NISTIR 8283

# Verifying Executability of SysML Behavior Models Using Satisfiability Modulo Theory Solvers

Raphael Barbau
Conrad Bock

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8283

National Institute of
Standards and Technology
U.S. Department of Commerce

NISTIR 8283

# Verifying Executability of SysML Behavior Models Using Satisfiability Modulo Theory Solvers

Raphael Barbau
*Systems Integration Division*
*Engineering Laboratory*
*Engisis LLC*
*Bethesda, Maryland*

Conrad Bock
*Systems Integration Division*
*Engineering Laboratory*

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8283

February 2020


U.S. Department of Commerce
*Wilbur L. Ross, Jr., Secretary*

National Institute of Standards and Technology
*Walter Copan, NIST Director and Undersecretary of Commerce for Standards and Technology*

**National Institute of Standards and Technology Interagency or Internal Report 8283**
**Natl. Inst. Stand. Technol. Interag. Intern. Rep. 8283, 85 pages (February 2020)**

**This publication is available free of charge from:**
**https://doi.org/10.6028/NIST.IR.8283**

# Verifying executability of SysML behavior models using Satisfiability Modulo Theory solvers

Raphael Barbau*[2,1] and Conrad Bock†[1]

[1]National Institute of Standards and Technology, Gaithersburg, USA
[2]Engisis LLC, Bethesda, USA

February 2020

**Abstract**

This report presents an approach to verifying executability of system behavior models by treating them as logical constraint problems solved using Satisfiability Modulo Theory (SMT). With behavior models interpreted as logical constraints on execution order, these solvers can determine whether the models are executable by finding executions that meet those constraints, or not executable because they are overconstrained. The approach relies on Ontological Behavior Modeling (OBM) to unify behavior modeling in the Systems Modeling Language (SysML) under a logical framework based on its structural elements. Translation patterns are proposed between SysML structural models, OBM, and logical constructs in SMT-LIB, a language used as input to SMT solvers. Software developed based on these patterns automatically translates SysML models extended with OBM into SMT-LIB files. Finally, the approach and software are demonstrated by translating and solving example SysML behavior models.

## 1. Introduction

Engineered systems have an increasing number of components that behave and interact in increasingly complex ways. This is tackled with computerized models providing automated support to system design. System models can be automatically processed and transformed to derive new knowledge, including detection of errors that could be missed in engineering reviews (verification). This document focuses on verification of system model executability.

*barbau@nist.gov
†conrad.bock@nist.gov

The Systems Modeling Language (SysML) is a widely used graphical language for specifying systems [1]. It extends the Unified Modeling Language (UML) [2], a widely-used graphical language for specifying software.[1] SysML was created for systems engineers, who are responsible for coordinating activities of other engineers (mechanical, electrical, production, and so on). SysML includes elements for describing structural aspects of systems, and also has three ways to specify system behaviors: activities, state machines, and interactions. For example, activities can describe a sequence of actions taken on a product as it moves through a factory, state machines can describe states of machine tools, and interactions can describe how machine tools communicate.

The three behavior modeling techniques in SysML were originally developed separately, and then brought together in one language. This resulted in a lack of integration among them, with the same capabilities offered in different ways, and unique capabilities unable to be mixed in one diagram. To address these problems, Ontological Behavior Modeling (OBM) was developed to centralize aspects common to these ways of modeling behaviors [3]. This method uses elements of SysML usually only applied to structure (whole-part and part-part relationships) to model behaviors in these three ways, capturing what they have in common, and building on this to reflect their differences.

While the syntax of SysML imposes restrictions on how models can be built (using a restricted set of elements and relationships, and constraints on how they are assembled, and so on), it is still possible for behavior models to be have no possible executions due to the semantics SysML gives for its model elements (language semantics gives constraints models place on their allowed instances, in this case, their executions). Executability can be determined by attempting to find instances of a model, as allowed by the constraints of SysML semantics. If the attempt succeeds, the model is executable, otherwise it is not. In the field of logic, the possibility for a model to have instances meeting its language constraints (semantics) is the satisfiability of the model, and finding an example of those instances is satisfiability solving. To find out whether a behavior model is executable, an approach based on satisfiability expresses this model in logical statements, and then uses a solver to find instances that are consistent with all the logical statements.

Satisfiability is an old field of computer science that was originally limited to Boolean variables and expressions without quantifiers (SAT). Subsequent extensions introduced theories, which add new kinds of variables (e.g., Real, Integer, Arrays, custom datatypes) and predicates (e.g., equalities, inequalities) to use in Boolean expressions. Satisfiability Modulo Theories (SMT) are satisfiability problems that include such theories[4]. SMT-LIB includes a de-facto standard language for giving input to all the SMT solvers, as well as a large library of benchmark problems[5, 6].

This report describes a process for automatically transforming SysML behavior models into logical statements and checking satisfiability by finding solutions using

[1]The remainder of the paper will refer to SysML/UML as SysML, for brevity.

SMT solvers. A translation from a subset of SysML to SMT-LIB was proposed by the NASA Jet Propulsion Laboratory (JPL) [7]. That work focused on aspects of SysML typically used only for structure, which can be applied to complex behavior when combined with OBM. This report presents an extension of JPL's work to translate SysML behaviors extended by OBM into SMT-LIB. Section 2 reviews behavior modeling in SysML and OBM's unification of its techniques. Section 3 gives a detailed translation from SysML/OBM into SMT-LIB. Section 4 applies the translation to example behaviors, giving results of SMT verification. Finally, Section 5 presents future work.

## 2. Behavior modeling in SysML and OBM

This section briefly describes behavior modeling in SysML (Section 2.1), as well as an implementation of OBM that covers commonalities between these behavior modeling techniques (Section 2.2).

### 2.1 Behavior modeling in SysML

This subsection describes the main SysML concepts for representing system behaviors. Behaviors in SysML can be specified in three ways: activities describe sequences of actions in a process, state machines describe the states of a behavior and its reactions to external stimuli, and interactions describe messages exchanged between participants. Each method has its own constructs with corresponding diagrammatic notation. The following paragraphs will briefly present overlapping features offered by these methods.

**Composition** All three methods can compose behaviors from others. Activities have actions, some of which can call other behaviors (call behavior actions). State machines have states, which can nest other state machines (submachines). Interactions have messages grouped in fragments, some of which can use another interaction (interaction use fragments). In each case, when a behavior is executed, the lifetimes of executions that are part of a composition are within the lifetimes of the execution they are part of.

**Time ordering** All three methods can order behaviors in time: Activities have control flows between actions, state machines have transitions between states, and interaction fragments have general orderings between the start and end of messages. Time ordering can be further detailed by specifying alternatives, parallelism, and looping:

- Activities have decision nodes to select among alternative next actions, and merge nodes between any alternative previous actions and the next action.

State machines have choice and junction pseudostates for alternative transitions. Interactions have alternative interaction fragments within *alt* combined fragments.

- Activities have fork nodes for multiple next actions, and join nodes between multiple parallel actions and the next action. They also have parallel expansion regions. State machines have fork and join pseudostates for parallel transitions to states in other regions. Interactions have parallel interaction fragments within *par* combined fragments.
- Activity control flows can form a loops. Activities also have loop nodes and iterative expansion regions. State machines transitions can also form loops. Interactions have iterated interaction fragments within *loop* combined fragments.

**Start and end** Two methods have constructs for the start and end of behaviors. Activities have initial nodes and final nodes, while state machines have an initial pseudostate and a final state.

**Participants** Two methods can specify objects that participate in behaviors. Activities have partitions, while interactions have lifelines.[2]

**Transfers** Two methods can specify flow of items. Activities have object flows between object nodes, which can be pins on actions, stand on their own between actions (central buffer nodes), or be on the boundary of activities (parameter nodes), all of which can specify the type of item flowing. Interactions messages have arguments, which are also typed.

### 2.2 OBM extension

OBM describes behavior using SysML concepts that are usually only applied to structure. Section 2.2.1 reviews these concepts and their commonalities with the behavioral concepts introduced in the previous subsection.

#### 2.2.1 Applying structural concepts to behavior

System structure in SysML describes the kinds of components a system is made of (whole-part relationships), and how they are interconnected (part-part relationships). SysML takes a logical approach to this, treating system models as classes (sets) of the systems to be built (instances). SysML uses the term "block" instead of "class", as UML does, but this report uses "class." Classes can have properties that each instance

[2]State machines react to stimuli from elsewhere rather than from external participants that might have caused those stimuli. They can be used to specify behavior of each participant separately from their interaction.

can give values for. These values must be instances of the type of the property (a class or a datatype), with the number of values in each instance constrained by the multiplicity of the property. Property values often identify objects (instances of classes), but can be data such as numbers or booleans (instances of datatypes). An association is a relationship between two classes that specifies a property of each class to identify instances of the other class (association end properties). Instances of associations are links between instances (objects) of the associated classes, with each class instance identifying the other via an end property. Classes can have connectors typed by associations, which specify links between values of properties of the same object. Connectors are part-part relationships, and the connected properties are part-whole. A SysML model of these concepts is shown in Figure 1.

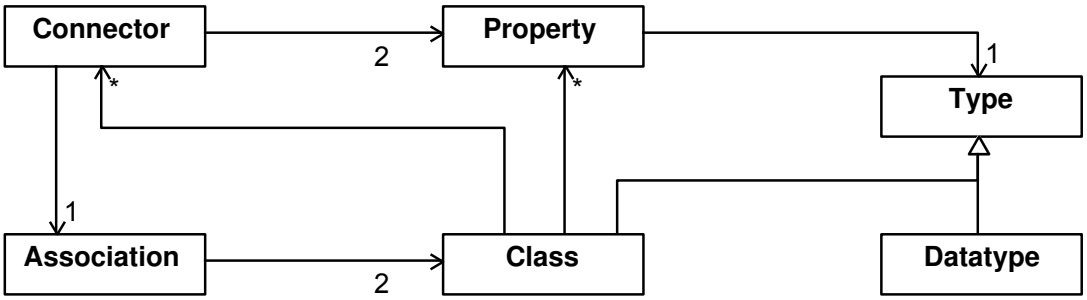


**Fig. 1.** SysML structural concepts

The three ways of modeling behaviors in SysML have a lot in common, as explained in the previous subsection. They also overlap structural modeling, because these behaviors are also classes (blocks in SysML), with their instances being executions of behaviors. OBM was developed to bring out more commonalities between behavior and structure modeling in SysML. OBM treats:

- Nodes, states, and participants/executions as composite properties (whole-part relationships)
- Edges, transitions, and messages as connectors (part-part relationships), with connector end multiplicities modeling parallel and alternative flows

### 2.2.2 OBM models

To facilitate modeling behaviors as structure, users can indicate which structural elements are also behavioral. There are two ways to achieve this (more information in [8–10]):

- Extend the modeling language to add more specialized language elements. Most UML tools do not support extending the UML metamodel directly, so UML

provides profiles with stereotypes to extend UML metaclasses (such as Class, Property, Connector). Users apply stereotypes to model elements to classify them by these extensions.

- Create or extend model libraries and use them in new models. UML does not have model libraries, but users can create classes, properties, and associations, in libraries available to other modelers. These modelers type their properties and connectors by library classes and associations, respectively, and extend library elements by specializating classes and associations, or subsetting and redefining properties, as needed.

The implementation of OBM in this report uses both the above techniques: behavior elements are classified by library and metamodel elements (as stereotypes) simultaneously. Language extensions are required for adding properties to language elements or when a classification should apply to an element but not its specialization (e.g. Interface stereotype). Otherwise, language extensions are not needed except for readability, and library elements can be used instead.

Figure 2 and 3 show the OBM library and the OBM profile used in this report, respectively. The root class of the library is Anything, all other library classes (indirectly) specialize it. Occurrence is a specialization of Anything classifying things that happen in time (objects and performance of behaviors). BinaryLink is an association between Anything and itself (specifying links between two things or between one thing and itself). Stereotype extension relations (filled arrowheads) in Figure 3 indicate which UML metaclass a stereotype extends (specializes). Generalization relations (empty arrowheads) indicate which other stereotype a stereotype specializes.

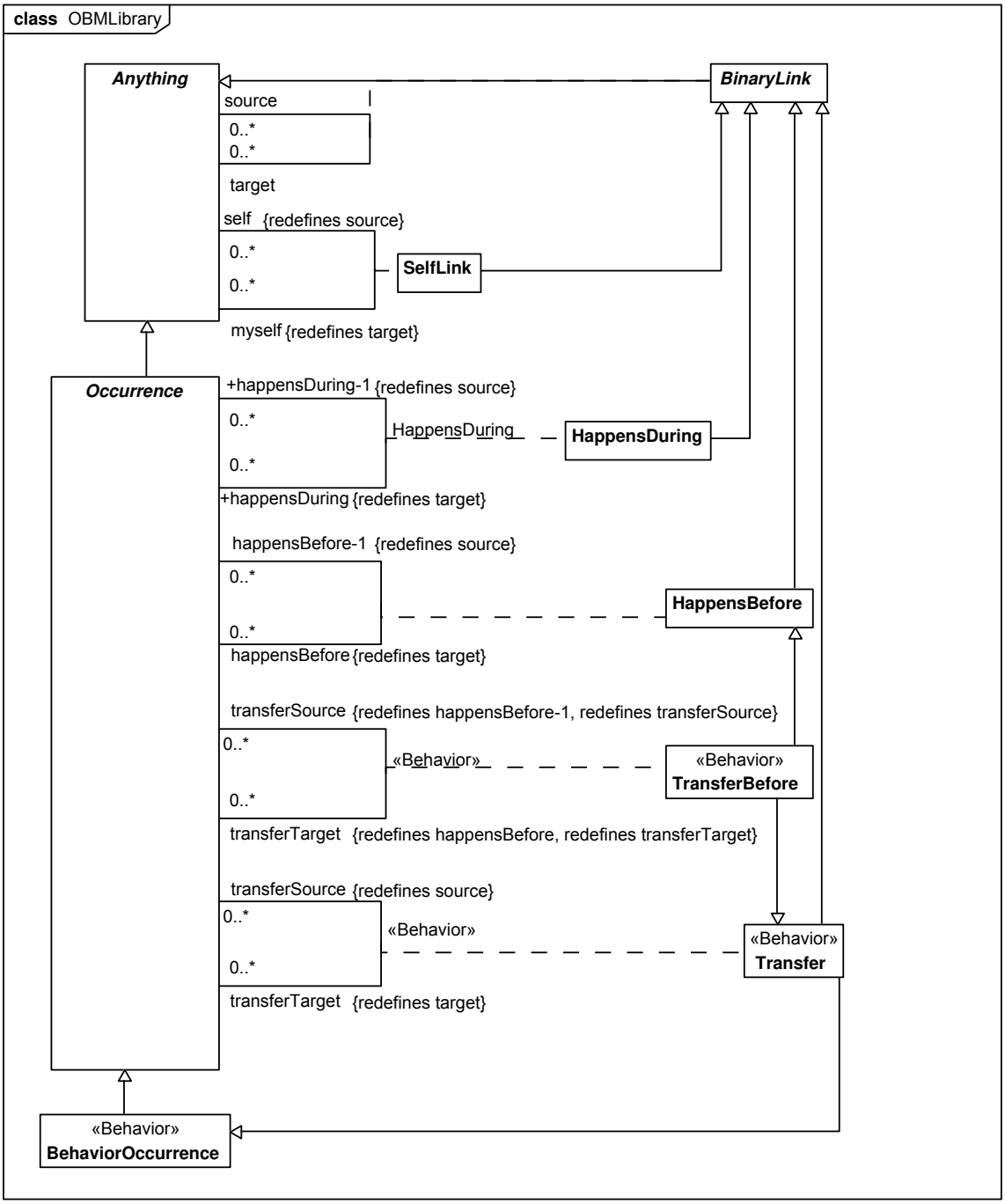


**Fig. 2.** OBM Library

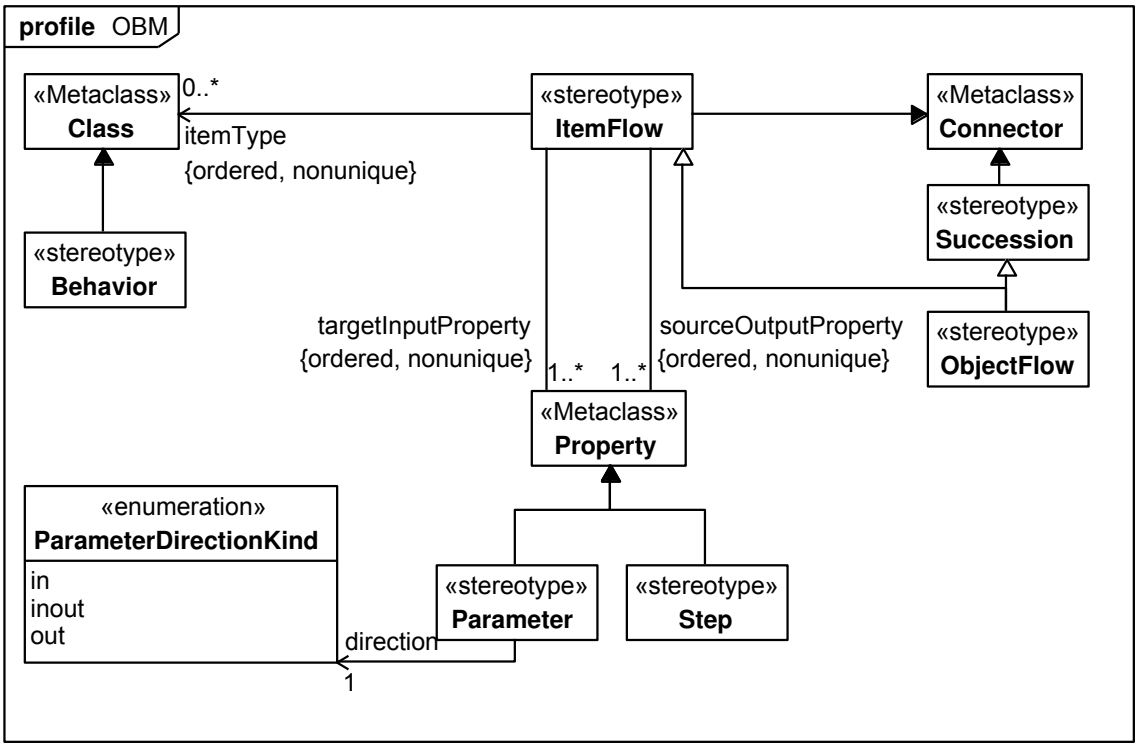


**Fig. 3.** OBM Profile

**Behaviors** The BehaviorOccurrence class (from the OBM library) specializes Occurrence for performances of behaviors. BehaviorOccurrence and its specializations have the Behavior stereotype (from the OBM profile) applied[3].

Specializations of BehaviorOccurrence can have properties typed by BehaviorOccurrence or a specialization of it, in which case the properties have the Step stereotype applied. When a property is typed by BehaviorOccurrence, its multiplicity indicates how many times the step must or might be executed during the owner's execution. Steps with multiplicity 0..* might occur multiple times, but also might occur once or not at all if there are other constraints on the steps, including others due to other steps in same behavior. An exception is initial nodes, which occur exactly once per execution of the behavior (multiplicity 1).

**Temporal relations** OBM treats Occurrences as time intervals, providing two temporal relations between them modeled as associations:

- HappensBefore links occurrences that are separate in time (except possibly overlapping at one time point), one happening before the other.

[3]A stereotype is needed because UML Behavior is abstract, it cannot directly classify model elements.

- HappensDuring links occurrences that are completely overlapping, one happening completely during the other (including happening exactly when another does).

Logical characteristics of these relations are:

1. Both temporal relations are transitive.
2. Both are asymmetric, except in these cases:
   (a) HappensBefore is symmetric between occurrences of zero duration that happen at the same time, which means it is reflexive for zero duration occurrences (all zero duration occurrences overlap themselves at one time point).
   (b) HappensDuring is symmetric between occurrences that happen at the same time, which means it is reflexive (all occurrences happen at the same time as themselves).
3. When occurrences are related by HappensDuring, all HappensBefore relations involving the occurrence of longer (or equal) duration also apply to the other occurrence.

These associations are adapted from Allen's interval logic [11]:

- Excluding the symmetric cases above, HappensBefore is equivalent to the union of Allen's before and meets interval relations, which are for intervals completely separate in time and overlapping at exactly one point, respectively. Allen's meets and equals are disjoint, preventing time intervals from having zero duration.
- HappensDuring is equivalent to the union of Allen's starts, during, finishes, and equals interval relations.

Restricted versions of these are specified in translation to SMT solvers (see Section 3.4) and applied to detecting inconsistency in models using these associations.

Connectors typed by HappensBefore and HappensDuring specify temporal relations between values of the connected properties, which are expected to be steps (properties typed by BehaviorOccurrence). Connectors typed by HappensBefore specify links where the source (a value of the first connected property) occurs before the target (a value of the second connected property). These connectors have the Succession stereotype applied. Connectors typed by HappensDuring specify links where the source occurs during the target. A HappensDuring link is implied when a step (property typed by BehaviorOccurrence) is owned by a specialization of BehaviorOccurrence.

