Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets © Jot, 2011
Online at http://www.jot.fm

Ontological Behavior Modeling

Conrad Bock[a] James Odell[b]

a. U.S. National Institute of Standards and Technology

b. Computer Sciences Corporation

Abstract This article gives an example of improving the effectiveness of behavior modeling languages using ontological techniques. The techniques are applied to behaviors in the Unified Modeling Language (UML), using the logical meanings for classification introduced in UML 2. The article suggests unifying UML's three kinds of behavior languages around the abstract syntax and semantics of composite structure, UML's model for capturing interconnection of parts of classes. This significantly simplifies the UML metamodel, provides a formal semantics to clarify ambiguities in the current informal semantics, and increases the expressiveness of UML behaviors.

Keywords Behavior, Ontology, UML

1 Introduction

Language standards are more effective when people understand them the same way. This requires that everyone see the same real-world implications of things said in the languages (in "sentences"). For example, if we all understand "The dog chases the cat" the same way, it is because we have similar ideas about what constitutes real dogs, cats, and chasing. The relationship between sentences and real world interpretations of them is *semantics*. The relationship between a language and its sentences is *syntax*, which determines the sentences allowed in the language [Genesereth87].

Formal approaches to semantics aid language understanding by specifying how to properly interpret sentences in terms of real world things. This enables them to cover the many uses of a language more completely, avoiding piecemeal approaches of examples and manually defined compliance tests, as well as inadvertent standardization introduced in reference implementations. A disadvantage of formal

C. Bock, J. Odell: "Ontological Behavior Modeling," Journal of Object Technology, vol. 10, 2011, pages 3:1-36. Available at doi:10.5381/jot.2011.10.1.a3

approaches is they often are difficult to learn, and time consuming to apply, though some of the time is paid back as language definers are forced to communicate their intent more clearly, uncovering problems early in the language development cycle where they can be more easily addressed. One of the primary barriers to formal approaches to semantics is lack of integration with commonly used techniques, and the need to use them wholesale rather than gradually.

This article suggests ontological approaches to formalizing language standards as a practical way to gain the benefits of formality while reducing its cost. Ontological techniques use commonly understood notions of categories and conditions of membership to increase formality while preserving accessibility (this is sense of "ontology" in description logics [Borgida07]). They focus attention on the potential members of categories, facilitating communication based on concrete examples, and supporting more complex membership conditions as necessary. These techniques enable languages to be built up from smaller, simply defined elements to larger ones, by creating new categories with membership conditions based on existing simpler categories, in thin enough layers that languages are more easily understood in a uniform way. Ontologically defined languages can be translated to more informal or formal languages as desired. They can determine some properties of behaviors without reference implementations or compliance tests, but make these easier to develop, because behaviors are more clearly defined. Ontological approaches also facilitate consolidation of common practice and communication by providing simpler, more widely understandable basic notions, gradually combining them into more sophisticated ones.

Ontological approaches are already used in some mainstream organizations, such as the Object Management Group (OMG), which introduced logical meanings for classification in a major revision of Unified Modeling Language (UML 2) [OMG10a], and adopted a standard UML extension and mapping for the Web Ontology Language (OWL) [W3C09][OMG09] (OWL is a standard for interchanging a particular kind of description logic [Horrocks06]). These efforts and others showed that much of UML class semantics is the same as OWL's [Berardi05], and that it is suitable as an ontological language generally [Guizzardi04]. This view of UML classes treats them as categories with conditions for membership, as they are in OWL, rather than as encapsulated code or instance factories. It fulfills one of UML's original purposes in modeling systems without committing to whether the behaviors of the system are performed manually, enacted by a combination of workflow systems and people, or executed automatically by software or hardware [Cocks04]. OMG uses the ontological capabilities of UML 2 to define languages by specializing and extending others, either through metamodels or metamodel profiles [OMG10b].

This article continues the work of applying ontology to UML, focusing on the abstract syntax and semantics of behaviors in the UML to strengthen their common basis [OMG10a]. UML has three ways to specify behaviors, each emphasizing different aspects:

- Activities for inputs and outputs between actions, and their time ordering.
- State machines for reacting to notification of external events.
- Interactions for messages between objects.

UML has some common abstract syntax and semantics for these three specialized behavior languages, but it is a small portion of them and does not provide a way to integrate them. This article expands the commonality between UML's three specialized behavior languages and provides linkages between them.

Some ontological techniques are currently used in UML behaviors, but not completely applied. UML currently supports behavior as ontological classes and instances. The article builds on this by specializing behavior from composite structure, UML's model for interconnecting of parts of classes, where parts of behaviors are actions in Activities, states in State Machines, and messages in Interactions. It adds a simple temporal model to link parts of behavior together. This article sketches how the framework is specialized for the most salient aspects of UML's three behavior languages. It does not suggest any changes in UML notation (it uses composite structure notation for user model examples, but these are not proposed as a concrete syntax for behaviors).

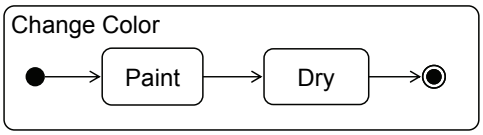
Section 2 defines the notion of behavior semantics in general, drawn from conventional mathematical definitions rather than UML. Section 3 covers how UML behavior semantics is defined currently. Section 4 applies ontological techniques to behavior modeling, in part through UML composite structure. Section 5 outlines related and future work. Section 6 summarizes the article. Sections 3 and later assume familiarity with the UML metamodel and notation [OMG10a], especially its composition model [Bock04], and OMG's metalevel architecture [OMG10c].

2 Behavior Semantics in General

The real-world implications of anything said in behavior languages are what occurs when behaviors actually happen. For example, a factory operation for changing the color of an object happens many times every day, at many factories, each involving a different object, different colors, and so on. Each time the behavior happens is a separate *behavior occurrence*, usually at different times, involving different objects and colors, and at different factories. Occurrences might be

performed manually, enacted by a combination of workflow systems and people, or executed automatically by software or hardware.

The semantics of behavior languages specify which occurrences "follow" the behaviors expressed in those languages, in the sense that occurrences are supposed to obey models of behaviors. The occurrences following the three behaviors in Figure 1 happen to be the same, but this is only clear from the semantics of the languages, rather than the syntax in Figure 1. For example, the figure does not say the arrows, punctuation, and symbol placement imply painting must complete before drying starts, even though many explanations of these languages assume it is understood. The semantics is more apparent from the occurrences in Figure 2. This shows behaviors on the vertical axis, time on the horizontal, and occurrences as interval bars on the graph. The dashed oval contains two groups of occurrences following the behavior in Figure 1, assuming that painting is supposed to complete before drying starts. The group outside the oval on the right does not follow the behavior. Figure 2 has only a few example occurrences, rather than a complete semantics. Complete semantics of the behavior languages used in Figure 1 provide general rules to determine whether an occurrence follows the particular process specifications in Figure 1.




```
<process name="Change Color">
    <sequence>
      <invoke operation="Paint"></invoke>
      <invoke operation="Dry"></invoke>
    </sequence>
</process>

void ChangeColor
{ Paint();
  Dry();
}
```

Figure 1: Behavior Notation Examples

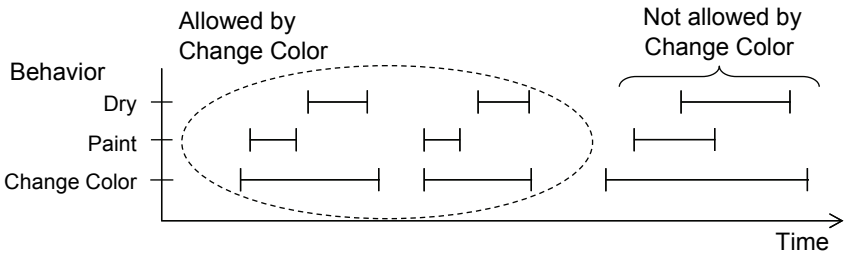


Figure 2: Behavior Occurrences

The same occurrence can follow multiple, partially specified behaviors, as illustrated in Figure 3 for the two behaviors at the top. The Change Color #1 definition on the left specifies that painting happens before drying. The Change Color #2 definition on the right specifies that spray painting happens before cleanup. The group of occurrences on the lower left only follows Change Color #1, because it dries, and brush painting is a kind of painting, but it does not clean up, and brush painting is not spray painting. The occurrences on the right only follow Change Color #2, because they do not dry. The occurrences in the middle follow both behaviors, because cleaning up and drying both occur after painting, assuming the arrows in the behavior language have temporal precedence semantics rather than imperative, and spray painting is a kind of painting.

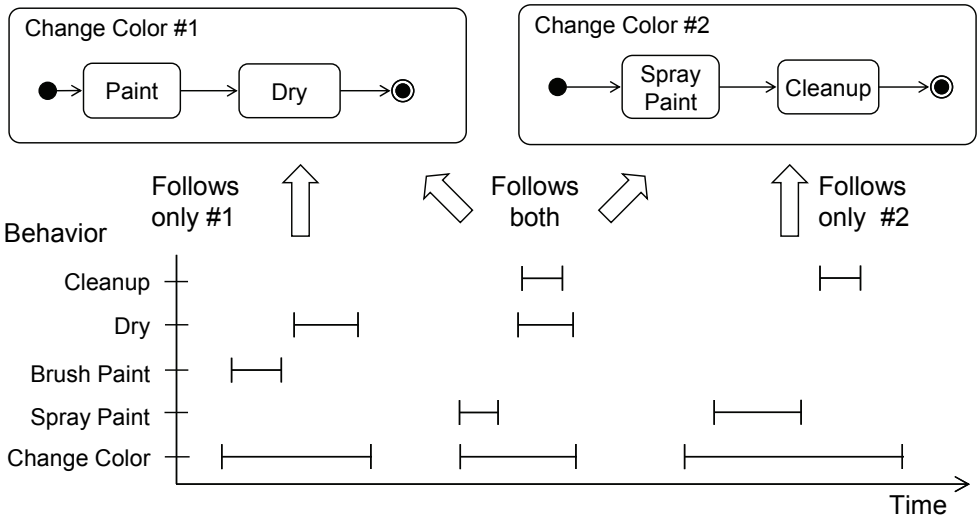


Figure 3: Overlapping Behaviors

Behavior languages can include elements that require occurrences following one behavior to also follow another (*generalization*). Figure 4 shows one behavior generalizing another, using the UML notation for generalization (see Section 3). This means any occurrence following the more special behavior Change Color #3 also follows the general behavior Change Color #1. It is not possible for occurrences to follow only the specialized behavior and not the general one. Figure 5 shows two behaviors generalizing a third one. Occurrences following Change Color #4 also follow Change Color #1 and Change Color #2, but it is possible to have occurrences of the general behaviors separately. Figure 6 shows behaviors that cannot have common occurrences, they are inconsistent. The Change Color #5 behavior allows only drying right after painting, using an operational or imperative semantics, while Change Color #6 allows only shipping. No occurrences can follow both of these behaviors.

