Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities

Michelle L. Crane and Juergen Dingel

Queen's University, Kingston, Ontario, Canada

Abstract

An interpreter for UML 2 actions and activities is presented. It is based on two novel features in UML 2: the three-layer semantics architecture and the new token offer semantics for activities, which is intended to generalize the token flow semantics of Petri nets. The interpreter offers an array of analysis capabilities, ranging from random execution to reachability properties and assertion and deadlock checking. The design of the interpreter makes it suitable as the basis for a more comprehensive UML virtual machine.

1 Introduction

Recently, there has been much interest in the use of models in software development in the hopes of improving productivity by increasing the levels of abstraction, automation, and analysis. One of the main notational contexts in which model-driven software development has been studied is the Unified Modeling Language (UML), the *de facto* standard in software modeling. UML has been very successful: in terms of industrial acceptance, UML is by far the most successful software modeling language. For instance, activity modeling via activity diagrams "was embraced by business-process modelers and by systems engineers, who tend to view many of their systems as interconnecting signal processors" [26].

Despite this success, there also seems agreement among UML standard-setters that its oft-mentioned lack of an unambiguous, formal semantics is preventing UML from affording its users more automation and analysis gains. In an effort to improve the support for model-driven development in general and to facilitate the development of a formal semantics in particular, the UML 2 specification introduced a novel three-layer semantics architecture [19, 25] as shown in Figure 1. Each layer depends on those below it, but not vice versa. The bottom layer represents the structural foundations of UML, including concepts such as values, objects, links, messages, etc. On top of that layer lie UML actions. On top of actions lie UML behavioral specification diagrams, e.g., interactions, state machines, and activities. Therefore, activities make use of actions in order to specify behavior; these actions are expressed in terms of constructs in the structural foundation.

While this architecture helps clarify the runtime semantics of UML, a comprehensive formalization of the semantics is still an open research problem. For instance, the runtime semantics of a collection of UML models is typically defined as "the constraints that models place on the runtime behavior of the specified system" [18]. However, to the best of our

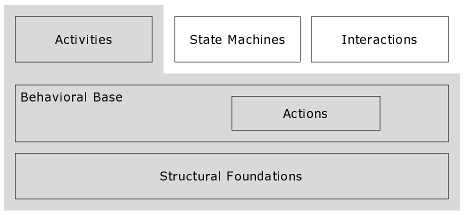


Figure 1: The UML three-layer semantic architecture. Shading indicates the parts covered in this paper

knowledge, no comprehensive list of these constraints has ever been compiled. One complicating factor is the large number of different kinds of diagram: How exactly do the constraints expressed in one diagram interact with constraints expressed in another? Another complicating factor is the large number of features that each diagram type typically supports. Activities, for instance, allow for different kinds of flow (e.g., data, control) using, e.g., multiplicities and weights, different kinds of behavior invocation (e.g., direct or indirect, synchronous or asynchronous), a variety of different kinds of control node (e.g., initial, final, fork, join, decision, merge), structure concepts (e.g., loop, conditional), and mechanisms for unstructured control flow (e.g., interruptible regions, exceptions). Additionally, the semantics of activities has changed from being viewed as a special kind of state machine to a rather complex token offer semantics, which generalizes Petri net semantics and contains delicate features such as deadlock avoidance rules and competition.

Many proposals for a semantics of UML already exist. However, most only deal with a small subset of the diagram types offered in UML, or provide unnecessarily restrictive answers to the question of which constraints a collection of UML models impose on the runtime behavior of the system. Recently, a formalization effort has been initiated that respects the three-layer architecture and specifically aims at producing a relatively comprehensive, yet minimally restrictive formalization due to its highly declarative style [4, 5, 6]. The approach, which we refer to as the *System Model*, is based on a formalization of the lowest layer of the three-layer architecture and uses compositions of timed state machines to capture the runtime constraints expressed in a model.

In this paper, we describe an interpreter for UML 2 actions and activities. The interpreter conforms to the UML 2 specification [19] in the sense that an activity's execution is consistent with its description of expected runtime behavior. The interpreter offers an array of analysis capabilities, ranging from random execution to reachability properties and assertion and deadlock checking. Finally, the interpreter is based on the System Model and shows how, for instance, an execution affects the structural foundation.

In 2005, the Object Management Group (OMG) issued a Request For Proposal for a foundational subset of the UML 2 metamodel that would provide a shared foundation for higher-level UML modeling concepts such as activities, state machines, and interactions and thus define "a basic virtual machine for the Unified Modeling Language...enabling compliant models to be transformed into various executable forms for verification, integration, and deployment" [18]. Using the System Model, we have created a partial implementation of such a virtual machine. While our implementation currently only supports actions and activities, the architecture and the underlying semantic domain facilitate the extension to other behavioral formalisms, such as state machines and interactions and thus the development of a comprehensive UML virtual machine.

The remainder of this paper is structured as follows: Section 2 discusses actions and activities and provides background on the System Model. Section 3 describes the process of using our interpreter, including a running example. Analysis techniques provided by the interpreter are discussed in Section 4. Section 5 outlines related work, while conclusions and future work are described in Section 6.

2 Background

2.1 Actions and Activities

A fundamental premise of UML behavioral semantics is the assumption that "all behavior in a modeled system is ultimately caused by actions" [19, §6.3.1]. Actions are "fundamental units of behavior in UML and are used to define fine-grained behaviors" [19, §6.3.2]. More precisely, every kind of behavior specification in UML is supposed to be expressible in terms of sequences of actions.[1] There are a total of 45 of these "fairly low-level actions" [21] defined in UML, 36 of which are concrete. These actions can perform a myriad of tasks, such as creation (e.g., CreateObjectAction), reading state information (e.g., ReadStructuralFeatureAction), updating state information (e.g., AddVariableValueAction), testing equality (e.g., TestIdentityAction), destruction (e.g., DestroyLinkAction), and invoking behavior (e.g., CallOperationAction).