Connector end multiplicities indicate how many links each value of the connected properties might/must have. For example, two common connector end multiplicities are:

- 1..1: each value has exactly one link for that connector.

- 0..1: each value has zero or one link for that connector.

Exclusivity constraints are currently represented as a OneOf constraint on connector ends of 0..1 multiplicity. This ensures that each value of a property with multiple incoming or outgoing connectors of 0..1 multiplicity will have exactly one link for one of the connectors, and 0 for the others. Connector end multiplicities are used to represent control nodes, see Section 4.1.1.

**Transfers** Transfer is an association between Occurrence and itself, specializing BinaryLink and BehaviorOccurrence (since transfers are performances of behaviors, rather than objects). Interactions between participants are modeled as connectors typed by Transfer (or one of its specializations).[4] These connectors have the ItemFlow stereotype applied, which places additional restrictions on their transfers:

- itemType: kind of items flowing
- sourceOutputProperty: properties from which the items are coming (items are values of these properties at the time a transfer begins)
- targetInputProperty: properties to which the items are going (items are values of these properties at the time a transfer ends).

**Object flows** TransferBefore is an association between Occurrence and itself, specializing Transfer and HappensBefore. It specifies flows between occurrences that happen after the source ends and before the target begins. This ensures transfers are between the same occurrences that are in temporal order. Connectors typed by TransferBefore have the ObjectFlow stereotype applied.

**Behavior parameters** Behavior may have parameters, which are properties that are accessible to other behaviors. Parameters are modeled as properties stereotyped by the Parameter stereotype. Parameters have a direction specifying whether the parameter value can be read (in), written into (out), or both (inout)[5]. The return direction value is not used in this implementation.

## 3. Translating SysML behaviors to satisfiability problems

This section explains how we translated OBM-based SysML behaviors into satisfiability problems. Section 3.1 introduces satisfiability and its variants, Section 3.2 introduces SMT-LIB, Section 3.3 describes how to translate basic SysML structural features (as needed by OBM) into SMT-LIB, Section 3.4 describes how to translate OBM-specific

[4] Transfer connectors can be temporally ordered using a HappensBefore connector between their adjunct properties.

[5] A stereotype is needed because UML Parameter does not specialization Property

content into SMT-LIB, and finally Section 3.5 describes how the translation was implemented as a software.

### 3.1 Introduction to satisfiability

In the field of logic, finding such instances (interpretations) is called solving a satisfiability problem. Boolean satisfiability (SAT) refers to satisfaction of propositional logic formulas. These single formulas are constructed with Boolean variables[6], and the three logical operators (not, and, or). The SAT solver then attempts to find an interpretation (assignment of values to all variables) that satisfies the formula. In terms of complexity, finding satisfying interpretations to SAT problems is NP-complete, which means no polynomial-time algorithm exists that finds solutions to all SAT problems, but a polynomial-time algorithm exists to check solutions if they are found for individual problems. In addition, algorithms are available that find solutions to a many SAT problems.

Satisfiability Modulo Theories (SMT) is an extension of SAT in which various theories can be used to add non-Boolean variables and functions. Examples theories are linear arithmetic, arrays, datatypes, etc. Theories should be decidable, meaning that an algorithm exists to find a satisfying interpretation or determine that no such interpretation exists, for all expressions and in finite time. This is however not the case with theories like non-linear integer arithmetic. Theories work with quantifier-free formulas (additional methods are needed to take care of quantifiers).

The overall objective of this work is to check whether behavior specifications can have instances (be executed, are satisfiable). Instances found by SMT solvers do not change over time, unlike procedurally generated instances of UML/SysML models. The SMT solution is a collection of variable assignments given all once, rather than over time. Changes to instances (variables) that satisfy a model might make them no longer satisfactory. Satisfiability solvers can still work with temporal models, however, finding instances that are ordered or nested in time, or that happen at specific points in time.

### 3.2 Introduction to SMT-LIB

SMT solvers combine a SAT solver, decision procedures that work for particular theories (quantifier-free), and a module for quantifiers specifically. Many SMT solvers have been developed, and a textual language called SMT-LIB serves as input for all these solvers. Hence, using SMT-LIB as a target language for translation of behaviors guarantees that the result can be read by many tools[7].

[6]These are called variables because they are assigned values by solvers, once per solution, rather than because their values can change over time.

[7]Small variations currently exist for newer capabilities, but are expected to be reconciled by the solvers in the future.

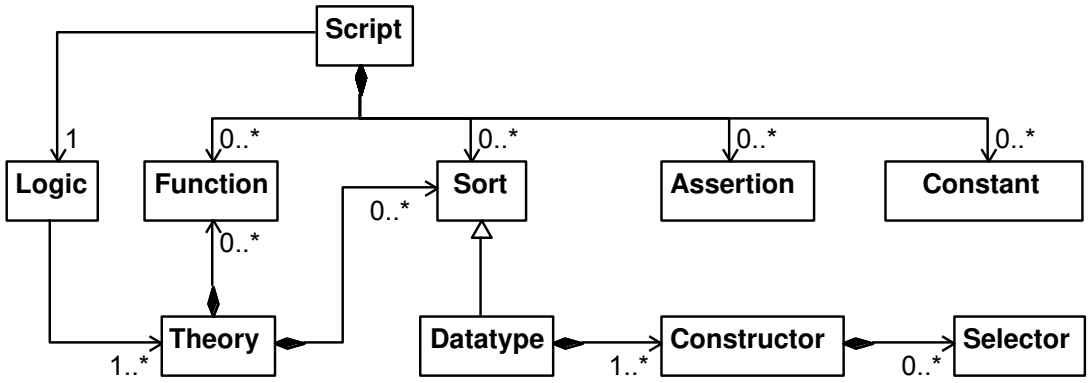


**Fig. 4.** SMT-LIB concepts

An overview of the SMT-LIB concepts is shown in Figure 4. An SMT-LIB file defines scripts, which are sequences of commands. The main commands declare and/or define functions and constants[8], as well as sorts (what programming languages call types, and SysML calls datatypes). Examples of sorts include Boolean, Real, Integer, Array, or user-defined datatypes. Functions are mappings from argument values to a result value. Constants are elements that have a value. Datatypes are kinds of sorts that have one or more constructors. Each constructor has zero or more selectors, which are characteristics (properties, fields) of the datatype. All function arguments, function results, datatype selectors, and constants are assigned a sort that restricts the values they can have. Assertions are logical expressions that must hold true for all solutions. Solving an SMT problem finds assignments for functions and constants for which the assertions are true: constants are given a value, and functions are given a mapping between function parameters and the returned value.

SMT-LIB defines the following logical functions: and, or, not, eq (equality), distinct, => (implication), and ite (if-then-else). SMT-LIB also defines the some and forall quantifiers.

### 3.3 Translating SysML structural concepts to SMT-LIB

This section presents the SMT translation of SysML structure (that is extended by OBM, see 3.4), mostly following the approach in [7]. Section 3.3.1 explains SMT-LIB logics and their selection. Sections 3.3.2, 3.3.3, 3.3.4, and 3.3.5 address translation of SysML primitive datatypes, blocks, collections, SysML properties, and associations, respectively. Section 3.3.7 covers translation of operations defined by constraints, while Section 3.3.8 presents translation of constraints on instances. Section 3.3.9 presents the final steps to check the satisfiability of translated models.

[8]These are called constant because their value cannot change over time.

### 3.3.1 Logic selection

The beginning of SMT-LIB scripts usually specify which parts of SMT-LIB to use (particular theories, absence of quantifiers), which are called logics, supporting more efficient solvers in some cases. SMT-LIB does not currently define a logic for datatypes, so we select the ALL logic defined by the solver we use (Z3) to indicate no restriction:[9]

```
(set-logic ALL)
```

### 3.3.2 Primitive datatypes

SMT-LIB has sorts corresponding to the major SysML primitive datatypes. The correspondence between the two is as follows:

- SysML Real: SMT-LIB Real
- SysML Boolean: SMT-LIB Bool
- SysML Integer: SMT-LIB Int

### 3.3.3 Blocks

SysML models include blocks, which are classes. Instances of classes (objects) have identity, enabling other instances to refer to them, while instances (values) of datatypes have no identity and must be replicated by other instances to reuse them. SMT-LIB only supports datatypes, so a mechanism is needed to provide class semantics.

**Block definition** Non-abstract SysML blocks are translated as datatypes with one constructor. Because the names of all elements (datatypes, constructors, selectors, functions, constants) in an SMT-LIB file must be unique, a naming convention is needed to avoid collision when translating SysML blocks. The name of the datatype corresponding to a SysML block is the fully qualified block name, with "$" used as namespace separator. The name of its one constructor is the datatype name prefixed with "cons-".

For example, translating an empty block DrillPress (from the package Report) results in the following code:

```
(declare-datatypes ()(
  (Report$DrillPress
    (cons-Report$DrillPress))))
```

[9] Other solvers may use a different name for this logic.

**References** The translation introduces a sort Ref, a type for giving identity to SMT-LIB datatype values. Each value of Ref identifies a single object (instance of a class), with one value per object. Since values of Ref can point to values of only one sort, and the translation generates a unique sort (datatype) for each non-abstract SysML block, an intermediate datatype Any is inserted between Ref and the datatypes corresponding to non-abstract SysML blocks. Any has one constructor per non-abstract SysML block, and each constructor has one selector typed by the datatype corresponding to the SysML block. The name of the constructor is the name of that datatype prefixed with "create-", and the name of the selector is the name of that datatype prefixed with "get-". Identity is provided by associating a Ref value to an instance of Any, which contains an instance of a datatype corresponding to a SysML block.

The following code shows the definition of Any when only the block DrillPress is translated:

```
(declare-datatypes ()(
  (Any
    (create-Report$DrillPress
      (get-Report$DrillPress Report$DrillPress)))))
```

The translator supports two implementations of reference:

- Heap-based: Ref is defined as an integer. An associative array called heap maps these integers to instances of Any. A get-object function returns the instance for a given reference by looking up the integer in the array and returning the associated instance. New instances are created as needed by the solver, and the number of instances is not limited. The definition of Ref in this approach is:

  ```
  (define-sort Ref() Int)
  (declare-const heap (Array Ref Any))
  (define-fun get-object ((ref Ref)) Any (select
    heap
    ref))
  ```

- Constant-based: Ref is defined as a datatype with a limited number of constructors, each pointing at one instance of type Any. Each instance is declared as the value of a unique constant. A get-object function maps every reference to its associated constant. All instances are are declared before solving, none can be created by the solver. The definition of Ref in this approach is (for 3 instances):

  ```
  (declare-datatypes ()(
    (Ref
      (ref0)
      (ref1)
      (ref2))))
  (declare-const obj0 Any)
  (declare-const obj1 Any)
  ```

```
(declare-const obj2 Any)
(define-fun get-object ((ref Ref)) Any (ite
  (=
    ref
    ref2)
  obj2
  (ite
    (=
      ref
      ref1)
    obj1
    obj0)))
```

In the constant-based approach, the solver will fail if the number of instances is too small. The heap-based approach is more appropriate when the approximate number of instances is not known. Example instances for both implementations are provided in Section 3.3.9.

**De-references** To retrieve data type values from Refs (de-referencing), three functions are defined for every concrete block (X is a datatype corresponding to a SysML block):[10]

- deref-is-X determines whether an Any instance corresponding to a given reference contains an object of datatype X.
- deref-X returns the object of datatype X corresponding to the given reference.
- deref-isa-X returns whether the Any object corresponding to the given reference contains an object of datatype X, or of a datatype corresponding to any specialization of the block X in SysML.

For example, dereferencing functions for a block MachiningTool and its specialization MillingMaching would be:

```
(define-fun deref-is-Report$MachiningTool ((this Ref)) Bool (
  is-create-Report$MachiningTool (get-object this)))
(define-fun deref-Report$MachiningTool ((this Ref))
   Report$MachiningTool (
  get-Report$MachiningTool (get-object this)))
(define-fun deref-is-Report$MillingMaching ((this Ref)) Bool (
  is-create-Report$MillingMaching (get-object this)))
(define-fun deref-Report$MillingMaching ((this Ref))
   Report$MillingMaching (
```

[10]The deref-is-X function and the deref-X function are not defined for abstract classes in SysML, because they cannot directly classify instances.

```
  get-Report$MillingMaching (get-object this)))

(define-fun deref-isa-Report$MachiningTool ((this Ref)) Bool (
   or
  (deref-is-Report$MachiningTool this)
  (deref-is-Report$MillingMaching this)))
(define-fun deref-isa-Report$MillingMaching ((this Ref)) Bool (
  deref-is-Report$MillingMaching this))
```

### 3.3.4 Collections

SysML provides four kinds of collections, depending on whether the elements in the collection are unique and/or ordered. This gives four combinations:

- Sets: unique and non-ordered (the default)
- Bags: non-unique and non-ordered
- Sequences: non-unique and ordered
- Ordered sets: unique and ordered collections.

Some SMT solvers provide support for some kinds of collection [12], but these have not been standardized in SMT-LIB yet. There are various alternatives to implement these collections using purely SMT-LIB. Associative arrays from Ref to Bool or Int can represent sets or bags. List datatypes can be used for sequences. One major issue with associative arrays is that cardinality constraints (multiplicity) cannot be conveniently enforced. As a result, the implementation uses List datatype for all collections. Uniqueness is enforced as needed, but ordering is not, since Lists are naturally ordered, and nothing prevents set and bags from being ordered as well, but ignored. The following code shows the definition of a list of type Ref, as well as a function to check lists for presence of a given element:

```
(declare-datatypes (X)(
  (List
    (nil)
    (insert
      (head X)
      (tail List)))))

(define-fun-rec is-in-list ((elem Ref) (list (List Ref))) Bool
    (and
      (is-insert list)
      (or
        (= (head list) elem)
        (is-in-list elem (tail list)))))
```

This function is created for each sort being collected.

Functions can be created to enforce multiplicity and uniqueness restrictions on collections. These functions return false if a given list violates the restrictions. Multiplicities can be checked by declaring how many consecutive insert constructors in a given list appear. Uniqueness can be checked by stating that the heads of the previous insert constructors are distinct. For example, the following code shows a function for checking whether a list has exactly two distinct elements:

```
(define-fun listconstraint ((elem Ref)(list (List Ref))) Bool (
  let ((list0 list))(ite
    (is-insert list0)
    (and
      ; constraints on the head
      (let ((list1 (tail list0)))(ite
        (is-insert list1)
        (and
          ; constraints on the head
          (let ((list2 (tail list1)))(ite
            (is-insert list2)
            ; no more elements
            false
            ; termination
            (distinct (head list0) (head (list1))))
        false)))))
     false)))
```

### 3.3.5 Properties

Properties are translated as selectors of the constructor corresponding to their block. Since there is no generalization in SMT-LIB, properties inherited to a SysML block are translated as if they were owned by the block. Selector names are the block datatype name, followed by the "$" character and the property name. If the property multiplicity is 1, the selector sort depends on the SysML property type:

- Primitive datatypes: the selector sort is given in Section 3.3.2.
- Blocks: the selector sort is Ref.

If the property multiplicity is other than 1, the selector sort is a collection of the above.

For example, consider a block Factory containing a property manager of type Person and multiplicity 1, a property workers of type Person and multiplicity 0..* (unique, non-ordered), and a property revenue of type Real and multiplicity 1. The following code is generated:

```
(declare-datatypes ()(
  (Report$Factory
    (cons-Report$Factory
      (Report$Factory$manager Ref)
      (Report$Factory$workers (Set Ref))
      (Report$Factory$revenue Real)))))
(declare-datatypes ()(
  (Report$Person
    (cons-Report$Person))))
```

Functions are defined for each owned or inherited property in a class to access the corresponding selector of each specialization of that class. This makes instances of a class appear as if they were also instances of their generalized classes. Two functions are defined for each owned or inherited property p in a SysML block X:

- X!p returns the value of the selector p for a given reference of type X.
- X.p returns the value of the selector p for the type of a given reference, which might be more specialized than the type in the call (X).

The return type of these functions is the same as the type of the selector. Property redefinition is handled in the second function, by calling selectors corresponding to the redefining properties in a specialized class. For example, consider a block CarFactory that specializes the block Factory previously introduced

The following code is generated for property access:

```
(define-fun Report$CarFactory!manager ((this Ref)) Ref (
  Report$CarFactory$manager (deref-Report$CarFactory this)))
(define-fun Report$CarFactory.manager ((this Ref)) Ref (
  Report$CarFactory$manager (deref-Report$CarFactory this)))

(define-fun Report$CarFactory!workers ((this Ref)) (Set Ref) (
  workers (deref-Report$CarFactory this)))
(define-fun Report$CarFactory.workers ((this Ref)) (Set Ref) (
  Report$CarFactory$workers (deref-Report$CarFactory this)))

(define-fun Report$CarFactory!revenue ((this Ref)) Real (
  Report$CarFactory$revenue (deref-Report$CarFactory this)))
(define-fun Report$CarFactory.revenue ((this Ref)) Real (
  Report$CarFactory$revenue (deref-Report$CarFactory this)))

(define-fun Report$Factory!manager ((this Ref)) Ref (
  Report$Factory$manager (deref-Report$Factory this)))
(define-fun Report$Factory.manager ((this Ref)) Ref (ite
  (deref-is-Report$CarFactory this)
  (Report$CarFactory$manager (deref-Report$CarFactory this))
```

```
  (Report$Factory$manager (deref-Report$Factory this))))

(define-fun Report$Factory!workers ((this Ref)) (Set Ref) (
  Report$Factory$workers (deref-Report$Factory this)))
(define-fun Report$Factory.workers ((this Ref)) (Set Ref) (ite
  (deref-is-Report$CarFactory this)
  (Report$CarFactory$workers (deref-Report$CarFactory this))
  (Report$Factory$workers (deref-Report$Factory this))))

(define-fun Report$Factory!revenue ((this Ref)) Real (
  Report$Factory$revenue (deref-Report$Factory this)))
(define-fun Report$Factory.revenue ((this Ref)) Real (ite
  (deref-is-Report$CarFactory this)
  (Report$CarFactory$revenue (deref-Report$CarFactory this))
  (Report$Factory$revenue (deref-Report$Factory this))))
```

With these constructs, calling Factory.manager on an instance of CarFactory will return the value of CarFactory.manager as expected.

Constraints on properties (including enforcing reference types) are addressed in Section 3.3.8.

### 3.3.6 Associations and connectors

Associations are translated the same way as classes, with member ends owned by an association translated the same way as other properties. The translation does not currently support member ends owned by the classes being associated. Connectors are translated as properties of multiplicity 0..*, typed by an association.

### 3.3.7 Operations as constraints

SysML operations on classes are a kind of behavior that only specifies input and output/return parameters. Typically the rest of the information needed to execute an operation is given by a behavior owned by a class, where the behavior specifies steps taken over time to change property values. Another way is to give this information is to constrain the operation return values relative to inputs in particular ways. Since the translation does not support changing objects, only constraints on operations is currently supported. SysML models often use the Object Constraint Language (OCL) for expressing constraints, a language designed for use with UML and its extensions [13].

In the translation, OCL expressions are parsed to create an abstract syntax tree, which is scanned and translated to SMT-LIB statements. Property calls in OCL are translated to calls of functions as in the Section 3.3.5, and operations calls are translated to calls of functions as described below. Mathematical and logical operators are translated to their equivalent.

In SMT-LIB, functions corresponding to operations are given a parameter for the object on which the operation is called. Additional parameters are added as defined in the operation definition. The content of the SMT-LIB function is the translation of the operation's body condition, specified in OCL.

```
(declare-fun Behavior.isBefore (Ref Ref) Bool)
...
(define-fun Behavior.isBefore ((this Ref)(t Ref)) Bool
  (<
  (Behavior.end this)
  (Behavior.start t)))
```

### 3.3.8 Class restrictions (invariants)

SysML models express various restrictions on class instances:

- Property type: attribute values are required to be of this type. In SMT-LIB, datatype selectors typed by Ref or by a collection of Ref should be restricted to have an object compatible with the type of the property, translated to SMT-LIB as the corresponding deref-isa function.
- Property multiplicity (upper and lower): the number of values for a property must be between the lower and upper multiplicity number. When the multiplicity is 1, no constraint is needed in SMT-LIB since the property is translated as a Ref selector, which has exactly one value. When the multiplicity is not 1, the property is translated as a collection, with constraints added to ensure it has the correct number of elements.
- Connectors: treated as properties typed by an association, with values limited to links (instances of the association) between values of the connector end roles (properties). Connector end multiplicities specify how many links the connected property values may have for a given connector.
- Other restrictions in opaque expressions, written in OCL for example: all constraints on a class are translated into their equivalent in SMT-LIB.

An "invariant" function containing the SMT-LIB equivalent of these restrictions is defined for each SysML block. The function takes an instance of the corresponding datatype as input, and states what must be true for that object, based on the restrictions above. Constraints that apply to each property or connector value separately (e.g. type, number of links, not multiplicity) are defined in separate functions, which are called by the block invariant function. This simplifies translation of lists, as this separate function will apply constraints on property values to the head of the list, and call itself on the tail of the list.

