

Towards a Formal Account of a Foundational Subset for Executable UML Models

Michelle L. Crane and Juergen Dingel

School of Computing, Queen's University
Kingston, Ontario, Canada
{`crane,dingel`}@cs.queensu.ca

Abstract. A current Request for Proposal [1] from the OMG describes the requirements for an “Executable UML Foundation”. This subset of UML 2 would serve as a shared foundation for higher-level modeling concepts, such as activities, state machines, and interactions. In a sense, this subset would define a basic virtual machine for UML, allowing the execution and analysis of runtime behavior of models. Regardless of the executable subset chosen, a precise definition of execution semantics of UML actions is required. To the best of our knowledge, no formal semantics of such a subset yet exists. We present our work on clarifying the semantics and pragmatics of UML actions. In particular, we sketch a formalization of a subset of UML actions and discuss common usage scenarios for the most complex actions, identifying usage assumptions that are not explicit in the UML 2 specification.

Keywords: Executable UML, actions, activities, formal mapping, semantics, Model-driven Engineering.

1 Introduction

Early software development methods incorporating the idea of executable models include STATEMATE [2], the Shlaer-Mellor method [3] and ROOM [4]. In all of these methods, executable code is generated automatically from models of behavior. This feature makes them particularly suitable for the construction of automated testing and simulation environments and explains their popularity for the development of embedded systems where the testing of the software on a specific target system can be particularly difficult. The purpose of executable versions of the Unified Modeling Language (UML) is to make these advantages available to UML users by enabling “a chain of tools that support the construction, verification, translation, and execution of computationally complete executable models” [1].

An oft-voiced criticism of UML is its lack of a formal, unambiguous description of its semantics. While the UML 2 specification [5] is still lacking such semantics, it does offer a new three-layer semantics architecture to aid in the development of a formal semantics. This architecture provides a stratification of the description

of UML models that clearly separates ‘low-level’ (i.e., primitive) behavioral specification mechanisms such as actions from ‘high-level’ (i.e., derived) mechanisms such as activities, state machines, and interactions. The Object Management Group (OMG) has published a Request for Proposal [1], soliciting a definition of an “Executable UML Foundation”. This foundation would be a computationally complete and compact subset of UML, with fully specified executable semantics, which would “serve as a shared semantics foundation that would support additional semantics...covering the higher-level formalisms defined in UML.”

The UML specification defines a set of “fairly low-level actions sufficient to describe behavior” [6]. UML specifically does not choose one particular action language, in order to refrain from restricting modelers to any specific paradigm. That said, the specification does declaratively define an action semantics, i.e., what actions should do, but not how they should do it. In order to provide a truly executable modeling language, any executable UML foundation must be accompanied by an action language, preferably one which has been formally specified. To the best of our knowledge, no formal semantics for any UML 2 action language yet exists.

In addition to being informal, the description of actions in UML is rather fragmented, making it difficult to discern the pragmatics of actions, that is, the practical aspects of how the constructs and features of actions may be used to achieve various objectives. For instance, the semantics and pragmatics of actions designed to initiate the execution of a behavior are complicated by the fact that UML supports different computational paradigms (e.g., object-oriented, procedural, data flow, control flow, synchronous, asynchronous, etc.).

The purpose of this paper is to report on our work to clarify the semantics and pragmatics of UML actions. In particular, we will sketch a formalization of a subset of UML actions and discuss common usage scenarios for the most complex actions. The work described in this paper is the result of our ongoing efforts to create a virtual machine for UML.

This paper is structured as follows: Section 2 lays the groundwork by introducing UML’s semantic architecture and by describing the semantic domain, specifically those concepts necessary to understand the formal mapping presented in Section 3. Section 4 discusses the pragmatics of the most complicated actions—invocation actions. Related work is discussed in Section 5, while Section 6 concludes the paper.

2 Background

2.1 UML Semantic Architecture

UML is a “general-purpose visual modeling language” [6] that can be used in the analysis and design of software systems. Although well-documented, UML does not yet have a formal semantics. The abstract syntax is carefully laid out in over 1000 pages of specification, but the meaning of the syntactic elements is discussed in prose, with a smattering of Object Constraint Language (OCL) constraints.

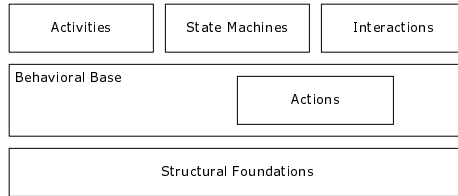


Fig. 1. The UML three-layer semantics architecture

The runtime semantics of UML is defined as a “mapping of modeling concepts into corresponding execution” [5]. This semantics is not formally defined in the UML specification. Instead, the specification outlines a three-layer semantic architecture, which “identifies key semantics areas...and how they relate to each other” [5].