Activities in UML are a behavioral formalism based on the flow of both data and control. As of UML 2.0, activities have been "redesigned to use a Petri-like semantics" [19, §12.3.4]. Specifically, the concept of token offering is a generalization of Petri net transition enablement [3]. However, complications due to the token offer semantics preclude treating activities simply as Petri nets. The complications include the fact that multiple control nodes may be adjacent to each other and therefore affect the 'locality' of token synchronization [24]. In addition, not only do tokens move concurrently and asynchronously in an activity (tokens in a Petri net move at the same time), but the decision of which token to move may involve multiple unconnected transitions and places [3].

Tokens move through an activity in accordance to a "traverse-to-completion" [2] semantics. In essence, a token only moves to a target if it can be used there immediately; the purpose of this strategy is to help avoid deadlock.

To illustrate token passing, consider the activity in Figure 2. Although a token can be offered from node #2 to #6 as soon as #2 has completed execution, that offer must actually be propagated through nodes #3, #4 and #5. Also, the token cannot actually be passed unless #6 has also received an offer from #10.

2.2 Notation Conventions

The activity in Figure 2 contains explicit object nodes (shown as rectangles), instead of the more widely-used pin notation. We make use of

[1]A hypothesis that, to the best of our knowledge, is still unproven.

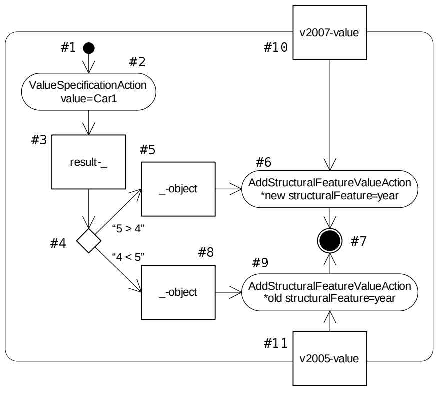


Figure 2: Sample activity diagram; this activity will change the value of *Car1.year*. The new value depends on the non-deterministic outcome of the decision node

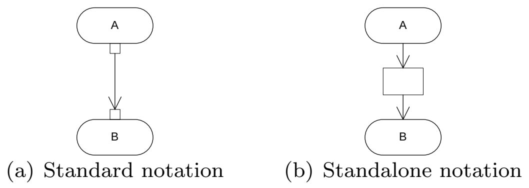


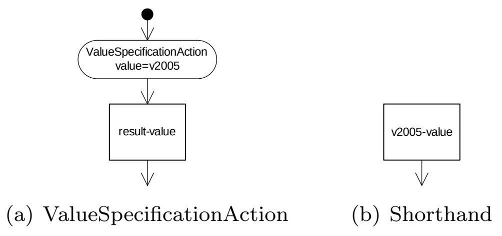
Figure 3: We use the standalone pin notation to explicitly define pins as object nodes between actions

the *standalone pin notation* [19, Figure 12.120], which is equivalent to the standard pin notation; see Figure 3.

In the standalone pin notation, object nodes in an activity represent pins, i.e., an object node represents both the output pin of one action and the input pin of another action. We do, however, make use of object nodes with no sources, as seen with the two 'value' object nodes in Figure 2. We use an object node without sources as shorthand for a ValueSpecificationAction,[2] where the object node refers to the result of that action; see Figure 4.

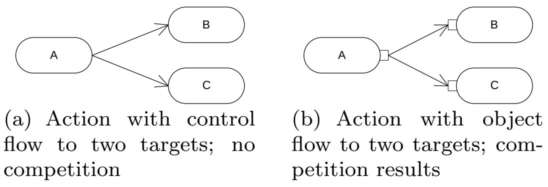
Pins are used to indicate object flow. Consider the activities in Figure 5. When there are no object nodes (pins), as in Figure 5(a), only control flow connects action A to its targets, B and C. When A finishes execution, it offers a control token to *both* B and C. These control tokens can be accepted by both targets, i.e.,

[2]This action is used to retrieve a value or object.



(a) ValueSpecificationAction (b) Shorthand

Figure 4: An object node with no source is shorthand for using a ValueSpecificationAction. The two examples shown are equivalent



(a) Action with control flow to two targets; no competition (b) Action with object flow to two targets; competition results

Figure 5: Two similar activities, with vastly different token passing

both `B` and `C` will execute. On the other hand, the activity in Figure 5(b) uses pins, so there is object flow between `A` and its targets. When `A` finishes execution, it offers a token (through its output pin) to both to targets `B` and `C`, through their input pins. In this case though, *only one* of the targets can accept the offer and thus receive the token (with its data). The other outstanding offer will be immediately revoked. In other words, when an object node has multiple targets, competition occurs.

2.3 System Model

Generally, the meaning of a UML specification is given by "the constraints that models place on the runtime behavior of the specified system" [18]. Described in a series of three documents [4, 5, 6], the goal of the System Model is to allow for these constraints to be captured and collected in their purest and most general form. Therefore, the System Model description intentionally relies solely on simple mathematical concepts such as sets, relations, and functions. In the System Model, the meaning of a model M is all implementations that satisfy the constraints expressed in M. This view facilitates the handling of collections of models. More precisely, the System Model contains a formalization of UML's structural foundation (i.e., the bottom layer of the architecture in Figure 1), which provides a formal notion of state. The constraints of a single behavioral diagram are captured by a (possibly timed) state machine. The meaning of a collection of behavioral UML diagrams is given by the composition of the state machines for each diagram.

The reader is encouraged to refer to the System Model documentation [4, 5, 6] to get a full description of all System Model concepts. In this paper, we only present a very brief overview. More precisely, we will only sketch the notion of *state* used in the System Model. First, a number of universes are postulated, such as the universe of type names, the universe of values, the universe of class names, the universe of object identifiers, the universe of states, etc. Then, types and values are organized into a type system as shown in Figure 6. *Values*, on the right, are elements of carrier sets of *TypeNames*, on the left. A third high-level concept, *VariableName* (not shown), represents attribute and variable names. References are "pointers" to values, whereas locations can hold values and correspond to the "slots" referred to in UML's structural foundation [19]. Type names can be composed into record types. A class name is a type name, defined as a reference to a record type. On the value side, an object identifier is defined as a reference to a record value.