Regarding connectors, consider a SysML connector c between property a to property b, with a as the role of the first connector end, and b as the role of the second connector end. For brevity, we will refer to the first end of links specified by the connector as source, and the second end as target. The translator adds two kinds of constraints to ensure:

- The ends of links specified by the connector are values of the properties at its ends. For every link of c, the source must be equal to (or within the collection of) the value of a of some instance of the connector's owning block (or one of its specializations) and the target must be equal to (or within the collection of) the value of b of that same instance.

- The number of incoming and outgoing links specified by a connector satisfies its end multiplicities for each end property value. For example, if the end multiplicity of c on the b side is 0..1, then each value of a can be the source for no more than 1 link of the connector. If the connector end multiplicity on the a side is 1, then each value of b must be the target of exactly one link. Utility functions are created to assert that there are exactly "0", "1", or "0 or 1" incoming and outgoing links.

  When a OneOf constraint is applied to multiple connector ends with 0..1 multiplicity, the resulting SMT-LIB constraint is that exactly one of the end has to have a 1 incoming/outgoing link, while the other have exactly 0 incoming/outgoing link. This is used to model the equivalent of SysML decision and merge nodes.

For example, consider a block ToyFactory with the same properties as Factory (manager, workers, revenue), plus two properties w1 and w2 of type Workstation and multiplicity 1, and a connector typed by Conveyor between these two properties. Conveyor is an association between two instance of Workstation, with multiplicity 0..1 on one end, and multiplicity 1 on the other end. The utility functions for E would be generated as follows (assuming a size limit of 2 for collections):

```
(define-fun has0outReport$Conveyor ((elem Ref)(collection (Set
   Ref))) Bool (
let ((list0 collection))(or
  (is-nil list0)
  (and
    (is-insert list0)
    (distinct
      elem
      (Report$Conveyor.end1 (head list0)))
    (let ((list1 (tail list0)))(or
      (is-nil list1)
      (and
```

```
        (is-insert list1)
        (distinct
          elem
          (Report$Conveyor.end1 (head list1)))
        (is-nil (tail list1)))))))))
(define-fun has01outReport$Conveyor ((elem Ref)(collection (Set
    Ref))) Bool (let ((list0 collection))(or
  (is-nil list0)
  (and
    (is-insert list0)
    (or
      (and
        (distinct
          elem
          (Report$Conveyor.end1 (head list0)))
        (let ((list1 (tail list0)))(or
          (is-nil list1)
          (and
            (is-insert list1)
            (or
              (and
                (distinct
                  elem
                  (Report$Conveyor.end1 (head list1)))
                (is-nil (tail list1)))
              (and
                (=
                  elem
                  (Report$Conveyor.end1 (head list1)))
                (has0outReport$Conveyor
                  elem
                  (tail list1))))))))
      (and
        (=
          elem
          (Report$Conveyor.end1 (head list0)))
        (has0outReport$Conveyor
          elem
          (tail list0)))))))
(define-fun has1outReport$Conveyor ((elem Ref)(collection (Set
   Ref))) Bool (let ((list0 collection))(and
  (is-insert list0)
  (or
    (and
      (=
```

```
        elem
        (Report$Conveyor.end1 (head list0)))
      (let ((list1 (tail list0)))(or
        (is-nil list1)
        (and
          (is-insert list1)
          (distinct
            elem
            (Report$Conveyor.end1 (head list1)))
          (is-nil (tail list1))))))
    (let ((list1 (tail list0)))(and
      (is-insert list1)
      (or (and
        (=
          elem
          (Report$Conveyor.end1 (head list1)))
        (is-nil (tail list1))))
      (distinct
        elem
        (Report$Conveyor.end1 (head list0)))))))))
(define-fun has0incReport$Conveyor ((elem Ref)(collection (Set
   Ref))) Bool (let ((list0 collection))(or
  (is-nil list0)
  (and
    (is-insert list0)
    (distinct
      elem
      (Report$Conveyor.end2 (head list0)))
    (let ((list1 (tail list0)))(or
      (is-nil list1)
      (and
        (is-insert list1)
        (distinct
          elem
          (Report$Conveyor.end2 (head list1)))
        (is-nil (tail list1)))))))))
(define-fun has01incReport$Conveyor ((elem Ref)(collection (Set
    Ref))) Bool (let ((list0 collection))(or
  (is-nil list0)
  (and
    (is-insert list0)
    (or
      (and
        (distinct
          elem
```

```
          (Report$Conveyor.end2 (head list0)))
        (let ((list1 (tail list0)))(or
          (is-nil list1)
          (and
            (is-insert list1)
            (or
              (and
                (distinct
                  elem
                  (Report$Conveyor.end2 (head list1)))
                (is-nil (tail list1)))
              (and
                (=
                  elem
                  (Report$Conveyor.end2 (head list1)))
                (has0incReport$Conveyor
                  elem
                  (tail list1)))))))))
      (and
        (=
          elem
          (Report$Conveyor.end2 (head list0)))
        (has0incReport$Conveyor
          elem
          (tail list0)))))))
(define-fun has1incReport$Conveyor ((elem Ref)(collection (Set
   Ref))) Bool (let ((list0 collection))(and
  (is-insert list0)
  (or
    (and
      (=
        elem
        (Report$Conveyor.end2 (head list0)))
      (let ((list1 (tail list0)))(or
        (is-nil list1)
        (and
          (is-insert list1)
          (distinct
            elem
            (Report$Conveyor.end2 (head list1)))
          (is-nil (tail list1))))))
    (let ((list1 (tail list0)))(and
      (is-insert list1)
      (or (and
        (=
```

```
          elem
          (Report$Conveyor.end2 (head list1)))
        (is-nil (tail list1))))
      (distinct
        elem
        (Report$Conveyor.end2 (head list0)))))))))
```

The invariant definition for the block ToyFactory looks like the following:

```
(define-fun Report$ToyFactory$manager.invC ((prop Ref)(owner
   Ref)) Bool (
  deref-isa-Report$Person prop))
(define-fun Report$ToyFactory$workers.invC ((prop (Set Ref))(
   owner Ref)) Bool (
  let ((list0 prop))(or
    (is-nil list0)
    (and
      (is-insert list0)
      (deref-isa-Report$Person (head list0))
      (let ((list1 (tail list0)))(or
        (is-nil list1)
        (and
          (is-insert list1)
          (deref-isa-Report$Person (head list1))
          (and
            (is-nil (tail list1))
            (distinct
              (head list0)
              (head list1)))))))))))
(define-fun Report$ToyFactory$w1.invC ((prop Ref)(owner Ref))
   Bool (and
  (deref-isa-Report$Workstation prop)
  (has01outReport$Conveyor
    prop
    (Report$ToyFactory.ToyFactory_connector5_end1_end2 owner)))
      )
(define-fun Report$ToyFactory$w2.invC ((prop Ref)(owner Ref))
   Bool (and
  (deref-isa-Report$Workstation prop)
  (has1incReport$Conveyor
    prop
    (Report$ToyFactory.ToyFactory_connector5_end1_end2 owner)))
      )
(define-fun Report$ToyFactory$ToyFactory_connector5_end1_end2.
   invC ((conn (Set Ref))(owner Ref)) Bool (
  let ((list0 conn))(or
```

```
    (is-nil list0)
    (and
      (is-insert list0)
      (deref-isa-Report$Conveyor (head list0))
      (=
        (Report$Conveyor.end1 (head list0))
        (Report$ToyFactory.w1 owner))
      (=
        (Report$Conveyor.end2 (head list0))
        (Report$ToyFactory.w2 owner))
      (let ((list1 (tail list0)))(or
        (is-nil list1)
        (and
          (is-insert list1)
          (deref-isa-Report$Conveyor (head list1))
          (=
            (Report$Conveyor.end1 (head list1))
            (Report$ToyFactory.w1 owner))
          (=
            (Report$Conveyor.end2 (head list1))
            (Report$ToyFactory.w2 owner))
          (and
            (is-nil (tail list1))
            (distinct
              (head list0)
              (head list1)))))))))
(define-fun Report$ToyFactory.invC ((this Ref)) Bool (and
  (Report$ToyFactory$manager.invC
    (Report$ToyFactory.manager this)
    this)
  (Report$ToyFactory$workers.invC
    (Report$ToyFactory.workers this)
    this)
  (Report$ToyFactory$w1.invC
    (Report$ToyFactory.w1 this)
    this)
  (Report$ToyFactory$w2.invC
    (Report$ToyFactory.w2 this)
    this)
  (Report$ToyFactory$ToyFactory_connector5_end1_end2.invC
    (Report$ToyFactory.ToyFactory_connector5_end1_end2 this)
    this)))
```

Applying the invariant to instances depends on the way references are implemented (see Section 3.3.3). When using the heap-based approach, the invariant application

looks like the following:

```
(assert (forall ((this Ref))(=>
  (deref-is-ToyFactory this)
  (ToyFactory.inv this))))
```

Quantifiers must be used to make sure all references are processed.

When using the constant-based approach, the invariant application looks like the following:

```
(define-fun invariants ((ref Ref)) Bool (and
  (=>
    (deref-isa-ToyFactory ref)
    (ToyFactory.invC ref))))
(assert (invariants ref0))
(assert (invariants ref1))
(assert (invariants ref2))
```

### 3.3.9 Satisfiability check

The translation process starts with a root SysML block, and operates incrementally on its dependencies. An assertion states that there must be exactly one instance of the root block. A constant ROOT is created to store that instance.

Finally, commands to check satisfiability and show instances that prove it are inserted.

For example, the following commands are generated for the block ToyFactory:

```
(assert (forall ((this Ref))(=>
  (deref-isa-Report$ToyFactory this)
  (Report$ToyFactory.invC this))))
(declare-const ROOT Ref)
(assert (deref-is-Report$ToyFactory ROOT))
(assert (forall ((x Ref)(y Ref))(=>
  (and
    (deref-is-Report$ToyFactory x)
    (deref-is-Report$ToyFactory y))
  (=
    x
    y))))
(check-sat)
```

Running that last command using the heap-based implementation yields the following result:

```
sat
(model
  (define-fun ROOT () Int
```

```
    0)
  (define-fun heap () (Array Int Any)
    (_ as-array k!0))
  (define-fun k!0!5 ((x!0 Int)) Any
    (ite (= x!0 0)
      (create-Report$ToyFactory (cons-Report$ToyFactory 1 (
         insert 9 nil) 0.0 2 3 (insert 8 nil)))
    (ite (= x!0 8) (create-Report$Conveyor (
       cons-Report$Conveyor 2 3))
    (ite (= x!0 2) (create-Report$Workstation
       cons-Report$Workstation)
    (ite (= x!0 3) (create-Report$Workstation
       cons-Report$Workstation)
      (create-Report$Person cons-Report$Person))))))
  (define-fun k!4 ((x!0 Int)) Int
    (ite (= x!0 8) 8
    (ite (= x!0 3) 3
    (ite (= x!0 1) 1
    (ite (= x!0 0) 0
    (ite (= x!0 9) 9
      2))))))
  (define-fun k!0 ((x!0 Int)) Any
    (k!0!5 (k!4 x!0)))
)
```

The first line is the result of the (check-sat) command, indicating the problem is satisfiable. The lines after that give the model discovered by the solver. The heap is defined as an array, which calls a function k!0 that gives the Any object corresponding to the value of the parameter x!0, defined in k!0!5:

- Reference 0 corresponds to an instance of ToyFactory with the reference 1 as value of the property manager, a list consisting of referenced 9 as value of the property workers, 0.0 as value of the property revenue, the reference 2 as value of the property w1, the reference 3 as value of the property w2, and the reference 8 as value of the connector.
- Reference 8 corresponds to an instance of the association Conveyor, with the reference 2 on the first end and reference 3 on the second end.
- Reference 2 and 3 return an instance of Workstation
- Any other reference (i.e. 1, and 9) returns an instance of Person

Running the satisfiability checking command using the constant-based (with six objects) approach yields the following result:

```
sat
(model
  (define-fun obj1 () Any
    (create-Report$Person cons-Report$Person))
  (define-fun obj3 () Any
    (create-Report$Workstation cons-Report$Workstation))
  (define-fun obj0 () Any
    (create-Report$ToyFactory (cons-Report$ToyFactory ref1 nil
       3.0 ref2 ref3 (insert ref4 nil))))
  (define-fun obj4 () Any
    (create-Report$Conveyor (cons-Report$Conveyor ref2 ref3)))
  (define-fun obj2 () Any
    (create-Report$Person cons-Report$Person))
  (define-fun ROOT () Ref
    ref0)
)
```

The first line is the result of the (check-sat) command, still indicating the problem is satisfiable. The lines after that give the value of the five constants:

- obj0 is an instance of ToyFactory, with the reference 1 as value of the property manager, an empty list as value of the property workers, 3.0 as value of the property revenue, reference 2 as value of the property w1, reference 3 as value of the property w2, and reference 4 as value of the connector.
- obj1 is an instance of Person.
- obj2 and obj3 are instances of Workstation.
- obj4 is an instance of the association Conveyor with reference 2 on the first end, and reference 3 on the second end.

### 3.4 Translating the OBM extension to SMT-LIB

This section describes the translation of the OBM extension of SysML (see Section 2.2.2) to SMT-LIB. This is in addition to the translation of structural aspects of SysML that OBM extends (see Section 3.3).

**Temporal relations** OBM defines two associations to model temporal relations between behavior occurrences (HappensBefore and HappensDuring, see Section 2.2.2) with several logical characteristics that are restricted in the implementation. The relations are required to be completely asymmetric, which includes irreflexivity. This means occurrences do not happen at the same time as others (occurrences can only happen during others of longer duration, with longer ones having non-zero duration). It also means occurrences cannot be related by HappensBefore and HappensDuring at

the same time.[11] Transitivity is not restricted. These narrow the correspondence to Allen's relations to

- HappensBefore is equivalent to the union of Allen's before and meets interval relations without the exclusions for symmetry in Section 2.2.2.
- HappensDuring is equivalent to the union of Allen's starts, during, and finishes (equals is not included).

The translation to SMT-LIB introduces two Boolean functions (logical predicates) before and during corresponding to HappensBefore and HappensDuring, as restricted above. The translation provides two approaches to specify that these functions are transitive, asymmetric, and relate to each other per characteristic 3 in Section 2.2.2:[12]

- Without timepoints: Asserts the (restricted) logical characteristics of the functions above, corresponding to:
  - $\forall x, y, z \;\; before(x, y) \wedge before(y, z) \implies before(x, z)$
  - $\forall x, y \;\; before(x, y) \implies \neg before(y, x)$
  - $\forall x, y, z \;\; during(x, y) \wedge during(y, z) \implies during(x, z)$
  - $\forall x, y \;\; during(x, y) \implies \neg during(y, x)$
  - $\forall x, y, z \;\; before(x, y) \wedge during(z, y) \implies before(x, z)$
  - $\forall x, y, z \;\; before(y, x) \wedge during(z, y) \implies before(z, x)$
- With timepoints: Adds two properties to BehaviorOccurrence: start and end, both of type Real and multiplicity 1, and a constraint stating that $start < end$. The functions above are defined with these properties to express transitivity and asymmetry, corresponding to:
  - before(b1,b2) $\iff b1.end \leq b2.start$
  - during(b1,b2) $\iff (b1.start > b2.start \wedge b1.end \leq b2.end) \vee (b1.start \geq b2.start \wedge b1.end < b2.end)$

  These definitions use numeric inequalities to ensure transitivity, asymmetry, and mutual exclusivity.

[11] Derived from asymmetric HappensBefore and characteristic 3 in Section 2.2.2 relating it to HappensDuring.

[12] This could have been specified as OCL in OBM, but it is simpler to express these characteristics directly in SMT-LIB than to translate the additional OCL constructs required. Translating these constructs could result in unbounded lists for values of end properties of HappensBefore, or possibly complex existential quantification over the reference heap.

The before function is true between occurrences linked by instances of HappensBefore, which exist between values of properties at the ends of connectors typed by HappensBefore (see invariant functions in Section 3.3.8). The same is true for during and HappensDuring connectors (see Figure 47 in Section 4.1.6), but for simpler modeling, it is also taken to be true between occurrences when one is classified by BehaviorOccurrence (or one of its specialization) and the other is a value of a property (such as OBM step) of the first occurrence typed by BehaviorOccurrence (or one of its specializations). Both functions are also true due to transitivity, but this is deduced without corresponding transitively deduced links. Examples in Section 4 have separate figures for links (instances of HappensBefore and HappensDuring) and temporal functions (before and during).

**Transfers, item flows, and object flow** Transfers describe movement of items between participants. They are modeled as a Transfer association, which connectors can type. Transfer is a specialization of BehaviorOccurrence. An ItemFlow stereotype indicates which properties of the participants give the transferred item. Another association named TransferBefore is for transfers that order their participants (behavior occurrences) in time.

### 3.5 Translator Implementation

The translation described in the previous sections was implemented in software [14] that takes a SysML file in XMI format as input, performs the translation, and generates an SMT-LIB file that can be executed on SMT solvers such as Z3[15].[13] The translator accepts various parameters, such as a root SysML block that is being translated, the way collections should be implemented, and whether timepoints are used. The main parts of the translator are:

- XMI readers and writer, based on Eclipse UML2. The main addition is an XMI reader preprocessor, which accommodates multiple versions of XMI, UML, and SysML.
- SMT-LIB Java classes, generated from an SMT-LIB metamodel.
- SysML to SMT-LIB translator, which implements the mapping from the previous section to create SMT-LIB Java objects from SysML Java objects.
- OCL to SMT-LIB translator, which translates opaque expressions written in OCL.
- SMT-LIB reader, which reads logics and theories defined as part of the SMT-LIB standard.

[13] We tested the generated files using Z3 version 4.7.1

- SMT-LIB writer, which generates an SMT-LIB file from SMT-LIB Java objects.
- Z3 bridge, which communicates the generated content directly to the Z3 solver.

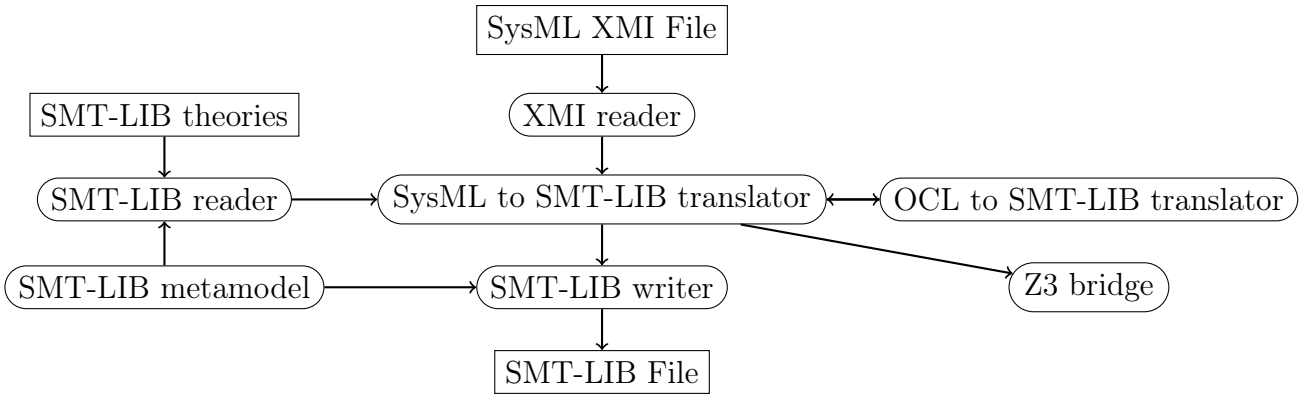


**Fig. 5.** Overview of the translator

Experiments with the Z3 solver showed that reasoning was influenced by the way models were serialized. Particularly, the solver would either timeout or give an answer within a second depending on the order of the Any constructors. We could not find a ordering strategy that would work for all our test models. As a result, the implementation uses a timeout of 10 s, and shuffles the Any constructors up to 5 times, giving 60 s in total to find an answer. With that approach, all models have been successfully checked (although this is not guaranteed). Benchmark numbers are useful since the solving process is not deterministic (it is not guaranteed to behave the same way given the same inputs).

## 4. Translation Examples

This section contains example translations. Many of them use a class AtomicBehavior that is a specialization of BehaviorOccurrence for behaviors that have no steps. Each example shows an OBM model along with the equivalent behavior represented using a SysML diagram. Instances are shown in diagrams from the translations. The notation shows instances as rectangles containing their identifier, type, and data values, if applicable (this is not UML notation). The diagrams have two parts:

- Upper parts show structural relations (links of associations) between instances, represented as solid arrows. Links for composite associations are indicated by a black diamond tail.
- Lower parts show temporal relations (see Section 3.4) between the instances, with before and during represented as short-dashed arrows and long-dashed

arrows respectively. To simplify the diagrams, temporal relations deduced by transitivity are not displayed.

Section 4.1 shows basic examples (e.g., control nodes, composition, and transfers), while Section 4.2 shows more advanced examples (e.g., behavior inheritance, redefinition).

