper, we focus on behaviour aspects of software systems. They are represented by several UML language units describing state machines, activities and interactions, as detailed in Section 2. Each type of diagram is useful by itself, offering different facilities to exhibit different properties of a system: State machines emphasise the changes made to a system's state due to the occurrence of events, activities emphasise control and data flow, and interactions emphasise the sequence of messages between the lifelines of objects.

While the UML can be profitably used to describe requirements and specifications, one of its shortcomings is the lack of a precise semantics. This has been addressed in recent years by research formalising the semantics of state diagrams [1, 2, 10, 5], activity diagrams [19, 20, 8, 16] and sequence diagrams [21, 6, 15]. The frameworks used include Abstract State Machines, graph transformations and basic formalisms such as relations and traces. A semantics of class diagrams is usually implicit, since behaviour applies to objects in the UML. For a detailed discussion of these and further approaches, see [14].

Hence the current state is that formal semantics have been separately defined for the various kinds of diagrams specifying behaviour. What is notably missing, however, is a semantics describing their combined use. This may be due to the

fact that the UML specification itself is not very elaborate about this issue, but already the name *Unified* Modelling Language aims at such an integrated use. It is time to give the UML a *unified* semantics.

We use Abstract State Machines (ASM) [3] to define the semantics. ASMs have been used to formalise the semantics of several programming and specification languages, including the UML diagrams for behaviour we are interested in. We can therefore base our unifying work on the existing specifications. At the same time, the resulting ASM rules are precise, comprehensible and executable.

We investigate all UML behaviour diagrams and identify various means to compose them. We formalise the semantics of composition by adding new ASM rules and by modifying appropriate parts of the established ASM specifications. The new rules coordinate the instantiation of the different UML diagrams, the computation of their respective context, and their interplay by communication. We thus achieve an integrated semantics of UML behaviour.

ASMs and the UML language units are briefly introduced in Section 2. In Section 3 we discuss a number of ways to combine different kinds of diagrams and define the corresponding semantics using ASM rules. The communication aspect is elaborated in Section 4.

Our formalisation is high-level enough to reveal problematic issues concerning the UML specification, as we discuss throughout the paper. Besides the exposure of semantical problems, the benefits of a precise semantics are numerous. Among other things, it reduces the space for interpretation and thus clarifies the meaning of models. It also serves as a (necessary) foundation for the implementation of tools supporting model execution, code generation and automated reasoning. For the first time, these advantages are obtained for models combining different kinds of behaviour specifications as intended by the UML.

The present paper extends [13] by the following new contributions. We systematically derive and discuss each of the different ways to compose behaviour, present the corresponding ASM rules in detail, and give examples. We moreover describe the semantics of communication.

2 Basics

In this section we describe the UML language units dealing with the behaviour of software systems, and Abstract State Machines [3] as far as required to understand the rules in this paper. By writing UML we mean UML 2.1.2 as specified in [17] unless stated otherwise.

Higher-level behavioural formalisms of the UML are based on its language unit Common Behaviors, and extended in the units Activities, State Machines and Interactions. Additional units related to behaviour are Actions and Use Cases.

The *Common Behaviors* language unit comprises three subpackages. The BasicBehavior subpackage describes behaviour as the change of an associated context object. In all other respects, behaviour is left abstract, with activities, state machines and interactions as concrete instances. This abstraction is the

key to compose behaviour specified by different kinds of diagrams, as detailed in Section 3.1. The Communications subpackage provides the core structure for signal handling and operation calls. It is the basis for our treatment of communication in Section 4. The SimpleTime subpackage is relevant for the semantics of the individual behaviour diagrams, but has no impact on the present paper.

Activity diagrams coordinate lower-level behaviours by specifying their dependences and the allowed execution sequences. They are composed of basic actions connected by edges to indicate (possibly concurrent) control and object flow. Parameterised actions send and receive signals, and invoke behaviour specified elsewhere, for example, in other diagrams. Edges connecting actions may pass through control nodes (decision, merge, fork, join) that coordinate the flows in an activity diagram. Interruptible activity regions support the termination of parts of an activity diagram. Our unifying work uses the ASM semantics of UML 2.0 activities in [19] that sequences actions based on a token flow [20].