A state is a triple composed of a data store, control store, and an event store. Roughly, the data store contains the information explicitly contained in the models provided by the user, such as existing objects and attribute values. The control and event stores, on the other hand, contain interpreter-level information, such as existing threads and program counters (control store) and the contents of each object's message buffer (event store).

The System Model has been used for the formalizations of interactions [7], state machines [8], and actions [10]. An implementation of the System Model in the Isabelle theorem prover is currently underway [16].

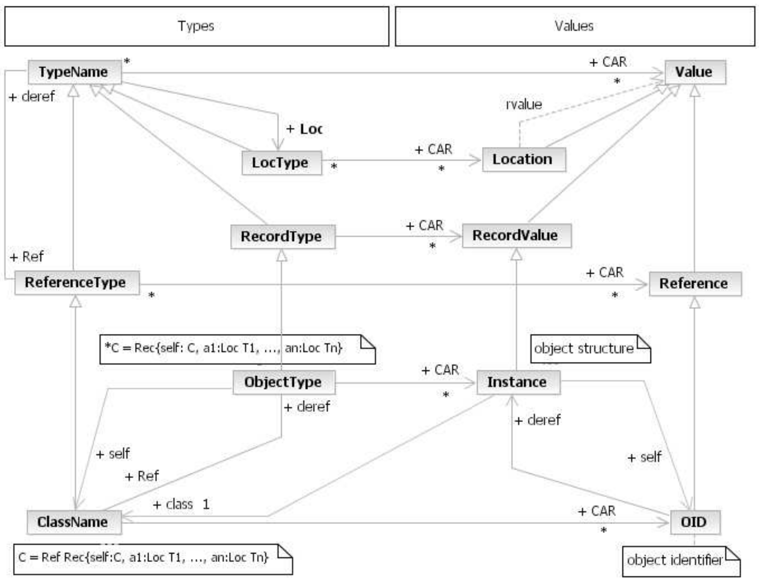


Figure 6: Class diagram representing System Model *TypeName* and *Value* concepts. Note the symmetry between the two sides

3 ACTi Interpreter

ACTi is our interpreter for UML actions and activities. It was designed to conform to the three-layer semantics hierarchy in Figure 1 and covers the shaded portion of that figure. ACTi was developed using Java and consists of approximately 16,000 lines of code, including the implementation of the System Model and actions. The structure of the implementation is shown in Figure 7. The System Model package implements the structural foundation of UML, the Actions package implements the individual UML actions, and the interpreter itself implements activities. As in the three-layer architecture, each layer depends on the layers below, but not vice versa.

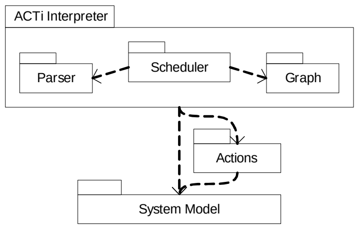


Figure 7: Structure of ACTi implementation conforms to semantics hierarchy in Figure 1

A high-level view of the process of using ACTi to execute an activity is shown in Figure 8. In essence, ACTi can take as input a text file representing an activity and provide as output one, or if possible, many paths[3] detailing the execution of an activity. As the activity executes, the underlying System Model state is modified. With the aid of a third-party tool, it is possible to visualize, and even navigate, the various states of the System Model as the activity is executed.

3.1 Example Execution

We use a running example to explore how ACTi can be used to execute and analyze an activity. Consider the activity in Figure 2. This activity takes a *Car* object, and changes the value of its *year* attribute. The new value of the attribute depends on the result of a decision. In this case, both outgoing edges of the decision have true guards, meaning that either outcome is equally possible.

The steps below relate to the labelled elements shown in Figure 8:

[3]Activities are concurrent by nature. The implementation, however, is restricted to sequential execution that simulates concurrency through interleaving.

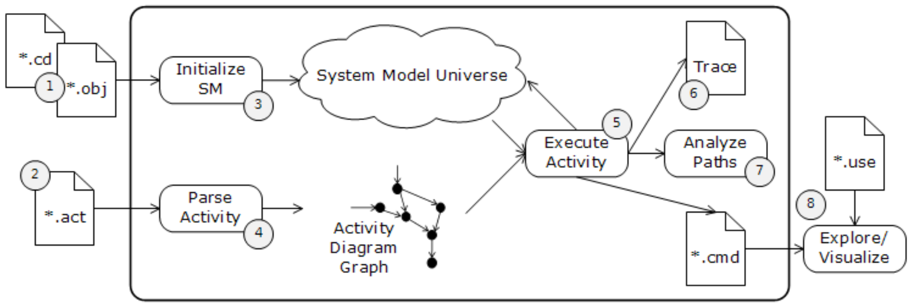


Figure 8: High-level view of how ACTi can be used to execute an activity. All steps inside the large rounded rectangle are automated. Eight elements (steps or artifacts) in the figure are labelled; these elements are discussed in detail in Section 3.1

```
type Int
  value v2000
  value v2005
  value v2007

class Car
  attr year:Int
```

Figure 9: *.cd file

```
obj Car1:Class_Car
  attr year=v2000
```

Figure 10: *.obj file

1. ***.cd and *.obj Files** The user is responsible for providing two text files that permit ACTi to initialize the System Model universe. The first, a *.cd file, details the static structure of objects that will be used—in essence, it is a simple representation of a class diagram. The second file is an optional *.obj file, used to detail the set of objects that exist when the activity begins execution, i.e., a user-defined initial state. Samples are shown in Figures 9 and 10.