This three-layer architecture is shown in Fig. 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. The middle layer is the behavioral base, which contains mechanisms for individual object behavior, as well as behavior involving more than one object. More importantly, this layer also contains the description of a set of UML actions. The top layer represents different behavioral formalisms in UML, all of which rely on the behavioral base. Activities, state machines, and interactions all make use of the actions in order to express behavior; these actions are explained in terms of constructs in the structural foundation.

The key to this architecture lies in the middle layer, i.e., actions. A fundamental premise of UML behavioral semantics is the assumption that “all behavior...is ultimately caused by actions” [5]. Actions are “fundamental units of behavior” [5]. As an analogy, actions are comparable to “executable instructions in traditional programming languages” [5].

The advantage of this approach to defining the runtime semantics is the fact that once UML actions are clearly mapped to the structural foundation, it should be relatively easy to define the semantics of different behavioral formalisms.

2.2 Semantic Domain

Our formalization leverages a previously developed semantic domain called the “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” [1]. Described in a series of three documents [7,8,9], 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 avoids the use of existing formal specification notations such as Z or Abstract State Machines and relies solely on simple mathematical concepts such as sets, relations, and functions. The System Model contains a formalization of UML’s structural foundation (i.e., the bottom layer of the architecture in Fig. 1), which results in a

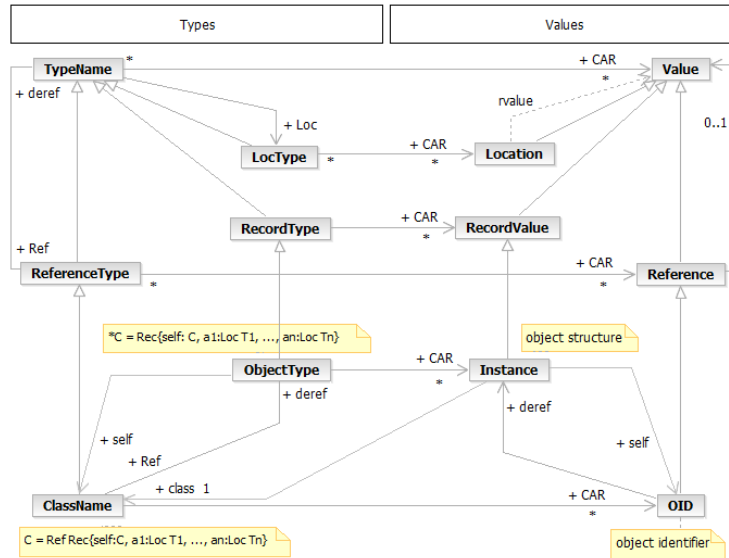


Fig. 2. Class diagram representing System Model *TypeName* and *Value* concepts. Note the symmetry between the two sides

formal notion of state. The meaning of a single behavioral diagram is 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 [7,8,9] to get a full description of all System Model concepts. In this paper, we present the essentials required to understand the formal mapping presented in Section 3.1. More precisely, we will only sketch the notion of state used in the System Model. First, a number of universes are postulated, such as *UTYPE* (type names), *UVAL* (values), *ULOC* (locations), *UVAR* (attribute and variable names), *UCLASS* (class names), *UOID* (object identifiers), *USTATE* (states), *UPC* (program counters), *UTHREAD* (threads), etc. Note that these universes encompass all UML models, all executions, and all behavioral formalisms. For example, *USTATE* is the universe of all states for all executions of all UML models, regardless of the type of behavioral formalism used, be it activities, state machines, etc. Then, types and values are organized into a type system as shown in Fig. 2. *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 [5]. 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. For instance, consider the class diagram in Fig. 3(a). If we instantiate an object of type *Class_Car*, the resulting System Model universe contains the elements shown in Fig. 3(b).

