Behavior as Composite Structure: A Common Semantic Basis for UML Behaviors

Conrad Bock, U.S. National Institute of Standards and Technology
James Odell, Computer Sciences Corporation
Tim Weilkiens, oose Innovative Informatik
James Baker, BAE Sytems
Antoine Lonjon, MEGA
August 17, 2009

This response to the Future Development of UML Request For Information [1] suggests unifying UML's three kinds of behavior around the abstract syntax and semantics of composite structures. This significantly simplifies the metamodel, provides a semi-formal semantics to clarify ambiguities in the current informal semantics, and increases the expressiveness of UML behaviors.

1 Introduction

Benefits of language standardization include more reliable communication and implementation. Language standards increase the likelihood that what is said will be the same as what is heard. This extends to implementers of tools for using the languages, increasing the likelihood that tools work seamlessly with each other and with the people using them.

The effectiveness of language standards depends on how uniformly people understand them, whether users or implementers. Perfect standards are understood the same way by everyone who uses the languages. Worst-case standards are understood differently by everyone who uses them. Many standards are in between, forming islands of reliable communication with bridges of partially reliable communication between some of them.

Understanding languages means seeing the real-world implications of things said using the languages (in "sentences"). For example, understanding "The dog chases the cat" means having some idea what constitutes real dogs, cats, and chasing. The relationship between sentences and the real world interpretations of them is *semantics*. The relationship between a language and the sentences is *syntax*, which tells which sentences are allowed in the language [2].

Effective language standards require uniform understanding of the real-world implications when something is said in the language (semantics). For example, an effective programming language standard enables users and implementers to have the same idea of what happens at runtime when using the reserved words of the language. Ineffective language standards result in people drawing different real-world implications from the same sentences. For example, an ineffective modeling language standard cannot be passed between people who do not know each other without misinterpretation.

There are various ways to achieve uniform understanding of the semantics of a language. It might be through common practice and communication already in place, informal documentation with many examples of the language being used and its real-world meaning, reference implementations, compliance tests, or more formal documentation based on existing standard languages, sometimes mathematical. These can be combined. For example, common practice and communication with informal documentation that leaves out well-understand details can be an effective standard.

Languages adopted at the Object Management Group (OMG) typically use informal documentation, but are often lacking in the details and examples necessary to achieve uniform understanding of the real-world implications of things said in the languages. OMG does not use reference implementations or compliance tests. Its adoptions rarely express semantics with more formal languages. Like many standards organizations, it is often working in areas that would benefit significantly from standard languages, but where common practice and communications are not uniform yet.

This response suggests incrementally increasing formality in OMG language standards as the most feasible approach to improving their effectiveness. Formality is relative notion, being the ratio of structure to informal text in a specification. This ratio can be increased by building up languages from smaller, simply defined elements to larger ones, in small enough layers that the language is more easily understood in a uniform way. Such *semi-formal* languages can be translated to more informal or formal languages as desired. They do not require reference implementations or compliance tests, but would make these easier to develop if desired, in part because formalism tends to attract additional resources from academia. Semi-formal approaches also facilitate consolidation of common practice and communication. Semi-formal approaches are already used in some OMG technology and can be easily understood by the OMG community. Metamodels provide language structure and specialization of one language from another is common, either through metamodels or metamodel profiles.

As a proof-of-concept for semi-formal approaches, this response applies them to the Unified Modeling Language (UML) to unify the abstract syntax and semantics of UML behaviors [3].[1] Some of these techniques are used in UML currently, but not completely applied, and are adopted in other OMG specifications, see Section 5. In particular, the response specializes behavior from composite structure. Starting with the existing concepts behavior classes and instances in UML, and adding a simple temporal model, composite structure can be specialized to capture the three existing UML behaviors. This response sketches how this is done for the most salient aspects of UML behaviors.

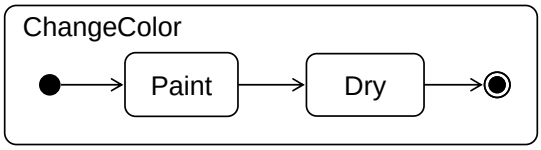
Section 2 defines the notion of semantics used in this response, drawn from conventional mathematical definitions. Section 3 covers how UML behavior semantics is defined currently. Section 4 applies composite structure to behavior modeling. Section 5 has links to other material on this topic.

[1] The response does not address concrete syntax or assume any changes in notation. It uses composite structure notation to show user models, but it is not expected to be the concrete syntax for behaviors.

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 (occurrences are "computation-independent" in OMG terminology).

The semantics of behavior languages specify which occurrences are allowed for behaviors expressed in those languages. Figure 1 shows three behaviors that happen to allow the same occurrences, 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, juxtaposition, and punctuation imply painting must be complete before drying starts, and even 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 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. Figure 2 is only a few example occurrences, rather than a complete semantics. A complete semantics of the behavior languages used in Figure 1 provides general rules to determine for any occurrence whether it follows the particular process specifications in Figure 1.




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

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

Figure 1: Behavior Notation Examples

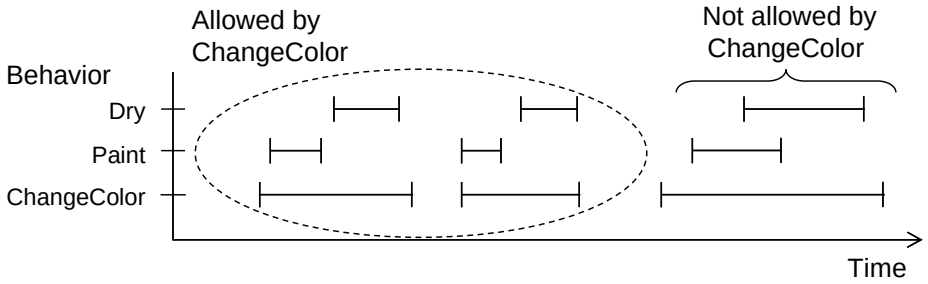


Figure 2: Behavior Occurrences

Multiple, partially specified behaviors can allow the same occurrence. Figure 3 illustrates this for the two behaviors at the top. The CHANGECOLOR #1 definition on the left specifies that painting happens before drying. The CHANGECOLOR #2 definition on the right specifies that spray painting happens before cleanup. The group of occurrences on the lower left only follows CHANGECOLOR #1, because it does not clean up, and brush painting is not spray painting, while the occurrences on the right only follow CHANGECOLOR #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.

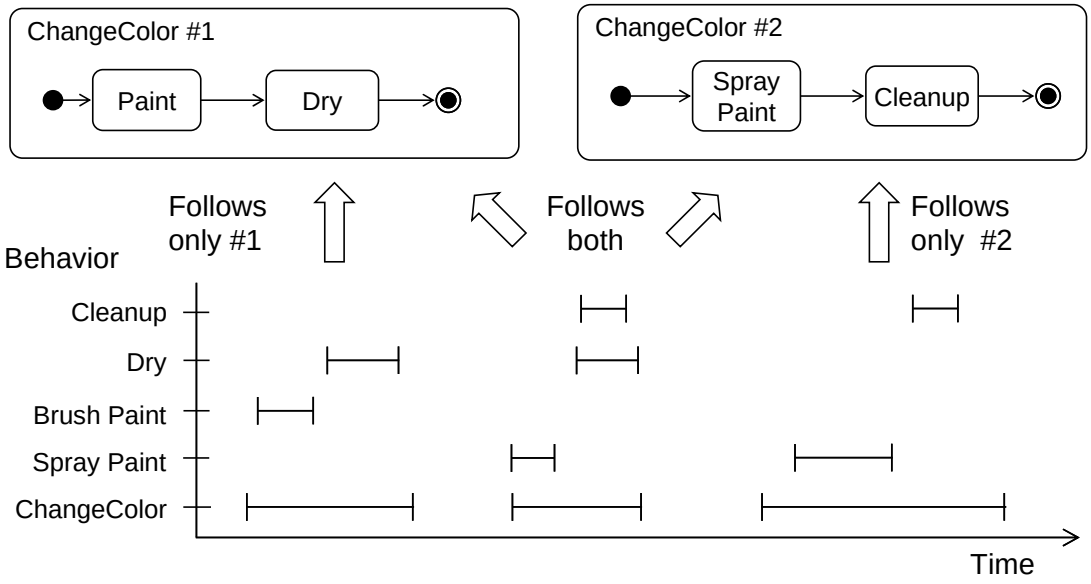


Figure 3: Overlapping Behaviors

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 (which has the same semantics, see Section 3. This means any occurrence following the more special behavior CHANGECOLOR #3 also follows the general behavior CHANGECOLOR #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

CHANGECOLOR #4 also follow CHANGECOLOR #1 and CHANGECOLOR #2, but it is possible to have occurrences only of the general behaviors separately. Figure 6 shows behaviors that cannot have common occurrences, they are inconsistent. The CHANGECOLOR #5 behavior allows only drying right after painting, using a closed or imperative semantics, while CHANGECOLOR #6 allows only shipping. No occurrences can follow both of these behaviors.