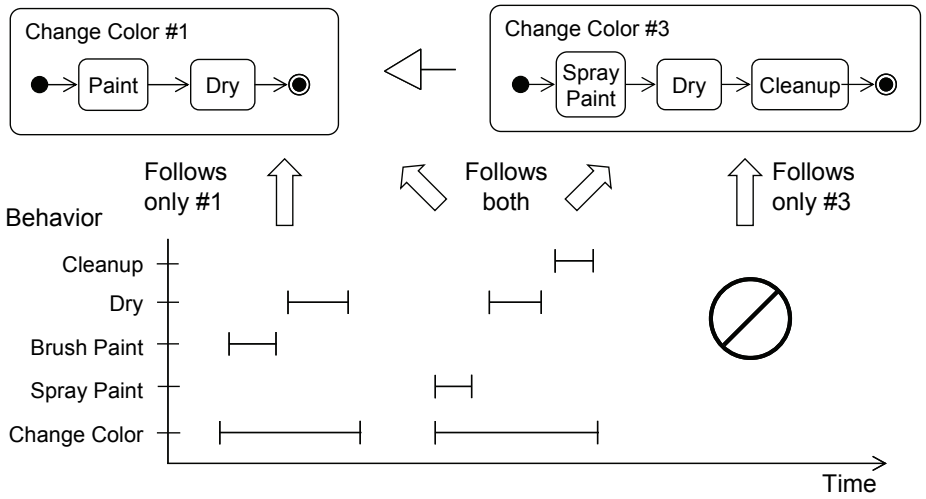


Figure 4: Behavior Generalization

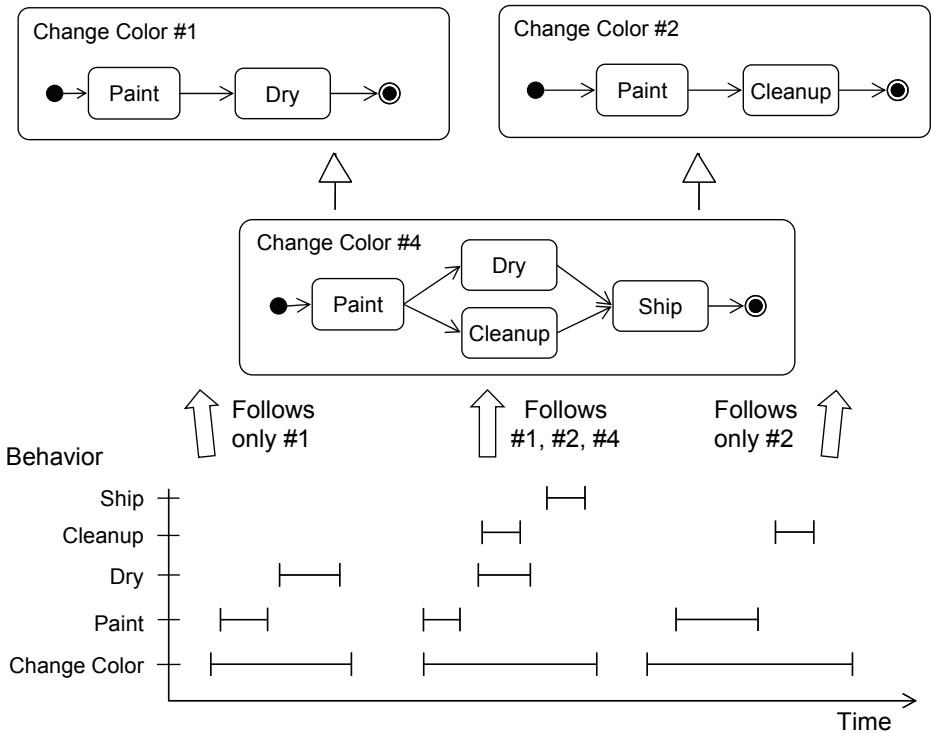


Figure 5: Multiple Behavior Generation

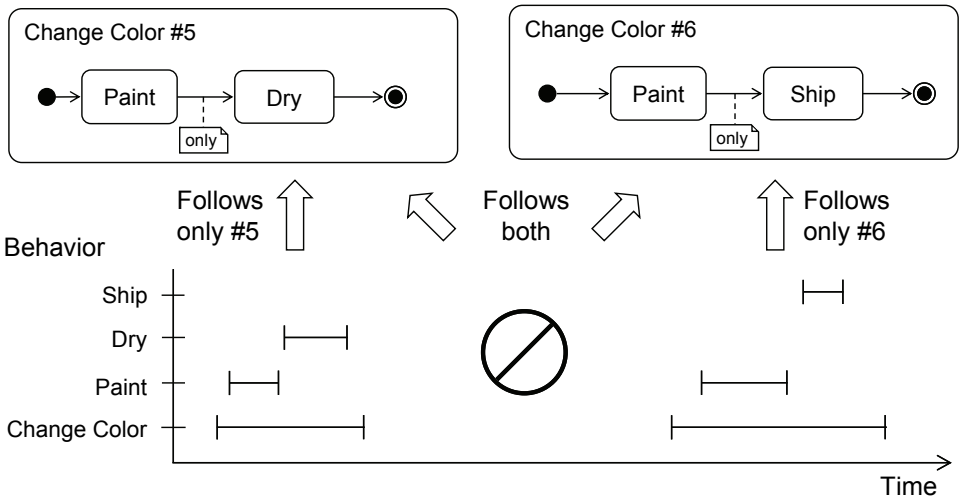


Figure 6: Inconsistent Behaviors

Venn diagrams are another way to visualize the examples so far, as illustrated in Figure 7 and Figure 8. The dots are occurrences and the ovals are the behaviors from Figure 3 through Figure 6. A dot inside an oval is an occurrence following a behavior, otherwise it does not. The ovals for Change Color #1 and Change Color #2 overlap, with the occurrences following both definitions populating the intersection, per Figure 3. The oval for Change Color #3 is completely contained in Change Color #1, because Change Color #1 generalizes Change Color #3, per Figure 4. The oval for Change Color #4 is completely contained in the intersection of Change Color #1 and Change Color #2, because they both generalize Change Color #4, per Figure 5. Some occurrences in the intersection are not contained by Change Color #4 because there can be occurrences satisfying Change Color #1 and Change Color #2 without the shipping step required by Change Color #4. The occurrences lying outside all the ovals do not satisfy any of the behaviors. Figure 8 is the Venn diagram for Figure 6. The intersection of the ovals for Change Color #5 and Change Color #6 do not contain any occurrences because they require conflicting things of the occurrences, per Figure 6.

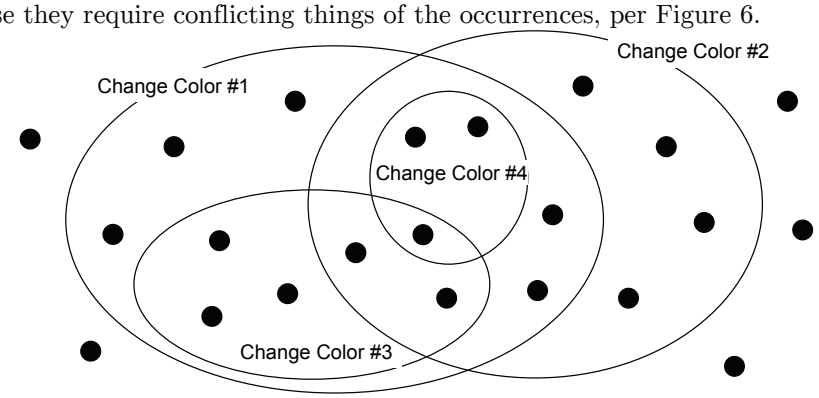


Figure 7: Venn Diagram for Figure 3 through Figure 5

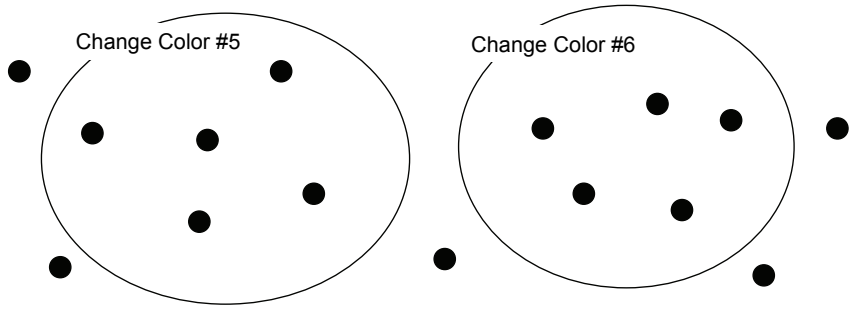


Figure 8: Venn Diagram for Figure 6

The examples in this section assume behavior languages specify which occurrences follow behaviors written in the language (semantics). There are various ways to specify behavior semantics, ranging from informal to formal. This article suggests an ontological approach that treats behaviors as models of occurrences, where models capture constraints on intended occurrences of the behaviors. This balances accessibility and formality, and can be augmented with more precise methods as needed. The rest of the article assumes familiarity with the UML metamodel and notation [OMG10a], especially its composition model [Bock04], and OMG's metalevel architecture [OMG10c].

3 Behavior Semantics in UML Currently

The most basic aspects of UML behavior semantics are the same as described in Section 2, captured in an ontological way by specializing Behavior from Class in the UML metamodel (M2), as shown in Figure 9 (this treats UML classes as ontological categories into which instances fall, rather than object-oriented classes that are instantiated, see Section 1). The dashed arrows across levels show *instances* at the tail of the arrows falling into classes at the head of the arrows (the arrow between Change Color #3 and Behavior is omitted for brevity, adopting the default UML constraint that specialized classes must be of the same or more special type than their generalizations). The instances of user-defined behaviors (M1) are occurrences (M0). Occurrences are shown at the lowest level because they do not classify anything, they are only instances, and cannot form generalizations. By comparison, behaviors are classes in the middle level, and generalize with the same semantics as classes: occurrences of specialized behaviors are occurrences of general behaviors. Behaviors support properties, associations, operations, and even other behaviors, such as state machines. This reflects common practice in systems that manage processes, for example, workflow and operating systems. Operations on behaviors might manage execution, such as starting, stopping, or aborting occurrences. The operations are specified in the behavior models (M1) and invoked

on occurrences (M0). Occurrences can have properties, such as how long the process has been executing or how much it costs, as well as links to objects, such as the performer of the execution, who to report completion to, or resources being used, and states of performance such as started and suspended. The properties are specified in the behavior models (M1) and are given values in occurrences (M0).