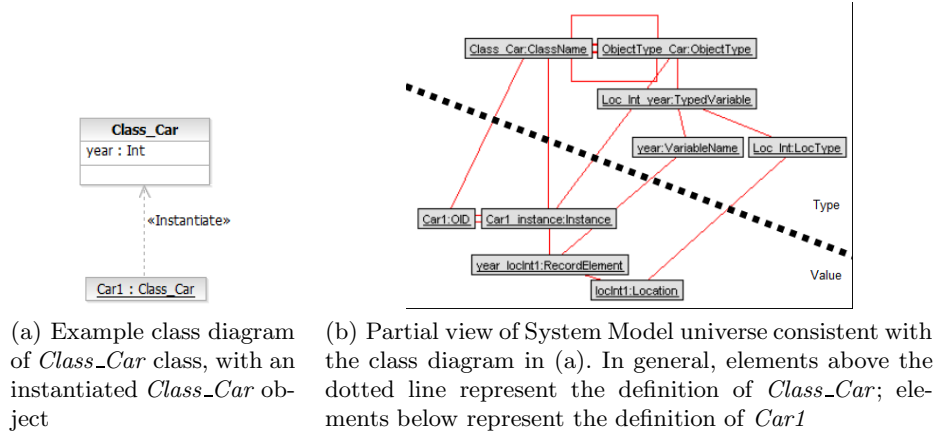


Fig. 3. Mapping from simple UML class and object to relevant System Model concepts

A state $s \in USTATE$ is defined as a triple $s = (ds, cs, es)$ where

- $ds \in DataStore$ is the data store, i.e., information about the current state of the UML model under examination. The data store (o, m) contains information about objects that currently exist in the execution ($o \subseteq UOID$), as well as mappings of those objects’ attributes to values ($m \subseteq ULOC \not\rightarrow UVAL$).
- $cs \in ControlStore$ is the control store, i.e., information related to the transitioning from state to state. We have added certain items to the control store description in order to allow the capture of the semantics of activity executions. The control store $(ad, pc, thr, vars)$ contains information about the activity diagrams currently being executed (one activity can call upon others), program counters ($pc \subseteq UPC$) and threads ($thr \subseteq UTHREAD$). In addition, we use the control store to store information about local variables, i.e., variables local to individual activities. The set $ad \subseteq ActivityDiagram$ is composed of individual activities, each defined by a unique identifier, a set of nodes, and a set of initial nodes. Every node is either an *ActionNode*, *ObjectNode*, or *ControlNode*; the union of these sets is the set *GraphNode*.
- $es \in EventStore$ is the event store, i.e., information related to messages, message passing, event buffers, etc. Each object has an associated event buffer.

When executing a typical imperative program, the control store’s program counter would hold the address of the next instruction to be executed. However, activities are by nature concurrent executions. We have introduced the concept of $Token \subseteq UPC$ to represent control tokens¹ in the executing set of activities. Thus, pc in the control store is now the set of tokens sitting on nodes. Moreover,

¹ As of UML 2, activities have been “redesigned to use a Petri-like semantics” [5]. Specifically, the concept of token offering is a generalization of Petri net transition enablement [10]. However, complications due to the token offer semantics precludes treating activities simply as Petri nets [10,11,12].

$SchedulerThread \subseteq UTHREAD$ is used to represent the runtime threads used during activity execution.

A token $t \in Token$ is defined as a 4-tuple $t = (id, thread, sittingOn, cameFrom)$ where

- $id \in String$ is a unique identifier
- $thread \in SchedulerThread$ is the runtime thread associated with this token
- $sittingOn \in GraphNode$ is the activity node upon which this token rests
- $cameFrom \in GraphNode$ is the activity node from which this token was sent

A scheduler thread $th \in SchedulerThread$ is defined as a pair $th = (id, token)$ where

- $id \in String$ is a unique identifier
- $token \in Token$ is the token associated with this thread

$Token_s$ is the set of tokens existing in state s and $Token = \bigcup_{s \in USTATE} Token_s$. In general, $Token_s \subseteq s.cs.pc$;² however, when dealing with activity diagrams, tokens are the *only* program counters and thus $Token_s = s.cs.pc$. Similarly, $SchedulerThread_s \subseteq s.cs.thr$ in general and $SchedulerThread_s = s.cs.thr$ when dealing with activities.

In general, threads contain information used during the execution of certain invocation actions. For the purposes of this paper, it is enough to state that tokens and scheduler threads are uniquely paired, i.e., every token has an associated scheduler thread, and vice versa:

$$\begin{aligned} & (\forall t \in Token \mid \exists ! th \in SchedulerThread \mid t.thread = th \wedge th.token = t) \\ & \wedge (\forall th \in SchedulerThread \mid \exists ! t \in Token \mid th.token = t \wedge t.thread = th). \end{aligned}$$

3 Semantics of Actions

As discussed earlier, an executing UML model is treated as a large state machine. Individual UML actions act like executable programming language statements, modifying the underlying state as they execute. Each action may have attribute and associations, which are used to provide static information to the action, e.g., the type of classifier to use with the `CreateObjectAction`. In addition, actions may have input and/or output pins, which are used dynamically, e.g., an output pin would hold the resulting object of the `CreateObjectAction`.