## 4.1 Basic examples

### 4.1.1 Control nodes

**Fork** Figures 6 and 7 show SysML activity and OBM representations of a behavior with a fork, respectively. In the OBM representation, BehaviorFork is a block with three properties typed by AtomicBehavior: p1, p2, and p3. The multiplicity on p1 is 1 (it corresponds to an initial node), indicating that the step happens exactly once, and the multiplicity on the two other properties is 0..*, indicating that the steps may happen any number of times (they correspond to actions). Two connectors link p1 with p2 and p3 respectively. The multiplicities on all the ends of the connectors are 1. This means each behavior occurrence in p1 must have a HappensBefore relationship to exactly one behavior occurrence in p2 and exactly one in p3. In addition, each occurrence in p2 or in p3 must have a HappensBefore relationship with exactly one behavior occurrence in p1. The flow is expected to go from p1 to both p2 and p3.

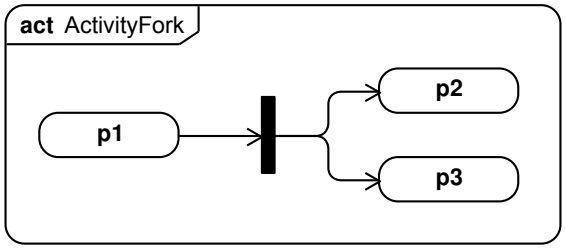


**Fig. 6.** Fork model (activity)

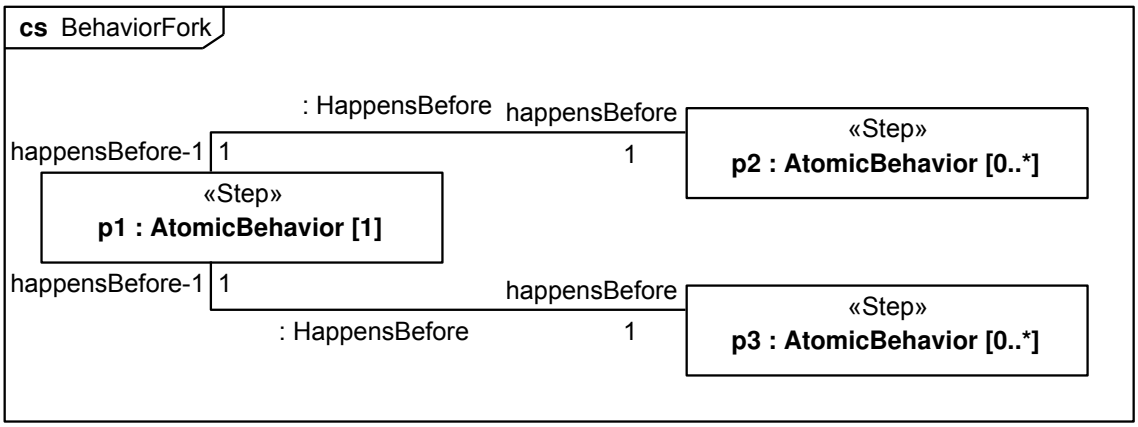


**Fig. 7.** Fork model (OBM)

Figure 8 shows two views of instances produced by the reasoner from SMT files generated from Figure 7 by the translator. The top part shows links between instances, while the bottom shows temporal relations. At the top is an instance #0 typed by BehaviorFork, with the instances #1, #4, and #5 in p1, p2, and p3. Two links typed by HappensBefore are present: #2 connects #1 to #4, and #3 connects #1 to #5. The lower part shows that #1 happens before both #4 and #5, and these three instances happen during #5. That means the flow went from p1 to both p2 and p3, as expected.

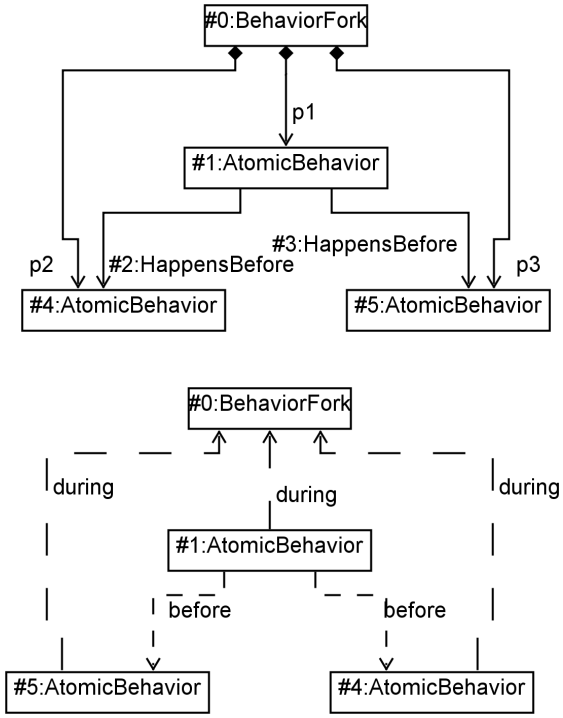


**Fig. 8.** Fork instance model

**Join** Figures 9 and 10 show SysML activity and OBM representations of a behavior with a join, respectively. In the OBM representation, BehaviorJoin is a block with three properties typed by AtomicBehavior: p1, p2, and p3. The multiplicity on p1 and p2 is 1, indicating that the steps happen exactly once, and the multiplicity on p3 is 0..*, indicating that the step may happen any number of times. Two connectors link respectively p1 and p2 with p3. The multiplicity on all the ends of the connectors is 1. This means each behavior occurrence in p1 and in p2 must have a HappensBefore relationship to exactly one behavior occurrence in p3. In addition, each behavior in p3 must have a HappensBefore relationship from exactly one behavior occurrence in p1 and exactly one in p2. The flow is expected to go from both p1 and p2 to p3.

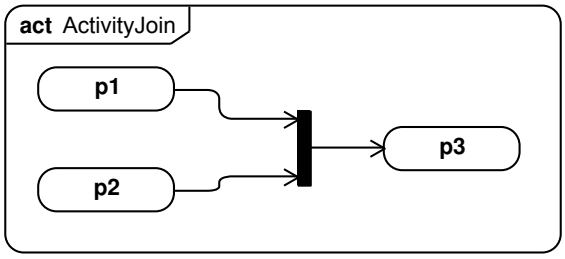


**Fig. 9.** Join model (activity)

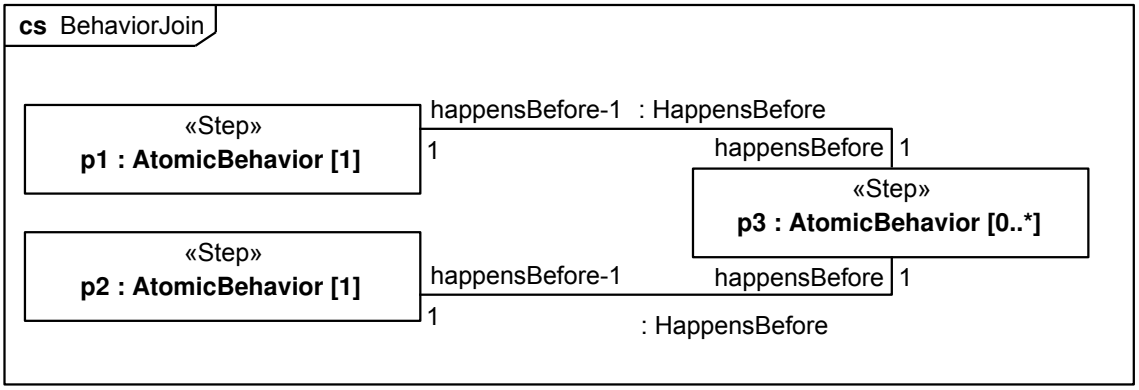


**Fig. 10.** Join model (OBM)

Figure 11 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of BehaviorJoin, with three steps: #1 as value of p1, #2 as value of p2, and #6 as value of p3. Both #1 and #2 happen before #6, so the model corresponds a join.

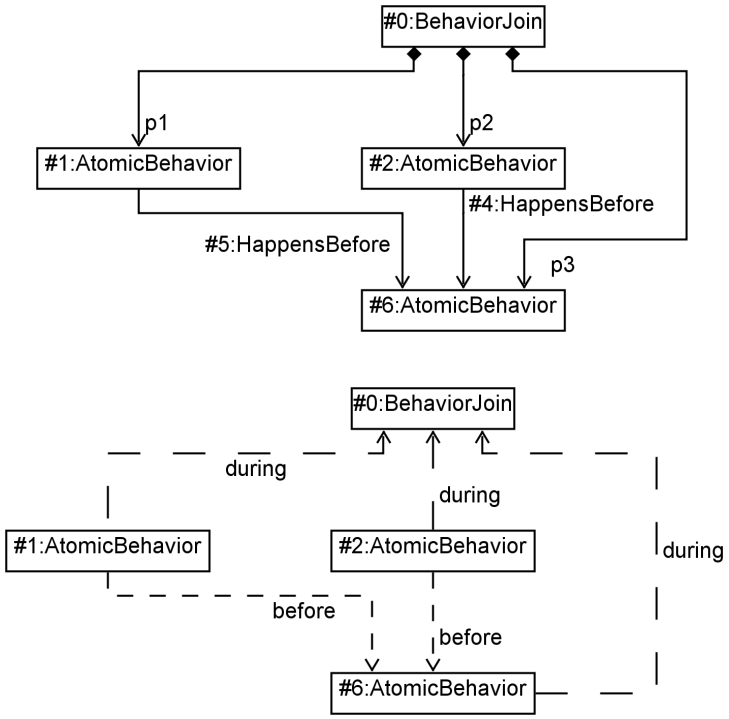


**Fig. 11.** Join instance model

**Decision** Figures 12 and 13 show SysML activity nd OBM representations of a behavior with a decision, respectively. In the OBM representation, BehaviorDecision is a block with three properties typed by AtomicBehavior: p1, p2, and p3. The multiplicity on p1 is 1, indicating that the step happens exactly once, and the multiplicity on the two other properties is 0..*, indicating that the steps may happen any number of times. Two connectors link p1 with p2 and p3 respectively. The multiplicities on the ends on p1 are 1, and the multiplicities on the ends of p2 and p3 are 0..1 with a OneOf constraint on them. This means each behavior occurrence in p1 must have a HappensBefore relationship to exactly one behavior occurrence, either in p2 or p3, but not both. In addition, each occurrence in p2 or in p3 must have a HappensBefore relationship from exactly one behavior occurrence in p1. The flow is expected to go from p1 to either p2 or p3.

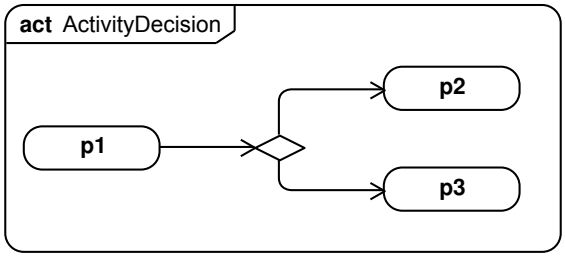


**Fig. 12.** Decision model (activity)

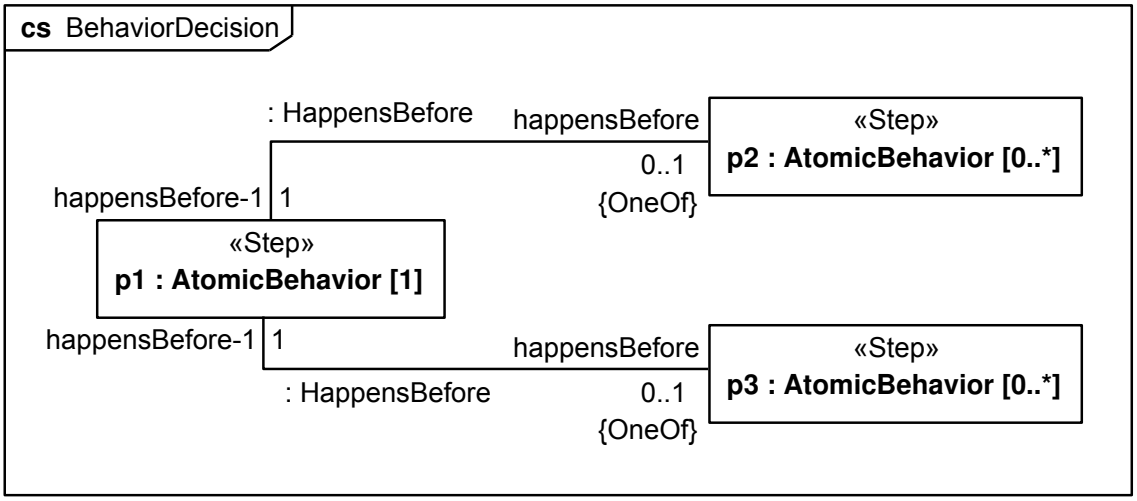


**Fig. 13.** Decision model (OBM)

Figure 14 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. `#0` is an instance of BehaviorDecision, with the instance `#1` as value of `p1`, and `#5` as value of `p3`. The temporal relations show that `#1` happens before `#5`, and since there is no value for `p2`. the model corresponds to a decision.

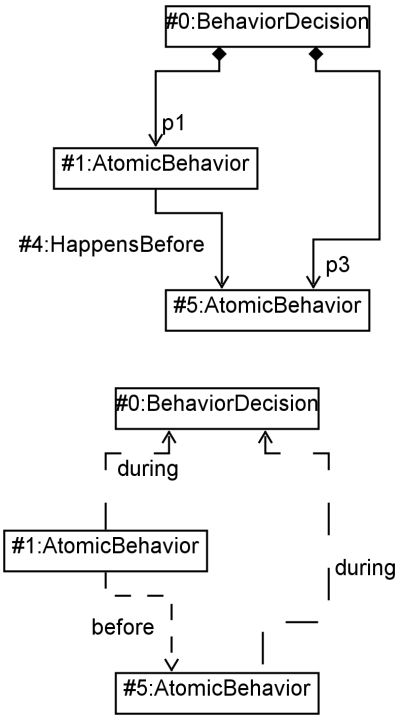


**Fig. 14.** Decision instance model

**Merge** Figures 15 and 16 show SysML activity and OBM representations of a behavior with a merge, respectively. In the OBM representation, BehaviorMerge is a block with three properties typed by AtomicBehavior: p1, p2, and p3. The multiplicity on p1 and p2 is 1, indicating that the steps happen exactly once, and the multiplicity on p3 is 0..*, indicating that the step may happen any number of times. Two connectors link p1 and p2 to p3. The multiplicities on the ends of p1 and p2 are 0..1 with a OneOf constraint, and the multiplicities on the ends of p3 are 1. This means each behavior occurrence in p1 or in p2 must have a HappensBefore relationship to exactly one behavior occurrence in p3. In addition, each behavior occurrence in p3 must have a HappensBefore relationship from exactly one behavior in either p1 or p2, but not both. The flow is expected to go from p1 to p3, and from p2 to p3.

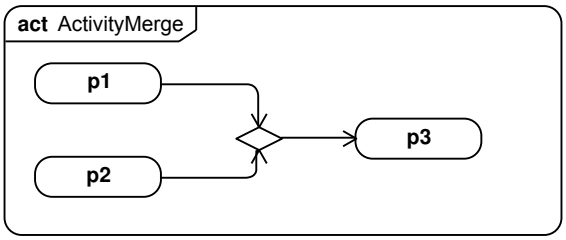


**Fig. 15.** Merge model (activity)

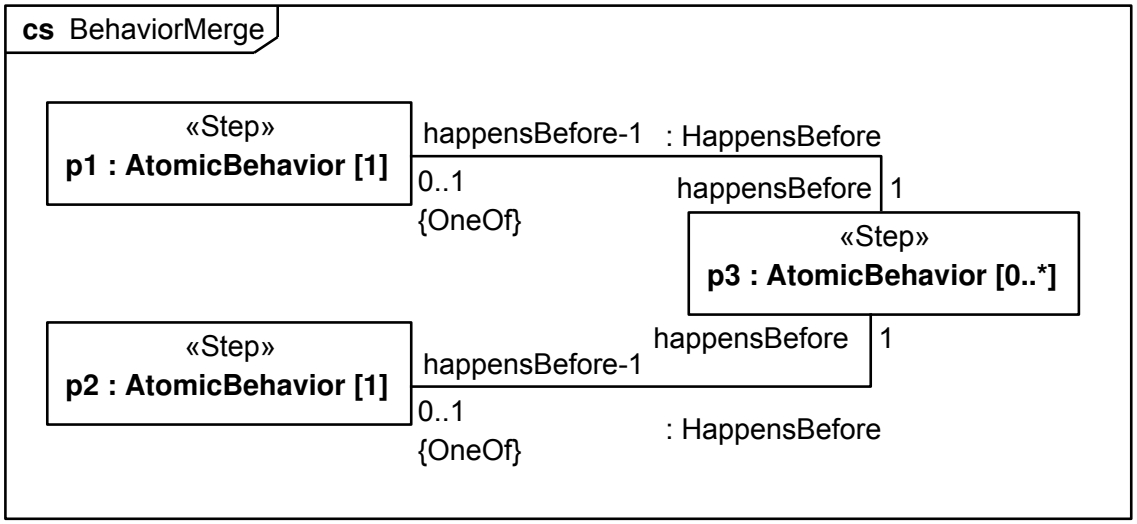


**Fig. 16.** Merge model (OBM)

Figure 17 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of BehaviorMerge, with the instance #1 as value of p1, #2 as value of p2, and both #6 and #7 as values of p3. The temporal relations show that #2 happens before #7, and #1 happens before #6. Since each value in the steps p1 and p2 has one successor in p3, the model corresponds to a merge.

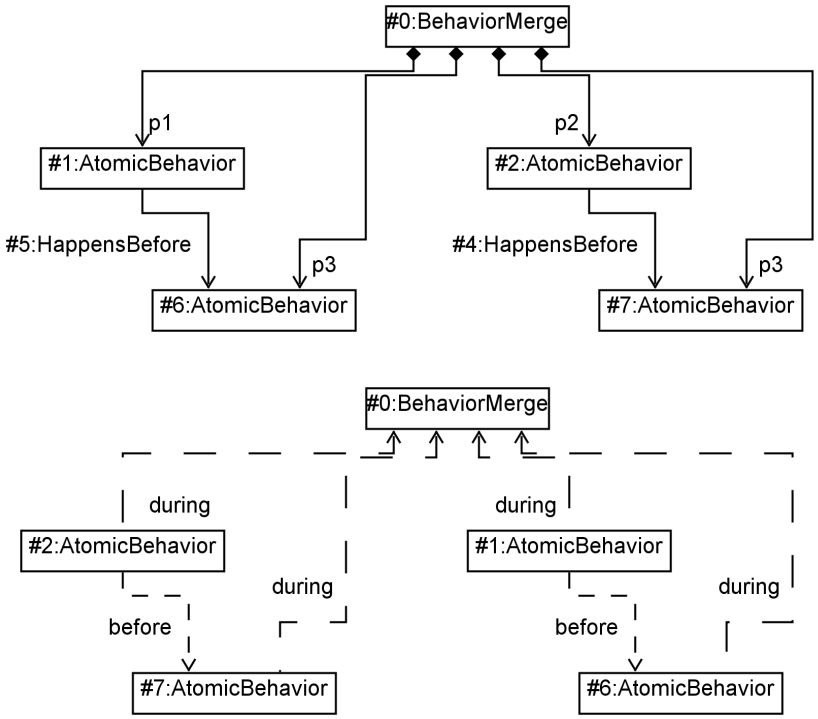


**Fig. 17.** Merge instance model

**Altogether** Figures 18 and 19 show SysML activity and OBM representations, respectively, of a behavior that combines the examples of fork, join, decision, and merge. In the OBM representation, ComplexBehavior is a block with seven properties, connected as follows (see previous sections for more detailed explanations):

- p1, p2, and p3 form a Fork,
- p2, p3, and p4 form a Join,
- p4, p5, and p6 form a Decision
- p5, p6, and p7 form a Merge.

The flow is expected to go from p1 to both p2 and p3, then from both p2 and p3 to p4, from p4 to either p5 or p6, and from either p5 or p6 to p7.

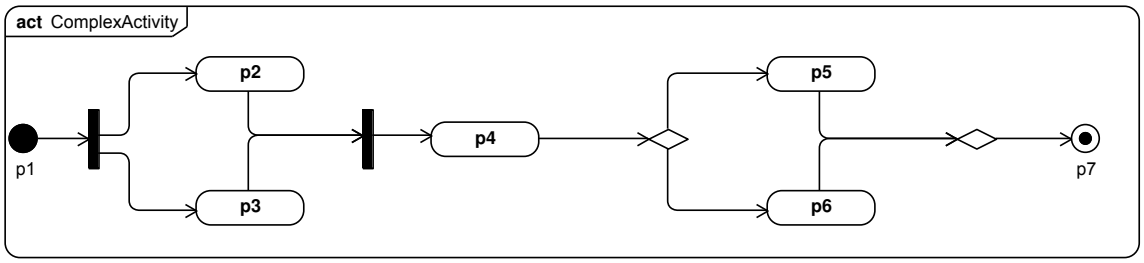


**Fig. 18.** Complete behavior (activity)

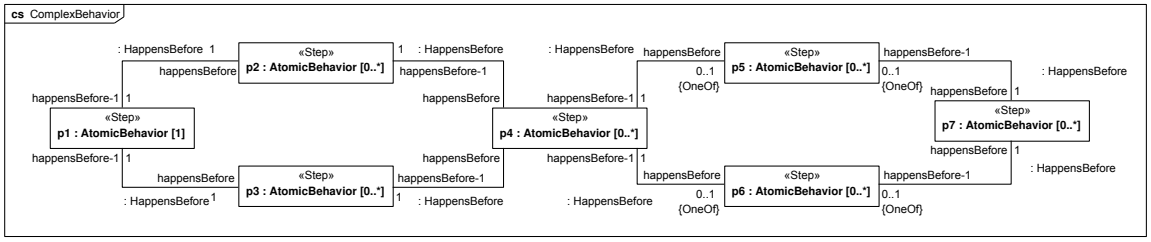


**Fig. 19.** Complete behavior (OBM)

Figure 20 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of ComplexBehavior, with the instance #1 as value of p1, #7 as value of p2, #9 as value of p3, #10 as value of p4, nothing for p5, #11 as value of p6, and #14 as value of p7. The temporal relations show that #1 happens before both #7 and #9 (forming a fork), both #7 and #9 happen before #10 (forming a join), #10 happens before #11 not does not happen before a value in p5 (forming a decision), and #11 happens before #14 (forming a merge).

