

## On the Semantic Foundations of Standard UML 2.0

Bran V. Selic

IBM Distinguished Engineer  
IBM Rational Software Canada  
bselic@ca.ibm.com

**Abstract.** This paper provides an overview of the foundations of the run-time semantics underlying the Unified Modeling Language as defined in revision 2.0 of the official OMG standard. One of the problems with the format used for that standard is that the information relating to semantics is scattered throughout the text making it difficult to obtain a global understanding of how the various fragments fit together. This has led many to incorrectly conclude that UML has little or no semantic content. One of the objectives of this paper is to provide a clear and concise description of the structure and essential content of UML run-time semantics. This can serve as a convenient starting point for researchers who want to work on the problem of UML semantics and, in particular, those who are interested in producing formal models of those semantics.

### 1 Introduction

An oft-repeated criticism of UML is that it has “no semantics”; that it is primarily a visual notation whose graphical constructs can be interpreted more or less according to whim. One objective of this paper is to dispel such unwarranted criticisms by explaining the structure and content of the semantic foundations that underlie UML based on the most recent revision of the official standard (UML 2.0).

The term “semantics” as used in this paper refers to the run-time interpretations of UML models, that is, the structural entities and behaviors that are represented by the different modeling concepts of UML. For some, the only acceptable definitions of language semantics are ones that are expressed in some well-known *mathematical* formalism, such as Z or CSP. In fact, it is often argued that an accepted mathematical formalism should be the starting point for defining a language, since that greatly increases the likelihood that the semantics will be clean, consistent, and amenable to formal analysis. Computer languages such as Lisp are often cited as successful examples of this approach. However, when it comes to modeling languages such as UML, this may not be the most suitable approach.

This is because UML is intended to model complete systems across a broad spectrum of different application domains. Such models typically go beyond modeling just software and computation. For example, in the analysis and design of real-time and embedded software systems, it may be necessary to model the behavior of some real-world physical entity such as a hardware device or human user. In general, physical things tend to be highly diverse and much more complex than most mathematical formalisms can handle.

An additional consideration related to formal models, is that it is often the case that the same entity may need to be modeled from different viewpoints related to different

sets of concerns. For example, modeling system users from the perspective of performance (e.g., inter-arrival times) is very different than modeling them from the point of view of human-computer interaction.

This suggests that basing UML on any specific concrete mathematical formalism would likely severely hamper one of its primary objectives: to unify a set of broadly applicable modeling mechanisms in a common conceptual framework. This aspect of UML must not be underrated and is one of the key factors behind its widespread adoption. The commonality that it provides makes it possible to use the same tools, techniques, knowledge, and experience in a variety of different domains and situations.

This is not to say that UML is incompatible with formalization. On the contrary – it is crucial that suitable formal representation of its semantics be defined for all the good reasons that such representations provide and in particular because it is one of the keystones of the MDA initiative [4]. In fact, another key objective of this paper is precisely to encourage the development of suitable formalizations of UML by providing a clear specification of what needs to be formalized.

There have already been numerous notable efforts to formally define the run-time semantics of UML (e.g., [2] [3] [12]). However, most of them only cover subsets of UML and are usually only loosely based on the semantics defined in the official UML standard specifications [6] [8] [7].

One problem shared by some of these efforts is that, because they are constructed for a specific purpose or domain, they tend to define a single concrete semantics. This immediately puts them in conflict with the broad scope of UML noted above. In essence, standard UML is the foundation for a “family” of related modeling languages<sup>1</sup>. This means that any formal semantics definition of standard UML should provide the same kind of latitude for different interpretations as the standard itself. In effect, a formalized semantics of standard UML should define a *semantic envelope* of possible concrete semantics.

A major impediment to any kind of formalization of UML semantics is that the standard documents do not cover the semantic aspects in a focused fashion. Instead, due to the idiosyncrasies of the format used for standards, the material is scattered throughout the documents, making it very difficult to develop a consistent global picture<sup>2</sup>. In addition, the standard omits much of the rationale and historical background that is required for a full understanding of these aspects<sup>3</sup>. To rectify these shortcomings, this paper provides a concise synthesized view of the structure, content, and rationale of the run-time semantics defined in the standard.

The following section starts this off with a high-level view of the semantics definitions of standard UML and their relationships. It includes an informal description of the UML view of cause-and-effect relationships that sit at the core of any dynamic semantics specification. The more substantive exploration starts in section 3, which describes the structural semantic foundations. This is followed by a description of the behavioral semantic base in section 4. This base is shared by all the different higher-

---

<sup>1</sup> The term “family of languages” for describing UML was first used by Steve Cook.

<sup>2</sup> To a certain extent, this fragmentation has had a negative impact on the actual definition of the semantics leading to minor inconsistencies and omissions. This issue is discussed further in section 5.

<sup>3</sup> In software, as with most things, it is often useful to know the history of how ideas evolved to fully understand them. Unfortunately, this is often unrecognized in technical cultures.

level behavioral paradigms of UML (e.g., state machines, activities). To assist those interested in a more detailed understanding, section 5 provides a guide that identifies where the various semantic elements described in the paper are located in the actual UML standard document. A brief summary and suggestions for related research are provided at the end of the paper.

## 2 The Big Picture

The launching of the MDA initiative by the OMG, with its emphasis on using models for more than just documentation or informal design “sketching”, generated a strong impetus to expand and clarify the semantics of UML. This was reflected in the definition of the first major revision of UML, UML 2.0, which provides a much more extensive and systematic coverage of semantics relative to earlier versions.

There are two fundamental premises regarding the nature of UML semantics in UML 2.0 that need to be stated up front. The first is the assumption that all behavior in a modeled system is ultimately caused by actions executed by so-called “active” objects (explained below). The second is that UML behavioral semantics only deal with *event-driven*, or discrete, behaviors. This means that continuous behaviors, such as found in many physical systems, are not supported. To be sure, this capability is not precluded, and it is likely that extensions will be defined to provide it. One possible opportunity for this might be the current effort to define a UML for system engineering [5].