2. ***.act File** The user is also responsible for providing the activity to be executed, in the form of an *.act file. An activity file is written in the Activity Diagram Linear Form (ADLF) [14, 15], a lightweight textual representation of UML activities, intended to be easy to read and write. Figure 11 contains an ADLF representation of the activity in Figure 2.

3. **Initialize SM** The interpreter prepares the underlying System Model universe using the *.cd and *.obj files.

4. **Parse Activity** The interpreter parses the incoming *.act file, creating a graph representing the activity. Each element of the activity is represented as a type of graph node, e.g., action nodes (shown as rounded rectangles) are linked to action instances. All nodes are labelled for identification purposes and instances representing control tokens are placed on the initial nodes[4] of the resulting graph. At this point, the activity is ready for execution.

```
<InitialNode>
 -> (VSA value=Car1)
 -> [result-_]
 -> <DecisionNode>{
   "5 > 4" -> [_-object] ->
    (ASFVA *new structuralFeature=year)
   -> <ActivityFinal *final>,
   "4 < 5" -> [_-object] ->
    (ASFVA *old structuralFeature=year)
   -> <*final>
   };
[v2007-value] -> (*new);
[v2005-value] -> (*old).
```

Figure 11: ADLF representation of activity in Figure 2. Due to space restrictions, we have abbreviated the action names: `VSA` for `ValueSpecificationAction` and `ASFVA` for `AddStructuralFeatureValueAction`. Further details on ADLF can be found at [14, 15]

5. **Execute Activity**

   ACTi supports six different modes of execution, permitting users to explore activities in various ways. Table 1 shows

[4]Initial nodes in the graph include initial nodes in the activity, as well as any node without incoming edges. In the case of the activity in Figure 2, the initial nodes are `#1`, `#10`, `#11`.

these six modes, categorized along two dimensions. One dimension refers to the level of abstraction employed during execution. A 'deep' execution applies the standard semantics of actions and thus avoids all abstraction. By default, actions are assumed to be specific UML actions, e.g., CreateObjectAction, ReadVariableAction, etc. However, it is also possible for a user to define their own actions in the 'deep' mode. These actions must be accompanied by executable code, which makes changes to the underlying System Model state. On the other hand, a 'shallow' execution applies a non-standard semantics, which treats every action as a 'no-op', leaving the data store unchanged but advancing tokens. This type of execution is useful for experimenting with control flow, e.g., looking for different paths through an activity, checking for deadlock, etc.

Table 1: Modes of execution

	Default Initial State	**User-Defined Initial State**
Deep Detail	Random Guided	Random Guided
Shallow Detail	Random Guided	

The other dimension refers to the initial state of the System Model when the activity starts executing. With the 'default' initial state, the System Model universe has been defined by the user, but no objects exist. Alternatively, after defining the universe, the user can also define a set of objects (with values for their attributes) that should exist when the activity begins execution.

In three of the quadrants[5] of the modes table there are listed two types of execution:

- **Random** During execution, all nondeterministic choices are resolved using randomization. For instance, when there are multiple enabled nodes ready to be executed, one node is chosen at random. Also, if there are multiple true edges out of a decision node (with targets that are willing to accept the token offer), one edge is chosen at random to be traversed. Finally, when competition occurs, e.g., an object node is the source of multiple edges, the winning edge is chosen at random. This randomization ensures that the more executions that are performed on a particular activity, the more likely it is that different paths will be found.
- **Guided** It is also possible to suggest a path to the interpreter, forcing it to execute certain nodes in a specific order. This mode of execution is useful for exploring how the System Model changes in response to a specific execution. The user is notified if the specified path cannot be followed.

[5]Note that there are no execution types listed in the lower-right quadrant of Table 1. When performing a shallow execution, there is no advantage in setting the initial state in the System Model, as the execution will not modify the System Model at all.

We will now describe deep, random execution in more detail; this is probably the most commonly used mode of execution. We will assume that the initial state has been defined by the *.obj file shown in Figure 10. Note that a guided execution would be similar, except that no choices need be made with respect to the next node to execute; that would be predetermined by the user. On the other hand, a shallow execution would make use of the same randomness described above; however, no changes would be made to the system model by the actions themselves. Finally, the use of the default initial state would not affect the described execution.

ACTi includes a scheduler, which uses the token passing semantics detailed in the UML specification. As an activity is executed, the underlying System Model state is changed; a *macro step* is the full transition from state to state. Inside this macro step are *micro steps*, specifically:

- **Choose Next Token** The scheduler examines all tokens in the current state. Of those tokens that are resting on nodes that are currently enabled,[6] one token is chosen randomly.

- **Execute node/action** The node that the chosen token is resting on is executed. The execution of the node itself takes place at the node level of detail (see Figure 12). Furthermore, if an action node is being executed, then its specific action is executed at the action level of detail. Assuming that the node can complete execution immediately,[7] token offers are created for the node's immediate targets.
- **Offer and Pass Tokens** Now that the executed node (usually an action) has modified the underlying state, all that remains is to process the offers and pass the tokens. The scheduler deals with all tokens in one batch; see Section 3.2.

The activity diagram in Figure 12 summarizes the execution of a macro step. We distinguish between three levels of detail for execution. Tasks at the activity level require access to the entire activity. Tasks at the node level are restricted to actual node instances, e.g., action node, control node, etc. Finally, tasks at the action level are linked to implementations of the individual UML actions (not used during shallow execution). Changes made to the System Model's data store are typically made at the action level, while changes to the control store are typically made at the activity level.

6. **Trace** It is possible to have ACTi provide differing levels of trace information as output. At the very minimum, ACTi can output simply a list of nodes; the order of these nodes constitutes the resulting path through the graph. At the other end of the extreme, ACTi can perform a full trace, detailing exactly what is happening at each point in the execution, including how the scheduler is choosing tokens for execution, how token offers are proceeding, how the System Model is being modified, etc. In fact, the trace can also include a detailed listing of the System Model universe as each action is executed. A trace with a medium amount of information is shown in Figure 13. It contains information about which node is executed, and how tokens are being offered and passed.