State diagrams model discrete behaviour by specifying the states of a system and the transitions between the states. Transitions are triggered by events, resulting in the change of state and the execution of associated behaviour. States may be composed of (orthogonal) subregions and (hierarchical) submachines. Our unifying work uses the ASM semantics of UML 1.4 state machines in [2] extended to UML 2.0 by [9].

We treat the 'most common variant' [17] of *interactions*, namely sequence diagrams. They show how several objects communicate by means of messages. An object's lifeline orders the occurrences of events that include sending and receiving messages, creating and destroying objects, and executing behaviour. In contrast to state machines and activities, which describe behaviour performed by an object, interactions describe *emergent behaviour* resulting from the participant objects. Our unifying work uses the ASM semantics of UML 2.0 sequence diagrams developed in [11].

UML *use case diagrams* capture the high-level requirements of a system, but do not specify behaviour on their own. They are instead linked to behaviour specified by the above language units. UML *actions* specify low-level transformations on the state of the system, and are modelled independently of the behaviours (primarily activities) containing them. Hence an action is likewise not a behaviour on its own. Because of these reasons, it is unnecessary to specially consider use cases and actions for the purposes of this paper.

We have adapted the existing ASM semantics to the same UML version 2.1.2. The modifications necessary to integrate the various diagrams are described in Sections 3.2–3.5 and 4.

The ASMs, used to formalise the semantics of the UML language units, can be read as 'pseudo-code over abstract data' [3]. An ASM comprises transition rules operating on a state composed of functions defined over a base set. The update rule $f(s_1, \ldots, s_n) := t$ modifies the value of f at $(s_1, \ldots, s_n)$ to t. Further constructs include abstractions using **let** ... **in**, multiway conditionals, and rule calls with call-by-name semantics. The rule **forall** x **with** φ **do** R executes R in parallel for each x satisfying φ. The rule **choose** x **with** φ **do** R chooses

some x satisfying φ and then executes R. Updates accumulated by these rules are performed in parallel unless sequentialised by **seq**.

In our work we use asynchronous multi-agent ASMs, allowing the concurrent execution of several ASM agents. Each performs its own rules as described above, and they communicate by shared functions. Further details of ASMs, including an operational semantics, are provided by [3].

3 Formal Semantics for Combining Language Units

Before we detail the ways to combine behaviour specifications, we briefly describe our general approach to the semantics of UML. The ASM rules need to access the concrete diagrams of the model whose semantics they define. To this end, we translate the UML syntax, also called the 'meta model', to static ASM domains (for classes) and functions (for attributes and associations). They are initialised to yield the particular values corresponding to the concrete model. Monitored ASM functions are used for information determined by the environment. Individual executions of behaviour are represented by asynchronous multi-agent ASMs. To model their interaction and signal handling, we use shared ASM domains and functions. The semantics is then specified by ASM rules acting on these functions such as those we describe in the following. Further details of this approach are provided in [19], and the complete set of rules in [14].

3.1 Combining Behaviour Specifications

In this section we systematically derive several different ways to combine activities, interactions and state machines. To this end, we investigate the UML meta model, looking for the places where the abstract class Behavior is used to specify behaviour. This way we identify the possible means to combine behaviour, which is then realised by specialising the abstract Behavior to the concrete classes Activity, Interaction or StateMachine.

We find that the occurrences of Behavior fall into two categories. First, behaviour is used for elementary data processing: Data is passed to the behaviour, which is expected to have no side effects; it produces a result that is further processed. Examples are the selection and transformation of tokens at object nodes and flows and decision nodes in activities, and the reduction of a collection by ReduceAction. Typically, this processing is low-level and will be described by code rather than another diagram. The code is easily integrated by specifying the computation directly as an ASM rule or in a language with ASM semantics such as Java or C♯.

Second, behaviour is invoked that can make full use of UML's specification facilities. In this case, we can distinguish the calling and the called behaviour. Typically, the called behaviour is specified by a diagram as discussed in the following. If it is low-level, it may also be given by code and integrated as above.