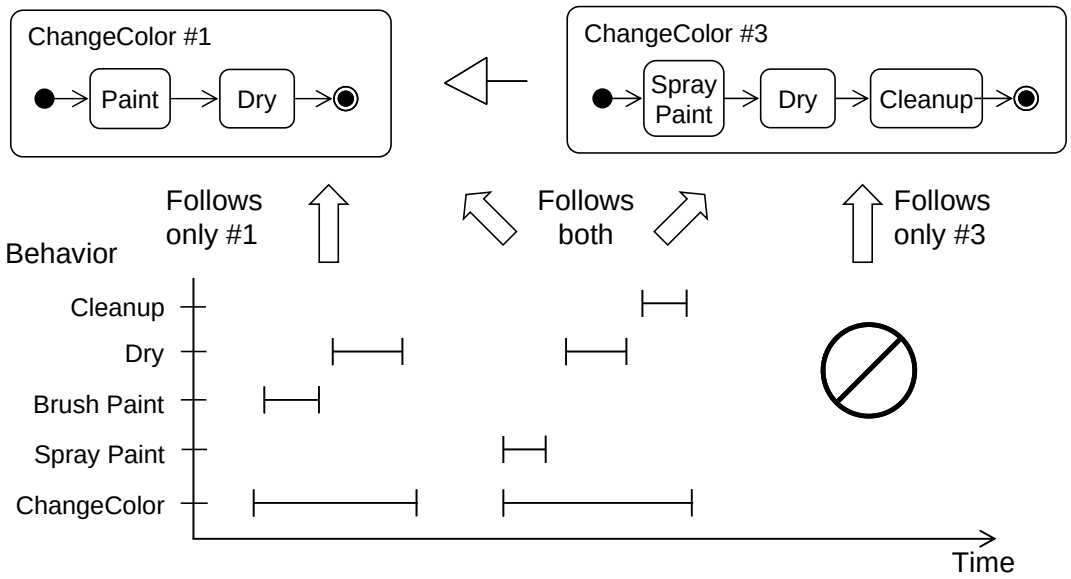


Figure 4: Behavior Generalization

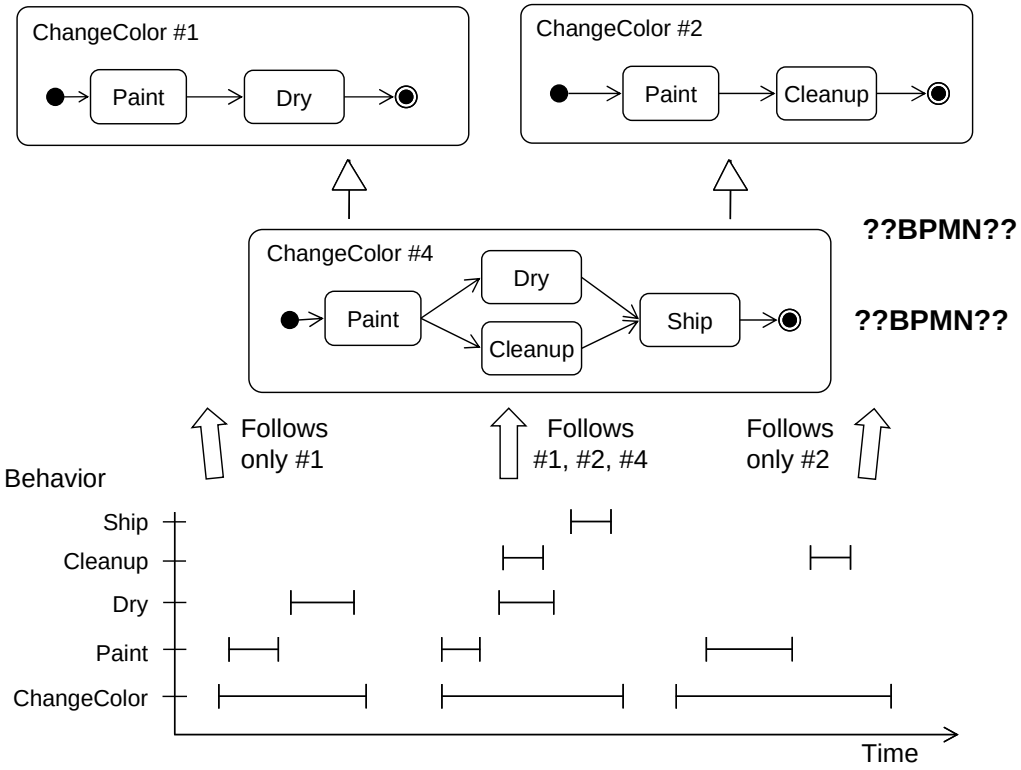


Figure 5: Multiple Behavior Generalization

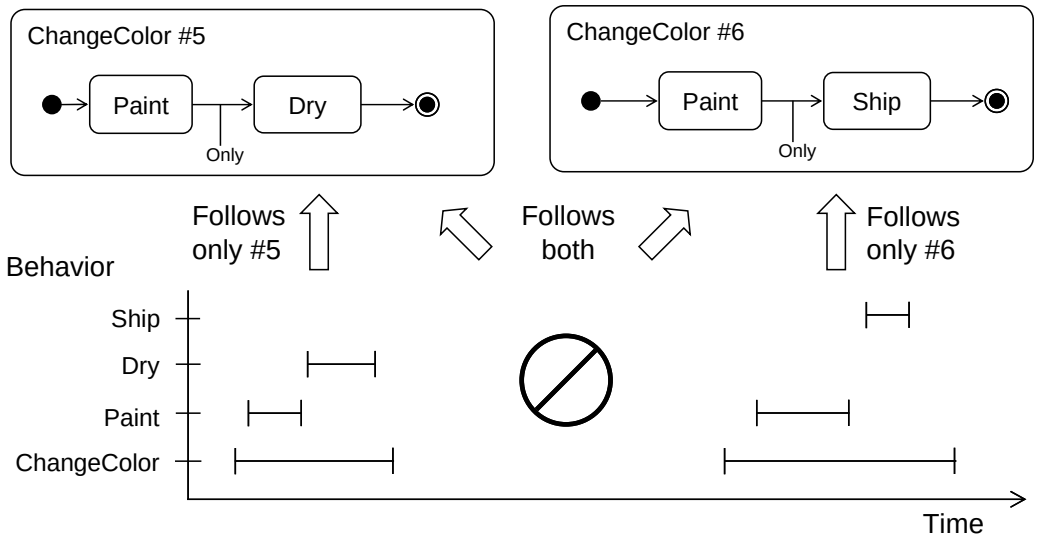


Figure 6: Inconsistent Behaviors

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 CHANGECOLOR #1 and CHANGECOLOR #2 overlap, with the occurrences following both definitions populating the intersection, per Figure 3. The oval for CHANGECOLOR #3 is completely contained in CHANGECOLOR #1, because CHANGECOLOR #1 generalizes CHANGECOLOR #3, per Figure 4. The oval for CHANGECOLOR #4 is completely contained in the intersection of CHANGECOLOR #1 and CHANGECOLOR #2, because they both generalize CHANGECOLOR #4, per Figure 5. Some occurrences in the intersection are not contained by CHANGECOLOR #4 because there can be occurrences satisfying CHANGECOLOR #1 and CHANGECOLOR #2, but without the shipping step required by CHANGECOLOR #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 CHANGECOLOR #5 and CHANGECOLOR #6 do not contain any occurrences because they require conflicting things of the occurrences, per Figure 6.