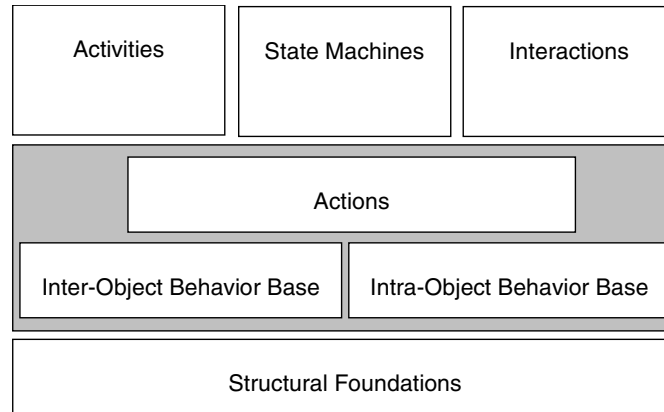
### 2.1 The Semantics Architecture

Fig. 1 identifies the key semantics areas covered by the current UML 2.0 standard. It also shows the dependencies that exist among them.

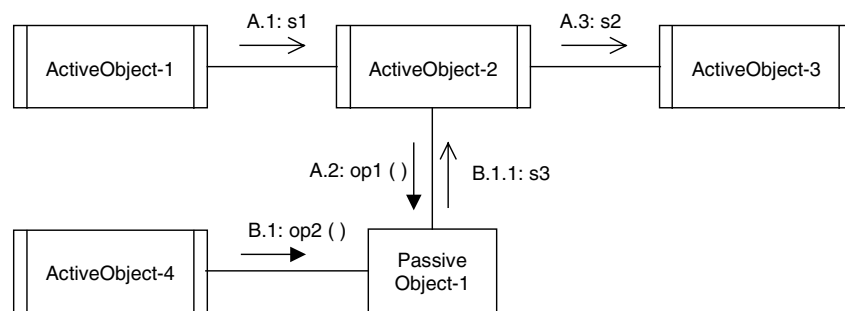
At the highest level of abstraction, it is possible to distinguish three distinct layers of semantics. The foundational layer is structural. This reflects the premise that is no disembodied behavior in UML – all of it emanates from the actions of structural entities. The next layer is behavioral and provides the foundation for the semantic description of all higher-level behavioral formalisms<sup>4</sup>. This layer (represented by the shaded rectangle in Fig. 1) is called the Behavioral Base and consists of three separate sub-areas arranged into two sub-layers. The bottom sub-layer consists of the *inter-object behavior base*, which deals with how structural entities communicate with each other, and the *intra-object behavior base*, which addresses the behavior occurring within structural entities. The *actions* sub-layer is placed over these two. It defines the semantics of individual actions. Actions are the fundamental units of behavior in UML and are used to define fine-grained behaviors. Their resolution and expressive power are comparable to the executable instructions in traditional programming languages. Actions in this sub-layer are available to any of the higher-level formalisms to be used for describing detailed behaviors. The topmost layer in the semantics hierarchy defines the semantics of the higher-level behavioral formalisms of UML: *activities*, *state machines*, and *interactions*. These formalisms are dependent on the semantics provided by the lower layers.

---

<sup>4</sup> The term “behavioral formalism” as used throughout this text does not imply formal definition in the mathematical sense, but denotes a distinct behavioral paradigm.



**Fig. 1.** The UML semantics layers: the Semantics Foundation consists of the bottom two layers – the Structural Foundations and the Behavioral Base (shaded area)



**Fig. 2.** Example illustrating the basic causality model of UML

This paper deals with the two bottom layers only. These two layers represent the semantic foundations of UML. This is shared substrate that ensures that objects can interact with each other regardless of which specific high-level formalisms are used to describe their behaviors.

## 2.2 The Basic Causality Model

The “causality model” is a specification how things happen at run time. Instead of a formal treatment, this model is introduced using the example depicted in the collaboration diagram in Fig. 2. The example shows two independent and possibly concurrent threads of causally chained interactions. The first, identified by the thread prefix ‘A’ consists of a sequence of events that commence with ActiveObject-1 sending signal s1 to ActiveObject-2. In turn, ActiveObject-2 responds by invoking operation op1( ) on PassiveObject-1 after which it sends signal s2 to ActiveObject-3. The second thread, distinguished by the thread prefix ‘B’, starts with ActiveObject-4 invoking operation op2( ) on PassiveObject-1. The latter responds by executing the method that realizes this operation in which it sends signal s3 to ActiveObject-2.

The causality model is quite straightforward: objects respond to messages that are generated by objects executing communication actions. When these messages arrive, the receiving objects eventually<sup>5</sup> respond by executing the behavior that is matched to that message. The dispatching method by which a particular behavior is associated with a given message depends on the higher-level formalism used and is not defined in the UML specification (i.e., it is a semantic variation point).

In the course of executing its behavior, an object may send messages to other objects, and so on. Note that the term “message” is used here in a generic sense to cover both synchronous and asynchronous communications. In fact, a synchronous communication (call) requires two messages. First is the message from the source to the target identifying the operation to be invoked as well as the values of any arguments. This is followed by a reply message from the target back to the calling object that includes values to be returned to the caller. Because the call is synchronous, the caller is suspended until the reply is received.

### 2.3 Active Objects: The Sources of Behavior

UML objects are categorized into *active* objects and *passive* objects. The primary difference between them is the manner in which incoming messages are handled by different kinds of objects. Active objects respond only when they execute a receive action. In a sense, active objects control when they interact with their environment. The following definition taken from the UML 2.0 specification describes this crucial characteristic of active objects:

*[the] point at which an active object responds to communications from other objects is determined solely by the behavior of the active object and not by the invoking object.*

In the context of state machine formalisms, this property of active objects is also known as “run-to-completion”. That is, once the active object has accepted a message, no further messages are accepted until the next receive action is executed. For state machines this takes place only after a stable state is reached.