In principle, each kind of diagram can be used independently to specify the caller and the callee. In practice, it is problematic to call interactions, since they

specify emergent behaviour [17, page 482]. This kind of behaviour results from the interaction of all participant objects and is not performed by a particular object [17, page 419]. Hence in general it is not meaningful to let a given object call emergent behaviour. We will investigate this issue in further work, but do not allow interactions to be called in the present paper. However, interactions may appear as the caller, for example, to specify test scenarios as in Section 3.5. For the callee, we are thus left with activities and state machines, and we now argue that these two diagrams can indeed be called from activities, interactions and state machines.

Let us exemplify this argument for state machines as the caller by considering the relevant part of the UML meta model [17, page 525]. It clearly shows that a behaviour may be associated to a state as its entry, exit and do-activity, as well as the effect of a transition. The intended meaning of these behaviours is described by the UML specification in text form. Each of these behaviours can independently be specified as an activity or as a state machine, since they specialise the abstract class Behavior.

We carry out the same procedure for activities and interactions to identify their possible combinations. For activities, we obtain that behaviour is attached to CallBehaviorAction [17, page 245]. This kind of action may be used in activity diagrams, for example, for hierarchical decomposition. For interactions, we obtain that behaviour can be associated to BehaviorExecutionSpecification [17, page 467]. This is a specialisation of InteractionFragment, whose instances are the (partially ordered) constituents of interactions.

We have thus identified the ways to compose UML behaviour diagrams. In the following section, we formalise the calling mechanism of diagrams. It is applied to formally define the semantics of calling behaviour from activities, state machines and interactions, respectively, in Sections 3.3–3.5.

3.2 Calling Mechanism

In the existing ASM semantics of UML behaviour diagrams, there is just one place describing the invocation of behaviour specified by a diagram: The calling of an activity from an activity by a CallBehaviorAction [19]. We abstract the calling mechanism from that description and generalise it to other kinds of diagrams for the caller and the callee. The resulting ASM rule takes as arguments the called behaviour, its context object, its parameters and a flag indicating whether the call is synchronous or asynchronous. Observe that these arguments resemble the ingredients of calling mechanisms in programming languages.

```
StartBehaviour(behaviour, context, input, isSynch) ≡
  case behaviour in
    Activity:     if behaviour.isSingleExecution ∧ isRunning(behaviour)
                    then ReCall(behaviour, input)
                    else Call(Activity(behaviour, context, input))
    StateMachine: Call(StateMachine(behaviour, context, input))
    Interaction:  Call(Interaction(behaviour, input))
  if isSynch then Self.mode := waiting else Self.mode := completed
```

For activities, we distinguish if an existing execution is used or a new one created according to the isSingleExecution attribute [17, page 317]. In the former case, treated by the rule ReCall, we notify the already running execution that new input tokens are available using the internal *StartEvent*. In the remaining cases of StartBehaviour, the rule Call creates a new agent executing the corresponding handler from the ASM semantics for the individual diagrams.

ReCall(*activity*, *input*) ≡
  let *exec* = *agent*(*activity*) **in**
    exec.callers := *exec.callers* ∪ {*Self*}
    AddEvent(*exec*, *StartEvent*(*input*))
    Self.*calledExec* := *exec*

Call(*rule*) ≡
  let *exec* = *new*(*Agent*) **in**
    exec.callers := {*Self*}
    ASM(*exec*) := *rule*
    Self.*calledExec* := *exec*

The called behaviour is informed about its callers. This is necessary so that they can be notified about termination even in the case of asynchronous calls (this mechanism is used, for example, to implement the so-called completion transitions in state machines). If the call is synchronous, we put the calling agent into a mode waiting for the called execution to terminate.

Let us finally point to the absence of the context object for interactions in the rule StartBehaviour. This reflects the fact that they specify emergent behaviour of several participating objects, as discussed in Section 3.1. Although interactions cannot be called from other diagrams, they are included in StartBehaviour to set up the behaviour of the initial objects when the modelled system is started. To this end, we assume that interactions are instantiated as objects and thus run in their own context [17, page 430].

InitialiseBehaviour ≡
  forall *object* **with** *object* ∈ *BehaviouredObject* **do**
    StartBehaviour(*object.classOf.classifierBehaviour*, *object*, ∅, *false*)

In the initial case, there are no parameters and the calls are asynchronous since all initial objects act concurrently.