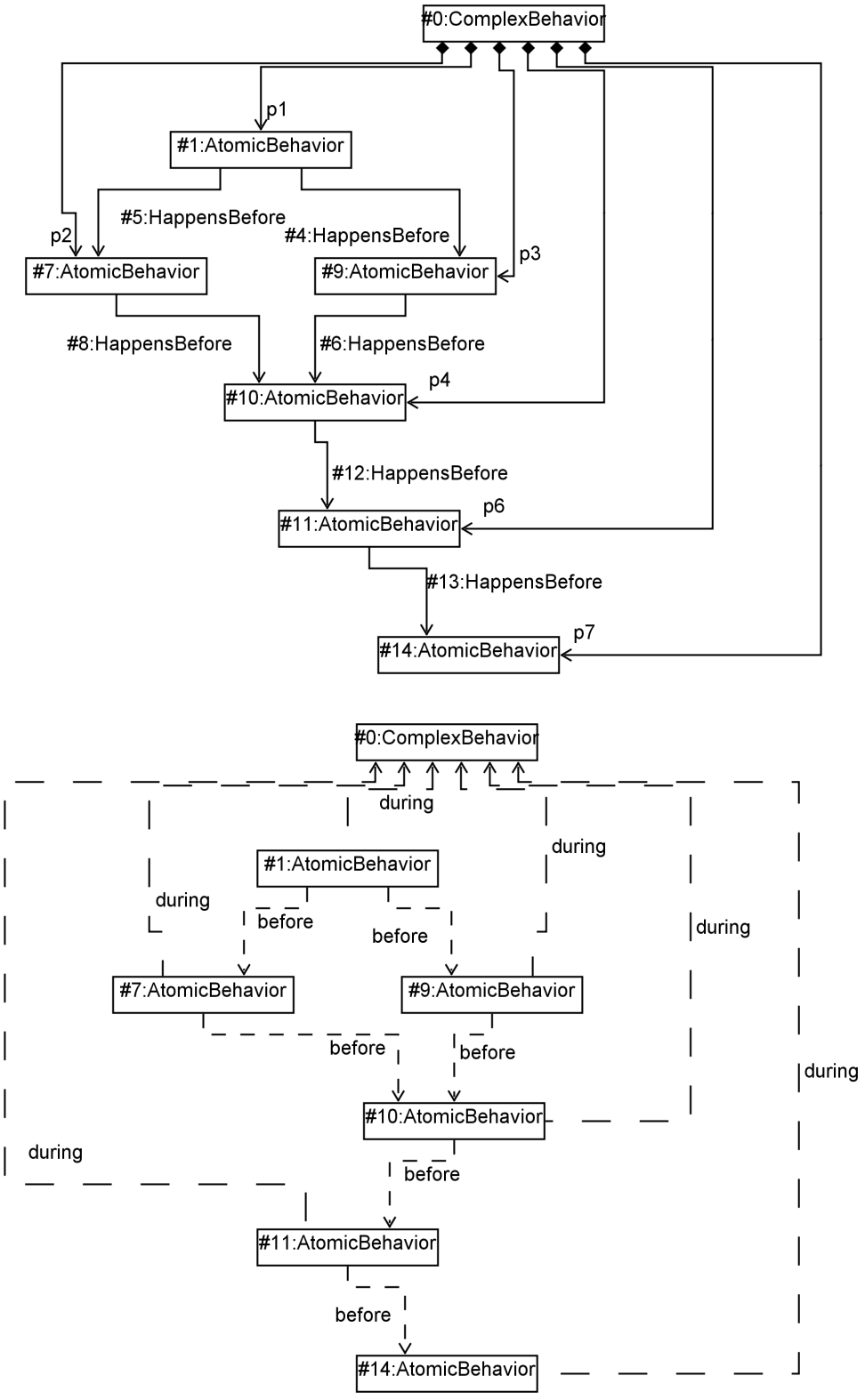


**Fig. 20.** Complete behavior instance model

### 4.1.2 Loops

Figures 21 and 21 show SysML activity and OBM representations of a behavior with a loop, respectively. In the OBM representation, Loop is a block with three properties typed by AtomicBehavior: p1 has multiplicity 1, p2 has multiplicity 2..* (to require at least two occurrences in the loop), and p3 has multiplicity 1..*. There are HappensBefore connectors between p1 and p2, p2 and p3, and from p2 to itself. The connectors around p2 are exclusive, with outgoing connectors acting like a decision, while incoming ones act like a merge (see Section 4.1.1).

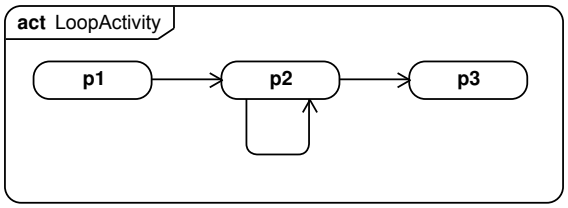


**Fig. 21.** Loop (activity)

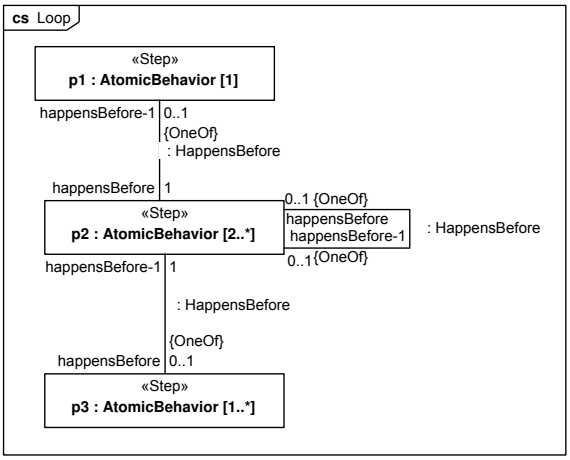


**Fig. 22.** Loop (OBM)

Figure 23 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations in the lower part. #0 is an instance of Loop, with instance #1 as value of p1, instances #8 and #10 as values of p2, and instance #7 as value of p3. HappensBefore connectors exist between #1 and #8, #8 and #10, and #10 and #7. The flow goes a second time to p2 before going to p3, forming a loop. See Section 4.2 for examples of object flows in loops.

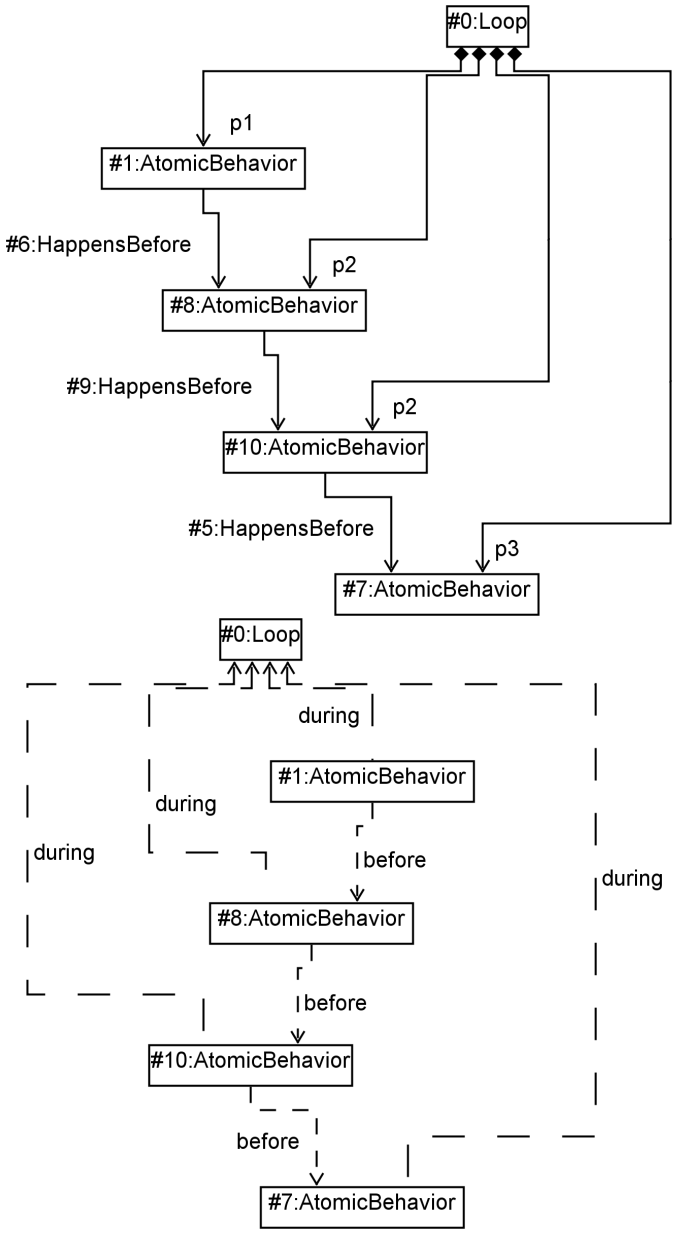


**Fig. 23.** Loop instance model

### 4.1.3 Calling behaviors

Figures 24 and 25 show SysML activity and OBM representations, respectively, of a behavior that contains a composed behavior in addition to atomic behaviors. In the OBM representation, ComposedBehavior is a block with one property p2 typed by NestedBehavior, and two properties p1 and p3 typed by AtomicBehavior. Connectors link p1 to p2, and p2 to p3. NestedBehavior itself has two properties p4 and p5 typed by AtomicBehavior and linked. All multiplicities are equal to 1. The flow is expected

to go from p1 to p2, and from p2 to p3. Within p2, the flow is expected to go from p4 to p5.

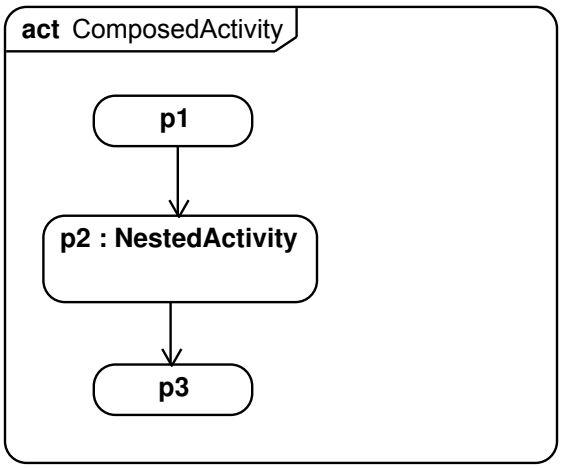


**Fig. 24.** Composed behavior (activity)

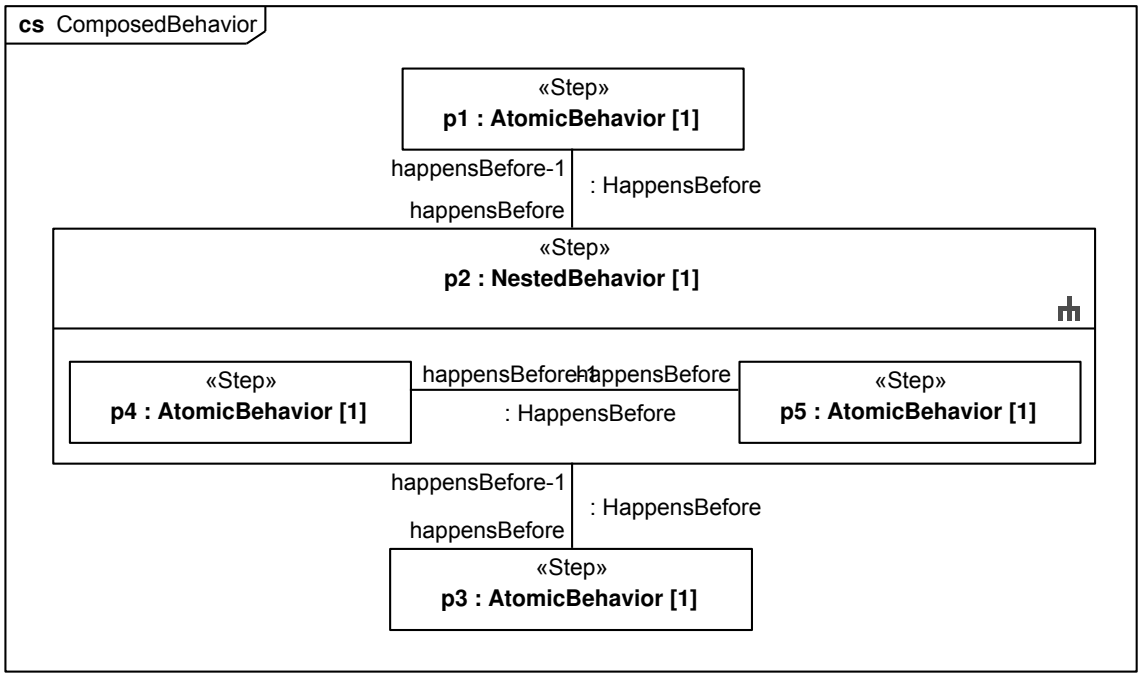


**Fig. 25.** Composed behavior (OBM)

Figure 26 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of ComposedBehavior, with the instance #1 as value of p1, #2 as value of as p2, and #3

as value of p3. The instance #2 itself has the instances #8 and #9 as values of p4 and p5. The temporal relations shows that #1 happens before #2, #2 happens before #3, these three instances happen during #0, #8 and #9 happen during #2, and #8 happens before #9. The steps of the composed behavior happen as expected.

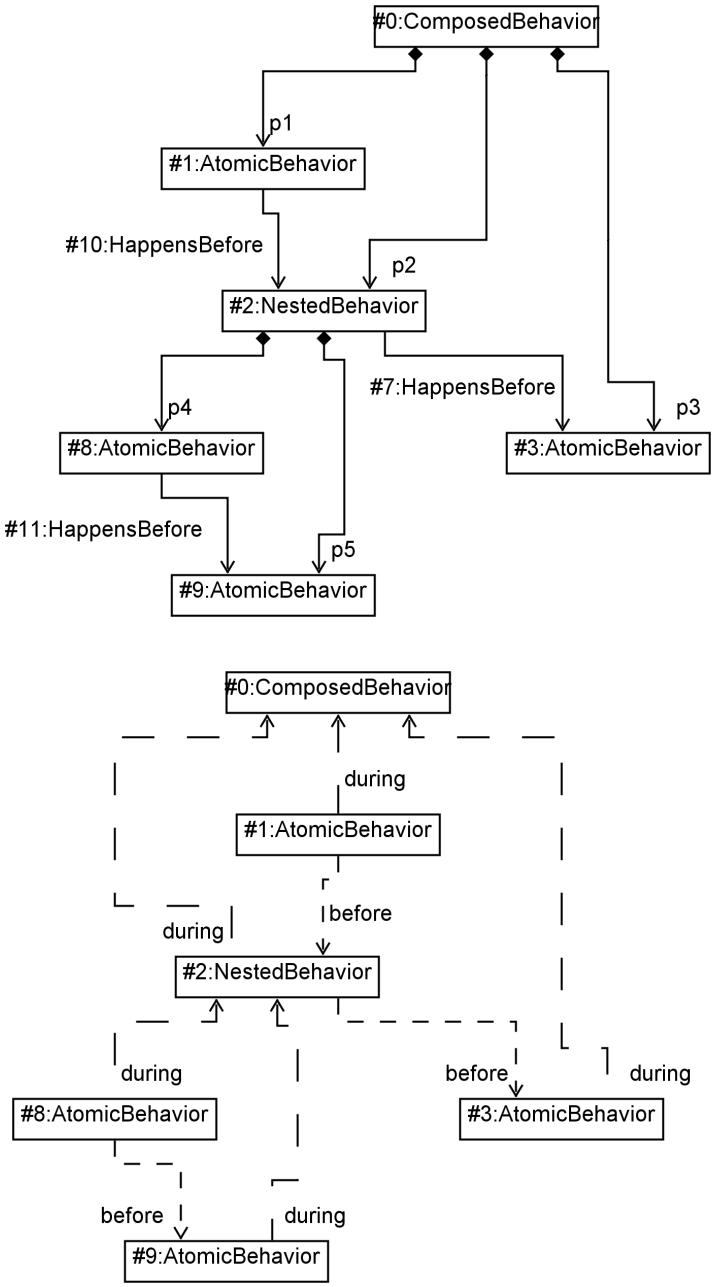


**Fig. 26.** Composed behavior instance model

### 4.1.4 Item flows and parameters

This section presents the use of transfers and behavior parameters, used to pass items to inner behaviors.

**Itme flows** Figures 27 and 28 shows a SysML internal block diagram and an OBM representation of transferring a property value between two participants, respectively. ParticipantTransfer is a block containing a property supplier of type Supplier with a suppliedProduct, and a property customer of type Customer with a receivedProduct. These two properties are linked by a connector of type Transfer and stereotyped by ItemFlow. The stereotype values indicate the flowing item comes from the suppliedProduct property of the source and goes to the receivedProduct property of the target. Solving is expected to show that the product is the same in both participants. The translation does not support changing objects, so the solver cannot produce instances of the two participants when they do not have the product (customer before the transfer, and supplier after the transfer).

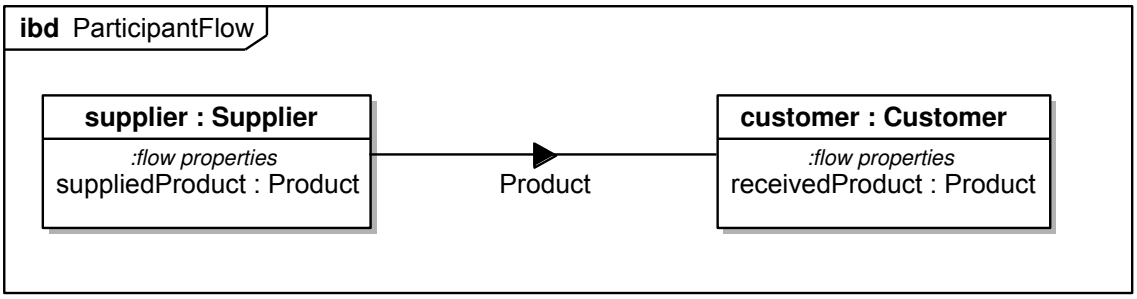


**Fig. 27.** Participant transfer (item flow)

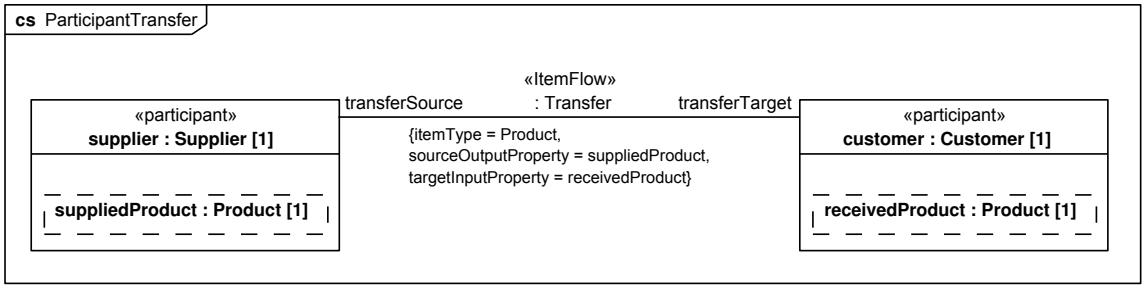


**Fig. 28.** Participant transfer (OBM)

Figure 29 shows an instance model produced by the reasoner.#0 is an instance of ParticipantTransfer, with instance #1 as value of supplier and #2 as value of customer. #8 is a connector typed by Transfer that connects #1 to #2. Both #1 and #2 point

have the product #7 as respectively suppliedProduct and receivedProduct, showing that the model transfers the value from supplier to customer as expected.