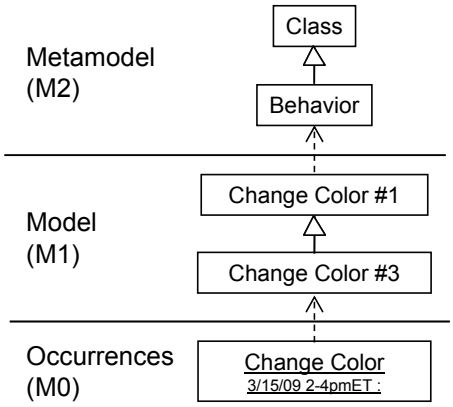


Figure 9: UML Behavior Currently

The rest of UML behavior semantics is less formally expressed, except for the slightly more formal overview in Common Behavior, which has a non-normative model of the real-world implications of behaviors, and the brief use of trace semantics in the Interactions chapter. Neither of these is used for the semantics of the other kinds of UML behavior languages described in Section 1, except some terminology reuse between the Common Behavior semantic model in state machines, and in interactions often conflated with the model level. UML has a general metamodel for time, time intervals, and durations, but it is not used to specify the semantics of behaviors, perhaps because it does not cover ordering in time. Interactions have an event ordering metamodel with briefly described trace semantics, but it is not well developed or used for specifying the semantics of other UML behaviors.

4 Behavior Semantics for UML Future

The ontological approach to UML behavior semantics in this section builds on the existing foundation of behavior classes described in Section 3. Section 4.1 uses composite structure to capture how behaviors coordinate other behaviors in time. Section 4.2 applies class modeling to events. Section 4.3 uses properties to identify things participating in behaviors and associations. Section 4.4 extends the models of the previous sections to capture transfer of things between behaviors and participants.

4.1 Composition and Time

One of the primary purposes of behaviors is to coordinate other behaviors in time. For example, in Figure 1 of Section 2, Change Color coordinates Paint and Dry, in this case ensuring the occurrences of painting and drying happen during occurrences of changing color, and in the proper order. The occurrences Paint and Dry are *suboccurrences* of the occurrences of Change Color.

Coordinating behaviors in time requires at least two kinds of constraint on allowed occurrences (semantics):

1. Between occurrences and suboccurrences, to ensure suboccurrences happen during the occurrence they are "under," for example, between Change Color occurrences and Paint occurrences.
2. Between suboccurrences, to ensure they happen in the desired order, for example, between Paint occurrences and Dry occurrences under Change Color occurrences.

The above are the whole-part and part-part relations of composition [Bock04], respectively, applied to temporal relations between occurrences. They are addressed in Sections 4.1.1 and 4.1.2, respectively.

4.1.1 Whole-part for Behavior

UML has various concrete syntaxes for the first behavior coordination semantic (occurrence to suboccurrence):

- Activities have actions that compose behaviors directly, or indirectly through operations.
- State Machines have submachine states that compose state machines, and states have behaviors that happen on entry, exit, and during the state.
- Interactions have interaction uses that compose other interactions directly, as well as messages and actions that compose behaviors indirectly.

UML does not have a common abstract syntax or semantics for the above, except for some semantic elements in the mostly informal overview in Common Behavior, which are used sporadically in specifying the concrete behavior metamodels.

The basis for the first behavior coordination semantic above is some occurrences happen during others. Specifically, the time intervals of some occurrences are within the time intervals of others (the beginning of one occurrence is at the same time or after the beginning of another, and the end of the first occurrence is at the same time or before the end of the other, equivalent to the disjunction of "during" and "=" in Allen's temporal logic [Allen83]). For example, the beginning of a Change Color occurrence is at the same time or before the beginning of its Paint

suboccurrence, and the end of the suboccurrence is at the same time or before the end of its Change Color occurrence. This can be captured as a happensDuring association between occurrences as shown in Figure 10 at M1 (the type of Property is actually Type in UML, but the semantics is same for the purposes of this article). The Behavior Occurrence class is the most general behavior, added in an M1 library. It generalizes all user-defined behaviors, and classifies all M0 occurrences (properties and operations for all occurrences can be defined on Behavior Occurrence, such as their start and end times, see other examples in Section 3). It makes no constraint on occurrences at all, it allows all of them, like an intentionally empty behavior specification. Any occurrence happening during another will be linked via happensDuring. The happensBefore association in Figure 10 is used for the second behavior coordination semantic, see Section 4.1.2.

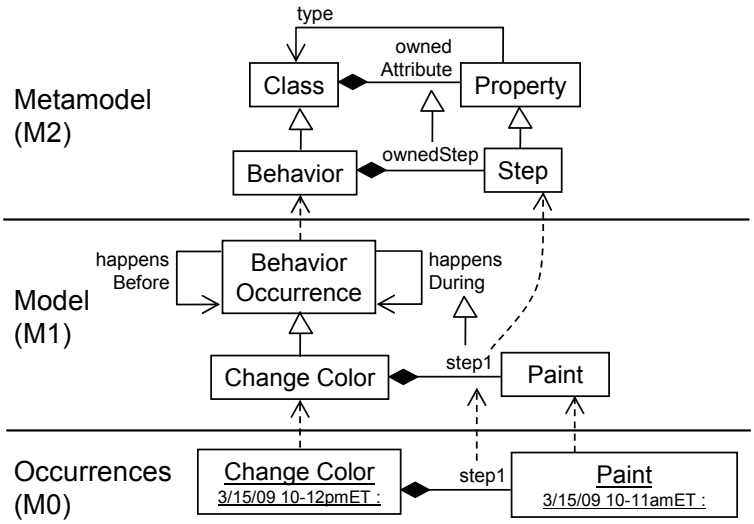


Figure 10: Step Properties

The happensDuring association must be specialized to link occurrences to suboccurrences, because there are potentially many unrelated occurrences happening at the same time that are not suboccurrences. For example, a factory will have many occurrences happening while changing the color of a particular object, such as products being placed on the loading dock. Most of these are unrelated to changing color. Suboccurrences can be distinguished by specializing Property to classify behavior properties at M1 that have suboccurrences as values at M0, see the Step metaclass in Figure 10 (the ownedStep metaproperty is subsetted from ownedAttribute to indicate a behavior's steps are included in its attributes. Generalization notation is used for UML property subsetting for brevity). The types of step properties at M1 are the "subbehaviors," such as Paint being the type of the step1 property on the Change Color behavior. Each occurrence of Change Color will have an occurrence of Paint as the value of its step1 property. Step properties are subsetted from the end of happensDuring at M1 that identifies the suboccurrence, ensuring suboccurrence time intervals are

within those of the occurrences they happen under. A similar model captures the drying step also, linking each occurrence of Change Color to an occurrence of Dry.

4.1.2 Part-part for Behavior

UML has three concrete syntaxes for the second behavior coordination semantic (suboccurrence to suboccurrence), depending on the kind of behavior:

- Activities have control flow between actions.
- State Machines have transitions between states.
- Interactions have temporal orderings between messages.

UML does not have a common abstract syntax or semantics for the above.

The basis for the second behavior coordination semantic is some occurrences happen before others. Specifically, the time intervals of some occurrences are before the time intervals of others, except possibly at an end point (the end of one occurrence is at the same time or before the beginning of another, equivalent to the disjunction of "<" and "meets" in Allen's temporal logic [Allen83]). For example, the end of the Paint suboccurrence is at the same time or before the beginning of the Dry suboccurrence under Change Color occurrences. This can be captured as a happensBefore association between occurrences as shown at M1 in Figure 10 in Section 4.1.1. Any occurrence happening before another will be linked to it via happensBefore.

The happensBefore association must be limited to each occurrence of the composed behavior separately. For example, a factory will have many occurrences of Change Color, but painting is only required to happen before drying under each occurrence separately. Under different Change Color occurrences drying might happen before painting, because drying can happen under one occurrence before painting happens under another. This can be captured by specializing UML Connector to classify those at M1 connecting step properties and typed by happensBefore, see Succession in Figure 11 at M2 (the /role property is introduced to elide connector ends for readability, and composite structure notation is used, but it is not suggested as a concrete notation for UML behaviors). Succession connectors between step properties require the occurrence values of steps to be ordered in time by linked by the happensBefore association. For example, the succession between step1 and step2 in Change Color ensures the painting suboccurrence of each Change Color occurrence happens before the drying suboccurrence under that same occurrence of Change Color, rather than others. This is the "contextualization" provided by connectors, compared to using the happensBefore association directly between Paint and Dry, which would allow all occurrences of painting to happen before drying, regardless of what occurrence of

Change Color they are under (this assumes structural semantics of connectors, rather than message passing semantics [Bock04][OMG00]).

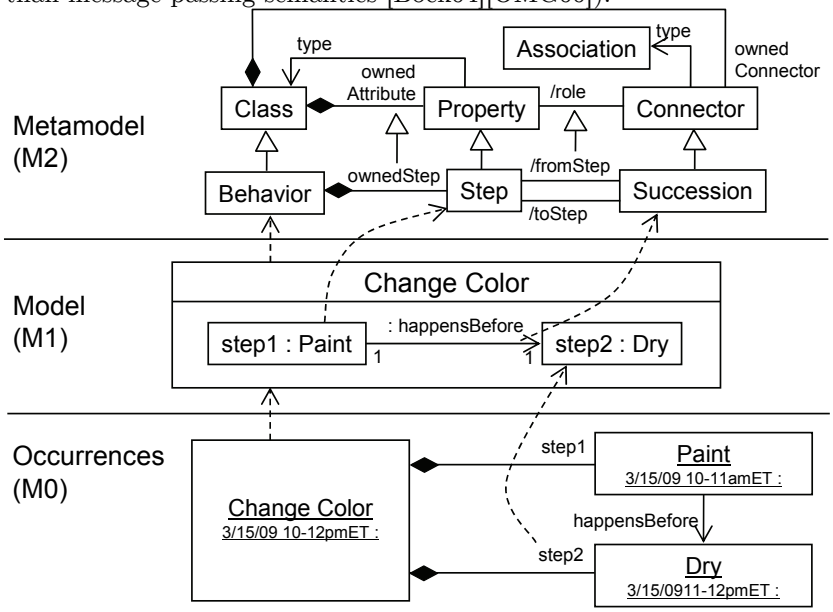


Figure 11: Succession Connectors

The semantics of successions must require existence of later occurrences at the proper time, and give the correct time ordering of occurrences when successions form loops. They should require occurrences happening later to exist when earlier occurrences have happened, and prevent occurrences from appearing later in a behavior when they were supposed to follow those happening earlier. Behaviors with succession loops can potentially have multiple occurrence values for the same step, which means some occurrences in steps appearing later syntactically will happen before occurrences in steps appearing earlier. The semantics of successions should not link all the values of earlier step properties to values of later ones.