We have thus established the mechanism to call diagrams, namely the rule StartBehaviour. It allows for a simple integration of further kind of behaviour having an ASM semantics. The following sections implement the calling of behaviour by this mechanism. To abort a called behaviour, we adapt the termination mechanisms described in [19, 9].

3.3 Calling Behaviours from Activities

As our running example, we use a simplified model of an MP3 music player whose behaviour is specified in Figure 1.

The general behaviour is modelled by the activity diagram on the left: Its first action shows a welcome message on the display, then the user can simultaneously edit the play list with the action EditList and listen to music with the action PlayMusic. These actions are instances of CallBehaviorAction, detailed in further UML diagrams. The behaviour EditList is modelled by an (omitted) activity diagram. The calling of activities from activities is described in [19].

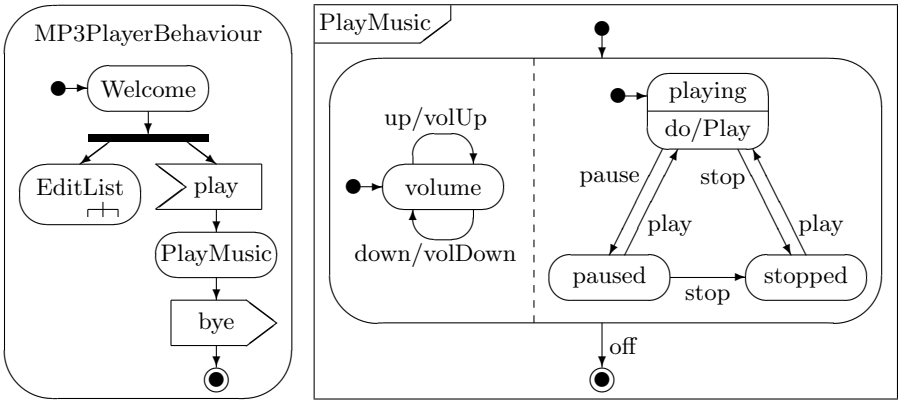

Fig. 1. Simplified MP3 player

In this paper we focus on the right path of the activity diagram which specifies the playing of music. If the play signal is received – generated by the environment, for example, by the user pressing a button – a CallBehaviorAction is executed which activates the behaviour PlayMusic. In our model, this is adequately specified as a state machine on the right of Figure 1. It contains two regions which model the behaviour of volume control and playing music.

The ASM rule ExecuteCallBehaviourAction handles the calls of other behaviours. For space reasons we only show the parts of the rule which are relevant for calling another behaviour. The detailed semantics of CallBehaviorAction can be found in [19].

```
ExecuteCallBehaviourAction ≡ ...
  if Self.mode = enabled then
    let cTag = tagValue(Self.node, CallContext, context) in
      if cTag = undefined
        then context := Self.activityExecution.context
        else context := computeContext(Self.activityExecution.context, cTag)
    seq
    StartBehaviour(Self.node.behaviour, context,
      {(pinToParameter(n), ts) | (n, ts) ∈ Self.input}, Self.node.isSynchronous)
  if Self.mode = waiting then ...
```

At first the context object of the called behaviour is computed. This is left unclear by the UML specification [17, page 429], since the called behaviour is not owned by the CallBehaviorAction. We therefore add a context tag as in [19] from which the context is computed. If the tag is not defined, the context of the caller is used. The function *pinToParameter* maps the inputs, which are provided by the input pins of the calling action, to the parameters of the called behaviour. With these arguments we start the associated behaviour. The waiting mode of the rule waits for the termination of synchronous calls.

In the MP3 player the context of the called state machine is the activity's context. The state machine can thus access attributes of the context object. No parameters are passed and the call is synchronous by default, hence the calling activity waits until the state machine terminates.

The behaviour within the playing state of the MP3 player (do-activity) is specified by the activity diagram Play, not detailed here. Calling activities from state machines is described next.

3.4 Calling Behaviours from State Machines

The state machine on the right of Figure 1 is interpreted by an ASM agent, called 'top agent'. As soon as the composite state is reached, another two agents are started to interpret the orthogonal regions. As described in Section 3.1, calling behaviours from state machines can occur during (internal or external) transitions or when entering, exiting or being in a state. Each agent holds the currently active states and executes the rules required to perform transitions and to call behaviours in its region. Transitions must be performed atomically, including any behaviour invoked while they take place. We can therefore no longer use the rules of [2] since these apply to UML 1.4, which allowed only actions, not behaviours to be executed. To ensure atomic transitions, we introduce appropriate modes for the ASM agents.