Any messages arriving at an active object between successive receive actions are simply stored in a holding buffer. (Note that this buffer does not necessarily have to be a queue, since stored messages do not have to be processed in the order of arrival. The decision on which of potentially many saved messages to process when the next receive action is executed depends on the scheduling discipline and is a semantic variation point. For instance, priority-based schemes will order messages according to their priority and not according to the arrival order.)

In contrast, passive objects have no say in the matter, they respond when the message arrives, regardless of whether the object is already occupied processing a previous message. This means that concurrency conflicts may occur when accessing passive objects. If this is undesirable, special measures may be needed to ensure mutual exclusion.

No such problems occur for active objects. This freedom from concurrency conflicts without resorting to complex and error-prone conflict management mechanisms

---

<sup>5</sup> An object does not necessarily respond immediately to an arriving message – this depends on the system semantics as well as on the type of behavioral formalism used for the receiver.

is one of the most appealing features of active objects. (Of course, two active objects can still interfere with each other if they share an unprotected passive object.) The price paid for this is the possibility of *priority inversion*, which is a situation where a high-priority message may have to wait until a low-priority message is fully processed.

Following creation, an active object commences execution of its behavior, which it continues to execute until it completes or until the object is terminated by an external agency. In the course of executing its behavior, the active object may execute one or more receive actions.

Note that the semantics of UML active objects are defined independently of any specific implementation technology. Thus, there is no reference to “threads of control” or “processes” in their definition. This means that the concept to be interpreted quite broadly and can be used to model many different kinds of entities, including active entities in the real world.

### 3 The Structural Foundations

Although not shown in Fig.1, this foundational layer is decomposed into two sub-layers. The *elementary* sub-layer deals with the atoms of structure: objects, links, and the like. At the next level up is the *composites* sub-layer, which describes the semantics of composite structures.

#### 3.1 The Elementary Sub-layer

As noted, all behavior in a system modeled by UML is the result of objects executing actions. However, at this most basic semantics level, an object is not the most fundamental concept. The structural concepts defined at this level are:

- *Pure values*. Pure values are timeless and immutable. They include numbers, strings and character values, Boolean values (true and false), etc. A special kind of pure value is a *reference*, which is a value that uniquely identifies some structural entity<sup>6</sup>. In the UML metamodel, values are specified by instances of the ValueSpecification metaclass. References are represented by the metaclass InstanceValue, which is a subclass of ValueSpecification.
- *Cells*. These are named structural entities capable of storing either pure values or other cells. They are typed, which means that they can only store pure values or structural entities of a particular type. Cells are dynamic entities that can be created and destroyed dynamically. During their existence, their contents may change. Each cell has a unique identity, which distinguishes it from all other cells. The identity of a cell is independent of its contents. A reference is a value that is a concrete representation of the identity of a cell. If a cell contains another cell, the contained cell is destroyed when the containing cell is destroyed. (Note, however, that the contents of the cell are not necessarily destroyed – this is discussed further in section 3.2.) A cell defined in the context of a behavior is called a *variable*.

---

<sup>6</sup> It is interesting to note that value literals, such as “1”, can be viewed as references to an abstract entity representing the concept of “one”.

- *Objects* are simply cells that are typed by a UML class. This means that they may have associated behaviors and behavioral features. An object may have one or more *slots*, which are simply cells owned<sup>7</sup> by the object. Each slot corresponds to an attribute or part<sup>8</sup> of the class of the object. Active objects will also have cells for buffering messages that have arrived but which have not been processed yet by a receive action.
- *Links* are dynamic structural entities that represent ordered collections (tuples) of object references. Like objects, links are typed, except that the type of a link is an association. UML distinguishes between two kinds of links. *Value links* are immutable entities whose identities are defined by their contents. Consequently, the existence of a value link is dependent on the existence of the objects whose references it contains. *Object links*, on the other hand have an identity that is independent of their contents. Thus, the references contained by an object link can change over time but the link retains its identity. If the type of the object link is an association class, then the link may also have additional slots corresponding to the attributes of the class.
- *Messages* are dynamic passive objects that are created to convey information across links as a result of the execution of communication actions.

UML models may contain representations of structural entities, which serve to capture the state of these entities during some real or imagined instant or time interval. The metamodel for this is shown in Fig. 3. The metaclass InstanceSpecification is a general mechanism for representing instances of various kinds of classifiers, including objects (but also instances of more abstract kinds of classifiers such as collaborations). The structural features of the classifier (attributes, parts, link ends, etc.) are represented by slots and the values contained in the slots by value specifications.

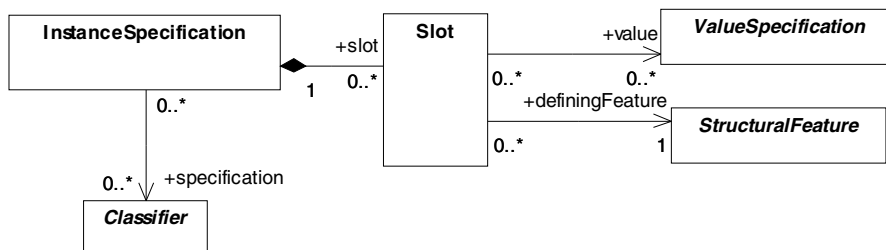


Fig. 3. UML metamodel concepts used for modeling run-time instances and their values

### 3.2 The Composites Sub-layer

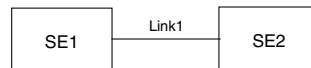
There are several ways in which structural entities combine with each other to produce composites. When considering composites, an important aspect is the way in which the various components of a composite depend on each other:

<sup>7</sup> Note that “ownership” is synonymous with containment in this case.

<sup>8</sup> In the UML metamodel, attributes and parts (of structured classes) are modeled slightly differently. However, this is primarily due to technicalities of the way that the metamodel is constructed and does not reflect any fundamental semantic differences.