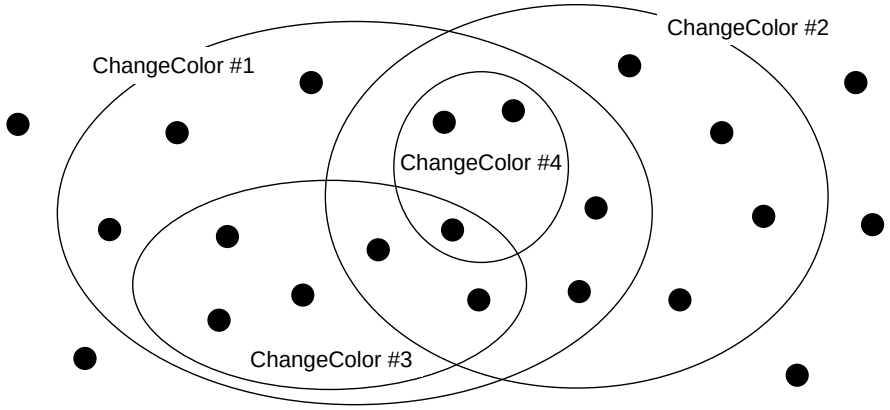


Figure 7: Venn Diagram for Figure 3 through Figure 5

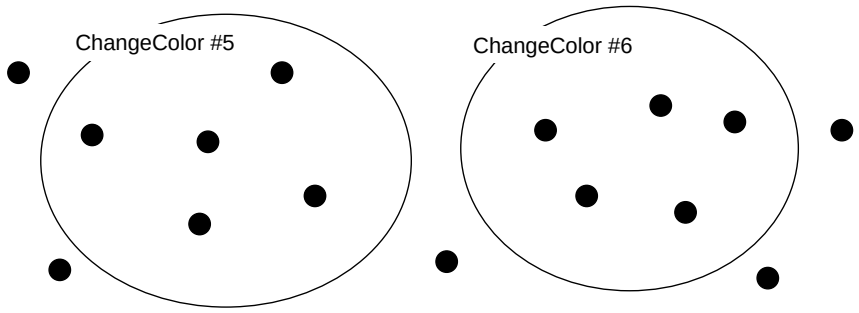


Figure 8: Venn Diagram for Figure 6

The examples in this section assume the behavior languages used in them specify how to tell which occurrences are allowed for any behavior written in the language (semantics). There are various ways to do specify semantics, ranging from informal to formal. This response suggests a semi-formal approach that treats behaviors as models of occurrences, where models capture constraints on intended occurrences of the behaviors. This has the advantage of being a more familiar specification technique than formal approaches, and can be augmented with more precise methods as needed.

3 Behavior Semantics in UML Currently[2]

The most basic aspects of UML behavior semantics is the same as described in the Section 2, expressed in a semi-formal way by specializing BEHAVIOR from CLASS in its metamodel (M2), as shown in Figure 9. The instances of user-defined behaviors (M1) are the occurrences (M0).[3] Behaviors can be generalized 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. Behaviors can support operations for managing execution, such as starting, stopping, aborting, and so on. They can have properties, such as how long the process has been executing or how much it costs, and 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, suspended, and so on.

[2] The rest of this response assumes familiarity with the UML metamodel and notation [3] and OMG's metalevel architecture [4].

[3] This assumes UML classes are treated as categories, rather than object-oriented classes (computation-independent semantics). The dashed arrows across levels identify elements falling into categories. The arrow between CHANGE COLOR #3 and BEHAVIOR is omitted for brevity.

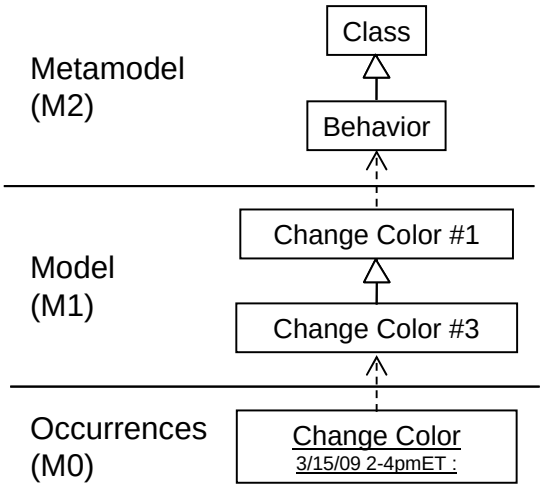


Figure 9: UML Behavior Currently

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 behaviors, except some terminology reuse between the Common Behavior semantic model in state machines, and in interactions conflated mostly with the model level. There is a general metamodel for time, time intervals, and durations, but it is not used to specify the semantics of behaviors, perhaps in part because it does not cover ordering in time. Interactions have an event ordering metamodel with a briefly described trace semantics, but it is not well developed or used for specifying the semantics of the other kinds of behaviors.

4 Behavior Semantics for UML Future

The semi-formal 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, CHANGECOLOR coordinates PAINT and DRY, in this case ensuring that occurrences of painting and drying happen during occurrences of changing color, and in the proper order.

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 CHANGECOLOR occurrences and PAINT occurrences.
2. Between suboccurrences to ensure suboccurrences happen the desired order, for example, between PAINT occurrences and DRY occurrences under CHANGECOLOR occurrences.

These are the whole-part and part-part relations of composition [5] applied to temporal relations between occurrences. They are addressed in Sections 4.1.3 and 4.1.4, respectively.

4.1.3 Whole-part for Behavior

UML has various concrete syntaxes for the first behavior coordination semantic (occurrence to suboccurrence), depending on the kind of behavior:

- 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, and messages and actions that compose behaviors indirectly.

UML does not have a common abstract syntax or semantics for the above, except some semantic elements in the mostly informal overview in Common Behavior, which is not used in specifying the specific 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 after the beginning of another, and the end of the first occurrence is before the end of the other). For example, the beginning of a CHANGECOLOR occurrence is before the beginning of its PAINT suboccurrence, and the end of the suboccurrence is before the end of its CHANGECOLOR occurrence. This can be captured as a property between occurrences as shown in Figure 10 at M1 (the type of PROPERTY is actually TYPE, but the semantics is same for the purposes of this response). The BEHAVIOR OCCURRENCE class is the most general behavior, provided in an M1 library. It generalizes all user-defined behaviors, and classifies all M0 occurrences.[4] It makes no constraint on occurrences at all, it allows all of them, like an intentionally empty behavior specification. The HAPPENSDURING association has BEHAVIOR OCCURRENCE at both ends. Any occurrence happening during another will be linked to it via HAPPENSDURING. (The HAPPENSBEFORE association is used for the second behavior coordination semantic, see below)

[4] Properties and operations for all occurrences can be defined on BEHAVIOR OCCURRENCE, such as their start and end times, and other examples described in Section 3.