First, we select one of the available events and a corresponding, enabled transition. This is done by the selecting and preparing modes of the rule PERFORMTRANSITION, which we discuss in Section 4. Second, we carry out the selected transition. If it is internal, only its effect has to be executed. In general, the transition crosses several state boundaries, hence a number of steps must be taken to execute the exit, effect and entry behaviours in the given order. Moreover, the running do-activities of the left nodes must be aborted, and new do-activities initiated after each state is entered. The correct sequence of these tasks is delivered iteratively by the rule NEXTTASK, updating the current state and task.

```
PERFORMTRANSITION ≡ ...
  if Self.mode = running then
    case currentTask of
      exit:   EXIT(currentState)          do:     DO(currentState)
      effect: EFFECT(currentTrans)        finish: FINISH(currentTrans)
      entry:  ENTRY(currentState)                 Self.mode := selecting
    NEXTTASK
```

Besides doing the necessary book-keeping, the rules EXIT, EFFECT, ENTRY and DO call the annotated behaviour. We first discuss EFFECT; the rules EXIT and ENTRY are similar. In our example it is invoked when the events up or down trigger a transition in the volume state, and hence the behaviour volUp or volDown. This is performed by a synchronous call since transitions are atomic. In contrast to activities, the context object is clearly specified by the UML as the context of the calling state machine. The parameters are obtained by the monitored function *getEffectParam*, since this is left open by the UML [17, page 574].

$$
\begin{array}{l}
\textsc{Effect}(trans) \equiv \\
\quad \textbf{let } context = trans.container.stateMachine.context \textbf{ in} \\
\quad\quad \textbf{let } param = getEffectParam(trans, context) \textbf{ in} \\
\quad\quad\quad \textsc{StartBehaviour}(trans.effect, context, param, true)
\end{array}
$$

In our example internal behaviour is started when the playing state is entered, either initially or as a result of a transition caused by the play event. This is performed by the rule Do analogously to Effect, except that the do-activity (Play) is called asynchronously since it must be performed concurrently; for example, a transition might leave the state while the do-activity is running.

3.5 Calling Behaviours from Interactions

To illustrate the calling of behaviour from interactions, we use them to specify test cases. Figure 2 shows a possible scenario for the MP3 player example. The sequence diagram specifies messages and their sequence between two lifelines, a user and the MP3 player. The events used in the behaviour of the MP3 player are created by the environment, in our case by the user. If the user turns on the player, the activity MP3PlayerBehaviour (see Figure 1) is executed. If the play message is received, the state machine PlayMusic is instantiated. The user then operates the player (for example, higher volume, pause the player) and the player reacts by sending messages to the user (for example, if the play list is finished). These reactions are defined in the do-activity of the playing state. After the user turns off the player, the state machine is terminated immediately and the activity sends the message bye to the user before it is stopped.