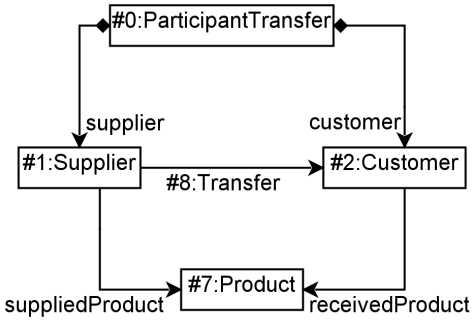


**Fig. 29.** Participant transfer instance model

**Behavior parameters** Figures 30 and 31 show SysML activity and OBM representations of a behavior with a parameter value passed between steps, respectively. In the OBM representation, ParameterBehavior is a block with three property a, b, and c of type A, B, and C respectively. B has two steps b1 and b2 of type B1 and B2 respectively. Each of these behaviors has a value that is being passed through an object flow:

- A has vout, with an outside read access
- B has vin, with an outside write access and an inside read access
- B1 and B2 have vinout, with an outside write and read access
- B has vout, with an outside read access and an inside write access
- C has vout, with an outside write access

Binding connectors link:

- B's vin to b1.vinout
- B's vout to b2.vinout

As a result, it is expected that values will be passed from a.vout to b.vin to b1.vinout to b2.vinout to b.vout to c.in.

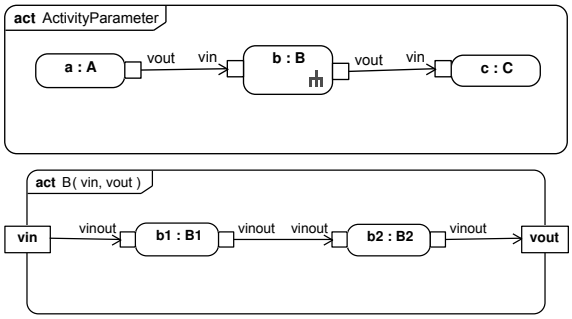


**Fig. 30.** Behavior parameters (activity)

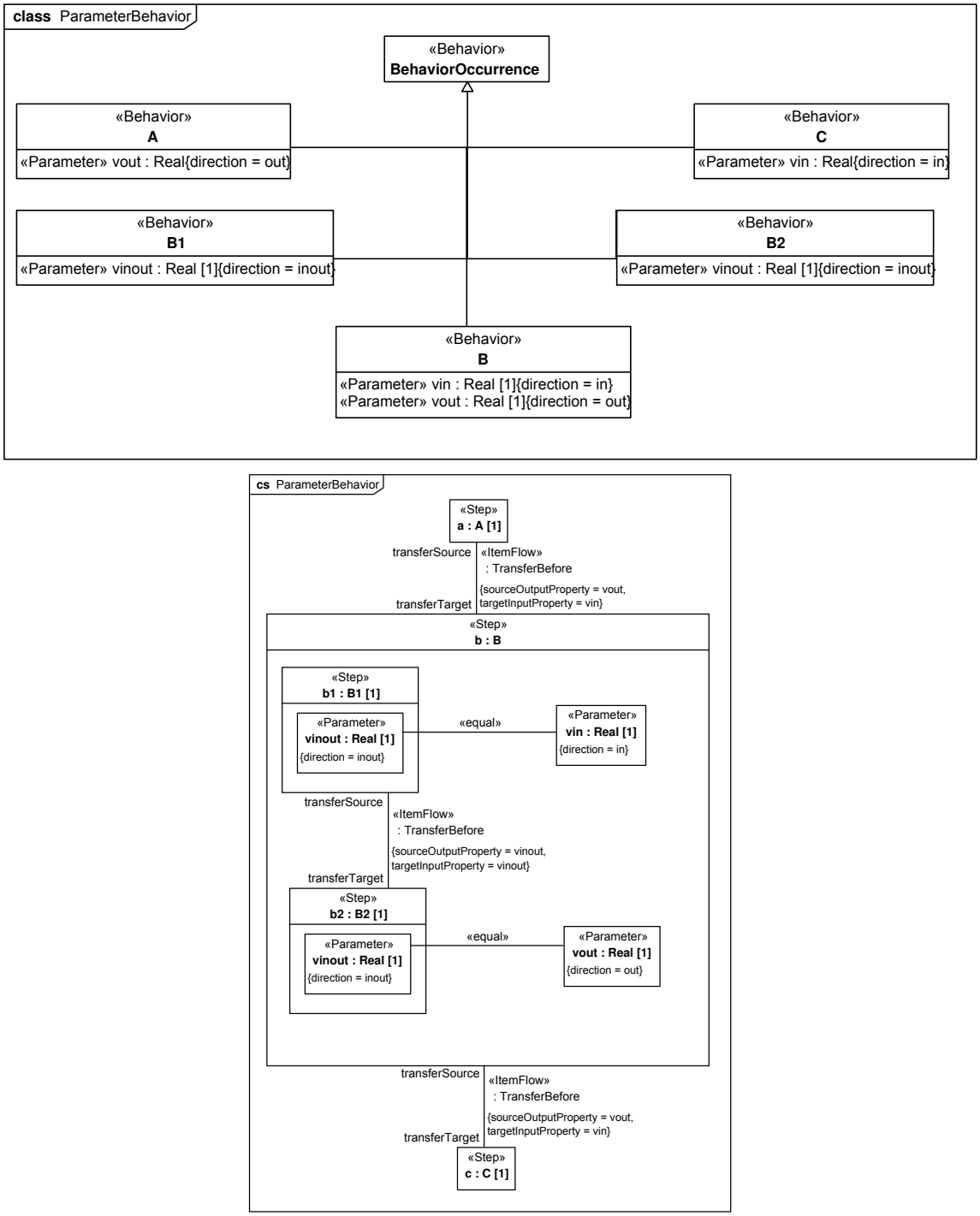


**Fig. 31.** Behavior parameters (OBM)

Figure 32 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of ParameterBehavior, with instance #1 as value of a, #2 as value of b, and #3 as value

of c. #2 has the instances #7 as b1 and #8 as b2. All these instances have the same value for the property vin/vinout/vout), confirming that the parameter was passed correctly between all these instances.

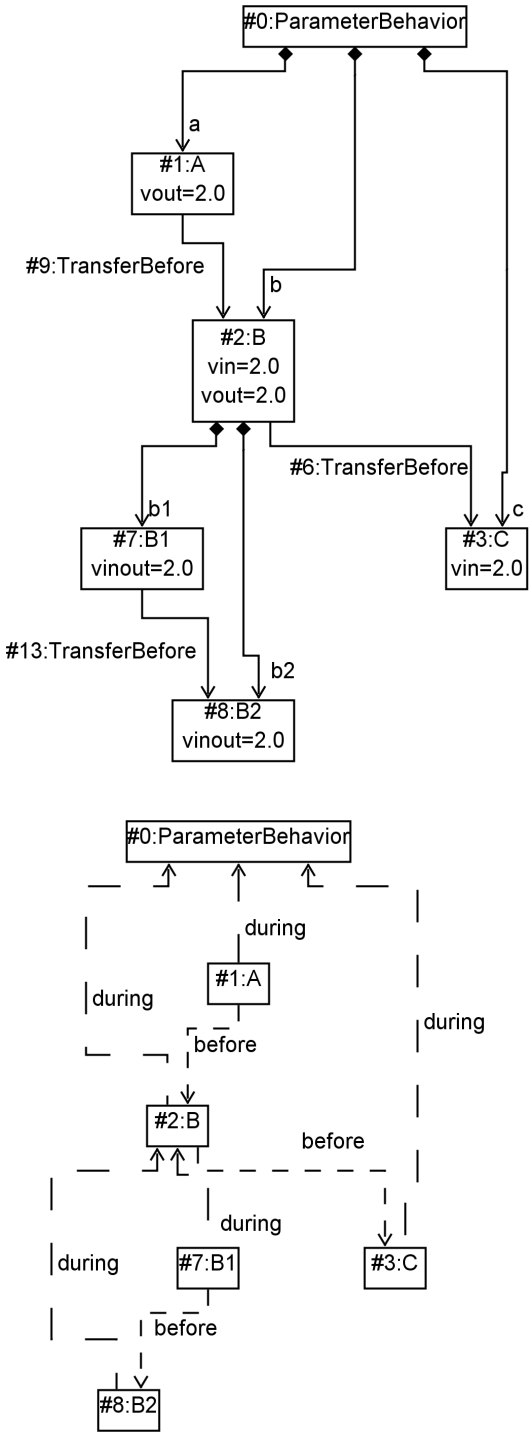


**Fig. 32.** Behavior parameters instance model

### 4.1.5 Steps with multiple executions

A BehaviorOccurrence can have steps that are executed multiple times.

**Control flows** Figures 33 and 34 show SysML activity and OBM representations, respectively, of a behavior with a fork and a join connected with control flows, with two occurrences on the first step (activity actions do not have multiplicity). In the OBM representation, ControlFlowBehavior is a block with four properties p1, p2, p3, and p4 of type AtomicBehavior. They are connected by HappensBefore connectors so that p1, p2 and p3 form a fork, and p2, p3 and p4 form a join. The multiplicity on p1 is 2, indicating that there should be two flows going from p1 to p4.

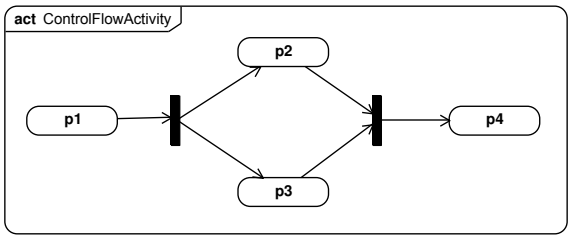


**Fig. 33.** Multiple control flows (activity)

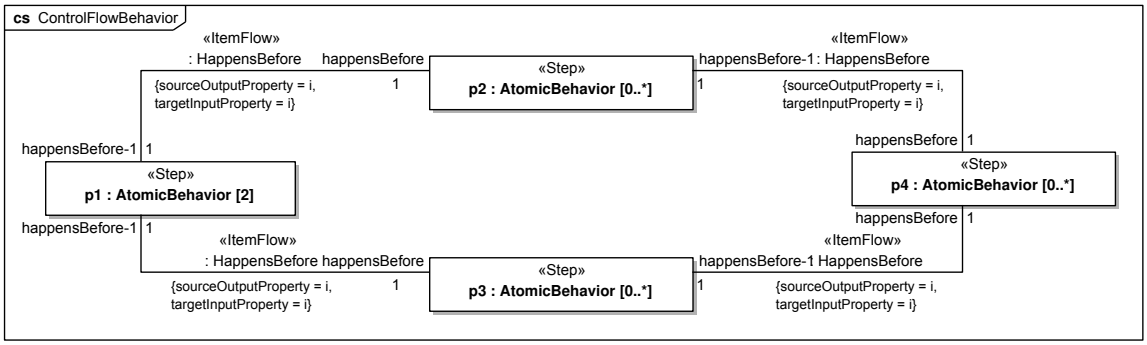


**Fig. 34.** Multiple control flows (OBM)

Figure 35 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of ControlFlowBehavior, with instances #12 and #5 as values of p1, instances #14 and #9 as values of p2, instances #1 and #6 as values of p3, instances #2 and #10 as values of p4. The first flow goes from #12 to #14 and #1, and from there to #2. The second flow goes from #5 to #9 and #6, and from there to #10.

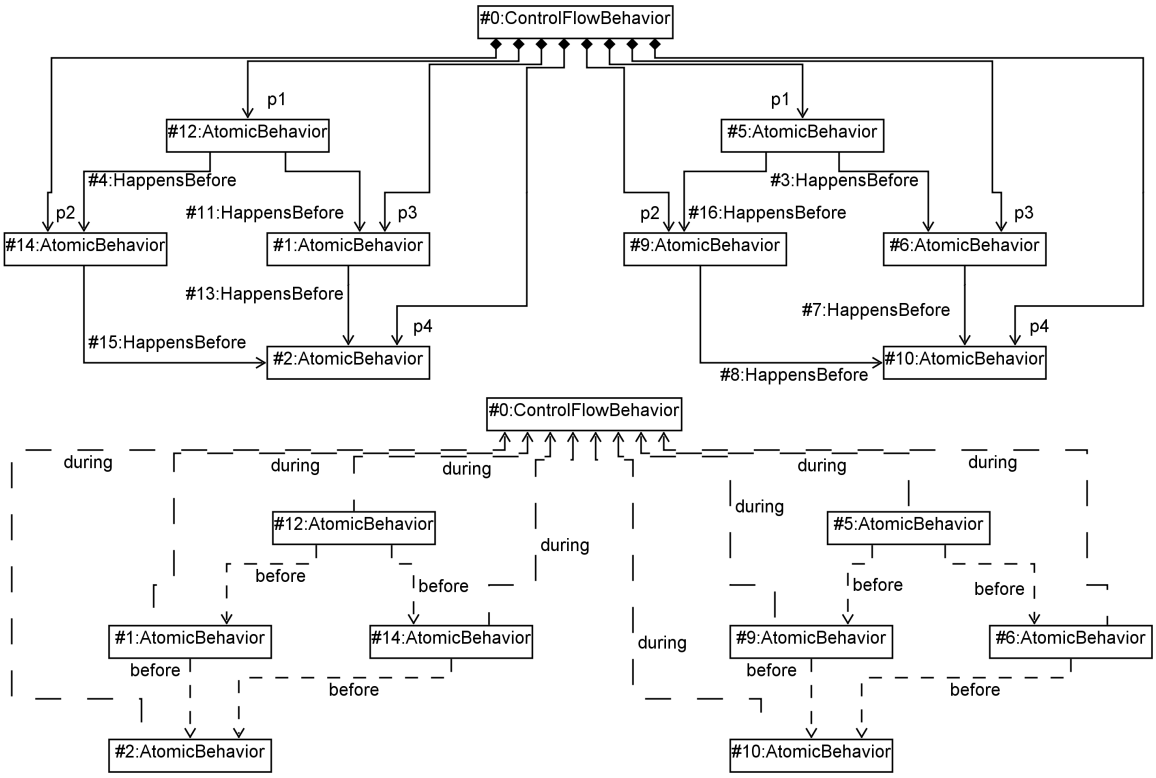


**Fig. 35.** Control flows instance model

In this example, control flows only specify temporal order between occurrences, nothing actually flows between them that could be taken as representing a "flow". Once the flow splits from a value of `p1`, it is possible that one branch goes to one value of `p4`, and the other branch goes to another value of `p4`, as shown in Figure 36. The flow can go from `#1` to `#10` instead of `#2`, and from `#6` to `#2` instead of `#10`

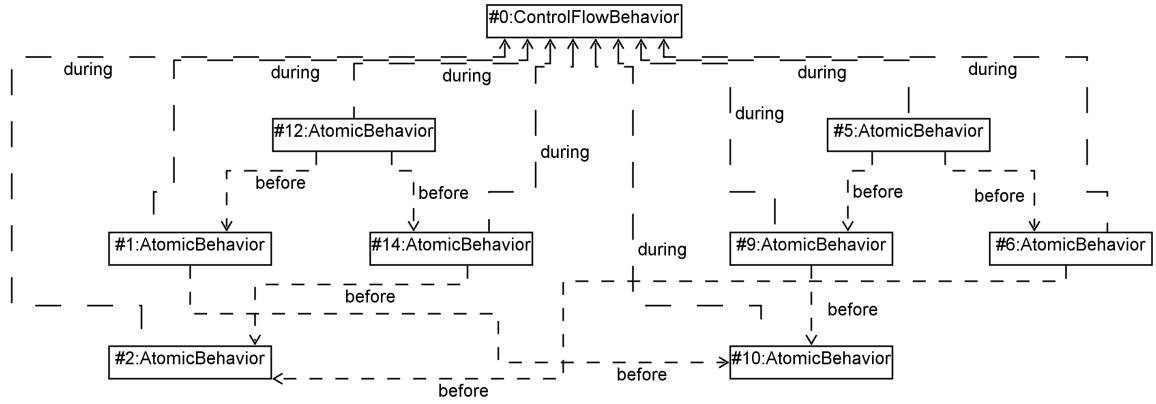


**Fig. 36.** Control flows instance model with mixed flows

**Object flows** Figures 37 and 38 show SysML activity and OBM representations, respectively, of a behavior with fork and a join connected with object flows, with two occurrences on the first step. In the OBM representation, ObjectFlowBehavior has four properties p1, p2, p3, and p4 of type BehaviorWithParameter. They are connected by TransferBefore connectors so that p1, p2 and p3 form a fork, and p2, p3 and p4 form a join. These connectors indicate that the value of i is copied between the steps involved. The multiplicity on p1 is 2, indicating that there should be two flows going from p1 to p4.

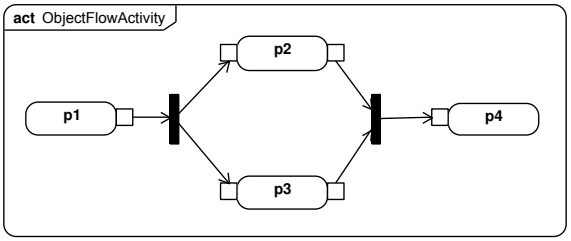


**Fig. 37.** Multiple object flows (activity)

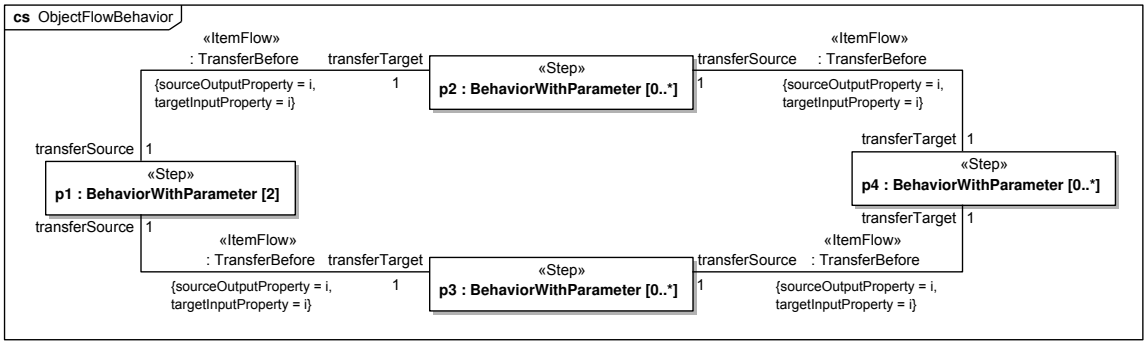


**Fig. 38.** Multiple object flows (OBM)

Figure 39 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of ObjectFlowBehavior, with instances #7 and #14 as values of p1, instances #8 and #15 as values of p2, instances #6 and #1 as values of p3, instances #12 and #2 as values of p4. The first flow, with a value of 3 for i, goes from #7 to #8 and #6, and from there to #12. The second flow, with a value of 10 for i, goes from #14 to #15 and #1, and from there to #2.

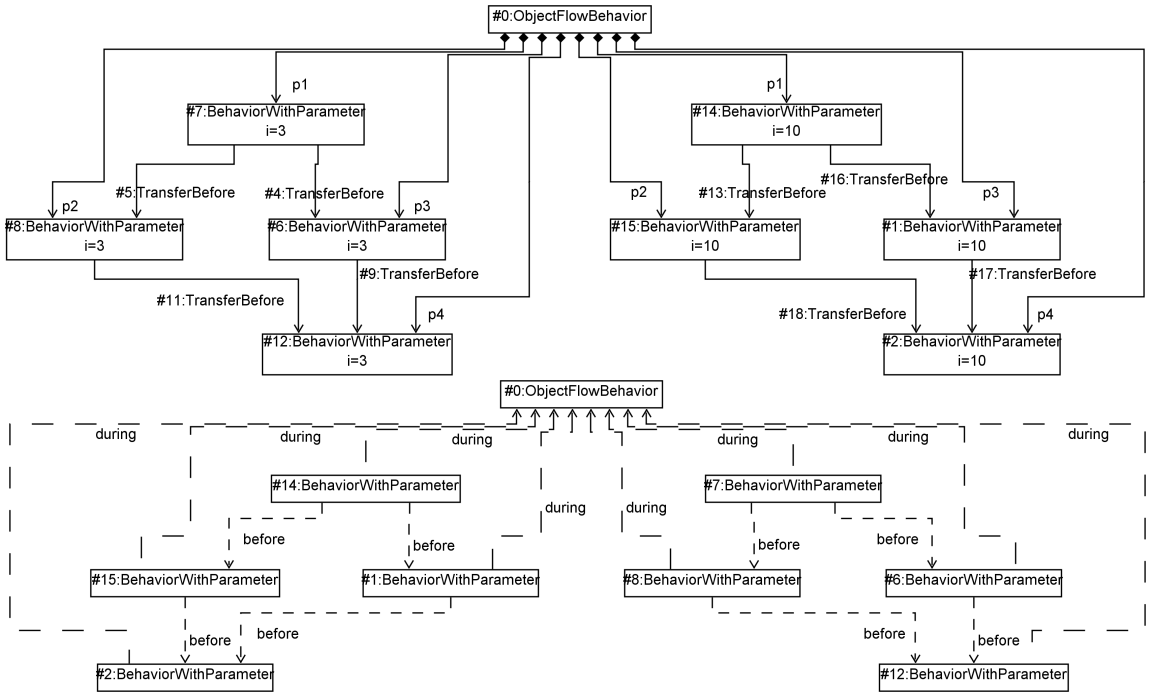


**Fig. 39.** Object flows instance model

In this example, the value of i in p1 is passed along all successors, so the value in p1 going to the two branches must end up going to the same value in p4.