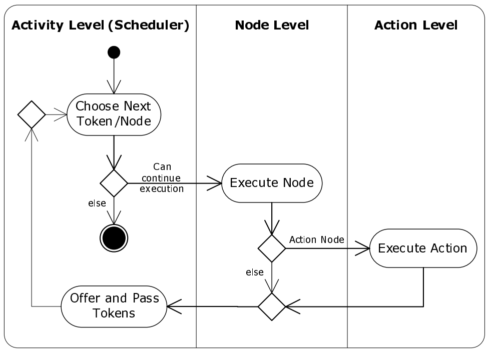


Figure 12: Activity showing tasks required during a *macro* step from state to state. Partitions are used to separate tasks by level of detail required; tasks on the left require access to the entire activity while those on the right are related to specific UML actions

7. **Analyze Paths** When configuring the execution of the desired activity, the user can also request different types of analysis. Although analysis can be performed on any single execution (random or guided), it is more useful when performed on multiple random executions. See Section 4 for a discussion of the analysis techniques that can be employed.

8. **Explore/Visualize** In addition to providing a textual trace output, ACTi can also generate a series of *.cmd files, which the user can incorporate as input into a third-party tool for visualization. The UML-based Specification Environment (USE) [1] is a powerful tool created at the University of Bremen that is

[6]An action node is enabled if it is not stalled (due to token offers and passing), it does not represent a call action waiting for a result, it does not represent an accept action waiting for an event, and it has sufficient tokens on its incoming edges.

[7]In almost all situations, a node can complete execution (and thus offer tokens to its targets) in one macro step. However, when either the CallOperationAction or the CallBehaviorAction are executed synchronously, the action calls the target behavior and then must wait for that behavior to complete. Only then can the calling action finish its execution and offer tokens.

used for the specification and analysis of systems. The user provides a USE model to create the structure of the specification. The tool provides for the visualization of the system as it is modified. We have created a USE model of the underlying System Model. That model, combined with the *.cmd files created when executing an activity, permits us to explore and visualize the state of the System Model as each node in the activity is executed.

3.2 Token Passing

As discussed in Section 2.1, the purpose of the token offer semantics is to avoid deadlock while executing an activity. The process of dealing with token offers and token passing is performed by the scheduler at the activity level (see Figure 12). When each node has completed execution, it creates token offers for its targets. In general, only action nodes can accept or reject offers; this means that an offer often needs to be propagated through the activity until an action node is encountered. All outstanding offers are maintained in a queue by the scheduler. The last part of the macro step is to process all offers in a batch. The scheduler examines each offer in the queue, determining whether or not the offer can be accepted by its target. Typically, when a token offer is accepted by a target node, the token gets passed to that node. However, many exceptions exist, for instance, a node with two incoming edges can only accept a token on one edge if it has also been offered a token on the other edge. To implement this semantics while processing the token offers, we perform a fixed-point iteration. Offers are iteratively processed until no more tokens can be transferred and no more offers have been propagated.

The trace in Figure 13 shows how complicated token passing can be. This is the trace of the macro step in Figure 2 in which the ValueSpecificationAction executes. The token offer and passing is complicated because the action must offer a token to another action node, e.g., either (or both) of the two AddStructuralFeatureValueAction nodes, which are on the other side of a decision node.

When node #2 finishes execution, it creates an offer to its immediate target, the object node #3. At this point, there are three offers in the scheduler's queue (because #10 and #11 have already been 'executed'). As a pin, #3 cannot accept the offer; it must pass the offer to its immediate target(s). In this case, the sole target is a decision node, which also cannot accept an offer. The decision node passes the offer along to all of its targets with positive guards, in this case, both of the AddStructuralFeatureValueAction nodes, #6 and #9. As can be seen from lines 53 and 57, both actions accept the offer. This acceptance is due to the fact that the other offers that these actions are waiting for have already been noted in this offer iteration. If it had been the case that either #10 or #11 had not yet executed, one or both of the actions would have rejected the offer.

In this case, both actions accept the offer; however, only one of them can be the recipient, resulting in competition for the token. The scheduler randomly chooses among the accepting targets. Here, #9 is the ultimate winner and a new token is passed to the action node.

The scheduler then iterates again through the offers. This time, the offer from #11 to #9 can now be accepted and passed also to the AddStructuralFeatureValueAction. At this point, that action is now considered enabled. The remaining offer from #10 to #6 can not be accepted, and indeed, will never be accepted in this execution.