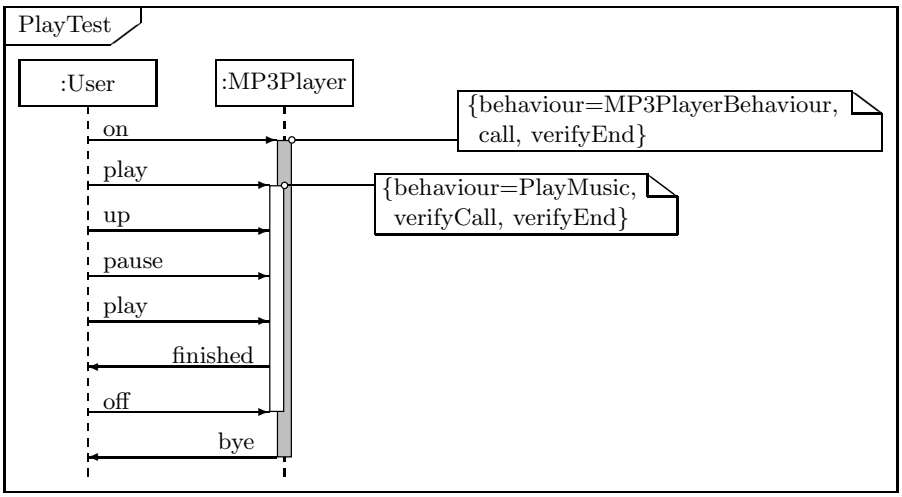

Fig. 2. Interaction defining test cases

A sequence diagram is interpreted by one ASM agent, processing the constituent interaction fragments. For the purposes of test case specification it is sufficient to consider one possible sequence of fragments matching the order imposed by the diagram (conveniently generated by ASM's **choose** rule). The fragments (which may be composed of other fragments) are processed iteratively. For each fragment, a rule defining the semantics according to its kind is executed. We only present the parts of the rule relevant for the combination of behaviours.

In the UML meta model the invocation of behaviour from a lifeline (see Figure 2) is represented as a BehaviorExecutionSpecification associated to two ExecutionOccurrenceSpecifications, namely its start and its finish. Note that both the execution and its two occurrence specifications are interaction fragments according to the UML. We base our semantics on the occurrences, since they offer the finer view. If the current fragment is such an execution occurrence, the following ASM rule is executed.

```
CaseBehaviourExecutionEvent(fragment) ≡
  let context = fragment.covered in
    let param = getBehaviourParam(fragment) in
      case behaviourKind(fragment) of
        call:       StartBehaviour(fragment.behaviour, context, param, false)
        end:        FinishBehaviour(fragment.behaviour, context)
        verifyCall: VerifyStartBehaviour(fragment.behaviour, context, param)
        verifyEnd:  VerifyFinishBehaviour(fragment.behaviour, context)
```

It distinguishes whether the associated behaviour starts or finishes. For test scenarios, we further distinguish who controls the start or finish of the behaviour. This may be the environment; then the behaviour must actually be started or aborted. Alternatively, the behaviour may be started by the specified diagrams under test; we then verify if the appropriate action takes place. The decision is taken according to annotations in the sequence diagram, see Figure 2. We propose that the context object of the behaviour is its lifeline, since this is left unclear by the UML.

4 Communication in UML

In [19] a semantics for the event-based communication between activities is stated. In this section, we describe the necessary modifications due the combination of different kinds of diagrams. The existing semantics uses an event pool for each behaviour execution to store event occurrences (for example, a signal causes an event at its target behaviour). While this suffices for activities, it does not comply with the UML, which requires the context objects (that may have more than one associated behaviour executions) to recognise event occurrences [17, pages 433 and 563].

Thus, we argue that the event pool is located at the context object of a behaviour, not at the behaviour itself. We specify rules to handle the procedure of sending signals via SendSignalAction or BroadcastSignalAction in compliance

with the specification. The rules create a request object (several for broadcasts), capturing among other things the sender and the target object. For specifying the target of a signal we use the signal path approach proposed in [18], adapted to our semantics.

Further changes are necessary for the semantics of the particular language units. We exemplify this for the top agent of state machines, which has to perform transitions upon receiving a signal.

```
PerformTransition ≡ ...
  if Self.mode = selecting then
    choose e with e ∈ Self.context.eventPool do
      dispatched := e ...
  if Self.mode = preparing then
    choose trans with trans ∈ fireableTransWithMaxPriority(dispatched) do
      FirstTask(trans)
      Self.mode := running
```

The rule chooses an event from the context object's event pool, the so-called dispatched event. It then chooses a transition fireable with the dispatched event and switches into running mode, see Section 3.4. The agents for the orthogonal regions execute a similar rule, performing transitions enabled by the dispatched event after synchronising with the top agent.

5 Conclusion

As discussed in the introduction, there are a number of works about the semantics of individual types of diagrams. Two further approaches aim at an integrated framework. Subsystems interacting by message passing are described in [12], but this applies to the old UML 1.4 only. Works initiated by the UML Semantics Project [4, 5, 8] are based on the 'system model' defined in terms of mathematics. However, the combination of different kinds of diagrams is not discussed; for example, activities can only call activities [7].

To overcome these limitations, we have presented a formal semantics of the combined use of activities, state machines and interactions, based on the same UML version 2.1.2. The resulting rules adhere to requirements present in or absent from the UML specification. They also allow the integration of code to implement low-level behaviour.