### 4.1.6 Unsatisfiable

The section presents some unsatisfiable OBM models.

**Due to connector end multiplicity** Figures 40 and 41 show SysML activity and OBM representations, respectively, of a behavior with a decision followed by a join, which is found to be unsatisfiable due to the multiplicities on connector ends. In the OBM representation, UnsatisfiableMultiplicity is a block with four properties typed by AtomicBehavior: p1, p2, p3, and p4. They are connected so that p1, p2, and p3 form a Decision, and p2, p3, and p4 form a Join. The model specifies that flow should go from p1 to either p2 or p3, but not both, and from p2 and p3 to p4. This is not possible because the decision prevents p2 and p3 from both having values to complete the join. SMT verification confirms the model is unsatisfiable.

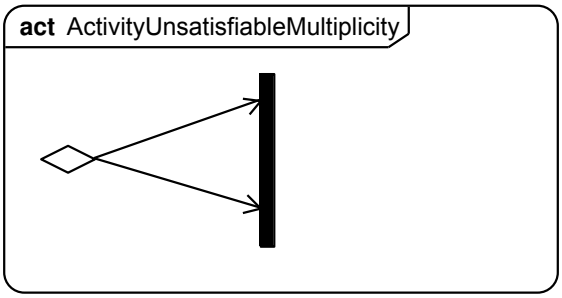


**Fig. 40.** Unsatisfiable model, connector end multiplicity (activity)

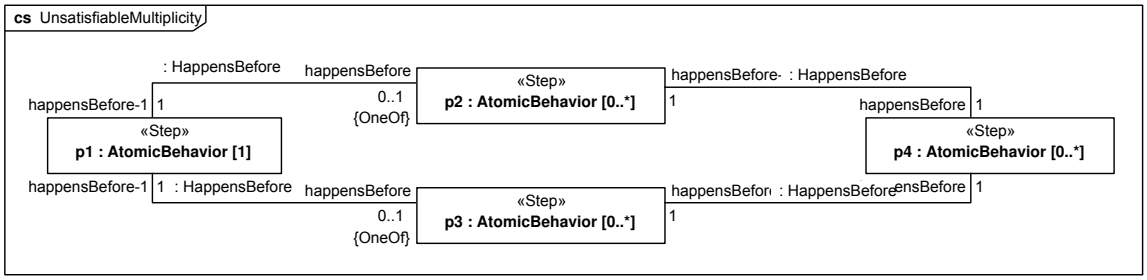


**Fig. 41.** Unsatisfiable model, connector end multiplicity (OBM)

**Due to asymmetry of HappensBefore** Figures 42 and 43 show SysML activity and OBM representations, respectively, of a behavior with two occurrences happening before each other, which is found to be unsatisfiable due to HappensBefore being asymmetric. In the OBM representation, UnsatisfiableAsymmetry is a block with two properties of multiplicity 1 typed by AtomicBehavior: p1 and p2. They are connected so that p1 happens before p2, and vice versa, which is not allowed by asymmetry. SMT verification confirms the model is unsatisfiable.

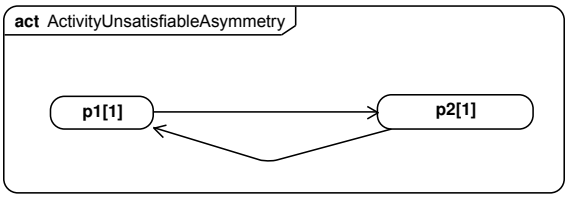


**Fig. 42.** Unsatisfiable model, asymmetry (activity)

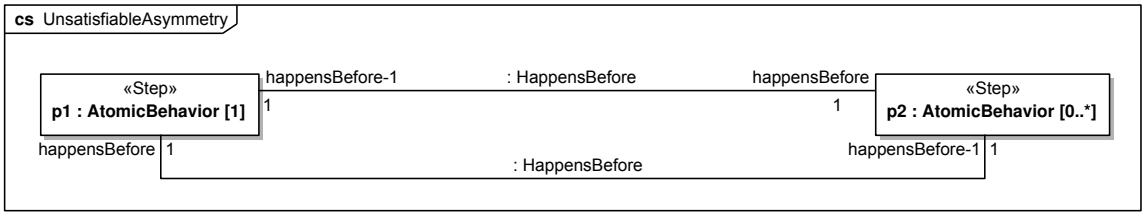


**Fig. 43.** Unsatisfiable model, asymmetry (OBM)

**Due to transitivity of HappensBefore** Figures 44 and 45 show SysML activity and OBM representations, respectively, of a behavior with three occurrences happening before each other, which is found to be unsatisfiable due to HappensBefore being transitive and asymmetric. In the OBM representation, UnsatisfiableTransitivity is a block with three properties of multiplicity 1 typed by AtomicBehavior: p1, p2, and p3. They are connected in a circle, implying by transitivity that they all happen before themselves and each other bidirectionally, which is not allowed by asymmetry. SMT verification confirms the model is unsatisfiable.

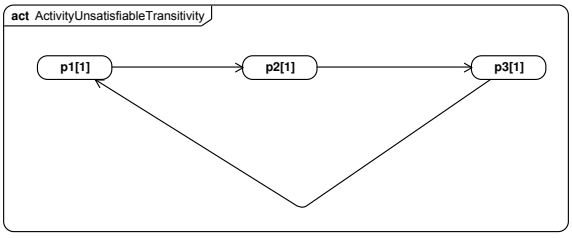


**Fig. 44.** Unsatisfiable model, transitivity (activity)

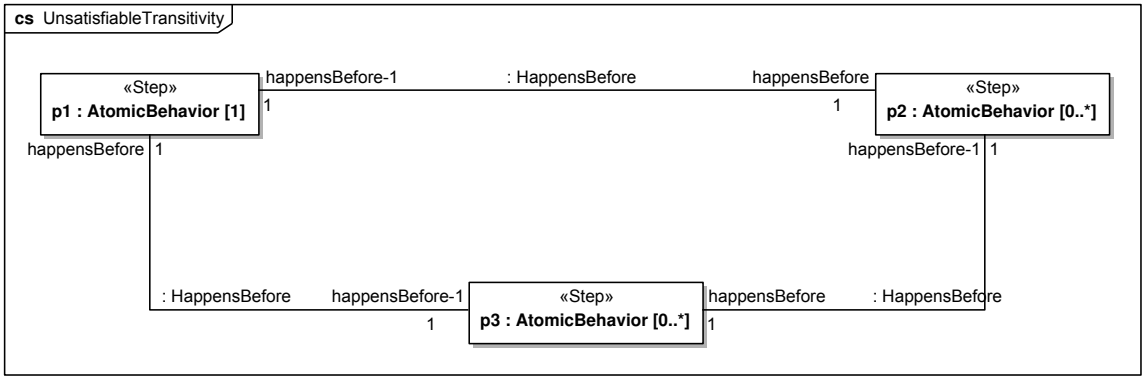


**Fig. 45.** Unsatisfiable model, transitivity (OBM)

**Due to logical interaction of temporal relations (1)** Figures 46 and 47 show SysML activity and OBM representations, respectively, of a behavior with three

occurrences connected by temporal relations, which is found to be unsatisfiable due to logical interaction of HappensBefore and HappensDuring. In the OBM representation, UnsatisfiableComposition1 is a block with three properties of multiplicity 1 typed by AtomicBehavior: p1, p2, and p3. They are connected so that p1 happens before p2, and p3 happens during p2, and p3 happens before p1. Logical interaction of the temporal relations implies p1 and p3 happen before each other, which is not allowed by asymmetry. SMT verification confirms the model is unsatisfiable.

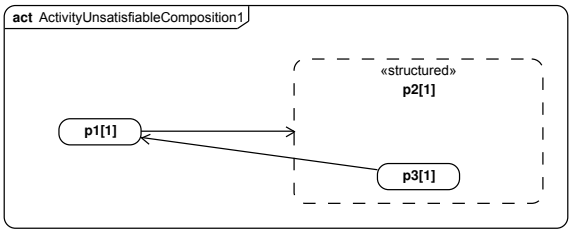


**Fig. 46.** Unsatisfiable model, logical implications 1 (activity)

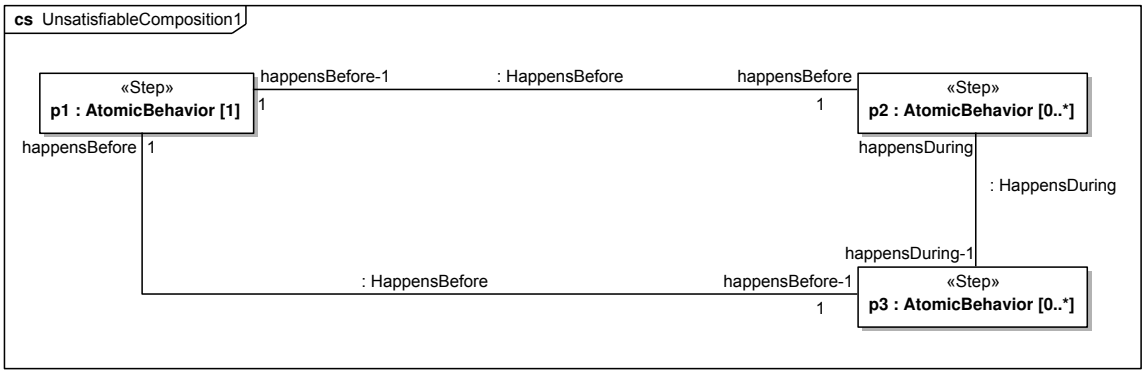


**Fig. 47.** Unsatisfiable model, logical implications 1 (OBM)

**Due to logical interaction of temporal relations (2)** Figures 48 and 49 show SysML activity and OBM representations, respectively, of a behavior with three occurrences connected by temporal relations, which is found to be unsatisfiable due to logical interaction of HappensBefore and HappensDuring. In the OBM representation, UnsatisfiableComposition2 is a block with three properties of multiplicity 1 typed by AtomicBehavior: p1, p2, and p3. They are connected so that p1 happens before p2, and p3 happens during p1, and p2 happens before p3. Logical interaction of the temporal relations implies p2 and p3 happen before each other, which is not allowed by asymmetry. SMT verification process confirms the model is unsatisfiable.

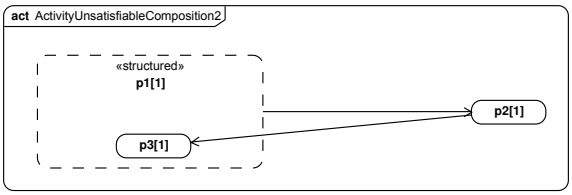


**Fig. 48.** Unsatisfiable model, logical implications 2 (activity)

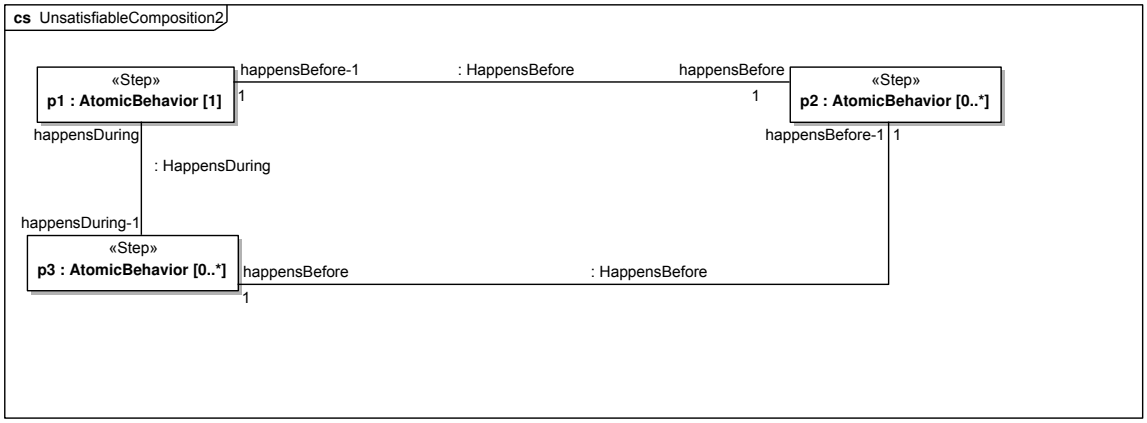


**Fig. 49.** Unsatisfiable model, logical implications 2 (OBM)

## 4.2 Advanced examples

This section shows examples adapted from [16], illustrating how to model taxonomy of behaviors using OBM. Section 4.2.1 shows examples that use control flows with a single occurrence per step (equivalent to UML activities with a single token), while Section 4.2.2 shows examples with object flows and multiple occurrences per steps (equivalent to UML activities with multiple tokens).

### 4.2.1 Control flow examples

**Generic Food service** Figure 50 shows a FoodService block, which is a generic definition of food service. The block has five properties: prepare, order, serve, eat, pay. The multiplicity of these properties is 0..*. The type corresponding to these properties are all specializations of BehaviorOccurrence: Order, Prepare, Serve, Eat, Pay. FoodService will be specialized and its properties redefined by concrete specializations, as needed.

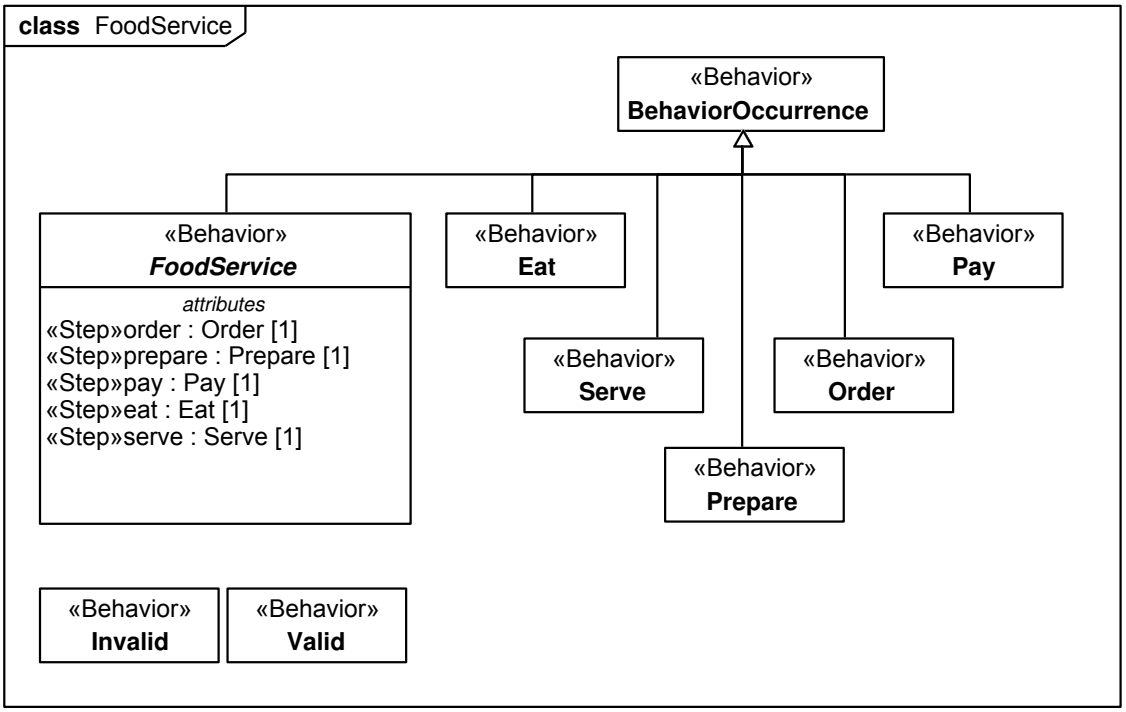


**Fig. 50.** Food service

Figures 51 and 52 show SysML activity and OBM representations of the generic FoodService, respectively. In the OBM representation, connectors are used to mark the temporal relationships that are true for all food services. These are the following: prepare and order must happen before serve, serve must happen before eat. Nothing is said about pay. The connectors will be inherited, and possibly redefined, by all specializations of FoodService.

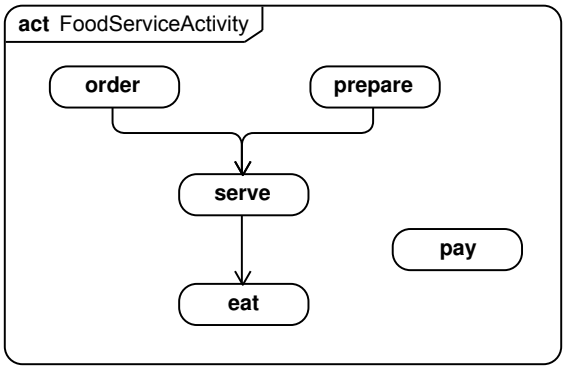


**Fig. 51.** Food service actions (activity)

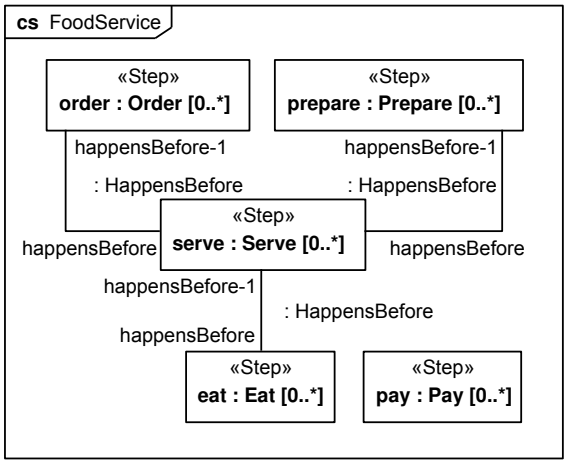


**Fig. 52.** Food service actions (OBM)

**Generic single food service** Figure 53 shows the SingleFoodService block, which is a specialization of FoodService that redefines all its properties to have a multiplicity of exactly one. This block has several specializations in which the steps are ordered differently: BuffetService, ChurchSupper, FastFoodService, and RestaurantService. The next subsections describe these specializations.

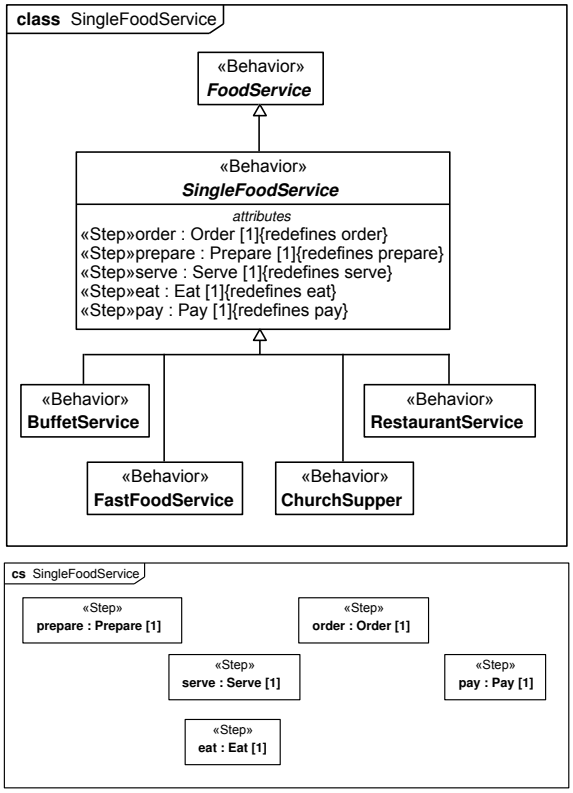


**Fig. 53.** Single food service

**Buffet** Figures 54 and 55 show SysML activity and OBM representations of a buffet food service, respectively. In the OBM representation, BuffetService is a specialization of SingleFoodService in which prepare occurs before order, and pay occurs after eat.

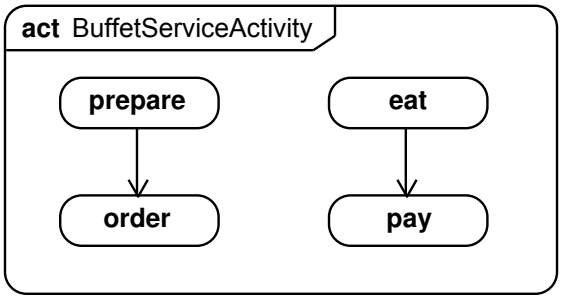


**Fig. 54.** Buffet service (activity)

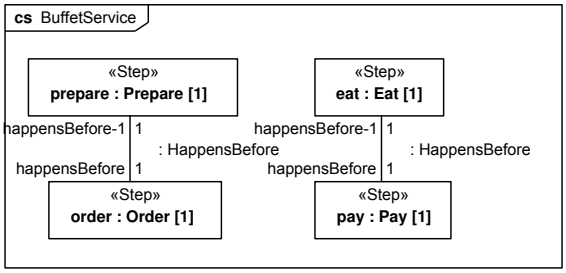


**Fig. 55.** Buffet service (OBM)

Figure 56 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of BuffetService with the instances #1, #2, #3, #4, #5 as values of order, prepare, serve, eat, pay. The temporal relations show that #2 happens before #1, #1 happens before #3, #4 happens before #4, #4 happens before #5.