- *Existence dependencies* are dependencies in which the existence (or non-existence) of one structural entity depends on the existence (or non-existence) of a different entity. In such relationships a dependent entity will be terminated if the structural entity that it depends on is terminated<sup>9</sup>. This is usually an asymmetric relationship since the inverse is generally not the case; that is, an entity may exist independently of its existence dependents.
- *Functional (behavioral) dependencies* are dependencies in which the successful functioning of one structural entity depends on the successful functioning of other functional entities.

Perhaps the most elementary combination of structural elements is the *peer* relationship (Fig. 4). This involves two or more entities communicating with each other over a link<sup>10</sup>. In this case, there may be behavioral dependencies between the communicating entities but there are no structural existence dependencies unless the link is a value link<sup>11</sup>.



**Fig. 4.** The “peer” composite relationship: in this case the existence of the entities at the ends of the link are independent of each other but there are usually functional dependencies between them

A second kind of fundamental structural relationship is *containment* (Fig. 5), which comes in two basic variants:

- *Composition*, in which there is an existence dependency between the containing element and the contained element and
- *Aggregation*, in which the contained element has no existence dependency relative to the container<sup>12</sup>.

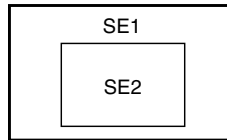
To understand the difference between the two forms of containment, recall that objects are simply cells that may contain other cells. The distinction between composition and aggregation is whether the object contained in a cell is destroyed along with its containing cell or not. In the case of composition, the object is destroyed when the cell is destroyed, whereas in the case of simple aggregation, the object that was contained in a cell that was destroyed (because its containing cell is destroyed) remains in existence.

<sup>9</sup> Note that there are lots of subtleties involved in this seemingly simple notion. For example, do the depending on and dependent entities disappear at the same instant or is there some interval of time involved? Is the dependent entity created at the same time as the entity that it depends on or not? These are semantic variation points that may be defined differently in different domains or situations.

<sup>10</sup> Not all links are necessarily used for communications. However, the distinction between communication and other types of links is not considered in this paper.

<sup>11</sup> Recall that the existence of value links depends on the existence of their ends. If one of the ends disappears, the link no longer exists.

<sup>12</sup> The two forms of containment are differentiated in the UML notation by the difference between the “full diamond” and the “empty diamond” composition notations.



**Fig. 5.** The composition relationship: the structural entity SE2 is contained within structural entity SE1 and may have an existence dependency on it

The most common use of containment is to encapsulate elements that are part of the hidden implementation structure of the container. (This implies that the container has a functional dependency on the contained element.) However, a container does not necessarily own all of the parts that it uses in its implementation. In case of aggregation, the part is merely “borrowed” from its true owner for the duration of the container. For example, a container might ask a resource manager, such as a buffer pool manager, to “lend” it a set of memory buffers that it needs to do perform its function. Once the task has been completed, the container may return the resource to its owner. The “borrowed” part is also returned automatically to the owner when the container itself is destroyed.

In addition to these two fundamental structural patterns, it has been suggested [11] that a third basic structural pattern exists: *layering*. This is a pattern in which the upper layer has existence and functional dependencies on the lower layer (similar to containment), but the lower layer has no dependencies of any kind on the upper layer. However, at present, there is no direct support for layering in UML, although it can be constructed from a combination of more primitive patterns.

#### 4 The Behavioral Base

In speaking of behavior in UML, care should be taken to distinguish between two subtly different uses of the term. In one case, “behavior” denotes the general intuitive notion of how some system or entity changes state over time. Alternatively, it refers to the specific UML concept shown in the metamodel fragment in Fig. 6<sup>13</sup>. To distinguish these two, the capitalized form of the word will be used when referring specifically to the UML concept (i.e., “Behavior” versus “behavior”).

In UML, Behavior is an abstract concept that represents a model element that contains a specification of the behavior of some classifier. At present, UML supports four different concrete specializations of this abstract concept: interactions, activities, and state machines (see Fig. 1). Each of them has its specific semantics as well as its own graphical notation and dedicated diagram type.

The specific semantics of these higher-level behavioral formalisms are outside the scope of this paper. However, it should be noted that, for historical reasons, not all of them fully exploit the available capabilities of the shared base. For instance, the base provides common mechanisms for describing the flow of behavior, including concepts for things such as decision points, iteration, and the like, yet these same types of constructs are also defined in state machines. This is because a large part of the shared behavioral base was introduced at a later stage of UML evolution. Consequently, some work still remains to fully integrate these formalisms on the new shared base.

<sup>13</sup> Only the case of class classifiers is considered in this section.

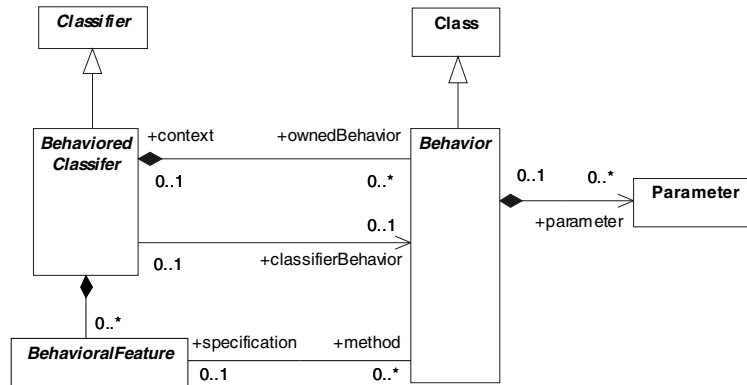


Fig. 6. The relationship between structure and behavior in the UML metamodel

The general Behavior concept captures what is common across all these forms: that they can be attached to a classifier (for a variety of different reasons) and that they may have input and output parameters.

Note that Behavior is a specialization of the Class concept, which means that when behaviors start executing they take on a structural form as kinds of objects. This may seem strange at first, but recall that a behavior may need local variables to keep temporary information that is used to pass information between different actions as well as keeping a record of its execution (e.g., a data and call stack, one or more program counters). The precise form of this execution record is yet another semantic variation point that depends on the specific behavioral form and implementation.