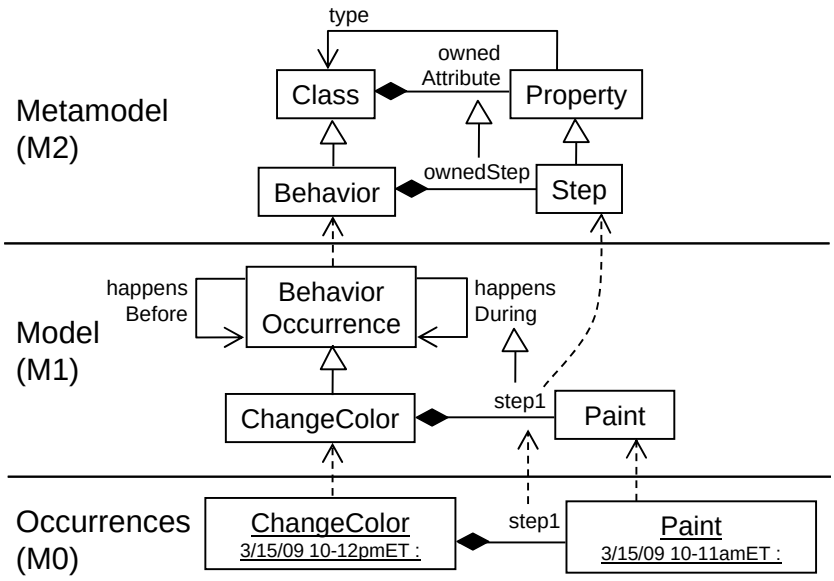


Figure 10: Step Properties

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 (generalization notation is used for property subsetting for brevity and readability). The types of step properties at M1 are the subbehaviors, such as PAINT being the type of the step1 property on the CHANGECOLOR behavior. Each occurrence of CHANGECOLOR 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 points to the shorter occurrence, 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 CHANGECOLOR to an occurrence of DRY.

4.1.4 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 general orderings between messages.

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

The basis for the second behavior coordination semantic above is some occurrences happen before others. Specifically, the time intervals of some occurrences are not overlapping and one is before the other (the end of one occurrence is before the beginning another). For example, the end of the DRY suboccurrence is before the beginning of the PAINT suboccurrence under CHANGECOLOR occurrences. This can be captured as a property between occurrences as shown in Figure 10 at M1. The HAPPENSBEFORE association has Behavior Occurrence at both ends. Any occurrence happening before another will be linked to it via HAPPENSBEFORE.

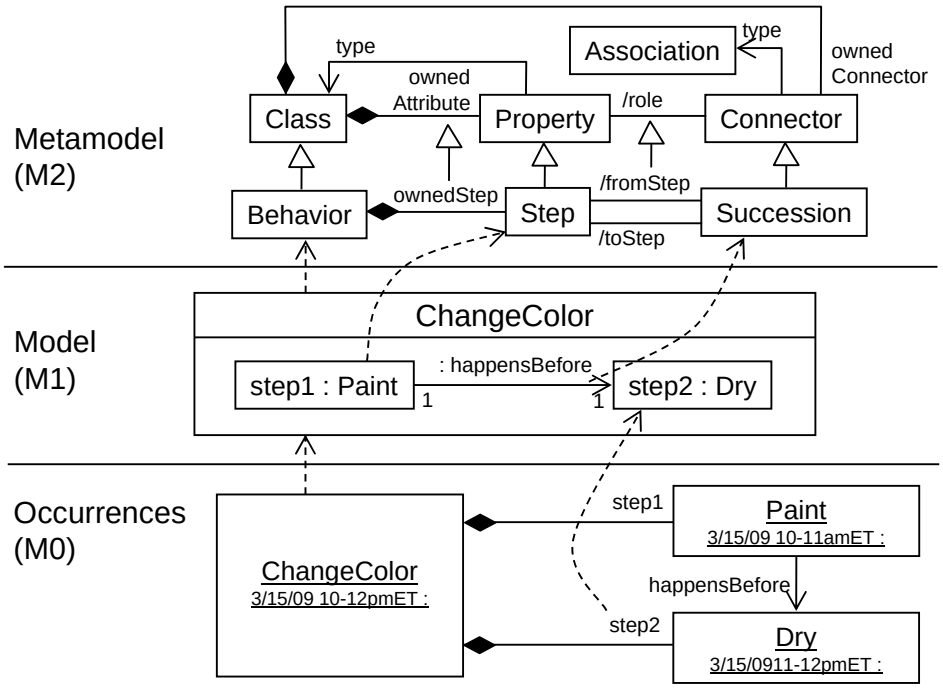


Figure 11: Succession Connectors

The HAPPENSBEFORE association must be "contextualized" to order suboccurrences in time, because the composed behavior will occur many times, and the ordering of suboccurrences must be limited to each one separately. For example, a factory will have many occurrences of CHANGECOLOR, but painting is only before drying under each occurrence separately. It is allowed to have drying suboccurrences before painting suboccurrences as long as they are under different CHANGECOLOR occurrences. This can be captured by specializing CONNECTOR to classify those at M1 that connect step properties and are typed by HAPPENSBEFORE, see SUCCESSION in Figure 11 at M2 (the /ROLE property is introduced to elide connector ends for readability).[5] Succession connectors between step properties require the occurrence values of those properties to be ordered in time by being linked by the HAPPENSBEFORE association. For example, the succession between STEP1 and STEP2 in CHANGECOLOR ensures the painting

[5] Figure 11 uses composites structure notation at M1 for readability, but it is not suggested as a concrete notation for UML behaviors, see footnote 1 in Section 1.

suboccurrence of each CHANGECOLOR occurrence happens before the drying suboccurrence under that same occurrence of CHANGECOLOR, rather than others.[6]

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 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.[7] 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:

- 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.[8] 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 loops, see next item.
  - 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

[6] This is the contextualization provided by connectors, as compared to using the HAPPENSBEFORE association directly between PAINT and DRY, which would require all occurrences of painting to be before drying, regardless of what occurrence of CHANGECOLOR they were under. This assumes contextualized association semantics of connectors, rather than message passing semantics [5][6][7].

[7] This is 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 successions, other links of HAPPENSBEFORE are created because HAPPENSBEFORE is transitive. Succession connector multiplicities do not apply to these links, see footnote 9.

[8] This is an example of how declarative semantics subsumes imperative by declaring existence of occurrences.

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.[9] 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. And with succession loops, STEP1 occurrence values could be linked to STEP2 occurrence values that happened earlier in the loop.

- 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.[10] 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.[11] 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

[9] Occurrence values of STEP1 can have other HAPPENSBEFORE links not due to successions, see footnote 7. For example, in the presence of succession loops, some values of STEP1 will link to multiple values of STEP2 due to the transitivity of HAPPENSBEFORE. Links created due to transitivity are not due to the connector, and are not restricted by connector multiplicity.

[10] See footnote 9.

[11] Occurrence values of STEP2 can have other HAPPENSBEFORE links not due to successions, see footnote 9. For example, in the presence of succession loops, some values of STEP2 will link backwards to multiple values of STEP2 due to the transitivity of HAPPENSBEFORE. Links created due to transitivity are not created due to the connector, and are not restricted by connector multiplicity.

semi-formally the "token" semantics informally described in some UML behaviors.[12] This can be summarized as the "array" formation of links due to connector ends with multiplicity of 1, see Figure 9.23 of [3].

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, one step with multiple outgoing successions to other steps means the first step happens before the others, while another step with multiple incoming successions from other steps means the last step happens after the others. Everything in the separate paths between these two steps have no time ordering constraints, they happen in parallel. Other more expressive coordinating constructs in UML go beyond successions, effectively constraining 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. 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 more formally with a constraint language on the occurrences [8].

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

[12] See UML activities, and though UML state machine semantics does not use the term "token," it uses quite a bit of Petri net terminology, and is effectively a token semantics.

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 semi-formally in a similar way to behaviors by specializing EVENT TYPE from CLASS in the metamodel (M2), as shown in Figure 12. The instances of user-defined event types (M1) are the event occurrences (M0).[13] 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, provided 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 associated 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.[14] 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 behavior occurrences, or both, when an event occurrence happens before a behavior occurrence. Event occurrences can happen during behavior occurrences, and event occurrences during event occurrences mean they happen at the same time.