The above requirements for the semantics of successions can be captured with UML multiplicity 1 on both ends of the connectors. Connector end multiplicities specify the minimum and maximum number of links created for each value of the connected properties on each instance of the class owning the connector. These are different from the multiplicities of the associations typing the connector, which constrain the number of links regardless of which connector creates them in which structured classifier. Connector multiplicities only constrain links created in a single instance of the structured classifier due to a single connector. Links of the association typing the connector can be created by other connectors of that type, or for other reasons entirely (for example, in the presence of succession loops, the transitivity of happensBefore will cause some values of step1 to link to multiple values of step2, and some values of step2 to link backwards to multiple values of

step1. Links created due to transitivity are not due to the connector, and are not restricted by succession connector multiplicity).

Multiplicities have lower and upper bounds, which have different semantic effects for successions depending on whether the multiplicities are on the later or earlier ends of the succession:

- Later end of successions:
    - Lower multiplicity of 1 means a succession will link every occurrence value of the earlier step through happensBefore to at least one occurrence value of the later step. In the example of Figure 11, this requires a drying occurrence value of step2 if there is a painting occurrence value of step1, because the connector must create at least one happensBefore link for each value of step1 to a value of step2. Without the lower multiplicity on the later end of successions, step2 would not be required to have a value, unless the step property has a minimum multiplicity of 1, which does not work in the presence of conditionals, see end of this section.
    - Upper multiplicity of 1 means the succession can link each occurrence value of the earlier step through happensBefore to no more than one occurrence value of the later step. Behaviors with succession loops can potentially have multiple occurrence values for the same step. In the example of Figure 11, the upper multiplicity limits the connector to link each painting occurrence value of step1 through happensBefore to no more than one drying occurrence value of step2. Without the upper multiplicity on the later end of successions, an occurrence value of step1 could be linked to multiple occurrences in step2 even though only one drying occurrence in step2 results from each painting occurrence in step1.
- Earlier end of successions:
    - Lower multiplicity of 1 means the succession will link every occurrence value of the later step through happensBefore to at least one occurrence value of the earlier step. In the example of Figure 11, this requires a painting occurrence value of step1 for each value of drying occurrence value of step2. Without the lower multiplicity on the earlier end of successions, step2 could have values that did not happen after a value in step1. The lower multiplicity prevents occurrence in step2 spontaneously appearing without an occurrence in step1.
    - Upper multiplicity of 1 means the succession can link each occurrence value of the later step through happensBefore to no more than one occurrence value of the earlier step. Behaviors with succession loops can potentially have multiple occurrence values for the same step. In

the example of Figure 11, the upper multiplicity limits the connector to link each drying occurrence value of step2 backwards through happensBefore to no more than one drying occurrence value of step1. Without the upper multiplicity on the earlier end of successions, an occurrence value of step2 could be linked to multiple occurrences in step1 even though each drying occurrence in step2 results from only one painting occurrence in step1, and with succession loops, could link step2 occurrence values to step1 occurrence values that happened later in the loop.

Taken together, the multiplicities ensure a one-to-one correspondence between occurrences at the earlier end of the succession with those at the later end, capturing more formally the token semantics informally described in some UML behaviors. This can be summarized as the "array" formation of links due to connector ends with multiplicity of 1, see Figure 9.23 of [OMG10a].

Behavior languages usually include constructs for more expressive constraints between suboccurrences. In UML these are control nodes in activities, pseudostates in state machines, and operators in interactions. UML does not have a common abstract syntax or semantics for these constructs. Some of them have the same semantics as successions, such as forks and joins in activities and state machines, and the par operator in interactions. These constructs establish partial time orders (parallelism) between portions of the behavior that are synchronized at the end of those portions. Successions support partial time ordering with more than one succession from or to the same step. For example, a step with multiple outgoing successions means the step happens before those at the later ends of the successions. Another step with multiple incoming successions means the step happens after those at the earlier ends of the successions. Steps in the separate paths between these "forks" and "joins" have no time ordering constraints, they happen in parallel.

Other more expressive coordinating constructs in UML go beyond successions, constraining occurrences across successions, for example, decisions and merges in activities, junctions and choice in state machines, and the opening side of the alt operand in interactions. These require additional constraints on suboccurrences allowed by successions. For example, decisions, splitting junctions, and the alt operator require only one of the successions going out of a step to result in an occurrence in the downstream step. This could be captured informally with "guards" on successions, possibly augmented more formally using the Object Constraint Language (OCL) at the model level, generated for each M1 behavior by constraint patterns defined in the metamodel [OMG10d]. Another example of additional constraints is merges, junctions used as merges, and the closing side of the alt operand, which require each incoming succession into a step to result in a separate occurrence of the step. This could be captured informally with relations

between successions, possibly augmented with a more formal constraint language on occurrences [Bock05][Bock06].

4.2 Events

The real-world implications of modeled events are changes as they actually happen at particular times. For example, the arrival of a product at a loading dock of a factory will happen many times, each time being a separate occurrence of a modeled event. When the modeling and occurrence levels might be confused, events in models at M1 are called *event types* and real events at M0 are called *event occurrences*.

UML has a common abstract syntax for event types (which it calls "events"), and some semantics in the mostly informal overview in Common Behavior. This is mostly limited to the time at which objects become aware of events happening outside them, which are a kind of event also. The semantic terminology in the Common Behavior overview is used to varying degrees in the semantics of the other kinds of behaviors, sometimes conflated with the modeling level.

The semantics of events can be captured in a similar way to behaviors by specializing Event Type from Class in the metamodel (M2), as shown in Figure 12 (this treats UML classes as ontological categories into which instances fall, rather than object-oriented classes that are instantiated, see Section 1). The instances of user-defined event types (M1) are the event occurrences (M0). Event types can be generalized with the same semantics as classes: occurrences of specialized event types are occurrences of the general event types. The Event Occurrence class is the most general event type, added in an M1 library. It generalizes all user-defined event types, and classifies all M0 event occurrences. It makes no constraint on event occurrences at all, it allows all of them. Event types support properties and associations, such as the time they happen and the particular objects that change. They can be linked for time ordering, which is similar enough to behavior semantics to abstract up to Occurrence in the M1 library, see Figure 12. It classifies all M0 "happenings," whether they occur over time as behaviors do, or are considered instantaneous, like events. The happensBefore and happensDuring associations are promoted to Occurrence. Any event occurrence happening before another will be linked to it via happensBefore, as well as any behavior occurrence happening before another, or both, when an event occurrence happens before a behavior occurrence. Event occurrences can happen during behavior occurrences. Event occurrences happening during event occurrences mean they happen at the same time.

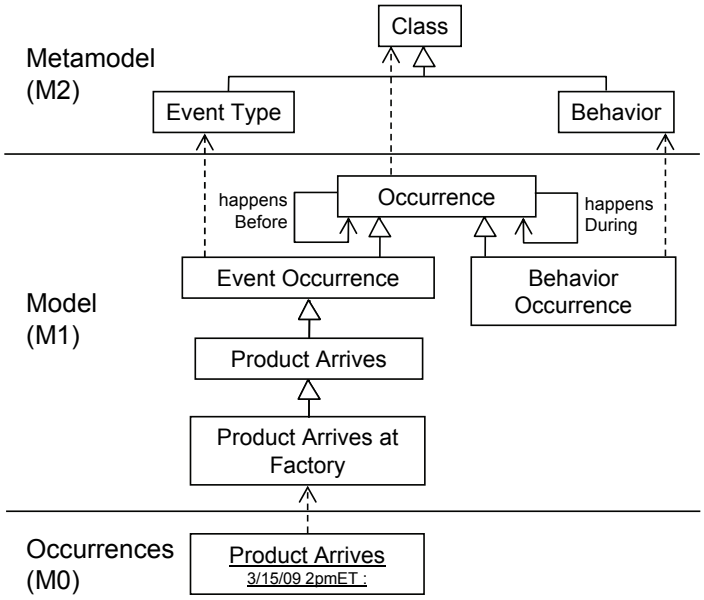


Figure 12: Event Types and Event Occurrences

Event types captured this way fold easily into behavior composition, because they can be the type of behavior properties linked by succession connectors. This assumes steps as in Figure 10 of Section 4.1.1 are generalized to be typed by behaviors or event types, enabled successions to connect both, as in Figure 13 (M2 omitted for brevity). In this example, the first step is a property typed by the arrival of a product at the factory. The occurrence value of this step is an event happening during the Change Color occurrence, due to subsetting of steps from happensDuring in Figure 10. The product arrival property is connected by succession to a painting step. This ensures painting occurs after the product arrives under each occurrence of Change Color.

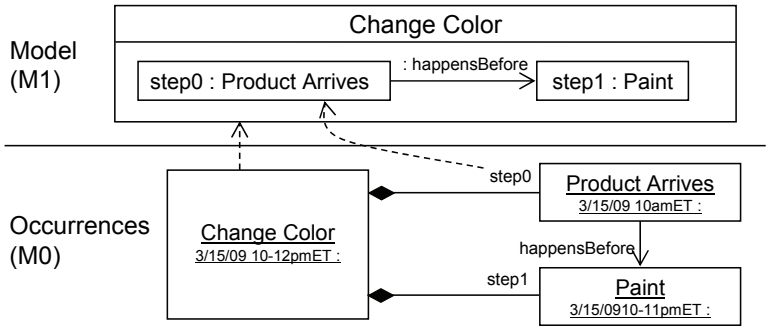


Figure 13: Event Steps

Properties of behaviors can capture events about occurrences themselves, for example, when occurrences start, end, whether they end abnormally, and so on. These can be captured in a taxonomy at M1, as shown in Figure 14. Behavior occurrences might end normally, whether or not they are successful in achieving their goals due to expected problems (normal ending), or might end unexpectedly due to actions of external or internal agents (abnormal and error ending). Taxonomies like these can be included in standard model libraries, and extended by modelers. Behavior properties typed by these can be connected by succession, as shown in Figure 15. The top class captures that all behavior occurrences have start and end properties, where the starting happens before ending under each occurrence. The Change Color behavior uses succession connectors on ports typed by behavior events to account for occurrences of Paint that fail, and require recycling of the product. It also requires that painting and ventilation abort at the same time (because events happening during others means they happen at the same time). Other precedence rules could be added, for example, that ventilation starts before painting starts.