Activities are used to compose individual actions into larger executions. We describe activities as directed graphs; each graph node $gn \in GraphNode$ represents some concept in UML activities, e.g., an action node representing a specific UML action or a control node representing a fork, etc.

² Dot notation is used as shorthand for projections on particular fields of a tuple. For example, state s is defined as a tuple (ds, cs, es) . The expression $s.cs$ retrieves the cs term of state s , i.e., the control store. $s.cs.pc$ retrieves the pc term of the control store, i.e., the set of program counters. This convention is used throughout the paper.

An activity can call on other activities. Executing such a set of activities consists of finding a suitable path through the directed graphs(s) corresponding to the activities. As each graph node is traversed/executed, changes are made to the underlying state. For example, executing an initial node results in the passage of control tokens to the initial node's targets. Executing an action node (a graph node representing a specific UML action) causes various changes to the state, depending on the type of the action, as well as on that action's attributes, associations and inputs. There are a total of 45 actions defined in UML, 36 of which are concrete. To date, we have formalized 22 of the concrete actions; these are listed in Table 1.

An action node $an \in ActionNode$ is an 8-tuple. For the purposes of this paper, we are interested in only four of its terms: $action \in Action$ (referring to the specific UML action represented by the node), $in, out \subseteq GraphNode$ (the sets of source nodes and target nodes of this node),³ and $annot \subseteq Annotation$ (representing the set of annotations on this node—these provide information to populate the action's attributes and associations).

An action $a \in Action$ is defined as a 4-tuple $a = (name, inPins, outPins, attrs)$, where $name \in String$ is the name of the action, e.g., "CreateObjectAction". $inPins, outPins \subseteq String \times UVAL$ are the input and output pins, respectively. Finally, $attrs \subseteq String \times (UVAL \cup UVAR \cup UTYPE)$ is the set of attributes and associations for the action.

Information about an action's attributes, associations, and pins can be found in the UML metamodel. For example, Fig. 4 shows the UML metamodel related to the AddStructuralFeatureValueAction. Figure 5 shows a sample activity diagram making use of this action. When describing this activity, the relevant action node an would refer to action

$$a = (\text{"AddStructuralFeatureValueAction"}, \{(\text{"object"}, Car1), (\text{"value"}, 2005)\}, \{\}, \{(\text{"structuralFeature"}, year)\}.$$

3.1 Sample Mapping

Using the System Model as our basic semantic domain, we formally define the semantics for the behavior of individual UML actions by clearly identifying how the underlying state $s = (ds, cs, es)$ changes as an action node representing each particular action executes. Here, we present a formal definition of the state changes for the AddStructuralFeatureValueAction, e.g., as used in the activity in Fig. 5.

Let $asfva$ be an action node representing an instance of AddStructuralFeatureValueAction. We explain the effect of executing $asfva$ by defining a function

$$\llbracket asfva \rrbracket : USTATE \rightarrow USTATE$$

such that $\llbracket asfva \rrbracket(ds, cs, es)$ for $(ds, cs, es) \in USTATE$ is defined as follows:

³ The sets in and out are the direct sources and targets of a particular node. However, token offering/passing typically occurs between action nodes only, skipping any pins or control nodes. Thus, the functions $realIn : GraphNode \rightarrow \mathcal{P} GraphNode$ and $realOut : GraphNode \rightarrow \mathcal{P} GraphNode$ are defined to find the 'true' targets of any graph node.

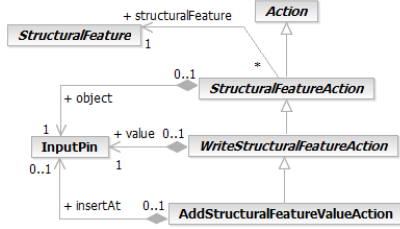


Fig. 4. UML metamodel for AddStructuralFeatureValueAction. StructuralFeature is the attribute to be modified, the “object” pin is the object to be modified and the “value” pin is the new value of the attribute. The “insertAt” pin is not used as we do not permit collections.

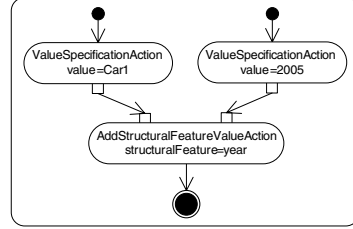


Fig. 5. Sample activity. The ValueSpecificationAction retrieves a value, e.g., the object identifier named *Car1*. The AddStructuralFeatureValueAction assigns a value to an attribute of an object. This diagram is equivalent to the pseudo-code statement `Car1.year = 2005`.