Fig. 6 shows that a classifier can own multiple behavioral specifications. The UML standard does not prescribe what purposes these behaviors serve. For example, a structured class may have behavior specifications that illustrate specific scenarios of interest associated with that class, such as a start-up scenario. However, at most one of those behavior specifications can be used to represent the behavior that starts executing when an object of that class is created and started<sup>14</sup>. This behavior is called the *classifier behavior*. For a passive class, this might be the constructor of the class. For active objects, it is the behavior that initiates the activity of the object and which continues until the object is terminated.

It is important to distinguish the classifier behavior from the overall behavior resulting from the combination of the classifier behavior and the behaviors of its internal part objects.

The model of behaviors in Fig.6 also identifies two other semantically relevant concepts:

- *Behavioral feature* is a *declaration* of a behavior that may be activated by other objects through invocations. Two kinds of behavioral features are defined in UML: *operations* and *receptions*. Operations are invoked through call actions whereas receptions are invoked through asynchronous signal sends (these are discussed in the following section).

<sup>14</sup> Note that creation and start of execution are not necessarily coincident. In some systems, a “start” of behavior may require an explicit action, while in others it may be automatic. This is a semantic variation point.

- *Methods* are Behaviors that realize behavioral features. Like classifier behaviors, methods can be specified using any concrete behavioral formalism deemed suitable by the modeler, including a simple action-based specification.

#### 4.1 The Inter-object Behavioral Base

This part of the Behavioral Base deals with communications. UML includes support for both synchronous and asynchronous interactions. Synchronous communications cause the invoking entity to suspend execution of its behavior until a reply is received. An asynchronous communication does not result in a suspension of the invoker who simply continues with execution of its behavior. UML defines several variations on these two basic forms:

- *Synchronous operation call* is an invocation of an operation of the target object. The invoking object is suspended until a reply is received from the target object.
- *Synchronous behavior call* is an invocation of the classifier behavior of the target object. The invoking object is suspended until a reply is received from the target object.
- *Asynchronous operation call* is a call to an operation of the target object. The invoker is not suspended but simply carries on executing its behavior. If there is a reply sent back from the operation, it is ignored.
- *Asynchronous behavior call* is like an asynchronous operation call except that the classifier behavior of the target object is invoked rather than a behavioral feature.
- *Asynchronous signal send* creates a signal with the appropriate parameters, which is then sent to the target object in a message. The target object has to have a corresponding *reception* defined that corresponds to that signal. The invoker is not suspended but simply carries on executing its behavior.
- *Asynchronous signal broadcast* is an asynchronous send with no explicit target objects specified. That is, the message is broadcast into a communications medium, such as a link, where any objects connected to the same medium can pick it up.

Regardless of the precise nature of the call, the underlying transmission mechanism is the same: a message object is created on the sending side that includes information about the invoked behavioral feature or behavior as well as information about the identity of the invoker. The message includes the argument values corresponding to the parameters of the invoked behavior (in case of signals, this information is part of the signal definition). Note that the details of whether the values are copied into the message or whether merely a reference to them is stored in the message are a semantic variation point. This means that both copy and reference semantics can be modeled.

Similarly, any reply information going in the reverse direction is packaged in a message and sent onwards by the invoked object. The semantics and characteristics of the transmission and medium and delivery process (reliability, performance, priorities, etc.) are not defined in standard UML, allowing different specializations according to the domain and circumstances.

UML provides specific actions for each of the above communication types as well as corresponding receive (accept) primitives. This allows modeling of variations of these basic primitives as well as construction of more sophisticated communication models.

In addition to the various modes of communication, the inter-object behavior base defines the following two semantically significant concepts:

- *Events and event occurrences.* An event is a specification of a kind of state<sup>15</sup> change. An occurrence of such a state change at run time is an event occurrence. Event occurrences are instantaneous. That is, a state change is assumed to occur at a particular instant rather than taking place over an interval of time. An event can be any kind of state change, such as the start of execution of an action, the sending of a signal, or the writing of a value to a variable. However, four special categories of state changes are particularly distinguished in the UML semantics since they are generally useful:
  - *Call events* represent the reception of a synchronous call by an object.
  - *Signal events* capture the receipt of a specific type of asynchronous signal by an object
  - *Change events* represent the change of value in a Boolean expression
  - *Time events* represent situations where a certain instant of time has arrived
- *Trigger* is a kind of event whose occurrence causes the execution of behavior, such as the trigger of a transition in a state machine. Any of the four kinds of events listed above can act as a trigger. However, not all events are triggers. For example, the sending of a signal does not trigger any behavior execution (although the receipt of that signal by the target does).

## 4.2 Intra-object Behavioral Base

This part of the Behavioral Base defines the essentials for representing actions and for combining them. The key notion here is that of a basic form of Behavior called an *activity*<sup>16</sup>. In essence, this is a generalization of the concept of procedure found in most common programming languages. Specifically, an activity is a kind of Behavior that may contain a set of *actions* and local variables.

**The Actions Model.** Actions are used to specify fine-grained behavior in UML, the kind of behavior corresponding to the executable instructions in traditional programming languages. This includes actions used to initiate and receive communications (described in section 4.1), create and destroy objects, read and write cells of various kinds, and so on. Note that, even though actions specify behaviors, they are not Behaviors in the UML sense.

The metamodel fragment shown in Fig. 7 depicts the general model of actions as defined in UML.

In essence, an action is a value transformer that transforms a set of inputs into a set of outputs as illustrated by the conceptual model in Fig. 8. Some actions may affect the state of the system, that is, they may change the values that are stored in the structural elements of the system (variables, object slots, links, etc.). The inputs and outputs of an action are specified by the set of *input pins* and *output pins* respectively.

<sup>15</sup> The term “state” here is used in its generic sense to describe some run-time situation existing at a particular instant and does not necessarily imply a formalism based on state machines.

<sup>16</sup> A more elaborate form of this same concept is used for the higher-level activity behavioral formalism.