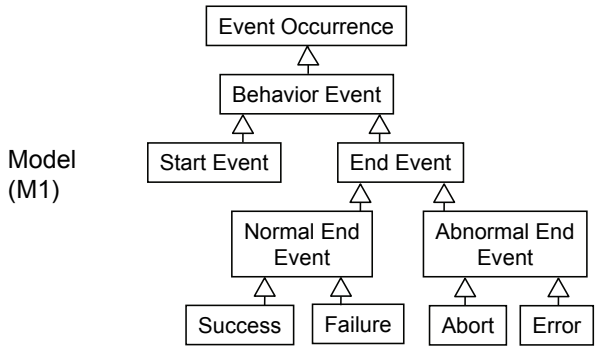


Figure 14: Behavior Event Taxonomy

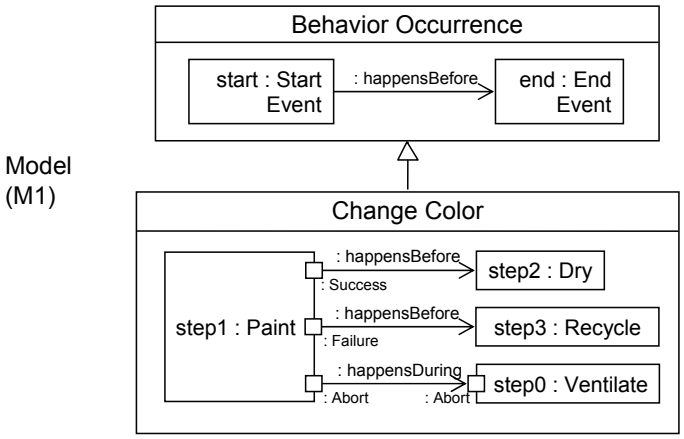


Figure 15: Successions between Behavior Events

Multiple behavior properties can have the same event type, which is useful for capturing the semantics of UML state machines. State machines are a compact notation for event-driven behaviors, in particular for specifying how objects respond to notification of external events. State machine semantics are mostly a subset of the other kinds of UML behaviors. An exception is pseudostates that machines "commit" to being accessible when used as submachines (all states of submachines are accessible, but entry and exit point pseudostates highlight that other machines use them for access to submachine states). This gives state machines multiple ways for "control" to enter and leave, which is not possible in the other kinds of UML behaviors. The semantics of entry and exit points can be captured with multiple behavior properties for start and end types on the same machine, respectively. State machines can have multiple properties typed by Start Event, each a separate entry point, while multiple properties can be typed by End Event, each a separate entry point. This distinguishes different "ways" of starting and ending the state, but without specialized event types as in Figure 15. When these machines are used by others as submachines, transitions to entry points and from exit points (through connection point references) correspond to successions to and from the event properties, as in Figure 15, assuming the merge semantics of transitions is addressed, see end of Section 4.1.2.

4.3 Participants

Behaviors involve objects that are behaving, by definition, and these objects can be identified by properties on behaviors. For example, a behavior for changing the color of objects in a factory involves at least the object having its color changed, the tools and materials used to change it, robots or people doing the changing, and so on. The behavior can have a property specifying the type of objects having their color changed, other properties specifying the types of tools and materials, and so on, as well as constraints on those objects, such as their size and other characteristics. Occurrences of the behavior have values for these properties that are instances of the specified object types, satisfying the specified constraints, and playing roles specified by the properties. For example, the values might be the object having its color changed during a particular painting occurrence, the individual tools being used in that occurrence, and so on.

Interactions are behaviors involving objects exchanging messages. Interactions can have properties specifying the types of objects involved, playing roles specified by the properties. For example, a buyer interacts with a seller. Occurrences of interactions have values for these properties that are instances of the specified types, playing roles specified by the properties. For example, the values might be a person who is buying and a company that is selling. Interactions do not usually specify internals of the objects exchanging messages, focusing only on externally

observable behavior (in particular, interactions do not need to specify the types of things exchanging messages, they might only describe the messages exchanged).

UML has various concrete syntaxes for specifying the objects involved in behaviors:

- Interactions have lifelines.
- Activities have object nodes, variables, and partitions.
- Behaviors have parameters.

UML does not have a common abstract syntax or semantics for the above.

It is not coincidental that associations also involve objects as behaviors do, by definition, and that these objects can be identified by properties on association classes. For example, an association for things owned by people has links involving at least individual things and an individual person. The association class can have properties specifying the type of things that can be owned and the type of people that can own them. Instances of the association class (links) have values for these properties that are instances of the specified types, playing roles specified by the properties.[1] Association classes can have other properties, for example, a property for how long the link has existed. UML does not currently have a standard way to specify which properties of an association class identify its end objects and which do not (the Systems Engineering Modeling Language, SysML, extends UML to support association participant properties [OMG10b]).

Behaviors can be considered specialized association classes if properties identifying objects involved in a behavior also identify its end objects as an association class. For example, a behavior for changing color can be considered an association between the object having its color changed and the tools used during the behavior. This enables behaviors to be types for connectors, which are needed to capture the semantics of object flow and messaging, see Section 4.4.

The basis for a semantics of participants is they can have lifetimes extending beyond those of the behaviors or associations they participate in. For example, an object being painted in a factory participates in a color-changing occurrence, but the object and the factory exist before the occurrence starts and after it ends. A person interacts with a company to purchase a product, but the person and company exist before the purchase begins and after it ends. A piece of furniture might be linked to an owner, but the piece of furniture and the owner usually exist before the link is created and after the link is destroyed.

---

[1] Association classes support multiplicities between their ends, for example, everything might be expected to be owned by at least one person. Ordinary classes with properties for ends cannot capture this, but they support multiplicities other than 1 for the ends themselves, see footnote 2.

The semantics of objects involved in behaviors or associations can be captured by first specializing Property to classify properties of association classes at M1 that will have end objects as values at M0, as shown in Figure 16 (the figures use M2 specialization instead of property subsetting for brevity and readability, but the semantics is the same). Then Association Class is specialized to Behavior, and Association Participant to Behavior Participant, for classifying properties of behaviors at M1 that have involved objects as values at M0. Further specializations are introduced for interactions, where the participants send messages to each other, see section 4.4. Figure 16 shows an interaction with participant properties for purchasing, including a buyer, a store, and a bank approving a credit card. A painting behavior might have participant properties identifying tools and materials used. [2]

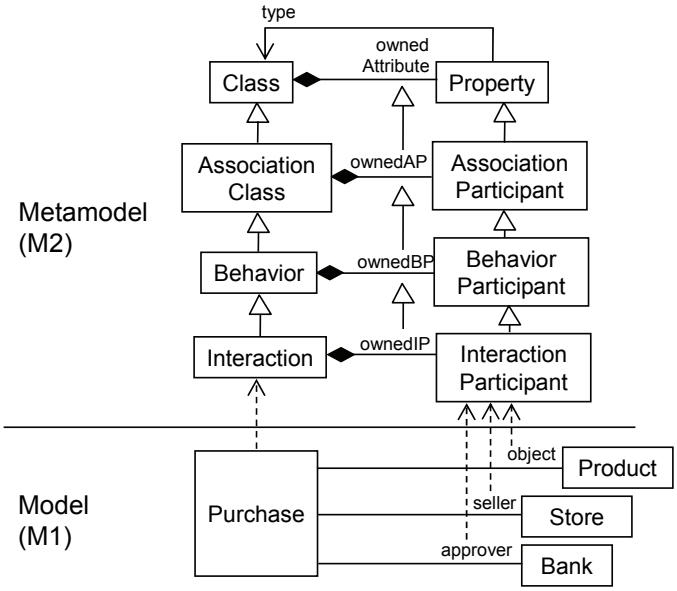


Figure 16: Participant Properties

[2] These specializations assume link ends at M0 might not have values, as in [Flatscher02][Bock97], whereas UML currently requires link ends to have exactly one object. Optional link ends are necessary because behaviors might have optional participants. The property specializations could be redefinitions, to prevent participants of the general kinds on the specialized classes. Behavior participant properties typically do not share values with step properties, but participants can be occurrences, see discussion of Figure 19. Requiring all behaviors to be association classes could be avoided with an upgrade to the Meta-Object Facility (MOF) for multiple classification [OMG10e]. Then M1 behaviors could additionally be classified as associations as needed, along with their participant properties, rather than using specialization Behavior from Association Class.

4.4 Object Flow and Messaging

The real-world implications of object flow and messaging are the "transfer" of entities, where the source and target of transfers can be anything capable of identifying the things being transferred. For example, an object in a factory can flow from a painting occurrence to a drying occurrence, even if the object does not physically move. The occurrences have properties identifying the object being painted or dried, and transfer is represented as removing the value of one property and re-assigning it to another. Object flow and messaging can also involve physical movement, for example, sending a package from one company to another.

UML has concrete syntaxes for object flow and messaging in different diagrams:

- Activities have object flows link pins on actions. Any kind of thing can flow that has elements at M0 (not operations calls, for example).
- Interactions have messages linking lifelines at points that can be identified by events. Messages can be signals or operation calls.

UML does not have a common abstract syntax or semantics for object flow and messaging (all UML behaviors can capture sending and receiving messages, but this is only the beginning and end of message transfers, rather than entire transfers).

The basis for a semantics of object flow and messaging is the transfer of entities happens over time, however small, which means they can be treated as behaviors. Occurrences of object flow and messaging behaviors start when an object begins flowing or a message is sent, and end when an object stops flowing, or the message is received. For example, an object in a factory can flow from a painting occurrence to a drying occurrence without being moved, but the transfer of participation from painting to drying occurrences will take at least some time in the real world, however small, and will start and end at particular times. Object flow and messaging that involve physical movement will obviously take at least the time to move the object or message, for example, to send a package from one company to another, and will also start and end at particular times.

The semantics of object flow and messaging can be captured by specializing Transfer from Behavior Occurrence at M1 to classify occurrences that transfer things, as shown in Figure 17. A behavior participant property on Transfer identifies the thing transferred at M0. It is typed by the class Thing, which is the most general class, provided in an M1 library. Thing generalizes all standard and user-defined M1 classes, and classifies all M0 elements of any kind. It makes no constraint on M0 elements at all, it allows all of them, like an intentionally empty class specification. Two other behavior participant properties on Transfer identify the source and target. User-defined transfers and the types of things transferred are specialized from Transfer and Thing respectively. The transferredThing property is redefined to limit the transferred things to the desired type, products in

this example (in UML, redefinition of a property by another of the same name restricts values of the property in the specialized class to the redefining type).

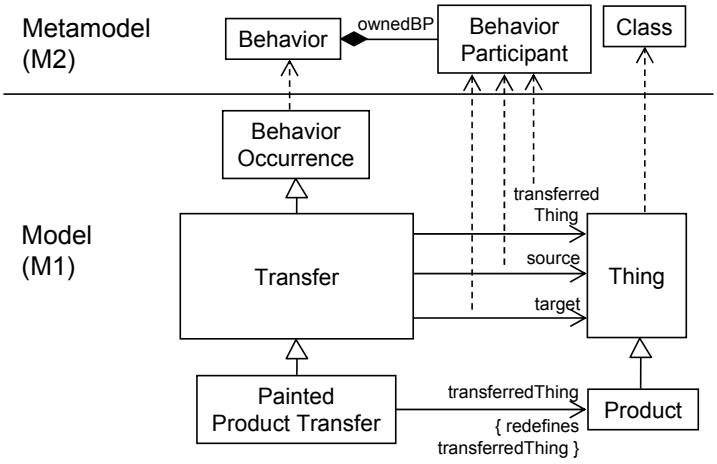


Figure 17: Transfers

Object flow and messaging differ only in the kinds of sources and targets they have, not the transfer itself. Object flow transfers things between behavior occurrences, while messaging transfers between objects. Behavior occurrences participating in object flows are like all participants in having lifetimes beyond the transfers. For example, when an object in a factory flows from a painting occurrence to a drying occurrence, the painting occurrence will start before the transfer, and the drying occurrence will end after it.