Pre-conditions. The action node *asfva* is *enabled*⁴ and has been selected by the scheduler to fire.

Post-conditions. If the pre-conditions are met, then

$$\llbracket asfva \rrbracket(ds, cs, es) = (ds', cs', es')$$

where

$$\begin{aligned} ds' &= setval(ds, oid, at, v) \\ cs' &= advance(cs, asfva) \\ es' &= es \end{aligned}$$

where

– $setval : DataStore \times UOID \times UVAR \times UVAL \rightarrow DataStore$ is defined as

$$setval((o, m), oid, at, v) = (o, m \oplus [*oid.at = v])$$

and

- $oid \in UOID$ is the object identifier whose attribute will be modified

$$(\text{“object”}, oid) \in asfva.action.inPins$$

- $at \in UVAR$ is the attribute to be modified

$$(\text{“structuralFeature”}, at) \in asfva.action.attrs$$

⁴ The precise definition of *enabled* has been removed due to space restrictions. An action node is considered enabled if it is not stalled (due to the nature of 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.

- $v \in UVAL$ is the new value of the attribute

$$(\text{“value”}, v) \in \text{sfva.action.inPins}$$

- \oplus is an operator that extends a mapping, e.g., $f \oplus [a = b]$ extends f by mapping a to b .
 - $*$ is the dereferencing operator. Recall that an object identifier oid is a reference to a record; $*oid$ is that record.
- $\text{advance} : \text{ControlStore} \times \text{GraphNode} \rightarrow \text{ControlStore}$ is a function defined as

$$\text{advance}((ad, pc, thr, vars), gn) = (ad, pc', thr', vars)$$

and the new set of program counters is the same as the old set, minus any tokens sitting on the currently executing node, plus a new token for each of the current node’s true targets, i.e.,

$$pc' = pc \setminus \text{removed} \cup \text{added}$$

where

$$\text{removed} = \{t \in pc \mid t.\text{sittingOn} = gn\}$$

and

$$\begin{aligned} \text{added} = \{t \in \text{Token} \setminus pc \mid t.\text{cameFrom} = gn \\ \wedge \exists! n \in gn.\text{realOut} \mid t.\text{sittingOn} = n\} \end{aligned}$$

and all threads in the new set of threads are matched with the new set of tokens

$$th' = \{t \in \text{SchedulerThread} \mid t.\text{token} \in pc'\}.$$

3.2 Summary of Changes to State

Table 1 lists all of the currently formalized concrete actions and indicates (with Δ) those parts of the state that change as each action is executed. We have also indicated exactly which parts of the control store are affected. Those parts of the state that are merely read during an action’s execution are marked with \checkmark .

UML actions are considered the “fundamental unit of behavior specification” [5]. For the most part, these actions can be considered primitive; they are defined to perform computation or access memory, but not both [5]. These blanket statements about UML actions are supported by several observations, drawn from the table:

- The execution of any action causes, at the very minimum, an advancement in the program counters and threads of the control store.
- Several actions only cause a change to the program counters and threads of the control store; they do not affect the rest of the state. These actions are essentially ‘read’ operations, e.g., `ReadVariableAction`. In addition, several actions read information from the System Model at large, instead of from the data store. For instance, the `ValueSpecificationAction` retrieves a value from the universe; the `TestIdentityAction` compares two values, etc.

Table 1. Summary of state changes caused by the execution of individual UML actions. Entries marked with \checkmark indicate that information is read only. Entries marked with Δ indicate that information is written.

Action	ds	cs			es
		vars	pc/th	ad	
AcceptCallAction			Δ		Δ
AcceptEventAction			Δ		Δ
AddStructuralFeatureValueAction	Δ		Δ		
AddVariableValueAction		Δ	Δ		
CallBehaviorAction			Δ	Δ	
CallOperationAction			Δ	Δ	Δ
ClearStructuralFeatureAction	Δ		Δ		
ClearVariableAction		Δ	Δ		
CreateObjectAction	Δ		Δ		
DestroyObjectAction	Δ		Δ		Δ
ReadExtentAction	\checkmark		Δ		
ReadIsClassifiedObjectAction			Δ		
ReadSelfAction			Δ	\checkmark	
ReadStructuralFeatureAction	\checkmark		Δ		
ReadVariableAction		\checkmark	Δ		
RemoveStructuralFeatureValueAction	Δ		Δ		
RemoveVariableValueAction		Δ	Δ		
ReplyAction			Δ		Δ
SendSignalAction			Δ	\checkmark	Δ
StartClassifierBehaviorAction			Δ	Δ	
TestIdentityAction			Δ		
ValueSpecificationAction			Δ		