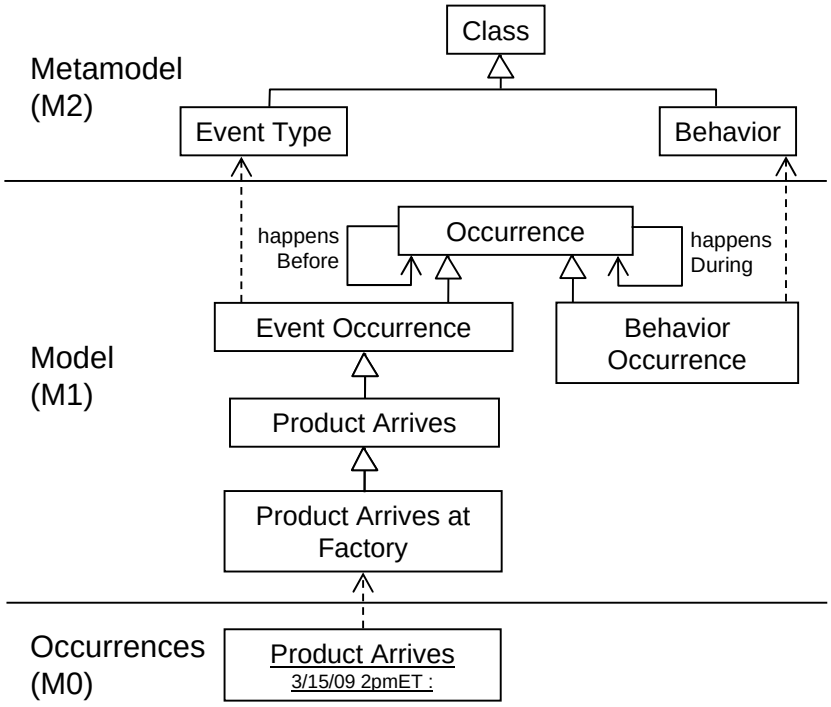


Figure 12: Event Types and Event Occurrences

[13] See footnote 3 in Section 3.

[14] Events might be treated as behaviors of short duration, where shortness is relative to the observer. For example, the assembly of a car might be considered an event when viewed in macroeconomic terms, but will be a behavior from the viewpoint of a factory worker.

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 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 CHANGECOLOR 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 CHANGECOLOR.

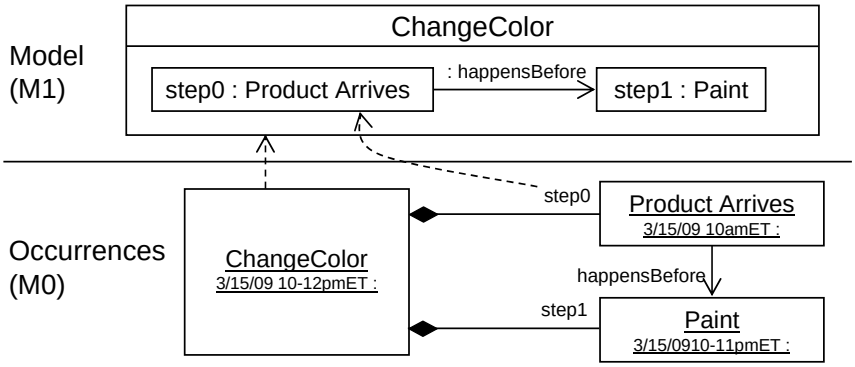


Figure 13: Event Steps

Behavior properties can include events about occurrences themselves, for example, when occurrences start, end, whether the end abnormally, and so on. These can be captured in taxonomy at M1, as shown in Figure 14. Behavior occurrences might end normally, but be successful in achieving their goals or not due to expected problems (normal ending), or might end forcibly from internal or external agents (abnormal 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 CHANGECOLOR 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 start before painting starts.

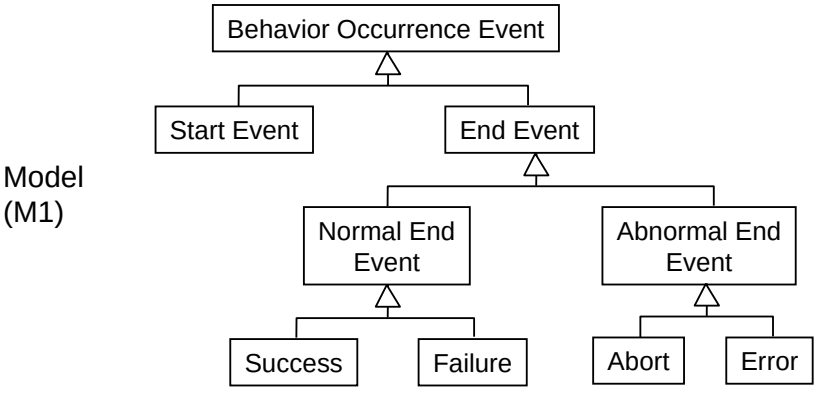


Figure 14: Behavior Event Taxonomy

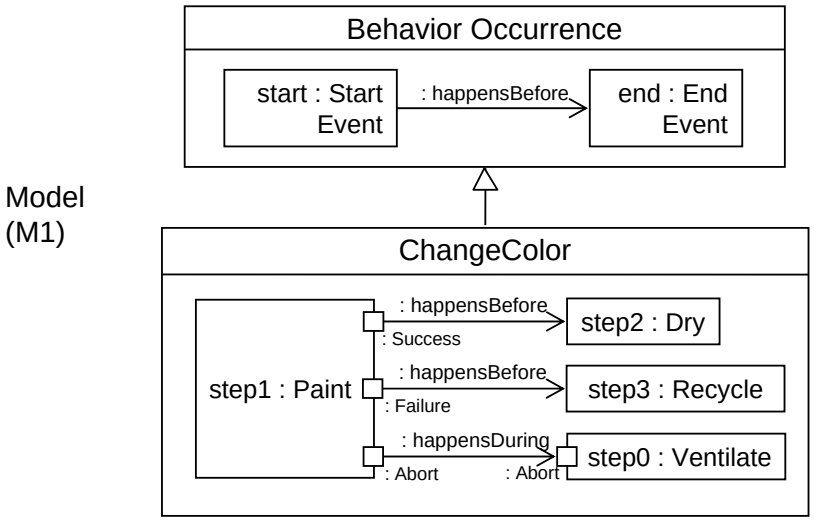


Figure 15: Successions between Behavior Events

Multiple behavior properties can be defined for 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 the UML sense of objects becoming aware of external events), in particular for notating how to respond to event notifications. State machines are semantically mostly a subset of the other kinds of UML behaviors, which enables concise notation, but an exception is pseudostates that machines "commit" to being accessible when they are used as submachines (entry and exit points).[15] 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, assuming the merge semantics of transitions is addressed, see end of Section 4.1.4.

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. This behavior can have a property specifying the type of objects having its color changed, properties specifying the types of tools and materials, and so on. Occurrences of this

[15] All states of submachines are accessible, but entry and exit points highlight that changes to them might affect other machines using them for access to submachine states.

behavior have values for these properties that are instances of the specified types, playing roles specified by the properties, for example, as the object having its color changed, the individual tools being used in that particular occurrence, and so on.

Interactions are behaviors involving objects that send messages to each other. Interactions can have properties specifying the types of objects involved, playing roles specified by the properties, for example, as the buyer interacting 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, a person who is buying and a company that is selling.

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, by definition, and that these objects can be identified by properties on association classes. A behavior can also be an association class by having some of its properties that identify objects involved in the behavior also identify end objects of the association class. For example, a behavior for changing color might 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 is needed for capturing the semantics of object flow and messaging, see Section 4.4.

UML does not currently have a standard way to classify properties of an association class that identify its end objects,[16] and the Meta-Object Facility (MOF) for representing UML models does not enable M1 behaviors to also be association classes.

The basis for a semantics of participants is their lifetimes can extend beyond those of the associations or behaviors they participate in. For example, 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. An object being painted in a factory participates in a color changing occurrence, but the object and the factory will exist before the occurrence starts and after it ends. A person interacting with a company to purchase a product, but the person and company will exist before the purchase begins and after it ends.

[16] The Systems Engineering Modeling Language (SysML) extends UML to support association participant properties [9].