It simplifies modeling to treat object flow and messaging the same way, with the difference implied by the kinds of source and target. This enables transfers between behavior occurrences and objects, for example between internal business processes and other companies. Behaviors accepting inputs and providing outputs through object flows (described below) can also receive and send messages between objects without wrapping them with a messaging layer. UML partially integrates object flow and messaging with actions or other model elements that are informally specified as sending or receiving messages. This still requires wrappers or other modification of object flow to work across objects. UML does not have a common abstract syntax or semantics for object flow and messaging integration, and the semantics of its partial integration is specified informally.

Another aspect of the semantics of object flow and messaging is they must be limited to the behavior occurrences in which they happen, in the same sense as happensBefore in Section 4.1. Object flows in UML are between actions happening under each occurrence of activities separately, while messaging is between lifelines under each occurrence of interactions separately. This means they can be captured by specializing Connector, as shown in Figure 18 (Change Color is classified as an

activity, which are the only UML behaviors that support object flows). Object flows are unified under a general Flow class, with the difference between object flow and messaging being the kinds of sources and targets, as described above. Flow connectors at M1 are typed by Transfer or its specializations, for example in Change Color by Painted Product Transfer from Figure 17, and in Purchase by transfers of credit card numbers, approvals, and products (flow connectors have no end for the thing being transferred, even though the transfer behavior as an association does. UML constraints are loose enough to accommodate this). Flows have the property /typeOfThingTransferred with values derived from the type of the transferredThing property of the connector type at M1 (transfers having no detail in them other than the thing that flows could potentially be omitted, leaving only the value of the typeOfThingTransferred property at M2). This is not the type of the transfer during which the thing is transferred, which is always TRANSFER or its specializations. The top M1 model in Figure 18 uses an object flow, because it connects properties identifying behavior occurrences in the flow, while the bottom model is using messaging, it connects properties identifying objects.

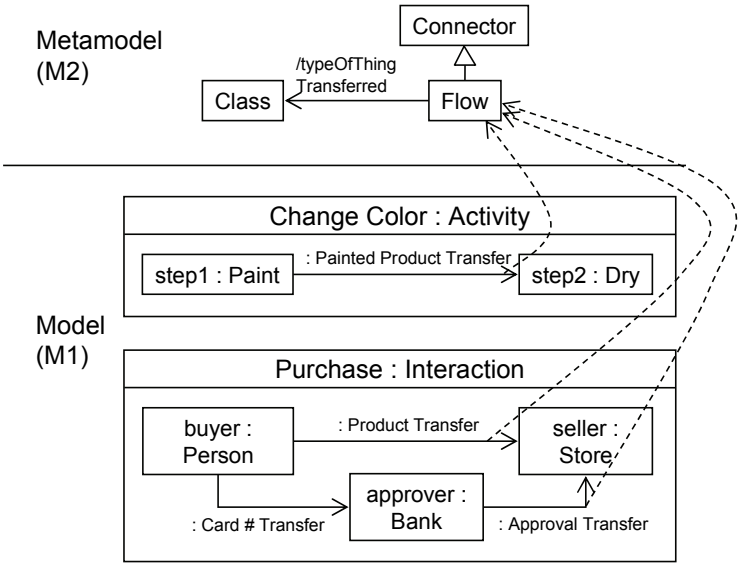


Figure 18: Flow Connectors

4.4.1 Inputs and Outputs

Some transfers come from and go to the "outside" of behaviors ("inputs" and "outputs"). For example, a behavior for changing the color of objects in a factory will get objects on which to operate from elsewhere in the factory, and will also give the changed object back to somewhere in the factory. Interactions usually do not have inputs and outputs, because they can add more participants to receive

and send more messages as needed, but interactions can have inputs and outputs to integrate with external behavior occurrences.

UML has a common abstract syntax for inputs and outputs (parameters), but the semantics is specified informally in different ways in interaction and activities (state machines have parameters, but do not provide elements for using them).

Inputs and outputs are semantically a kind of transfer with specialized participants for entities outside behaviors. This can be captured by a specialization of Behavior Participant, as shown in Figure 19.[3] In this example, the external entities for the changing color behavior are a feeder from which products are drawn to be painted, and a buffer to which they are put after drying completes (this assumes an upgrade to MOF supporting multiple classification for participants that are both external and interacting [OMG10e]). This highlights the flexibility of generalizing object flow and messaging. The transfer acts as object flow for the occurrences of painting and drying, but as messaging for the feeder and buffer (Change Color is not classified as an activity, as in Figure 18, because UML activities do not support flows to external objects). The external entities can also be behavior occurrences, as they typically are in business process modeling or functional programming languages, for example (and in UML, where behaviors are classes, see Section 3). In these applications there is one external "calling" occurrence from which inputs are accepted and to which outputs are provided for each occurrence of the behavior.

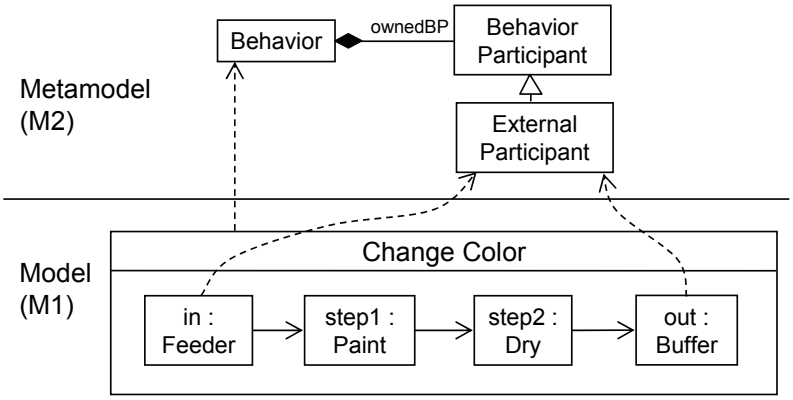


Figure 19: External Participants

[3] The external entities do not appear as ports in Figure 19, because they represent the external entities themselves, rather than a point at which things come out of or go into occurrences. They could be modeled as ports if the behavior is reused with equality connectors to the ports. Then connectors would link the ports to (properties identifying) entities outside the behavior.

The approach to inputs and outputs above is more expressive than UML parameters, because it can model the time order in which inputs arrive and outputs leave without specifying the internal details of the behavior, see Section 4.4.2. This is needed for long-lived behaviors that accept inputs and provide outputs at various times.

4.4.2 Flow Ordering

Transfers can be composed into larger transfers that order them in time ("protocols") as all occurrences can, see the two aspects of behavior coordination semantics in Section 4.1. A transfer happening during another means it starts and ends within the time interval of a larger transfer (occurrence to suboccurrence, whole-part semantics), while a transfer happening before another means one is completed before another starts (suboccurrence to suboccurrence, part-part semantics).

UML has three concrete syntaxes for flow ordering, depending on the kind of behavior:

- Interactions can order messages in time and reuse other interactions through interaction use.
- Protocol state machines can specify the order in which operations can be called on a class, and reuse other protocols as submachines.
- Activities can order actions for sending and receiving messages in time, and compose other activities through direct and operation calls.

UML does not have a common abstract syntax or semantics for the above.

Transfers and time ordering are captured as special kinds of connectors (flows and successions, respectively, see beginning of Section 4.4 and Section 4.1). This means capturing the time order of transfers requires succession connectors between flow connectors. Since connectors are always between properties, flow connectors must also be properties, the values of which are the M0 transfers as links (behaviors are association classes, occurrences are links between participating objects, see Section 4.3). It is useful to connect connectors generally, and this can be captured by specializing ConnectorProperty from Connector and Property as shown in Figure 20. Connector properties are typed by association classes, rather than ordinary associations, enabling them to have links as values (SysML extends UML to support connector properties [OMG10b]). The links in a composite are treated the same as the other objects in it. Figure 21 applies this to protocols by specializing Flow from ConnectorProperty, rather than from Connector, and also from Step to enable flows to be ordered in time. Successions can link flows as steps, for example in Figure 21 where the card transfer flow happens before the

approval flow, which happens before the product is given to the buyer. Successions can also capture the time order in which inputs arrive and outputs leave, as needed for long-lived behaviors that accept inputs and provide outputs at various times, such as streaming parameters in UML activities.

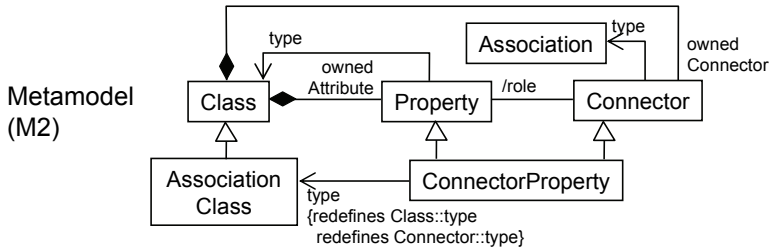


Figure 20: Connector Properties

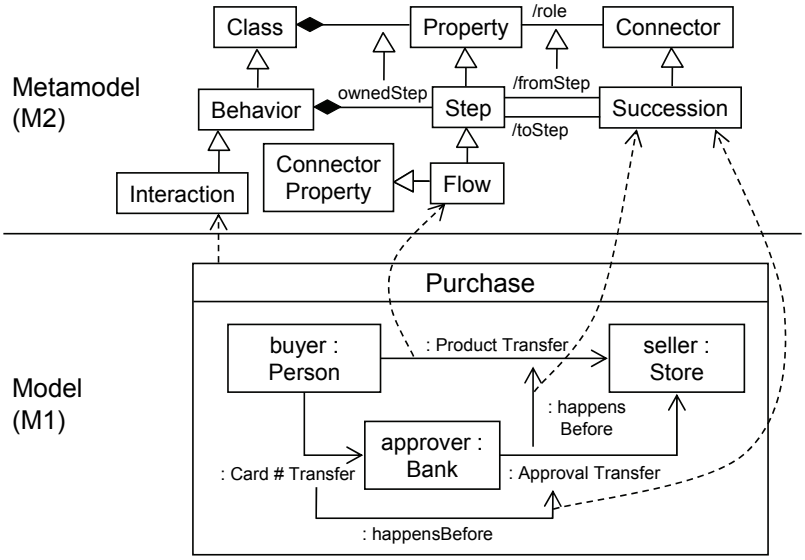


Figure 21: Flow Protocols

4.4.3 Composition with Flows and Participants

When behaviors coordinate other behaviors in time, as described in Section 4.1, they also coordinate flows between them and their participants. For example, a factory might use a behavior for changing color together with a behavior for assembling parts. The objects flowing out of changing color occurrences might be the same objects flowing into assembly. Similarly, a company might use a purchasing interaction having a particular organizational position as the buyer, using credit cards issued from the currently contracted bank, and purchased from currently approved stores. Coordination behaviors specify "bindings" that specify how flows and participants in the coordinated behavior are determined.