- Several actions require information about their context in order to execute. These are marked with a \checkmark in the *ad* column. For instance, the `ReadSelfAction` returns the object identifier of the object that owns the activity in which the action is found. To do this, the action needs to navigate to its surrounding activity diagram, and thence to the owning object of the diagram.
- Several actions do not read information about their context, but instead modify their context, i.e., the set of activity diagrams currently being executed. For example, the `CallOperationAction`, `CallBehaviorAction` and `StartClassifierBehaviorAction` all refer to behavior that might be contained in a separate activity diagram. As the call is executed, the additional diagram is incorporated into the set *ad*.
- In general, most actions are ‘specific’ in that they modify only one part of the state. For instance, structural feature actions only modify the data store, variable actions only modify the *vars* portion of the control store, etc.
- Thus far, we have identified only two ‘non-specific’ actions; they affect two or more areas of the state, in addition to the advancement of program counters and threads. `CallOperationAction` modifies both the set of current activities *ad* (by causing the called activity to be added) and the event store *es* (by

handling a call message). `DestroyObjectAction` removes an object from the data store, and its attendant event buffer from the event store.

- Finally, for the most part, actions can be considered ‘single-step’. By this, we imply that the action’s execution causes changes to the state, and then the action need not be revisited. In contrast, a ‘multi-step’ action must be visited twice in order to fully complete its execution. For instance, when either the `CallOperationAction` or `CallBehaviorAction` is used synchronously, the execution of the calling action is not considered complete until the called behavior (e.g., another activity) has been completely executed. It is, however, not the case that all invocation actions are multi-step. Either of the call actions can be executed asynchronously, which means that the invoked behavior is added to the current set of activities, and then the call is considered complete. Similarly, the send/broadcast actions never wait for a reply.

4 Pragmatics of Invocation Actions

In general, the majority of UML actions are straightforward, e.g., the `CreateObjectAction` creates a new object identifier, the `ReadVariableAction` reads a variable, etc. Invocation actions, on the other hand, can be quite esoteric. There are so many actions devoted to invoking behavior in UML that it can be rather complicated to determine exactly which action to use when modeling an activity.

Table 2 lists the various actions in UML that can be used to invoke behavior. The actions are categorized according to two dimensions: whether the invocation can be considered direct or indirect, and whether the invoked behavior can be executed synchronously or asynchronously. For our purposes, an invocation is direct when the target behavior is accessed without any intermediaries, e.g., there is no message or signal to be accepted before the invoked behavior can execute. Indirect execution relies on a call message, or a transmitted signal or object, in order to request the execution of a behavior. In other words, there is the possibility that a behavior invoked indirectly will not actually execute. On the other hand, behavior invoked directly should execute, barring unforeseen circumstances. With respect to the second dimension, a behavior that is invoked synchronously must execute and terminate before the calling behavior can complete its own execution. When a caller invokes a behavior asynchronously, however, the calling behavior can continue to execute while, or even before, the invoked behavior executes.

Direct Invocation

1. `CallBehaviorAction` with `isSynchronous = false`: This action is used to invoke a behavior directly, without using message or signal passing. There is no requirement for an accept action in the targeted behavior. An asynchronous `CallBehaviorAction` is considered completed when its called “behavior is started, or at least ensured to be started at some point” [5, §11.3.9].
2. `StartClassifierBehaviorAction`: Although not technically an invocation action (it does not inherit from `InvocationAction`), this action “initiates the behavior

Table 2. Summary of invocation actions. While not technically an invocation action, the `StartClassifierBehaviorAction` is included because its function is to initiate the classifier behavior of an object.

Invocation	Asynchronous	Synchronous
Direct	1. <code>CallBehaviorAction</code> isSynchronous = false 2. <code>StartClassifierBehaviorAction</code>	3. <code>CallBehaviorAction</code> isSynchronous = true
Indirect	4. <code>CallOperationAction</code> isSynchronous = false 5. <code>SendSignalAction</code> 6. <code>SendObjectAction</code> 7. <code>BroadcastSignalAction</code>	8. <code>CallOperationAction</code> isSynchronous = true

of the classifier of the input object” [5, §11.3.46]. There are no messages or signals and the targeted object has no choice but to execute the behavior; thus, this action is a direct invocation. It is asynchronous in that the calling behavior continues execution as soon as the classifier behavior is invoked.