Based on our semantics, we are currently extending our tool ActiveCharts (http://activecharts.de/), that directly executes UML activities, to simulate models specified by state machines and interactions. The tool can be used to find errors in the modelled system and to obtain a better understanding of it. Examples we have modelled include a lift and a traffic light control.

Acknowledgement. We thank Guido de Melo for his model of the MP3 player.

References

1. M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, December 2002.

2. E. Börger, A. Cavarra, and E. Riccobene. On formalizing UML state machines using ASMs. *Information and Software Technology*, 46(5):287–292, April 2004.
3. E. Börger and R. Stärk. *Abstract State Machines*. Springer, 2003.
4. M. Broy, M. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. 2nd UML 2 semantics symposium: Formal semantics for UML. In T. Kühne, editor, *Models in Software Engineering*, volume 4364 of *LNCS*, pages 318–323. Springer, 2007.
5. M. V. Cengarle, H. Grönninger, and B. Rumpe. System model semantics of statecharts. Informatik-Bericht 2008-04, TU Braunschweig, July 2008.
6. M. V. Cengarle and A. Knapp. UML 2.0 interactions: Semantics and refinement. In J. Jürjens, E. B. Fernandez, R. France, and B. Rumpe, editors, *Critical Systems Development with UML*, pages 85–99. TU München, 2004.
7. M. L. Crane. *Slicing UML's Three-layer Architecture: A Semantic Foundation for Behavioural Specification*. PhD thesis, Queen's University, January 2009.
8. M. L. Crane and J. Dingel. Towards a UML virtual machine: Implementing an interpreter for UML 2 actions and activities. In M. Chechik, M. Vigder, and D. Stewart, editors, *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 96–110. ACM Press, 2008.
9. M. Dausend. Entwicklung einer ASM-Spezifikation der Semantik der Zustandsautomaten der UML 2.0. Diploma thesis, Universität Ulm, June 2007.
10. H. Fecher and J. Schönborn. UML 2.0 state machines: Complete formal semantics via core state machine. In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 244–260. Springer, 2007.
11. J. Fürst. Entwicklung einer ASM-Spezifikation für die Semantik von UML 2 Sequenzdiagrammen als Grundlage zur Anbindung an ActiveCharts. Diploma thesis, Universität Ulm, February 2008.
12. J. Jürjens. Formal semantics for interacting UML subsystems. In B. Jacobs and A. Rensink, editors, *Formal Methods for Open Object-Based Distributed Systems V*, pages 29–43. Kluwer Academic Publishers, 2002.
13. J. Kohlmeyer. Executing UML 2 diagrams in ActiveCharts: A formal semantics for the combination of behavior specifications in the UML 2. In C. Bertelle and A. Ayesh, editors, *ESM 2008*, pages 94–101, October 2008.
14. J. Kohlmeyer. *Eine formale Semantik für die Verknüpfung von Verhaltensbeschreibungen in der UML 2*. PhD thesis, Universität Ulm, July 2009.
15. X. Li, Z. Liu, and J. He. A formal semantics of UML sequence diagram. In *Australian Software Engineering Conference*, pages 168–177. IEEE, 2004.
16. S. Marković and T. Baar. Semantics of OCL specified with QVT. *Software and Systems Modeling*, 7(4):399–422, October 2008.
17. Object Management Group. *UML 2.1.2 Superstructure Specification*, November 2007.
18. S. Sarstedt. Overcoming the limitations of signal handling when simulating UML 2 activity charts. In J. M. Feliz-Teixeira and A. E. Carvalho Brito, editors, *ESM 2005*, pages 61–65, October 2005.
19. S. Sarstedt. *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams*. PhD thesis, Universität Ulm, July 2006.
20. S. Sarstedt and W. Guttmann. An ASM semantics of token flow in UML 2 activity diagrams. In I. Virbitskaite and A. Voronkov, editors, *Perspectives of System Informatics: 6th International Andrei Ershov Memorial Conference, PSI 2006*, volume 4378 of *LNCS*, pages 349–362. Springer, 2007.
21. H. Störrle. Semantics of interactions in UML 2.0. In *Symposium on Human Centric Computing Languages and Environments*, pages 129–136. IEEE, 2003.