4 Analysis

As discussed in Section 3.1, there are six specific modes available for executing activity diagrams. One useful analysis is to determine that there is indeed a suitable path through the activity. Another very useful analysis is to find as many paths as possible by randomly executing the activity multiple times. For example, there are 30 unique execution paths through the activity in Figure 2. Technically, ACTi does not have the ability to examine the activity and exhaustively enumerate all possible paths. However, in random execution mode, the more times an activity is executed, the more likely it is that all paths will be found. ACTi also outputs the relative probability of

```
      // Executed so far: #1, #10, #11
38.--- Macro Step ---------------------------------------------------------------------------
39.   ...ACTi.....Looking for enabled token/node.............................................
40.        Out of all tokens: Token_4 (on #2), Token_2 (stalled on #11), Token_3 (stalled on #10)
41.        Scheduler has chosen Token_4, sitting on enabled #2(ValueSpecificationAction value=Car1)
42.        [Token_4] is/are at node #2, which is about to fire
43.   ...Node Level.....Executing Node.......................................................
44.        Firing #2(ValueSpecificationAction value=Car1)
45.   ...ACTi.....Offer Process..............................................................
46.      #2 --?-> #3 #2 has offered Car1 to #3
47.      Looking at offer [#10] --v2007--> (#6)       //Rejected
48.      Looking at offer [#11] --v2005--> (#9)       //Rejected
49.      Looking at offer (#2) --Car1--> [#3]
50.      #3 --?-> #4                                  #3 has offered (temporarily) Car1 to #4
51.      #4 --?-> #8                                  #4 has offered (temporarily) Car1 to #8
52.      #8 --?-> #9                                  #8 has offered (temporarily) Car1 to #9
53.      #8 <-Y-- #9                                  #9 has accepted the offer from #8
54.      #4 <-Y-- #8                                  #8 accepted the offer from #4
55.      #4 --?-> #5                                  #4 has offered (temporarily) Car1 to #5
56.      #5 --?-> #6                                  #5 has offered (temporarily) Car1 to #6
57.      #5 <-Y-- #6                                  #6 has accepted the offer from #5
58.      #4 <-Y-- #5                                  #5 accepted the offer from #4
59.      #3 <-Y-- #4                                  #4 has accepted the offer from #3
60.      #2 <-Y-- #3                                  #3 has accepted the offer from #2
         //Random choice: #9 wins
61.      #2 --!-> #3                                  #3 has won the offer from #2
62.      Passing new control Token_5 from #2 to #9.          #2 -o-> #9
63.      Deleting old control Token_4 from #2
64.      Looking at offer [#10] --v2007--> (#6)       //Rejected
65.      Looking at offer [#11] --v2005--> (#9)
66.      #11 <-Y-- #9          #9 has accepted the offer from #11
67.      #11 --!-> #9          #9 has won the offer from #11
68.      Passing new control Token_6 from #11 to #9.         #11 -o-> #9
69.      Deleting old control Token_2 from #11
70.      Looking at offer [#10] --v2007--> (#6)       //Rejected
```

Figure 13: Partial trace of execution of activity in Figure 2. This trace shows the macro step in which the ValueSpecificationAction executes, and the subsequent token offers and token passing

each path being chosen.

4.1 Path Analysis

In addition to being able to find paths through a specific activity, ACTi also has several analysis capabilities. These analyses are most useful when executing an activity multiple times. The result is that ACTi can function as a limited model checker, i.e., the analysis is performed for each path found. The following analyses are available:

- **Desirable Nodes** The user defines a set of desirable nodes. Any path that does not contain all desirable nodes is considered a failure. This analysis is especially useful in larger activities incorporating decisions, merges, forks and joins. It can be used for reachability analysis, i.e., asserting that a desirable node is always reached.
- **Undesirable Nodes** The user defines a set of undesirable nodes. Any path that contains any undesirable node is considered a failure. Similar to the previous analysis, this can be used to assert that an undesirable outcome is never reached.
- **Mutual Exclusivity** The user defines a pair of nodes. Any path that contains both nodes is considered a failure. This analysis can be used in situations where it is desirable that either one node or another is encountered, but never both. For example, analysis of the activity in Figure 2 shows that nodes #6 and #9 are indeed mutually exclusive.
- **Precedence** The user defines an ordered pair of nodes. Any path that contains both nodes, but out of order, is considered a failure. This analysis can be used for debugging activities where it is desired that if two given nodes are encountered, they are encountered in a specific order.
- **Execute Node n Times** Some activities

contain loops; this analysis can be used to confirm that a specific node is encountered at least n times in every path.

- **Assertion Checking** It is possible to create basic assertions over object attributes. For instance, for the activity in Figure 2, we could assert that *Car1.year* = *v2007*. This analysis would check this assertion at the end of every encountered path.

  As mentioned earlier, there is non-determinism inherent in the activity in Figure 2. The non-determinism is caused, in part, by the fact that both outgoing branches of the decision node are true. Given that either branch could be taken, we would expect that the value of *Car1.year* would be set to *v2005* about half the time; the other half the time, it would be set to *v2007*. Figure 14 shows the results of checking the assertion *Car1.year* = *v2007* for our example.

```
Checking assertion [Car1.year == v2007]...
=========================================
Those paths for which the assertion does
*not hold* are marked with an asterisk.

        1.   1 10 11 2 6 7
   *    2.   1 10 11 2 9 7
        3.   1 10 2 11 6 7
             ...
   *   28.   11 1 2 9 7
       29.   11 10 1 2 6 7
   *   30.   11 10 1 2 9 7

Summary:
  Is the assertion [Car1.year == v2007]
    true for every path? false
  15 / 30 = 50.0% of paths do not satisfy
    the assertion
```

Figure 14: Results of checking assertion *Car1.year* == *v2007* on the activity from Figure 2. The assertion is true for half of the paths

4.2 Unused Tokens or Offers

The ACTi scheduler will continue to execute, choosing the next token/node to execute, until there are no more tokens on enabled nodes. When this occurs, there may be tokens and/or offers remaining in the activity. Sometimes, this may be acceptable, as in our example. For instance, in the activity in Figure 2, a decision is made, choosing one outgoing branch over another. When the activity in our example finishes execution, there is an outstanding offer remaining on the un-chosen branch. In this case, the leftover offer does not constitute an error.

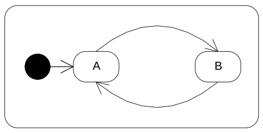


Figure 15: Simple activity that causes deadlock. Action `A` cannot execute until action B executes; action B cannot execute until action `A` terminates

In other cases, having leftover tokens or offers indicates that there is a problem with activity diagram, e.g., deadlock. Consider the activity in Figure 15. It is impossible for this diagram to terminate. The initial node will offer a token to action `A`; however `A` can never accept this token, because it also requires an offer from action `B`.

When the scheduler can no longer find an enabled node to execute, it warns the user if there are tokens or offers remaining in the activity. It is up to the user to confirm if these leftover tokens or offers indicate a deadlock or not, based upon the specific activity being executed.

4.3 Sanity Checks

ACTi also performs various 'sanity checks' during execution. Wherever possible, constraints from the specification are incorporated into the interpreter. For example, the following situations will raise warnings or errors:

- Flows into an initial or out of a final node.
- Mismatched flows into/out of a decision, merge, fork or join node.
- Missing or incorrect arguments for actions.
- Missing or incorrect static information for an action's attributes and/or associations.
- More than one 'else' branch leaving a decision node.
- A node is executed more than a predetermined number of times, i.e., rudimentary cycle checking.