The semantics of objects involved in behaviors and associations can be captured semi-formally by specializing PROPERTY to classify association properties at M1 that will have end object as values at M0, and another specialization to classify behavior properties at M1 that will have end object as values at M0, as shown in Figure 16.[17,18,19] 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. An example for a painting behavior might be participant properties identifying the tools and materials used.[20]

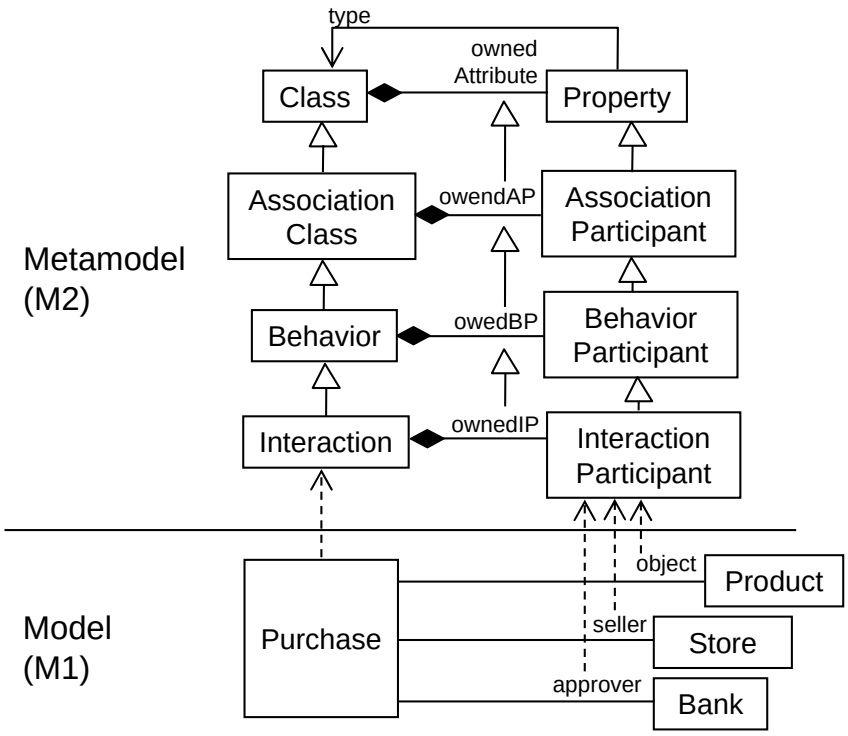


Figure 16: Participant Properties

[17] The metaclass names could include "Property" to clarify that the instances are not the actual participants, but only properties with M0 values that are the participants.

[18] This assumes link ends at M0 might not have values, as in as in [10][11] (UML currently requires link ends to have exactly one object). This is necessary because some behavior association classes will have more objects participating than are not link ends, for example, the transferred object in Section 4.4. It enables connectors to use only some of the association ends. Such a modification to association semantics could be avoided with MOF's upgrade to support multiple classification. Then M1 behaviors could additionally be classified as associations as needed, along with their participant properties, rather than using specialization BEHAVIOR from ASSOCIATION CLASS. The drawback of multiple M1 classification is the methodology for combining M2 classes is no longer clear. The figures use M2 specialization for brevity and readability.

[19] The property specializations could be redefinitions, to prevent participants of the general kinds on the specialized classes.

[20] In this example it would be useful if associations allowed more than one object per M0 link end, as in [10].

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 the transfer can be anything capable of referring to the things 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 referring to the object being painted or dried, and transfer means the removal of the value of one property and re-assignment of it to another. Object flow and messaging can involve also physical movement, for example, sending a package from one company to another.

The UML syntax for object flow and messaging appears in activities and interactions, respectively.[21] Object flows link pins on actions in activities. Any kind of thing can flow. Messages link lifelines in interactions 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.

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 of the object from painting to drying occurrences will take at least some time in the real world, 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, sending 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 semi-formally 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. It 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.

[21] All UML behaviors can capture the sending and receiving of messages, but this is only the beginning and the end of transfer of a message, rather than the transfer of the message.

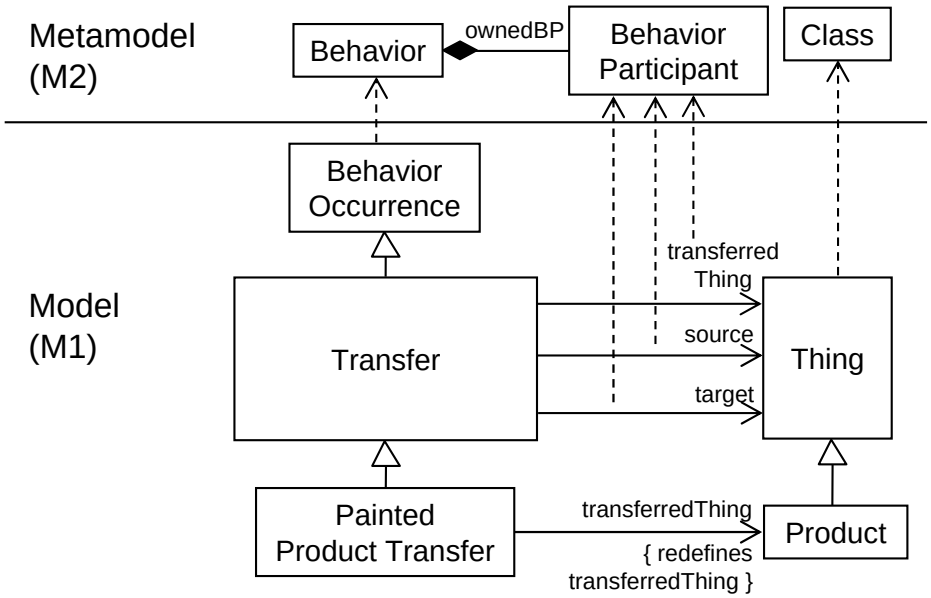


Figure 17: Transfers

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. The behavior occurrences participating in object flow transfers, like all participants, have lifetimes extending the transfers, see Section 4.3. 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 does, and the drying occurrence will end after the transfer.

Frequently transfers occur between behavior occurrences and objects, for example when crossing from internal business processes to interactions with other companies. It simplifies modeling to treat them the same way, with the difference implied by the kinds of source and target. For example, behaviors that accept inputs and provide outputs through object flow (described below), can be used to 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 behaviors to work across objects. UML does not have a common abstract syntax or semantics for object flow and messaging integration, and the semantics of this integration is specified informally.

Another aspect of the semantics of object flow and messaging is they are contextualized in the sense described in Section 4.1. Object flow is 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 semi-formalized by specializing CONNECTOR, as shown in Figure 17. They are unified under the general FLOW class, with the difference between object flow and messaging the kinds of sources and targets, as described above. Flow connectors at M1 are typed by TRANSFER or its

specializations, in CHANGECOLOR by PAINTED PRODUCT TRANSFER from Figure 17, and in PURCHASE transfers of credit card numbers, approvals, and products.[22] Flows have the property /TYPETRANSFERRED with values derived from the type of the TRANSFERREDTHING property of the connector type at M1.[23] This is specifies the type of thing being transferred, 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 is an object flow, because it connects properties identifying behavior occurrences participants in the flow, while the bottom model is messaging, it connects properties identifying objects.