UML has various concrete syntaxes for flow and participant binding:

- Activities have actions with pins matching called behavior parameters.
- Interactions have arguments matching behavior parameters, and can be used in conjunction with collaboration, collaboration uses, and collaboration role bindings.

UML does not have a common abstract syntax or semantics for the above (state machines have binding-like constructs, but these are for time ordering, rather than transfers, see the end of Section 4.2).

The basis for a semantics of bindings is they establish equality between the same things playing different roles in the coordinating and coordinated behavior occurrences. For example, a factory using a behavior for assembling might require transfers into assembly occurrences to be the same ones out of painting occurrences. Similarly, a company using a purchasing interaction might require the person participating as the buyer to be the same as the one participating in the company in the requisition position, and the credit cards used to be the same ones supplied in current banking contracts.

An aspect of the semantics of bindings is they must be limited to the behavior occurrences in which they establish equality, in the same sense as happensBefore in Section 4.1. The equality required by a coordinating behavior only applies to flows and participants within each occurrence of the coordinating behavior, and the occurrences being coordinated under it. This means bindings can be captured by specializing Connector, as shown in Figure 22 (SysML extends UML to support binding connectors [OMG10b]). In this example, a factory behavior uses binding connectors to equate M0 transfers out of its changing color suboccurrences to the transfers between changing color and assembling, and to equate those to transfer occurrences into its assembly suboccurrences. The nested composite structure diagrams in Figure 22 indicate reuse of the separately defined behaviors Change Color and Assembly by using them as types of steps. Bindings are directed, to prevent modification of reused behaviors, even though equality is symmetric mathematically. Using connectors this way in UML requires input and output flows to be ports, though they are not shown this way in Figure 22. Port connector properties can indicate which flows are accessible when a composite class is reused in another composite.

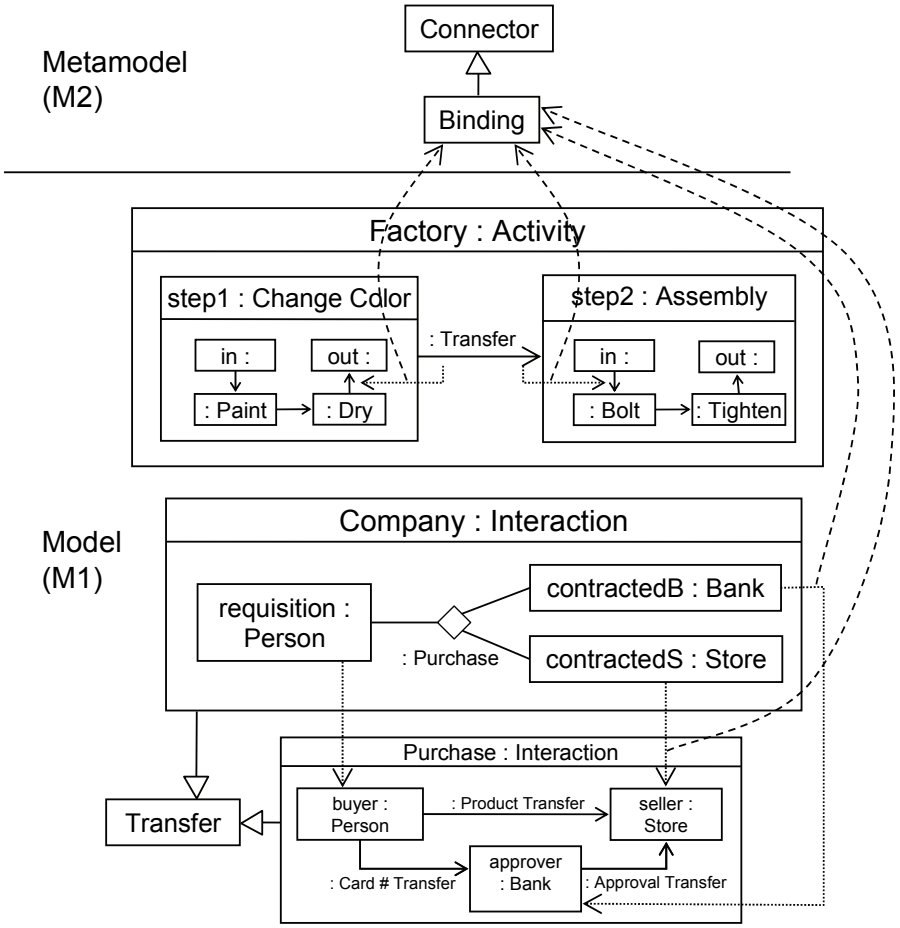


Figure 22: Binding Connectors

The bottom of Figure 22 shows an example of interaction reuse with participant bindings. The n-ary association notation applied in a composite structure is a three-end flow connector in an overall company interaction. The flow connector reuses a purchasing interaction, shown outside rather than inside as in the factory example. The purchasing interaction is specialized from Transfer, as all interactions are because their occurrences transfer things. This enables Purchase to be the type of the three-end flow in the Company interaction, because flows are connectors typed by transfers, see Section 4.4. The Company interaction has binding connectors between its participants and those of the Purchase interaction. These ensure the buyer in the purchase is the same person that fills the requisition position in the company, and the bank and store in the purchase are those contracted by the company. As in the factory example, using connectors this way in UML requires the participant properties to be ports, even though they are not shown this way in Figure 22. Port participants can indicate which participants are accessible when a behavior is reused in another behavior.

5 Related and Future Work

Many applications of ontology to dynamics focus either on modeling languages, or on the things being modeled (occurrences), but not both at once. A common approach is to use ontology languages for capturing modeling language syntax [Martin04][Dumitru05][Haller08]. This provides an accurate description of the way modelers can assemble language elements (syntax). However, it does not give them capabilities typical of ontology languages, such as specialization, or a way to tell when occurrences at M0 follow models written at M1 (semantics). At the opposite end of the spectrum, ontology languages can describe or constrain occurrences directly, with little emphasis on constructing behavior models [ISO06][Aitken02][Masolo02]. This gives a way to specify semantics for behavior modeling languages when desired, but not the semantics or syntax of any particular language. With these approaches, the modeler must repeatedly specify behaviors in all semantic detail, without the shorthands provided in behavior modeling languages. To address this, some researchers translate behavior modeling languages to ontologies of occurrences [Ren09][Gröner10]. This provides an easier "front end" for specifying allowed occurrences, but does not fully integrate modeling languages with their semantics, and in the particular work cited, requires significant limitations, such as sequentializing parallel tasks.

At least one framework enables the application of ontology to modeling languages and the things being modeled at the same time (syntax and semantics) [IDEAS09], but does not provide extensions for modeling dynamics. Another effort extends an upper ontology for modeling dynamics [Gangemi05]. However, it is either expressed too mathematically for the modeling community to use, or applies ontology languages only to syntax. Some are extending ontology languages to capture syntax and semantics at the same time, but have not applied the extensions to modeling dynamics [Jekjantuk10]. Work on formalizing UML with ontologies does not address its behavior models [Berardi05][Guizzardi04]. UML 2 introduces ontological meanings for classification, and these are applied to its syntax through the subset of UML used to define itself [OMG10e]. UML 2 also partially applies ontology to the semantics of behavior as classes. This paper continues this line of development, employing composite structure to capture temporal and other relationships between elements of behavior in a general enough way to apply to all three of UML behavior models.

The framework of this paper can be extended in future work to complete its specialization into the three UML behavior languages, including such topics as asynchronous and polymorphic invocations, interrupts, exceptions, and more expressive coordinating constructs for suboccurrences.

6 Summary

This article suggests improving the effectiveness of behavior modeling languages through ontological approaches, enabling users and implementers to understand them more uniformly. These approaches specify real-world implications of language sentences more rigorously than informal text, but not directly in mathematics. They start with common sense notions, building up incrementally to more complex ones. By taking smaller, accurately defined steps in language development, standards can increase the reliability of communication between users, tools, and implementers, enabling tools to work more seamlessly with each other and with the people using them.

A proof-of-concept for ontological approaches is provided by a common semantic basis for UML behaviors. It starts with the existing UML notion of behaviors as classes, where each instance is one occurrence of a behavior in time, see Section 2. The article treats elements of behavior as parts of a whole, as captured in UML composite structure. The two relationships of composition (whole-part and part-part) are applied to behavior through a common sense model of time: nested durations for subbehaviors, and time ordering for steps in behaviors, respectively, see Section 4.1, and summary in Figure 23 and Figure 24. Events are captured as classes, where each instance is one occurrence of an event in time. This enables them to type step properties and be ordered in time with other steps, see Section 4.2. Participants in behaviors and associations are treated as parts of a whole, and captured as properties in a composite structure, as summarized in Figure 24. This enables behaviors to act as links between participants, as associations do, and be used to connect parts of other behaviors, see Section 4.3. Specialized behavior associations between participants capture the transfer of objects in messaging and object flow, which are distinguished by the kind of source and target of the transfer (objects or behavior occurrences, respectively). Transfers connect elements of behaviors, including steps and participants, through composite structure, see the introduction to Section 4.4 and Section 4.4.1. Specialized properties identify links connecting objects and occurrences, which are combined with behavior steps to enable transfers to be ordered in time, as in messaging protocols, and inputs and outputs to long-lived behaviors, see Section 4.4.2. Finally, transfers can be equated (bound) to each other to enable behaviors to coordinate transfers when they use other behaviors see Section 4.4.3.