5 Related Work

Actions were originally introduced into UML 1.5; in UML 2.0, activities were brought more in line with these actions to produce a more procedural model [21]. We have formally defined the behavior of UML actions [10] and use activities as our action language to 'glue' actions together.

Defining the semantics of UML activities is a vibrant research area; although the fact that activities substantially changed with the introduction of UML 2 narrows the field somewhat. However, there has been research on defining the semantics of activities based on several formalisms, such as Abstract State Machines [23], Petri nets [27, 24], and dynamic metamodelling [12]. Other research has focused on model checking activities [13].

There are several commercial tools that support executable UML. xUML [20] is a subset of UML, used by Kennedy-Carter in their iUML tool, and which incorporates the Action Specification Language and permits the construction of executable models. xtUML [17] makes use of the Object Action Language, and is implemented by Mentor Graphics' BridgePoint tool suite. There also exists a plugin for IBM's Rational Modeling tools, that "enables the execution, debugging and testing of UML models" [11] using state machines and activities. Stemming from academic research, ActiveChartsIDE [23] is a plugin for Microsoft Visio, with an interpreter that supports the simulation and debugging of activity diagrams.

Our research differs from these initiatives in two ways. First, we are using the System Model as our semantic domain. The System Model has been used to describe the semantics of interactions [7] and state machines [8], but it has not yet been used with activities. Second, we base our interpreter on the actual UML actions, making it suitable for the foundational subset for executable models requested in [18]. (iUML and BridgePoint also target executable forms of UML, but employ their own action language; moreover, both of them are commercial tools.)

Finally, activities are becoming quite popular in terms of business process modeling/workflow modeling [22, 29], including different standards of analysis for activities. For instance, [12] discusses a set of *soundness* rules, originally described in terms of workflow modeling [28]. Essentially, an activity can be considered sound, if the following conditions hold [12]:

- There is exactly one initial node, and one final node.
- It must be possible to execute every action—not necessarily in every execution, but in at least one execution.
- When a token arrives at the final node, there must not be any other tokens left in the activity.
- A token eventually arrives at the final node.

Our interpreter could be used to check activities with respect to these soundness rules.

6 Conclusion

ACTi is our interpreter for UML actions and activities, designed in accordance with the runtime semantics described in the UML specification.

ACTi supports a subset of UML activities, including the following concepts: sequential and concurrent composition, fork, join, merge, decision (including the parsing and evaluation of basic guards[8]). We also support the token offer semantics as described in the specification. We do not support partitions, exceptions, interruptible regions, structured activity nodes, buffer nodes, collections, weights on edges, or multiplicities other than 1..* on pins.

This interpreter can be used to execute and analyze activities composed of UML, or even user-defined, actions. The execution can be performed in either a deep or shallow fashion. Deep executions modify the state of the underlying System Model. In addition, a user-defined initial state may be specified before the activity execution commences. Execution of an activity can either be random (resulting in a non-deterministic path through the activity)

[8]The guards used in Figure 2 are a trivial example. ACTi can also interpret guards involving object attributes and variables, e.g., $Car1.year > v2005$ and $done! = vTrue$.

or guided (the user provides a chosen path). In addition, the random execution can be performed multiple times, resulting in a list of possible paths through the activity.

While configuring the execution of an activity, the user can request various types of analysis. When performed with multiple random executions, these analyses are akin to limited model checking; assertions are checked for every one of the paths that the interpreter discovers. Analysis techniques include: checking for (un)desirable nodes in a path, checking for mutual exclusion between two nodes in a path, checking for precedence between two nodes in a path, and checking an assertion at the end of a path. Deadlock detection and various sanity checks are also performed during execution.

6.1 Observations

Our work provides positive evidence for the hypothesis that the three-layer architecture is suitable not only for the description of the runtime semantics of UML, but also for its implementation.

After examining the semantics detailed in UML 2.0, [27] describes how fork nodes need to be able to 'hold' tokens; this modification was incorporated into UML 2.1. We also have discovered minor errors in the specification that lead to confusion, e.g., erroneous multiplicities in meta-model diagrams or a question about what happens to tokens that are rejected by the outgoing edge of a join.

Finally, although the specification contains many detailed constraints, the majority of the burden of ensuring the well-formedness of an activity rests with the modeler. For instance, it is the modeler's responsibility to: consider how multiple streams of tokens interact [19, §12.3.4], arrange that each token through a decision node can traverse only one edge [19, §12.3.22],[9] ensure that downstream joins do not depend on tokens travelling through forks with guarded edges [19, §12.3.30], etc. In other words, the modeler is responsible for ensuring that their activities conform to the more complicated aspects of execution semantics. Even then, it is possible to create activities that are conformant to the specification and yet do not seem to serve a clear purpose. Consider, for example, the activity in Figure 16. This activity abides by all the constraints of the specification and yet it cannot be considered well-formed as it stalls almost immediately.

[9]Although we have chosen to permit multiple true guards (resulting in potential non-determinism), it would be simple to indicate a 'sanity check' error if multiple true guards were encountered.

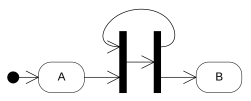


Figure 16: Activity that meets constraints in UML specification but that cannot be considered well-formed

6.2 Future Work

Future work with respect to the interpreter focuses on supporting more features of activities. Some features, such as multiplicities on pins, and weights on edges, will require incremental modifications to various parts of the interpreter. Specifically, the scheduler and encoding of token offer semantics would need to be modified. Other features, such as the inclusion of activity groups (and thus interruptible regions and structured activity nodes, such as loops and conditionals), would require high-level modifications. For instance, the creation of groups would require inserting a new level of abstraction—the group—between the graph and node levels.