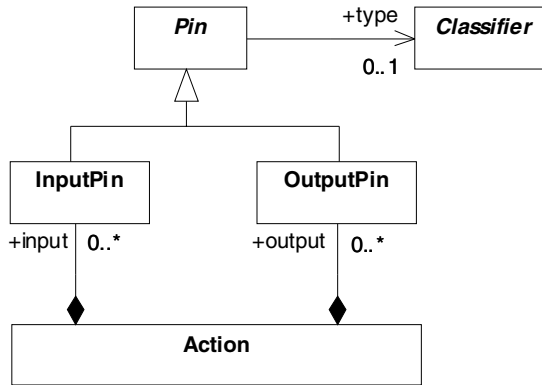


Fig. 7. The UML showing the relationship between actions and pins

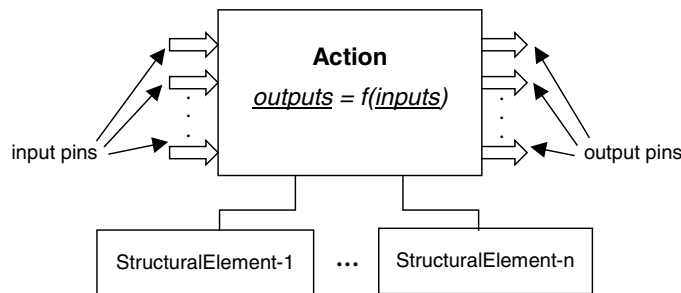


Fig. 8. The conceptual model of actions in UML: in general, an action transforms a set of inputs to a set of outputs; it may also read and write the values stored in one or more structural elements (variables, attributes, etc.)

Both input and output pins are typed elements, which means that only values of the appropriate type (including compatible subtypes) can appear on them.

An action can only execute if all of its input pins are enabled, that is, if all the values on all of the input pins are available. The point at which an action makes its outputs available is another semantic variation point. For example, in synchronous systems such as Esterel [1], all the outputs become available simultaneously. Other models might allow output values to be generated as soon as they become available and independently of whether the values on other output pins are available.

The run-time effect of an action can be described in terms of the difference in the state of the system from the instant just before the action is executed (pre-condition) to the instant just after execution completes (post-condition).

UML makes no assumptions on the duration of actions. Therefore, semantic models that require zero-time (i.e., instantaneous) execution as well as models that allow for finite execution times for actions are represented.

**Flows.** An action performs a relatively primitive state transformation and it is usually necessary to combine a number of such transforms to get an overall desired effect. This requires a means of combining actions and their effects to produce more

complex transformations. UML defines the concept of *flows* for this purpose. Flows serve to specify the order of execution of combined actions as well as the conditions under which those executions can take place. To provide maximum flexibility, UML supports two different ways of composing actions using either *control flows* and *object (data) flows*. These two correspond to two radically different models of computation. However, regardless of the type of flow used that the relationship between actions and flows can be conveniently modeled by a directed graph, with flows represented by the arcs (edges) of the graph.

Control flows are the most conventional way of combining actions. A control flow between two actions means that the execution of the action at the target end of the flow will start executing after the action at the source end of the control flow has completed its execution.

In some cases, it may be necessary to alter the execution flow in some special way (loops, conditional branches, etc.). This is achieved using special *control nodes*. These include standard control constructs such as forks and joins. In addition control flow can be modified through *guards* (specified by Boolean predicates) on flows which are used for conditional transfer of control. The different control nodes are discussed below.

Data flows, on the other hand, are used for computational models that are based on a fine-grained parallelism. In this case, an action executes as soon as all of its input pin values are available. This means that multiple actions may be executing concurrently, since an action may produce outputs before it has completed its execution and also because a given output can be fed to inputs on multiple different actions. Note that data flows do not connect actions like control flows; instead, they connect input and output pins on actions. Consequently, data-flow dependencies tend to be finer-grained than control flows, as some actions have multiple inputs and outputs.

The two types of flows can be combined for sophisticated control over execution. An example of this can be found in the higher-level activities behavioral formalism.

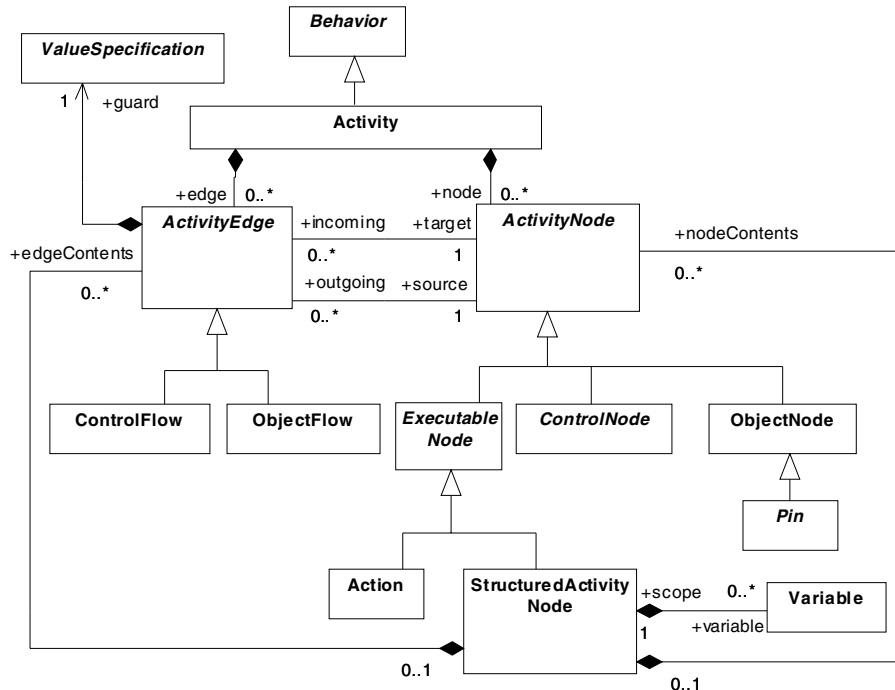
This flow-based model of computation behind actions was inspired by the Petri net formalisms [9] although no specific formal variant of Petri nets was used as its conceptual base. The difference between control and data flows is primarily in the nature of the tokens (control or data) that flow through them.