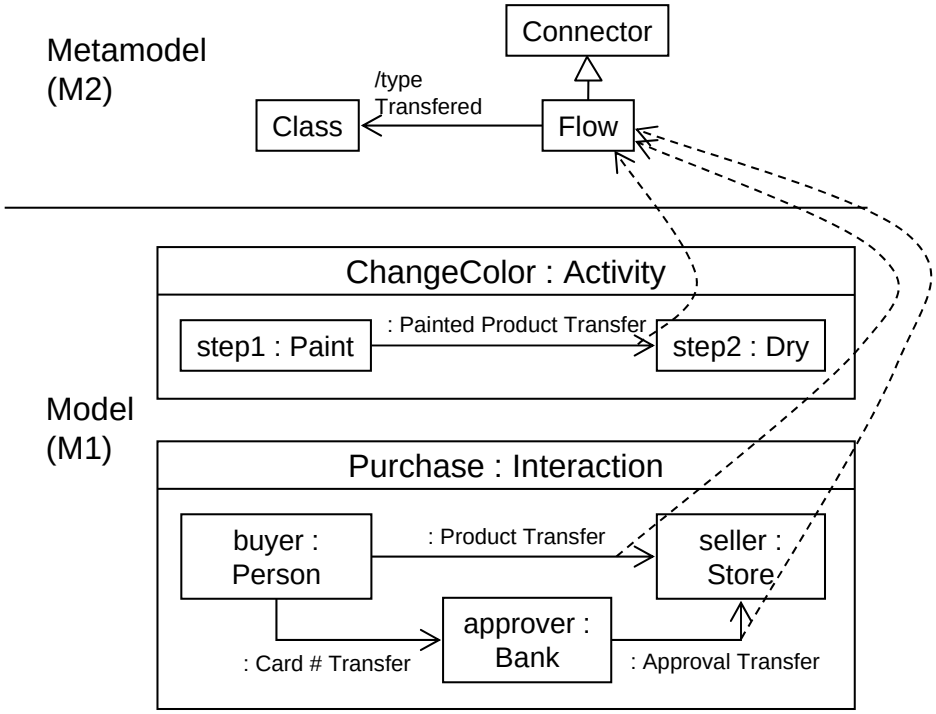


Figure 18: Flow Connectors

4.4.3 Inputs and Outputs

Some transfers come from and go to the "outside" a behavior ("inputs" and "outputs"). For example, a behavior for changing the color of objects in a factory will get the object to operate on 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 always add more participants to receiver and send more messages as needed, but interactions can have inputs and outputs to integrate with external behavior occurrences.

[22] The CHANGECOLOR behavior in Figure 18 is classified as an activity, which are the only UML behaviors that support object flows between occurrences.

[23] Alternatively TRANSFERREDTHING could be derived from TYPETRANSFERRED, they constrained against each other. OCL could be used instead of derived properties. Transfers that have no detail in them other than the thing that flows could potentially be omitted, leaving only the value of the TYPETRANSFERRED property at M2. The formality of the model is reduced, but the full model can be automatically created.

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, other than opaque behaviors).

Inputs and outputs are semantically a kind of transfer, requiring specialized participants representing the entities outside behaviors. This can be captured semi-formally by a specialization of BEHAVIOR PARTICIPANT, as shown in Figure 19.[24,25] In this example, the external entities for the changing color behavior are objects, a feeder from which products are drawn to be painted and a buffer to which they are put after drying completes. This highlights the flexibility of generalizing object flow and messaging. The transfer appears as an object flow to the occurrences of painting and drying, but as a message to the feeder and buffer.[26] The external entities can also be behavior occurrences, as typical in business process modeling or programming languages, for example. 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 being modeled.

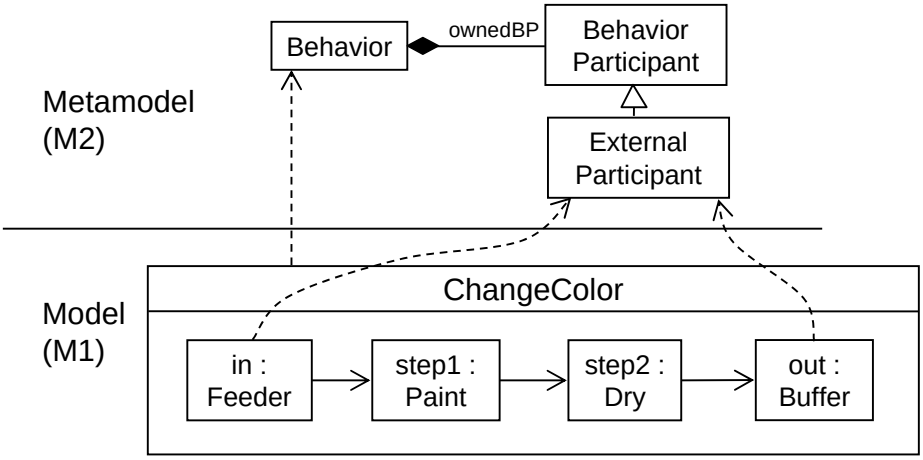


Figure 19: External Participants

The approach to inputs and outputs above is more expressive than parameters, because it can capture multiple roles played by entities outside the behavior, and has a uniform way to model 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 discussed next.

[24] 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, see below in this section. Then connectors would link the ports to (properties identifying) entities outside the behavior.

[25] This assumes a MOF supporting multiple classification, to support external interaction participants. Alternatively BEHAVIOR PARTICIPANT can have a Boolean property to indicate when a participant is external.

[26] This is why the CHANGECOLOR in Figure 19 is not classified as an activity, as in Figure 18, because UML activities do not support flows to external objects.

4.4.4 Flow Ordering

Transfers can be ordered in time and composed into larger transfers ("protocols"), like all occurrences, see Section 4.1. One transfer happening before another means one is completed before another starts, while a transfer happening during another means it starts and ends within the time interval of a larger transfer.

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 contextualized by special kinds of connectors (flows and successions, see beginning of Section 4.4 and Section 4.1 respectively). This means capturing the time order of transfers requires succession connectors between flow connectors. Since connectors are 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 generally useful to connect connectors, 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 just associations, to enable them to have links as values.[27] The links in a composite are treated the same as the objects in it. Figure 21 applies this to protocols by specializing FLOW from CONNECTORPROPERTY, rather than just 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.

[27] SysML extends UML to support connector properties [9].

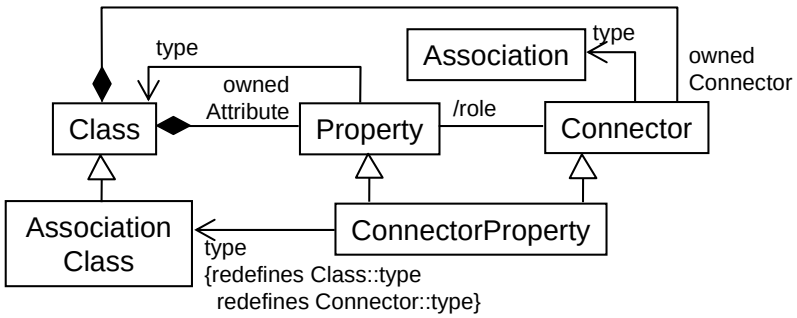


Figure 20: Connector Properties

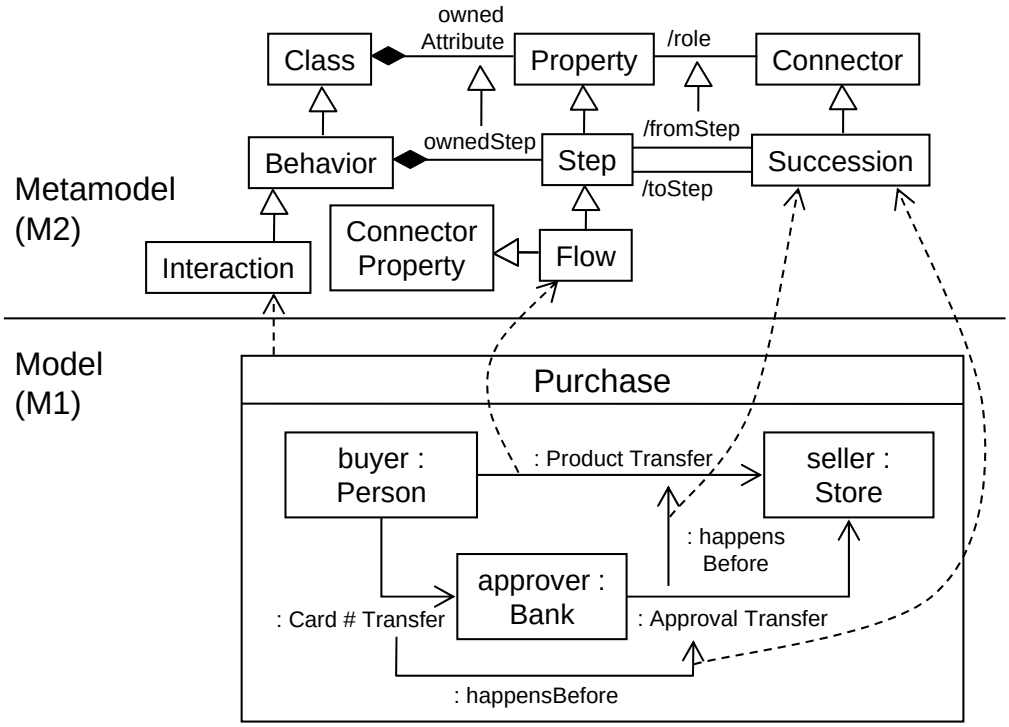


Figure 21: Flow Protocols

4.4.5 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 objects flowing into assembly. Similarly, a company might use a purchasing interaction with a particular position as the buyer, with credit cards issued from the currently contracted bank, and from currently approved stores. Coordinating behaviors specify "bindings" that specify how the flows and participants in the coordinated behavior are determined.