3. `CallBehaviorAction` with isSynchronous = true: When used synchronously, the `CallBehaviorAction` “waits until the execution of the invoked behavior completes and a result is returned on its output pin” [5, §11.3.9].

Indirect Invocation

4. `CallOperationAction` with isSynchronous = false: This is an indirect invocation action in that it “transmits an operation call request to the target object, where it may cause the invocation” [5, §11.3.10] of the desired behavior. When used asynchronously, the calling behavior can continue execution after the call request has been sent. On the receiving end, the `AcceptEventAction` is designed to be used for accepting asynchronous calls; however, the `AcceptCallAction` can also be used. In fact, we prefer using the `AcceptCallAction` so that the same target activity can be used to handle both synchronous and asynchronous calls.
5. `SendSignalAction`: This action creates a signal and transmits it to the target object, where it may cause the invocation of behavior [5, §11.3.45]. The calling behavior continues execution. On the receiving end, an `AcceptEventAction` is required.
6. `SendObjectAction`: Similar to the `SendSignalAction`, this action transmits an object to the target object, where it may invoke behavior [5, §11.3.44]. Here too, the calling behavior continues execution.
7. `BroadcastSignalAction`: This action is identical to the `SendSignalAction`, except that the signal is sent to multiple targets, where it may invoke behavior [5, §11.3.7].
8. `CallOperationAction` with isSynchronous = true: When used synchronously, the `CallOperationAction` must be accepted on the receiving end by an `AcceptCallAction` [5, §11.3.1], which will be linked to a `ReplyAction`. The `ReplyAction` completes the execution of the call, returning the result to the caller [5, §11.3.43], which can then terminate.

5 Related Work

The UML specification defines the *action semantics*, which describes the effects of a set of “fairly low-level actions sufficient to describe behavior” [6] but it does not define a concrete syntax for actions. Any executable UML foundation must be accompanied by an action language. We are primarily interested in formal semantics, specifically of UML actions. In order to examine the execution of actions, we use UML activities as our action language to ‘glue’ actions together.

Academic action languages tend to be part of tools focused on the analysis of models. OxUML is an OCL-based action language, which supports actions with side effects. The result is an executable UML, supported by a UML virtual machine [13]. [14] suggests a profile for UML that provides “complete and precise Aspect-Oriented behavior modeling” including UML’s action semantics. After weaving, the complete model can be executed in order to observe the behavior of the modeled aspects. ActiveChartsIDE [15] is a plugin for Microsoft Visio with an interpreter that supports the simulation and debugging of activity diagrams.

There are several commercial tools, which support executable UML; each of these includes their own proprietary action language. xUML [16] 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 is also a plugin for IBM’s Rational Modeling tools that “enables the execution, debugging and testing of UML models” [18] using state machines and activities. This execution engine does not specifically support UML actions; it uses Java as an action language.

Although a requirement for an executable UML is the inclusion of some kind of action language, none of the related work presented thus far focuses on the individual UML actions. These initiatives all make use of higher-level action languages. To the best of our knowledge, none of these action languages is accompanied by a formal semantics.

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. For instance, [19] provides a semantics for an older version of UML actions, but is obsolete because pins have since been introduced. On the other hand, there has been research on defining the semantics of UML 2 activities based on several formalisms, such as Abstract State Machines [15], Petri nets [11,12] and dynamic metamodeling [20]. Other research has focused on model checking activities [21]. Unfortunately, none of this particular research supports individual UML actions.

Thus far, the OMG has been presented with one submission [22] in response to their RFP for an Executable UML Foundation. This submission defines the Foundational Subset for Executable UML (fUML) and is supported by many of the large names in modeling technologies. Our research into the semantics of actions is most closely related to this submission. In our case, we are not so much interested in producing a “compact” subset of UML. For example, we are able to support calls, and thus `AcceptCallAction` and `ReplyAction`. We also

support variable actions, as well as the decision control node (with guards). Instead of decisions, fUML supports the loop and conditional nodes. Neither project supports links or associations.

6 Conclusion

We formally define the execution semantics of a subset of UML actions. This definition is expressed in terms of state changes to the global state machine that represents an executing UML model. We also discuss common usage scenarios for the most complex actions, i.e., invocation actions.