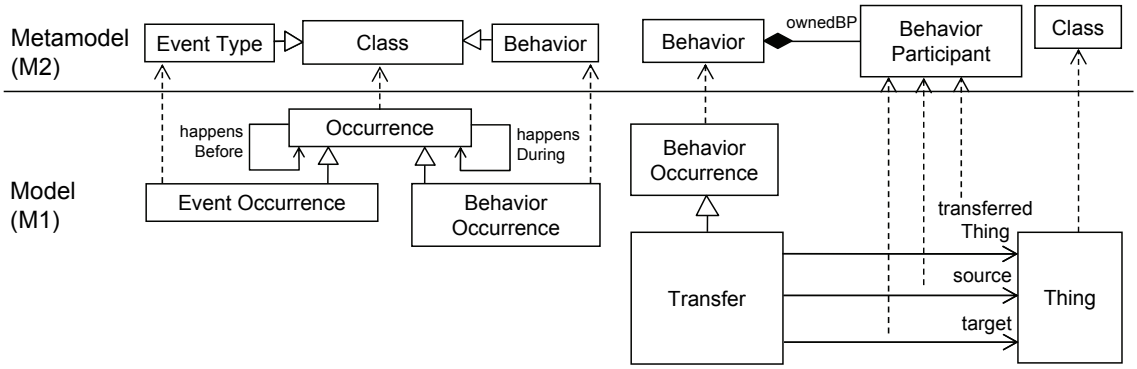


Figure 23: Model Library

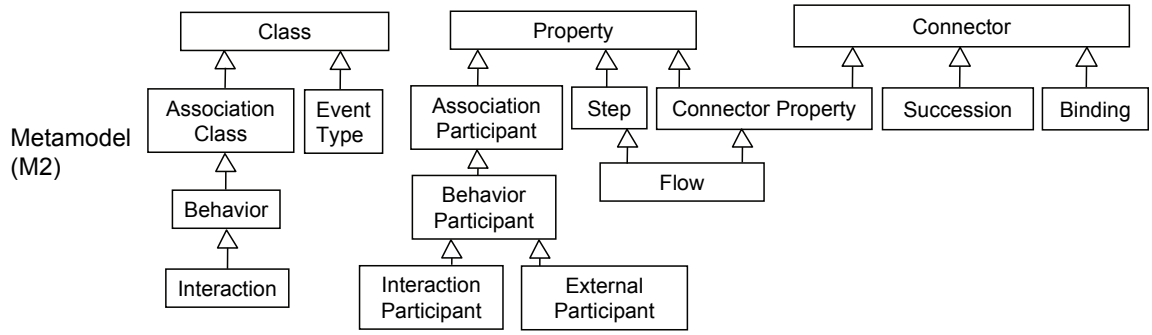


Figure 24: Metaclass Taxonomy

The ontological approach to language specification appears in the above models as simple notions, such as class as category, and properties specifying links between instances, also falling into categories, with both specialized in multiple, thin layers to more sophisticated constructions, such as flows between various kinds of behavior participants. At each stage, the implications of user models for the real world are captured (semantics), sometimes with reusable model libraries. This enables more uniform understanding and implementation of the three UML behavior models, and more expressiveness from their integration.

Acknowledgements

Many of the behavior modeling techniques in this article were applied largely independent of UML in the Business Process Definition Metamodel, developed with team members Antoine Lonjon, Cory Casanave, and others [OMG08]. A partial mathematical formalization is available in an executable subset of UML [OMG11].

Commercial equipment and materials might be identified to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the U.S. National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

[Aitken02] S. Aitken, J. Curtis: "A Process Ontology," *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web, Lecture Notes in Computer Science*, vol. 2473, pp. 263-270, 2002. doi:10.1007/3-540-45810-7_13

[Allen83] J. Allen: "Maintaining Knowledge about Temporal Intervals," *Communications of the Association of Computing Machinery*, vol. 26, no. 11, pp. 832-843, November, 1983. doi:10.1145/182.358434

[Berardi05] D. Berardi, D. Calvanese, G. De Giacomo: "Reasoning on UML class diagrams," *Artificial Intelligence*, vol. 168, no. 1-2, pp. 70-118, October 2005. doi:10.1016/j.artint.2005.05.003

[Bock04] C. Bock: "UML 2 Composition Model," *Journal of Object Technology*, vol. 3, no. 10, pp. 47-73, http://www.jot.fm/issues/issue_2004_11/column5, November-December, 2004. doi:10.5381/jot.2004.3.10.c5

[Bock05] C. Bock, M. Gruninger: "PSL: A Semantic Domain for Flow Models," *Software and Systems Modeling Journal*, vol. 4, no. 2, pp. 209-231, May 2005. doi:10.1007/s10270-004-0066-x

[Bock06] C. Bock: "Interprocess Communication in the Process Specification Language," *U.S National Institute of Standards and Technology Interagency Report 7348*, October 2006.

[Bock97] C. Bock, J. Odell: "A More Complete Model of Relations and Their Implementation, Part I: Relations as Object Types" *Journal of Object-Oriented Programming*, vol. 10, no. 3, pp. 38-40, http://www.conradbock.org/relation1.html, June 1997.

[Borgida07] A. Borgida, R. Brachman: "Conceptual Modeling with Description Logics," *The Description Logic Handbook: Theory, Implementation, and Applications*, 2nd ed., Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P (eds.), pp. 375-401, August, 2007. doi:10.2277/0521781760

[Cocks04] D. Cocks, M. Dickerson, D. Oliver, J. Skipper: "Model Driven Design," *International Council on Systems Engineering Insight*, vol. 7, no. 2, July 2004.

[Dumitru05] R. Dumitru, U. Kellera, H. Lausena, J. de Bruijna, R. Laraa, M. Stollberga, A. Polleresa, C. Feiera, C. Busslerb, D. Fensela: "Web Service Modeling Ontology," *Applied Ontology*, vol. 1, pp. 77-106, 2005.

[Flatscher02] R. Flatscher: "Metamodeling in EIA/CDIF-Meta-Metamodel and Metamodels," *Association of Computing Machinery Transactions on*

Modeling and Computer Simulation, vol. 12, no. 4, pp. 322-342, October 2002. doi:10.1145/643120.643124

[Gangemi05] A. Gangemi, S. Borgo, C. Catenacci, J. Lehman: "Task taxonomies for knowledge content," *Laboratory for Applied Ontology*, http://www.loa-cnr.it/Papers/D07_v21a.pdf , 2005.

[Genesereth87] M. Genesereth, N. Nilsson: *Logical Foundations of Artificial Intelligence*, Morgan Kaufman, 1987.

[Gröner10] G. Gröner., S. Staab: "Specialization and Validation of Statecharts in OWL," *Proceedings of of 17th International Conference on Knowledge Engineering and Knowledge Management by the Masses, Lecture Notes in Artificial Intelligence*, vol. 6317, pp. 360-370, 2010. doi:10.1007/978-3-642-16438-5_26

[Guizzardi04] G. Guizzardi, G. Wagner, H. Herre: "On the Foundations of UML as an Ontology Representation Language," *Lecture Notes in Computer Science*, vol. 3257, pp. 47-62, 2004.

[Haller08] A. Haller, M. Marmolowski, E. Oren, W. Gaaloul: "A Process Ontology for Business Intelligence," *Digital Enterprise Research Institute*, Technical Report 2008-04-1, April 2008.

[Horrocks06] I. Horrocks, O. Kutz, U. Sattler: "The Even More Irresistible SROIQ," *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 57-67, American Association of Artificial Intelligence Press, 2006.

[IDEAS09] International Defence Enterprise Architecture Specification Group: "The IDEAS Foundation Model," http://www.ideasgroup.org/foundation, 2009.

[ISO06] International Organization for Standardization: "Process Specification Language (ISO 18629)," ISO Technical Committee 184, Sub-committee 4, June 2006.

[Jekjantuk10] N. Jekjantuk, G. Gröner, J. Pan, E. Thomas: "Towards hybrid reasoning for verifying and validating multilevel models," *Proceedings of of 17th International Conference on Knowledge Engineering and Knowledge Management by the Masses, Lecture Notes in Artifical Intelligence*, vol. 6317, pp. 411-420, October 2010. doi:10.1007/978-3-642-16438-5_31

[Martin04] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, M. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, K. Sycara: "Bringing Semantics to Web Services: The OWL-S Approach," *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*, pp. 26-42, July 2004. doi:10.1007/b105145

[Masolo02] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari: "WonderWeb Deliverable D18: Ontology Library," *Laboratory for Applied Ontology*, http://wonderweb.semanticweb.org/deliverables/documents/D18.pdf, 2002.

[OMG00] Object Management Group: "UML 2.0 Superstructure Request For Proposal," http://doc.omg.org/ad/00-09-02, September 2000.

[OMG08] Object Management Group: "Business Process Definition MetaModel," http://doc.omg.org/formal/2008-11-03, http://doc.omg.org/formal/2008-11-04, November 2008.

[OMG09] Object Management Group: "Ontology Definition Metamodel," http://doc.omg.org/formal/2009-05-01, May 2009.

[OMG10a] Object Management Group: "OMG Unified Modeling Language, Superstructure," http://doc.omg.org/formal/2010-05-05, May 2010.

[OMG10b] Object Management Group: "OMG Systems Modeling Language," http://doc.omg.org/formal/2010-06-01, June 2010.

[OMG10c] Object Management Group: Unified Modeling Language: Infrastructure, http://doc.omg.org/formal/2010-05-03, May 2010.

[OMG10d] Object Management Group: "Object Constraint Language 2.2," http://doc.omg.org/formal/2010-02-01, February 2010.

[OMG10e] Object Management Group: "MOF Support for Semantic Structures (SMOF) - Beta 1," http://doc.omg.org/ptc/2010-11-39, November 2010.

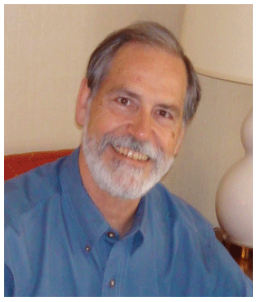
[OMG11] Object Management Group: "Semantics of a Foundational Subset for Executable UML Models (fUML)," http://doc.omg.org/formal/11-02-01 , February 2011.

[Ren09] Y. Ren, G. Gröner, J. Lemcke, T. Rahmani, A. Friesen, Y. Zhao, J. Pan, S. Staab: "Validating Process Refinement with Ontologies," *Proceedings of International Workshop on Description Logics*, July 2009.

[W3C09] World Wide Web Consortium: "OWL 2 Web Ontology Language, Document Overview," http://www.w3.org/TR/owl2-overview, October 2009.

About the authors

Conrad Bock is a Computer Scientist at the U.S. National Institute of Standards and Technology's Engineering Laboratory, specializing in formal product and process modeling. He was the founding editor for the Activity and Action models in the Unified Modeling Language and Systems Modeling Language at the Object Management Group, as well as a primary contributor to interaction modeling in the Business Process Model and Notation. He can be reached at conrad dot bock at nist dot gov.



James Odell is an international consultant specializing in applying object-oriented and agent-based techniques to build process- and event-aware enterprise systems. His commercial work involves understanding, communicating, and developing business systems and standards - especially those involving UML, business process management, data and meta-data modeling, service orientation, event-driven approaches, applied ontology, multiagent systems, and complex adaptive systems. These systems include software, machines, and people as agents and objects. He can be reached at email at jamesodell dot com.