The shallow mode of execution allows use of a non-standard semantics, and thus an, albeit rudimentary, form of abstract interpretation [9]. ACTi is designed to facilitate the application of other, more sophisticated non-standard semantics by, for instance, replacing one or more of the classes implementing actions. In future work, we want to use ACTi to help us identify notions of abstract interpretation that are useful for the analysis of activities.

Given the complicated nature of the semantics of activities, it would be useful to compile a collection of well-formedness rules for creating 'good' activities. In fact, it would be possible to categorize these rules according to the purpose of the activity. For instance, the soundness rules discussed in Section 5 could be considered a specialization of well-formedness. Once standards for well-formedness have been identified, it would be useful to add a well-formedness check to the analysis capabilities of ACTi.

In addition, the creation of a true model checker for activities would be useful. This work would require rewriting the scheduler to be able to exhaustively explore each path, e.g., a depth-first search through the graph, with backtracking.

Finally, ACTi currently represents only a partial implementation of the kind of virtual machine envisioned in [18]. In future work, we plan to broaden the scope of the interpreter to cover additional diagram types. Our formalization and implementation of actions together with the fact that the System Model was designed to support different types of behavioral specification diagrams (initial formalizations of interactions and state machines can be found in [7, 8]) will be extremely helpful here.

Acknowledgements

This research is supported by the Centre of Excellence for Research in Adaptive Systems (CERAS) and IBM. The authors thank Bran Selic of Malina Software Corporation and Conrad Bock of the National Institute of Standards and Technology for their invaluable assistance.

About the Authors

Michelle L. Crane is completing her Ph.D. at Queen's University. Her research interests include metamodelling, lightweight formal methods, and UML, specifically actions, activities and state machines. She can be reached at `crane@cs.queensu.ca`.

Juergen Dingel is an Associate Professor in the School of Computing at Queen's University. He heads the Applied Formal Methods Group and his research interests include the formal specification, verification and analysis of software systems in general, and software model checking, model-based testing and the foundations of model-driven development in particular. He can be reached at `dingel@cs.queensu.ca`.

References

[1] A UML-based Specification Environment. http://www.db.informatik.uni-bremen.de/projects/USE/, 2007. Web page of the USE project, from the University of Bremen.

[2] C. Bock. UML 2 activity and action models, part 4: Object nodes. *Journal of Object Technology*, 3(1):27–41, 2004.

[3] C. Bock. "re: Token/offer semantics for activities". E-mail to J. Dingel, Apr 25, 2008.

[4] M. Broy, M.V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The Structural Data Model. Tech. Report TUM-I0612, TUM, 2006.

[5] M. Broy, M.V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The Control Model. Tech. Report TUM-I0710, TUM, 2007.

[6] M. Broy, M.V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The State Machine Model. Tech. Report TUM-I0711, TUM, 2007.

[7] M.V. Cengarle. System model for UML – the interactions case. In *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, 2007.

[8] M.V. Cengarle, H. Grönniger, and B. Rumpe. System model semantics of statecharts. Technical Report (to appear), Technische Universität Braunschweig, 2008.

[9] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, Jun 1996.

[10] M.L. Crane and J. Dingel. Towards a formal account of a foundational subset for executable UML models. In *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2008. To appear.

[11] D. Dotan and A. Kirshin. Debugging and testing behavioral UML models. In *22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)*, pages 838–839. ACM, 2007.

[12] G. Engels, C. Soltenborn, and H. Wehrheim. Analysis of UML activities using dynamic meta modeling. In *9th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4468 of *LNCS*, pages 76–90. Springer, 2007.

[13] R. Eshuis. Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):1–38, 2006.

[14] D. Flater, P.A. Martin, and M.L. Crane. Rendering UML activity diagrams as human-readable text. Submitted.

[15] D. Flater, P.A. Martin, and M.L. Crane. Rendering UML activity diagrams as human-readable text. Technical Report NISTIR 7469, National Institute of Standards and Technology, November 2007.

[16] B. Gajanovic and B. Rumpe. ALICE: An advanced logic for interactive component engineering. In *4th International Verification Workshop (Verify'07)*, 2007.

[17] S.J. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture.* Addison-Wesley, 2002.

[18] Object Management Group. Semantics of a foundational subset for executable UML models. Request for Proposal ad/2005-04-02, Apr 2005.

[19] Object Management Group. Unified Modeling Language: Superstructure version 2.1. Document ptc/06-01-02, Jan 2006.

[20] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML.* Cambridge University Press, 2004.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 2005.

[22] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and P. Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *3rd Asia-Pacific Conference on Conceptual Modelling (APCCM)*, pages 95–104. Australian Computer Society, Inc., 2006.

[23] S. Sarstedt, J. Kohlmeyer, A. Raschke, M. Schneiderhan, and D. Gessenharter. ActiveChartsIDE. Poster at ECMDA 2005, Nov 2005.

[24] T. Schattkowsky and A. Förster. On the pitfalls of UML 2 activity modeling. In *International Workshop on Modeling in Software Engineering (MISE)*, pages 8–8, 2007.

[25] B. Selic. On the semantic foundations of standard UML 2.0. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185 of *LNCS*, pages 181–199. Springer, 2004.

[26] B. Selic. UML 2: A model-driven development tool. *IBM Systems Journal*, 45(3):607–620, Mar 2005.

[27] H. Störrle and J. Hausmann. Towards a formal semantics of UML 2.0 activities. In *Software Engineering*, volume 64 of *LNI*, pages 117–128, 2005.

[28] W. van der Aalst. Verification of workflow nets. In *18th International Conference on the Application and Theory of Petri Nets (ICATPN '97)*, volume 1248 of *LNCS*, pages 407–426. Springer, 1997.

[29] V. Vitolins and A. Kalnins. Semantics of UML 2.0 activity diagram for business modeling by means of virtual machine. In *9th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*, pages 181–194. IEEE Computer Society, 2005.