Our research has resulted in uncovering certain ambiguities and implicit assumptions about UML actions. For instance, the UML specification does not specify what should happen when the `DestroyObjectAction` destroys its owning object. In addition, the requirement to pair a synchronous `CallOperationAction` with an `AcceptCallAction` is not clear from the description of the former action. Finally, the fact that an asynchronous `CallOperationAction` should be matched by an `AcceptEventAction`, but could be matched by an `AcceptCallAction` is potentially confusing—the modeler would need to know whether the call was to be synchronous or asynchronous before creating the receiving behavior.

In addition to formally specifying the execution semantics of actions, a major contribution of our work is that we are validating the three-layer semantics hierarchy from Fig. 1. By using activities as our action language, we successfully map a higher-level behavioral formalism to actions. Then, we map these actions to the state transformers over the structural foundation, described by the System Model, a semantic domain specifically designed to capture the semantics of UML models on a maximally general level and to deal easily with collections of different kinds of UML diagrams.

Finally, this research is part of our ongoing efforts to create a virtual machine for UML. A discussion of our interpreter for actions and activities can be found at [23].

Acknowledgements

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

References

1. Object Management Group: Semantics of a foundational subset for executable UML models. Request for Proposal ad/2005-04-02 (April 2005)
2. Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts: The STATE-MATE Approach. McGraw-Hill, New York (1998)

3. Shlaer, S., Mellor, S.: *Object Lifecycles: Modeling the World in States*. Prentice-Hall, Englewood Cliffs (1992)
4. Selic, B., Gullekson, G., Ward, P.: *Real-Time Object-Oriented Modeling*. Wiley, Chichester (1994)
5. Object Management Group: *Unified Modeling Language: Superstructure version 2.1*. Document ptc/06-01-02 (January 2006)
6. Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual*, 2nd edn. Addison-Wesley, Reading (2005)
7. Broy, M., Cengarle, M., Rumpe, B.: *Semantics of UML – Towards a System Model for UML: The Structural Data Model*. Technical Report TUM-I0612, TUM (2006)
8. Broy, M., Cengarle, M., Rumpe, B.: *Semantics of UML – Towards a System Model for UML: The Control Model*. Technical Report TUM-I0710, TUM (2007)
9. Broy, M., Cengarle, M., Rumpe, B.: *Semantics of UML – Towards a System Model for UML: The State Machine Model*. Technical Report TUM-I0711, TUM (2007)
10. Bock, C.: *Re: Token/offer semantics for activities*. E-mail to J. Dingel (April 25, 2008)
11. Störrle, H., Hausmann, J.: *Towards a formal semantics of UML 2.0 activities*. In: *Software Engineering. LNI*, vol. 64, pp. 117–128 (2005)
12. Schattkowsky, T., Förster, A.: *On the pitfalls of UML 2 activity modeling*. In: *Proceedings of the International Workshop on Modeling in Software Engineering (MISE)*, p. 8 (2007)
13. Jiang, K., Zhang, L., Miyake, S.: *An executable UML with OCL-based action semantics language*. In: *Asia-Pacific Software Engineering Conference (APSEC)*, pp. 302–309 (2007)
14. Fuentes, L., Sánchez, P.: *Towards executable aspect-oriented UML models*. In: *10th International Workshop on Aspect-oriented Modeling (AOM)*, pp. 28–34. ACM Press, New York (2007)
15. Sarstedt, S., Kohlmeyer, J., Raschke, A., Schneiderhan, M., Gessenharter, D.: *ActiveChartsIDE*. In: *ECMDA 2005* (2005)
16. Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I.: *Model Driven Architecture with Executable UML*. Cambridge University Press, Cambridge (2004)
17. Mellor, S., Balcer, M.: *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, Reading (2002)
18. Dotan, D., Kirshin, A.: *Debugging and testing behavioral UML models*. In: *22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)*, pp. 838–839. ACM Press, New York (2007)
19. Ober, I., Coulette, B., Gandriau, M.: *Action language for the UML*. In: *Langages et Modèles à Objets (LMO)*, Hermes, pp. 277–291 (2000)
20. Engels, G., Soltzenborn, C., Wehrheim, H.: *Analysis of UML activities using dynamic meta modeling*. In: *Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS*, vol. 4468, pp. 76–90. Springer, Heidelberg (2007)
21. Eshuis, R.: *Symbolic model checking of UML activity diagrams*. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15(1), 1–38 (2006)
22. Object Management Group: *Semantics of a foundational subset for executable UML models*. Initial Submission ad/06-05-02 (May 2006)
23. Crane, M., Dingel, J.: *Towards a UML virtual machine: Implementing an interpreter for UML 2 actions and activities*. In: *2008 conference of the Centre for Advanced Studies on Collaborative research (CASCON) (to appear, 2008)*