In the UML metamodel, data and control flows are represented by *activity edges*, which can be either control flow edges (arcs) or object (data) flow edges as shown in Fig. 9.

**Nodes.** In the UML metamodel, the various types of nodes occurring in activity graphs are represented as special kinds of *activity nodes* (see Fig. 9). *Control nodes* are used to support of control-flow execution (see previous section), *executable nodes* contain behavior that needs to be executed (including actions), and *object nodes* represent values of some kind.

The following are the principal types of control nodes defined in UML:

- *Initial* nodes are used to initiate a flow when an activity starts executing; in effect, they and their outgoing flow identify which executable nodes will be executed first.
- *Flow final* nodes terminate the flows that terminate on them, but not necessarily the containing activity.



**Fig. 9.** A simplified version of the UML metamodel that shows the relationships between the key concepts in the Intra-object Behavioral Base: activities, actions, nodes, and flows

- *Final* nodes terminate the activity that contains them and, consequently, all of the active flows in that activity.
- *Decision* nodes are used as the source of a set of alternative flows, each guarded by a separate Boolean expression.
- *Merge* nodes are used to converge multiple incoming flows into a single outgoing flow. No synchronization takes place.
- *Fork* nodes are used to split a flow into multiple concurrent flows.
- *Join* nodes represent synchronization points, where flows come together but do not proceed until all of them arrive<sup>17</sup>.

Executable nodes represent nodes that contain actions. A special kind of executable node is a *structured activity node*, which roughly corresponds to the concept of a block in a block-structured programming language. A structured activity node may contain other activity nodes and edges as well as local *variables*. This recursive definition allows the construction of arbitrarily complex hierarchies of nested activity nodes. Specializations of this general concept include node types for explicit modeling alternative and iterative (looping) flows.

<sup>17</sup> There are numerous subtleties to the synchronization policies that depend on the nature of the flows, but these are beyond the scope of this paper.

Object nodes are used to specify the values that are transferred through an object (data) flow. Their most obvious use is to model pins (note that Pin is a kind of ObjectNode). That is, the flow of data between individual actions is specified by object flows that connect the output pins of a predecessor action to the input pins of its successor actions.

### 4.3 Actions

The topmost level of the Behavioral Base is a set of actions that perform different kinds of transformations as well as reading and writing structural entities such as variables and attributes. Actions can be packaged into activities that can then be invoked by any of the higher-level behavioral formalisms. For example, a state machine might use such an activity to describe the fine-grain behavior that occurs in a transition or in an entry action to a state.

The following categories of actions are defined in the Behavioral Base:

- *Communication actions* are actions used for interactions between objects. Their semantics are described in section 4.1 above.
- *Read-Write actions* are actions that access the values of structural entities (read actions) and, in some cases, change these values (write actions). This includes accessing attributes, links, and variables.
- *Structural actions* create and destroy structural elements such as objects, attributes, links, etc. (but not values). In addition, these include actions that dynamically change the classification (type) of structural entities.
- *Computation actions* specify actions that perform standard computations using arithmetic and logical operators, as well as more complex functional transforms.
- *Special actions* include actions for modeling exceptions and for starting the classifier behavior of objects.

In addition to the above action categories, the UML standard defines a set of actions for reading time values. However, the model of time implied in these actions is highly specialized, since it presumes an ideal global clock, such that the value of time is the same in all parts of the system. This makes it unsuitable for modeling many distributed systems or time-sensitive systems, in which the characteristics of individual clocks and phenomena such as time synchronization need to be taken into account. Hence, it is not considered as part of the general semantics foundation.

The details of the semantics of individual actions in these categories are outside the scope of this paper.

Note that some of the higher-level modeling formalisms provide additional actions for describing fine-grained functionality specific to their needs. Such actions are not part of the general semantics framework and are, therefore, not discussed further.

## 5 Mapping to the UML 2.0 Specification

The UML 2.0 specification is structured according to subject area. Each subject area has a dedicated chapter, which includes the relevant metamodel fragments and a set of

concepts defined for that subject area. Within a subject area, there is typically a short informal introduction followed by an alphabetically sorted list of concept descriptions. Each concept description includes a “semantics” section that describes the meaning of that modeling concept. Although this is a convenient organization for reference purposes, it makes it very difficult to deduce how the various semantics fit together.

One of the intentions of this paper was to collect in one place the information from the various dispersed fragments that deal with run-time semantics<sup>18</sup> and to provide the necessary system view. However, this paper is limited to a high-level description and anyone interested in understanding the finer details of these semantics will necessarily have to refer to the specification itself. To assist in this, this section explains where the various semantic elements described in this paper are located in the standard itself.

Note that all references to chapters and section numbers are with respect to the so-called Final Adopted Specification (FAS, for short)[8]. Unfortunately, this means that some of this information will be out of date as the standard and its corresponding specification evolve. (In fact, even as this text is being written, the FAS is being modified as part of the standard OMG standardization process. For instance, there have been some terminology changes related to the notion of events and triggers (these are reflected in the current text).)

Unfortunately, the overall semantics architecture shown in Fig. 1 and most of the terminology used to describe it do not appear anywhere in the specification. They have to be inferred from the semantic dependencies between different metamodel elements. Therefore, the diagram in Fig. 1 represents a synthesis of the relevant information contained in the various text fragments. In this, the author relied extensively on his familiarity with the often undocumented design intent behind many of the UML concepts, since he participated in the actual definition and writing of the standard<sup>19</sup>.

The general causality model is described in the introductory part of chapter 13 (CommonBehaviors) and also, in part, in the introduction to chapter 14 (Interactions) and the section on Interaction (14.3.7) and Message (14.3.14).