UML has various concrete syntaxes for flow and participant binding:[28]

- Activities have pins matching 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.

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 that the transfers into assembly occurrences will be the same ones out of painting occurrences. Similarly, a company using a purchasing interaction might require the person participating as the buyer be the same one participating in the company in the requisition position, and the credit cards used are the same ones supplied in current banking contracts.

An aspect of the semantics of bindings is they are contextualized in the sense described 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 semi-formalized by specializing CONNECTOR, as shown in Figure 22.[29] 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 CHANGECOLOR and ASSEMBLY by using them as types of steps. Bindings are directed, even though equality is symmetric mathematically, to prevent modification of reused behaviors. Using connectors this way in UML requires input and output flows to be ports, even though they are not notated this way in Figure 22. Port connector properties are a useful way to indicate which flows are accessible when a composite class is reused in another composite.

[28] State machines have binding-like constructs, but these are for time ordering, rather than transfers, see the end of Section 4.2.

[29] SysML extends UML to support binding connectors [9].

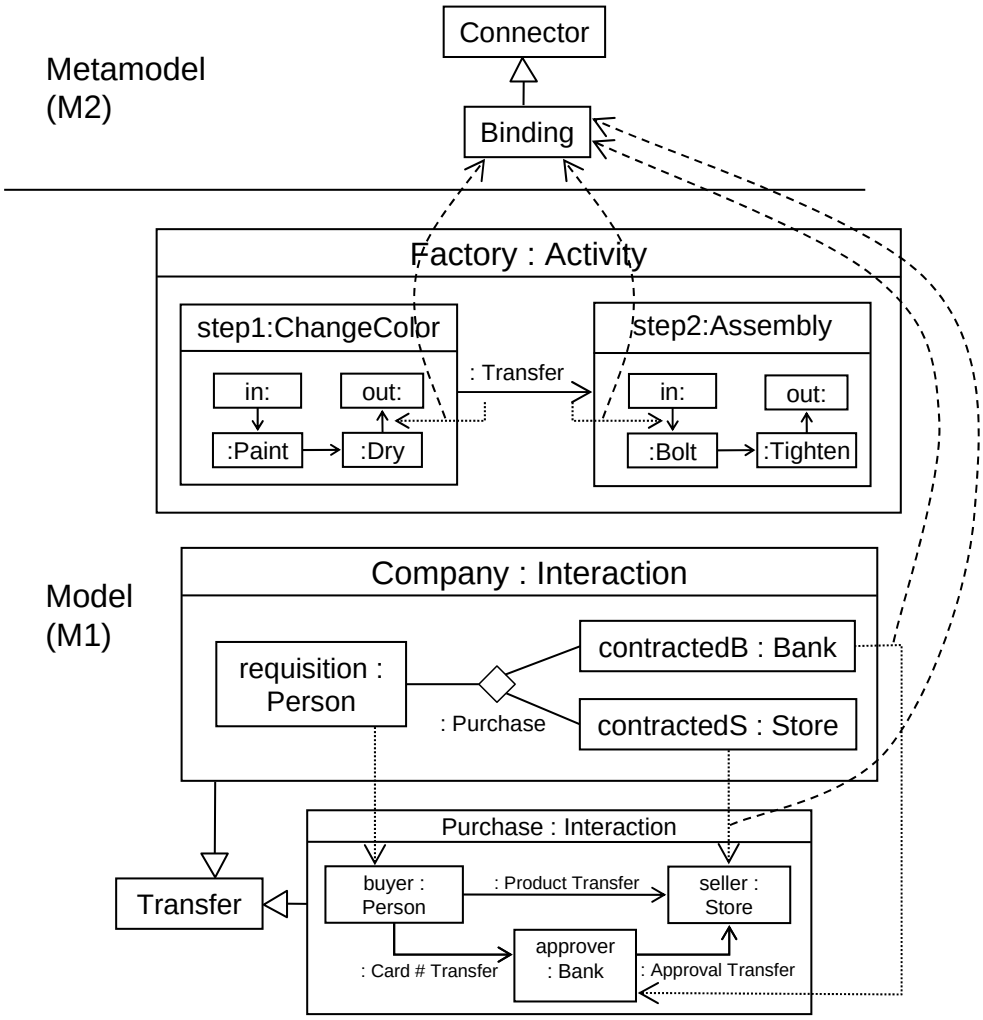


Figure 22: Binding Connectors

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 (flows just being 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 in the company. As in the factory example, using connectors this way in UML requires the participant properties to be ports, even they are not notated this way in Figure 22. Port participants are a useful way to indicate which participants are accessible when a behavior is reused in another behavior.

5 Other material

Other material about semi-formal unification of behavior modeling based on composite structure is available from the adoption of the Business Process Definition Metamodel (BPDM):

- BPDM Tutorial, Parts 1 and 2:
  http://doc.omg.org/omg/08-06-32
  http://doc.omg.org/omg/09-08-01

- BPDM/BPMN-S Design Rationale:
  http://doc.omg.org/bmi/2009-02-04, Section 4

- Rule-enable Process Modeling:
  http://doc.omg.org/bmi/07-12-03

- Execution Interoperability:
  http://doc.omg.org/bmi/2007-03-09

- BPDM specification:
  http://doc.omg.org/formal/2008-11-03, see Section 4.4 (Composition Model)
  http://doc.omg.org/formal/2008-11-04

A formal approach based on the semi-formal ones above is adopted in:

- Semantics of a Foundational Subset for Executable UML Models http://doc.omg.org/ptc/08-11-03, Section 10 (Base Semantics)

References

[1] Object Management Group, "Future Development of UML Request For Information," http://doc.omg.org/ad/2008-12-12, December, 2008.

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

[3] Object Management Group, "OMG Unified Modeling Language, Superstructure," http://doc.omg.org/formal/2009-02-02, 2009.

[4] Object Management Group, Unified Modeling Language: Infrastructure, http://doc.omg.org/formal/09-02-04, February 2009.

[5] Bock, C., "UML 2 Composition Model," in Journal of Object Technology, 3:10, pp. 47-73, http://www.jot.fm/issues/issue_2004_11/column5, November-December, 2004.

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

[7] U2 Partners, "2nd revised submission to OMG RFP ad/00-09-02: Unified Modeling Language: Superstructure," http://doc.omg.org/ad/2003-01-02, January 2003.

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

[9] Object Management Group, "OMG Systems Modeling Language," http://doc.omg.org/formal/2008-11-02, November 2008.

[10] Flatscher, R., "Metamodeling in EIA/CDIF-Meta-Metamodel and Metamodels," Association of Computing Machinery Transactions on Modeling and Computer Simulation, 12:4, pp. 322-342, October 2002.

[11] Bock, C., Odell, J., "A More Complete Model of Relations and Their Implementation, Part I: Relations as Object Types" Journal of Object-Oriented Programming, 10:3, pp. 38-40, http://www.conradbock.org/relation1.html, June 1997.