The structural foundations are mostly covered in two chapters. The elementary level is mostly covered in chapter 7, where the root concepts of UML are specified. In particular, the sections on InstanceSpecifications (section 7.7 of the FAS), Classes and Associations (section 7.11), and Features (section 7.9). The composites level is described primarily in chapter 9 (Composite Structures), with most of the information related to semantics contained in sections 9.3.12 (Property concept) and 9.3.13 (StructuredClassifier). In addition, the introduction to this chapter contains a high-level view of some aspects of composite structures.

The relationship between structure and behavior and the general properties of the Behavior concept, which are at the core of the Behavioral Base are described in CommonBehaviors (in the introduction to chapter 13 and in section 13.3.3 in particular).

---

<sup>18</sup> Not all modeling concepts in UML correspond to run-time concepts, hence, not all semantics descriptions deal with run-time semantics.

<sup>19</sup> In this regard, the UML specification follows the style used in many modern technical standards, which strive for a formality that only permits the inclusion of supposedly objective “facts”. Unfortunately, this often leads to the omission of invaluable information related to design rationale.

Inter-object behavior is covered in three separate chapters. The basic semantics of communications actions are described in the introduction to chapter 11 (Actions) and, in more detail, in the sections describing the specific actions (sections 11.3.1, 11.3.2, 11.3.6, 11.3.7, 11.3.8, 11.3.9, 11.3.18, 11.3.37, 11.3.38, 11.3.39). The concepts related to messages are defined in the Interactions chapter (sections 14.3.14 and 14.3.15), while the concepts of events and triggers are defined in the Communications package of CommonBehaviors (chapter 13). Event occurrences are defined in section 14.3.3 of the Interactions chapter.

The basic notion of actions, in the intra-object behavior base, is defined in the introduction to the Activities chapter (chapter 12) and in the section on the Action concept itself (12.3.1). The semantics of flows and nodes mechanisms are also described in the Activities chapter (sections 12.3.2, 12.3.3, 12.3.4, 12.3.6, 12.3.12, 12.3.13, 12.3.15, 12.3.17, 12.3.21, 12.3.22, 12.3.23, 12.3.24, 12.3.25, 12.3.27, 12.3.29, 12.3.32, 12.3.34, 12.3.37, 12.3.40).

The various shared actions and their semantics are described in chapter 11.

Finally, the higher-level behavioral formalisms are each described in their own chapters: Activities in chapter 12, Interactions in chapter 14, and State Machines in chapter 15.

One of the consequences of such a dispersed organization of semantics data is that there are bound to be some logical inconsistencies and omissions, given the complexity and informal nature of the specification. Some of these flaws have already been identified and are being fixed as part of the standard finalization process that is currently ongoing. (The FAS is not the final form of the standard; instead, it is published so that implementers and researchers can inspect it and provide feedback.) Undoubtedly, if a formal model of the semantics existed, it would be much easier to detect such flaws. However, as explained earlier, it is the author's opinion that it would not have been appropriate to start with a given mathematical formalism until the broad scope of design intent was described informally. This has now been done with the official UML 2.0 specification and the time may be ripe to fit an appropriate formal model over it.

## 6 Summary

This paper provides a high-level view of the run-time semantics foundation of standard UML, that is, the semantics that are actually described in the official OMG standard. The purpose is twofold: to dispel persistent and unjustified insinuations that "UML has no semantics" and to provide a convenient starting point for those interested in doing research on the topic of UML semantics.

As noted at the beginning, it is probably inappropriate to define a single concrete formalization of the semantics of UML, since it was intended to be used in a variety of different ways and for a variety of different domains. However, at this point, with the scope of UML defined by the UML 2.0 standard, it seems both feasible and highly desirable to define a general formal semantics of UML. Such a model would serve to more precisely define the *envelope* of possible concrete formalizations. This could then be used as a basis for specializations of that formal model for specific domains and purposes. Also, it could be used to validate alternative formalizations.

## References

1. Berry, G.: The Foundations of Esterel, In: G. Plotkin, C. Stirling and M. Tofte (eds.) *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, Cambridge MA (1998)
2. Kleppe, A., Warmer, J.: *Unification of Static and Dynamic Semantics of UML: A Study in redefining the Semantics of the UML using pUML Meta Modelling Approach*, Klasse Objecten, <http://www.cs.york.ac.uk/puml/mmf/KleppeWarmer.pdf>, Soest, Netherlands (2003)
3. Ober, I.: *Harmonizing Design Languages with Object-Oriented Extensions and an Executable Semantics*, Ph.D Thesis at Institut National Polytechnique de Toulouse, Toulouse, France (2004)
4. Object Management Group: *MDA Guide (version 1.0.1)*, OMG document ad/03-06-01, <http://www.omg.org/cgi-bin/doc?mda-guide> (2003)
5. Object Management Group: *UML for Systems Engineering – Request for Proposal*, OMG document ad/03-03-41, <http://www.omg.org/docs/ad/03-03-41.pdf> (2003)
6. Object Management Group: *UML 2.0 Infrastructure – Final Adopted Specification*, OMG document ad/03-09-15, <http://www.omg.org/docs/ad/03-09-15.pdf> (2003)
7. Object Management Group: *UML 2.0 OCL – Final Adopted Specification*, OMG document ad/03-10-14, <http://www.omg.org/docs/ad/03-10-14.pdf> (2003)
8. Object Management Group: *UML 2.0 Superstructure – Final Adopted Specification*, OMG document ad/03-08-02, <http://www.omg.org/docs/ad/03-08-02.pdf> (2003)
9. Peterson, J.: *Petri Nets*, *ACM Computing Surveys*, Vol. 9, Issue 3, ACM Press, New York, (1977) 223-252
10. Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual (2<sup>nd</sup> ed.)*, Addison-Wesley, Boston (2004)
11. Selic, B., Gullekson, G., Ward, P.: *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York (1994).
12. Sourrouille, J.L., Caplat, G.: *Constraint Checking in UML Modeling*, in *Proceedings International Conference SEKE'02, ACM-SIGSOFT*, (2002) 217-224