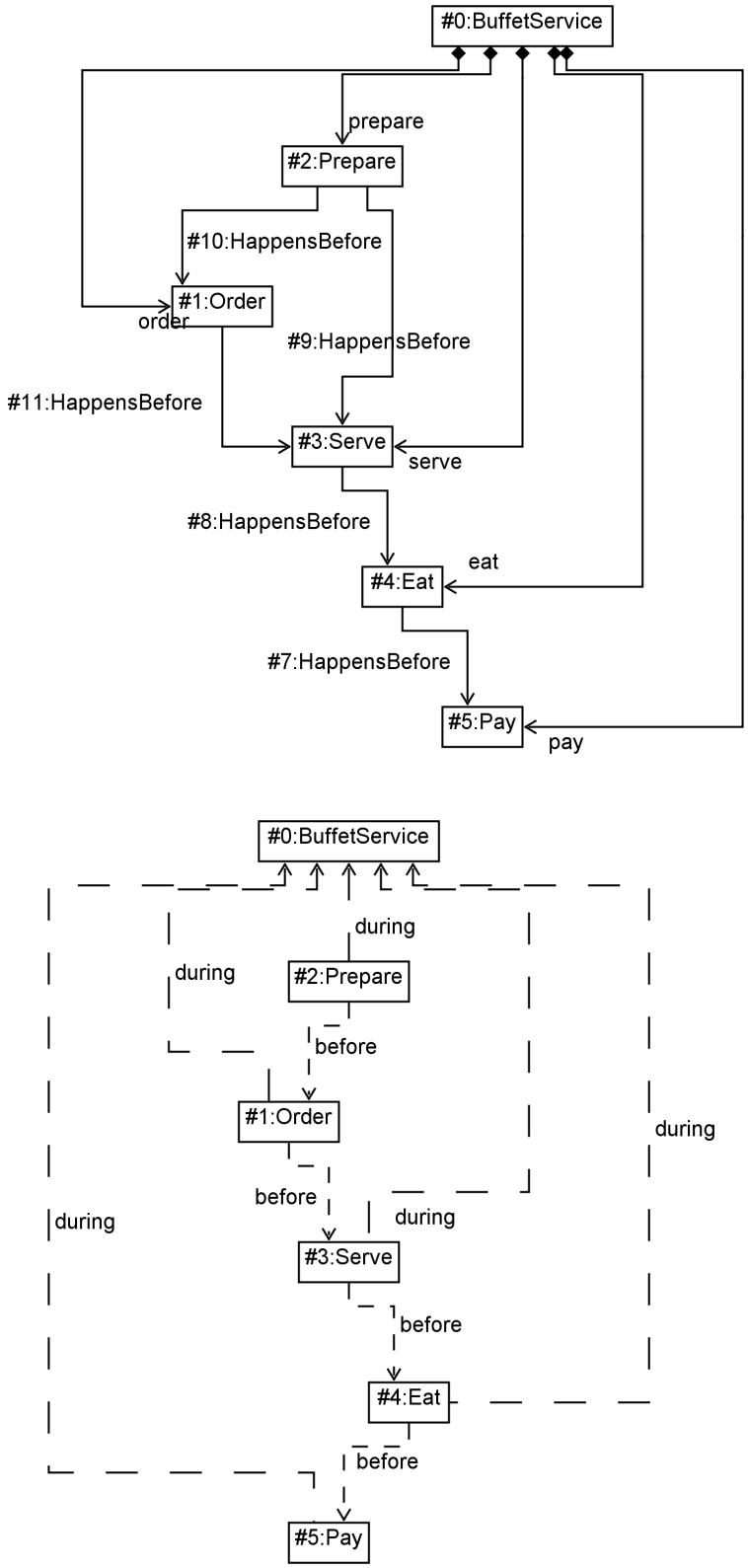


**Fig. 56.** Buffet instance model

**Church** Figures 57 and 58 show SysML activity and OBM representations of a church supper, respectively. In the OBM representation, ChurchSupper is a specialization of SingleFoodService in which pay occurs before prepare and order.

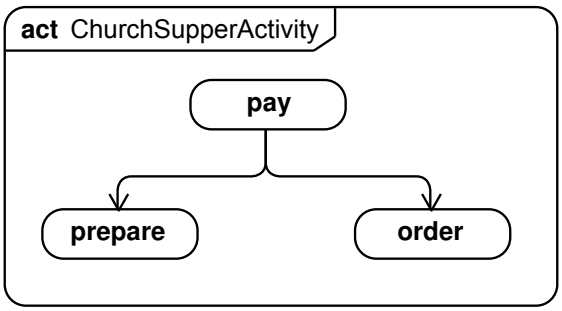


**Fig. 57.** Church supper (activity)

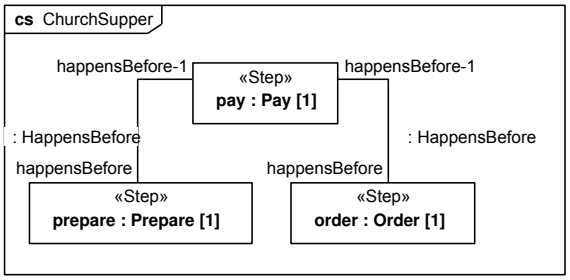


**Fig. 58.** Church supper (OBM)

Figure 59 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of ChurchSupper with the instances #1, #2, #3, #4, #5 as values of order, prepare, serve, eat, pay. The temporal relations show that #5 happens before #1 and #2, #1 and #2 happen before #3, #3 happens before #4.

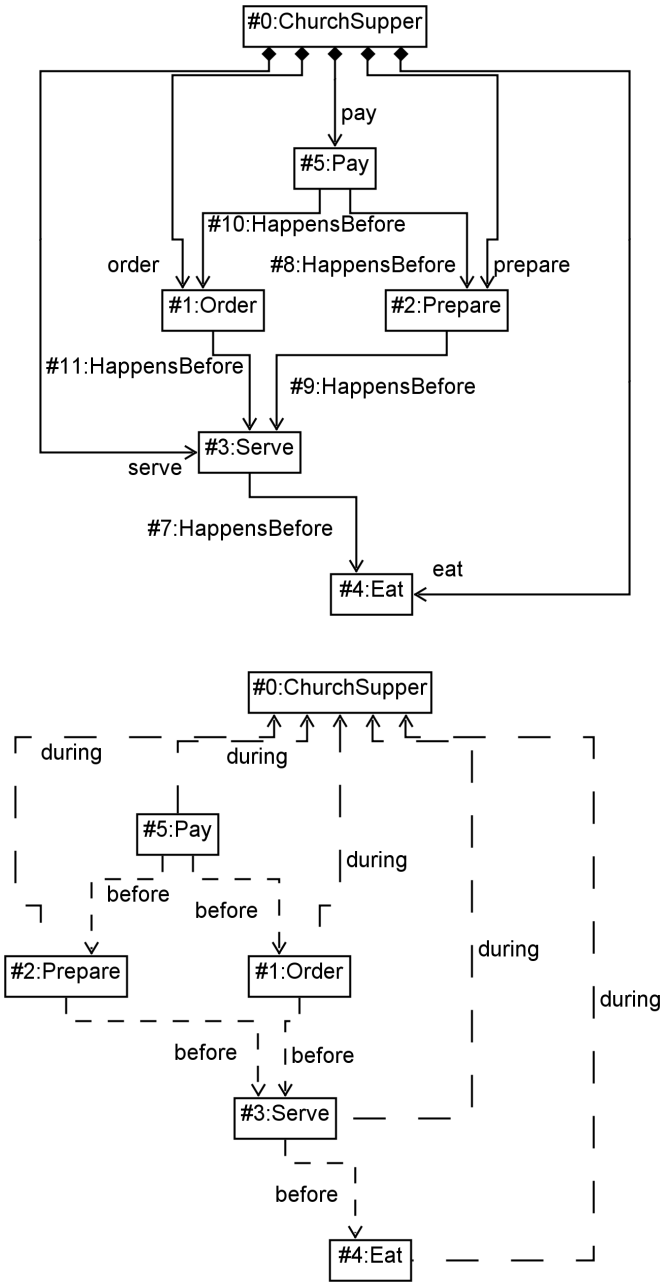


**Fig. 59.** Church supper instance model

**Fast Food** Figures 60 and 61 show SysML activity and OBM representations of a fast food service, respectively. In the OBM representation, FastFoodService is a specialization of SingleFoodService in which pay occurs before eat.

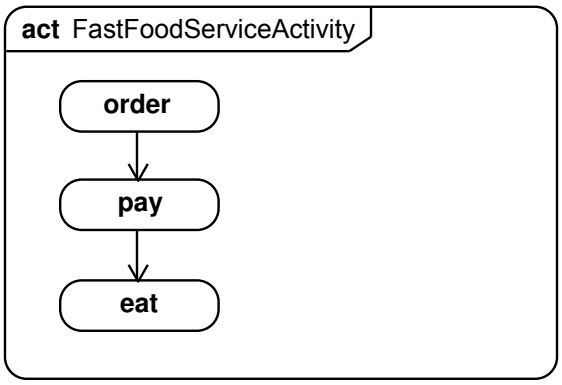


**Fig. 60.** Fast food service (activity)

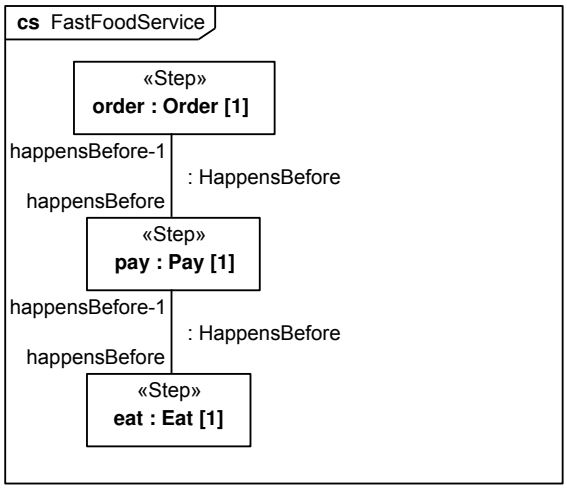


**Fig. 61.** Fast food service (OBM)

Figure 62 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of FastFoodService with the instances #1, #2, #3, #4, #5 as values of order, prepare, serve, eat, pay. The temporal relations show that #1 happens before #5, and #3 and #5 happen before #4.

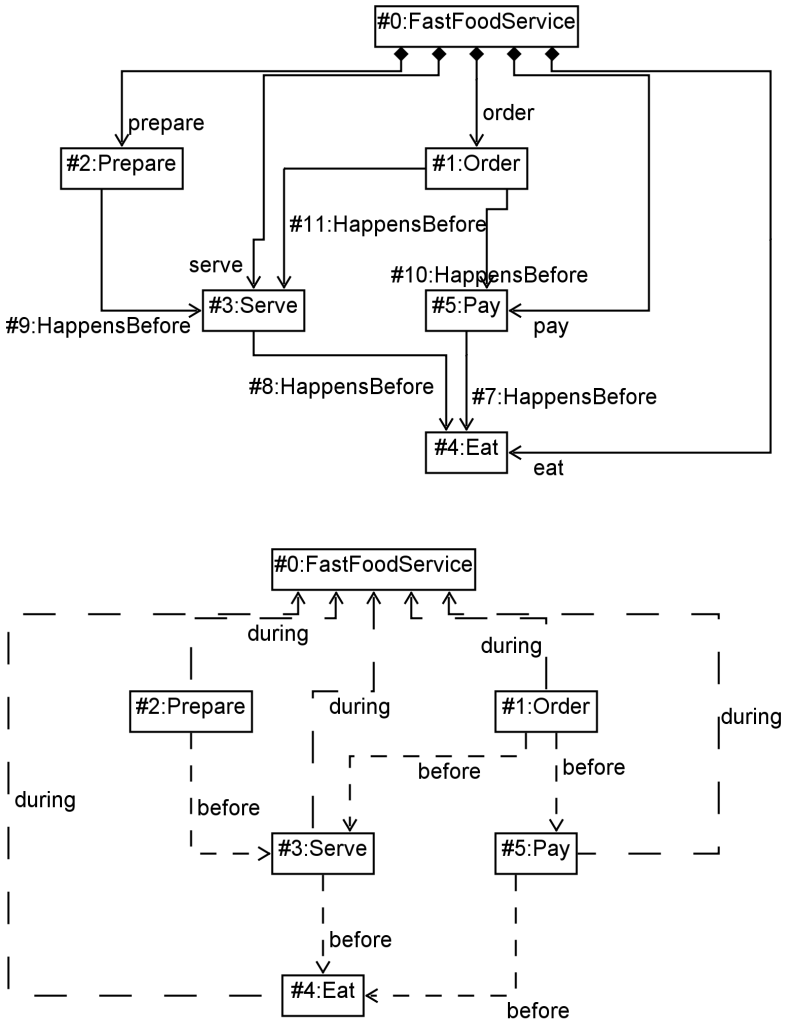


**Fig. 62.** Fast food instance model

**Restaurant** TheFigures 63 and 64 show SysML activity and OBM representations of a restaurant service, respectively. In the OBM representation, RestaurantService is a specialization of SingleFoodService in which pay occurs after eat.

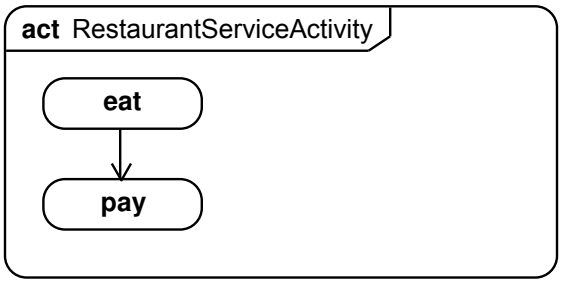


**Fig. 63.** Restaurant service (activity)

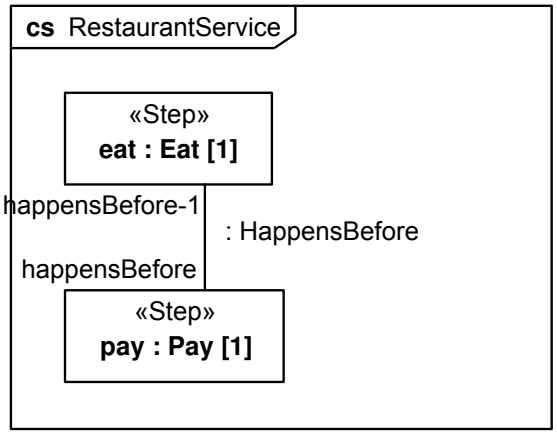


**Fig. 64.** Restaurant service (OBM)

Figure 65 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of RestaurantService with the instances #1, #2, #3, #4, #5 as values of order, prepare, serve, eat, pay. #1 and #2 happen before #3, #3 happens before #4, and #4 happens before #5.

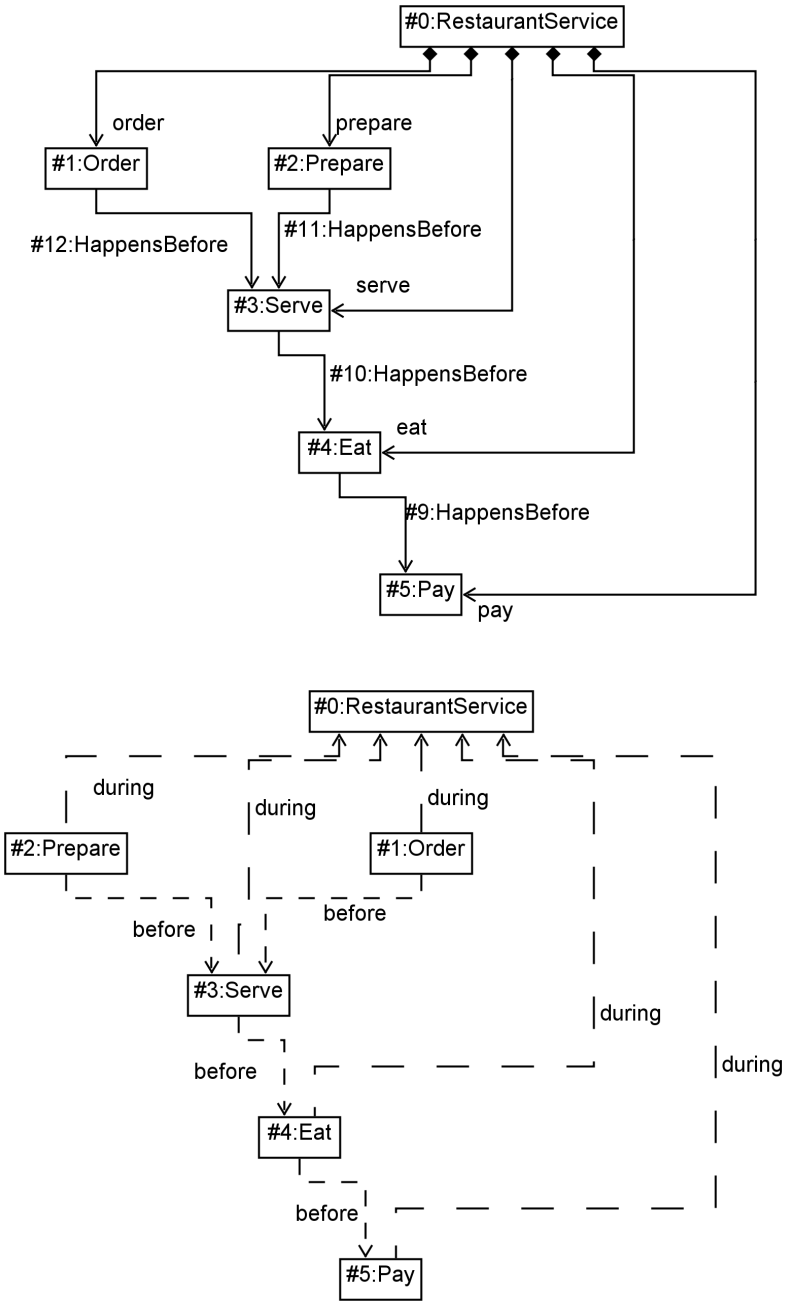


**Fig. 65.** Restaurant instance model

**Unsatisfiable** Figures 66 and 67 show SysML activity and OBM representations, respectively, of a food service that cannot have any execution. In the OBM representation, UnsatisfiableService is a specialization of SingleFoodService in which eat occurs before pay, and pay occurs before prepare. This service is unsatisfiable since these connectors imply that eat occurs before prepare.

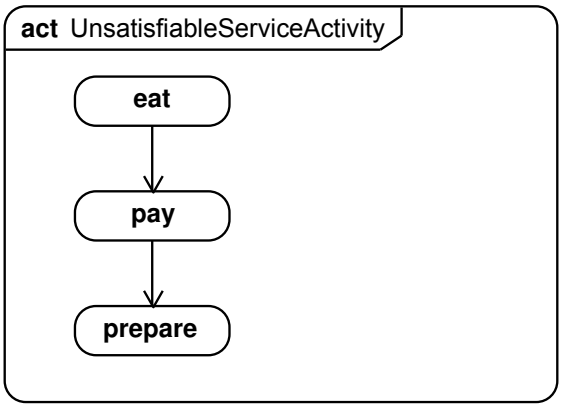


**Fig. 66.** Unsatisfiable service (activity)

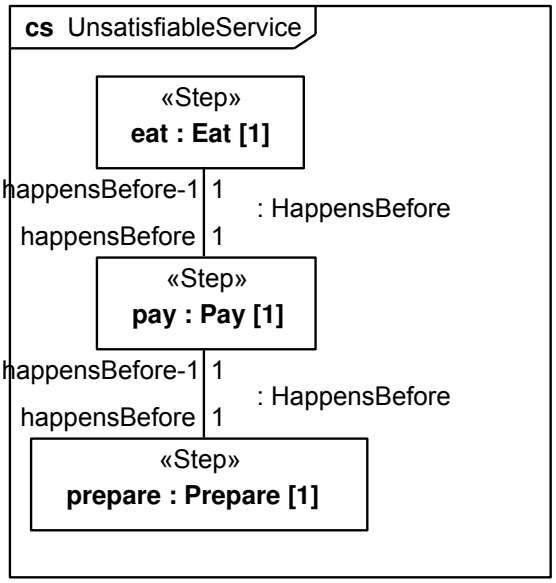


**Fig. 67.** Unsatisfiable service (OBM)

The verification process correctly returns that this model is unsatisfiable, since eat cannot occur before prepare.

### 4.2.2 Object flow examples

**Generic food item flow service** Figure 68 shows a generic item flow food service block IFFoodService, which is a specialization of FoodService in which prepare, serve, eat, and pay are redefined to have a specialized type that carries some information. The blocks IFOrder, IFPrepare, IFServe, IFEat, and IFPay have an additional property that correspond respectively to the ordered food, the prepared food, the served food,

the eaten food, and paid food item and the amount paid. These are common to any kind of food service tracking the food. More specializations are created for actions that exist in some "custom" food services. IFCustomOrder has properties for the destination of the food, and the amount of the order. IFCustomPrepare and IFCustomServe have a property for the destination of the food. An OCL constraint ensures the food item and the destination are unique for every order.

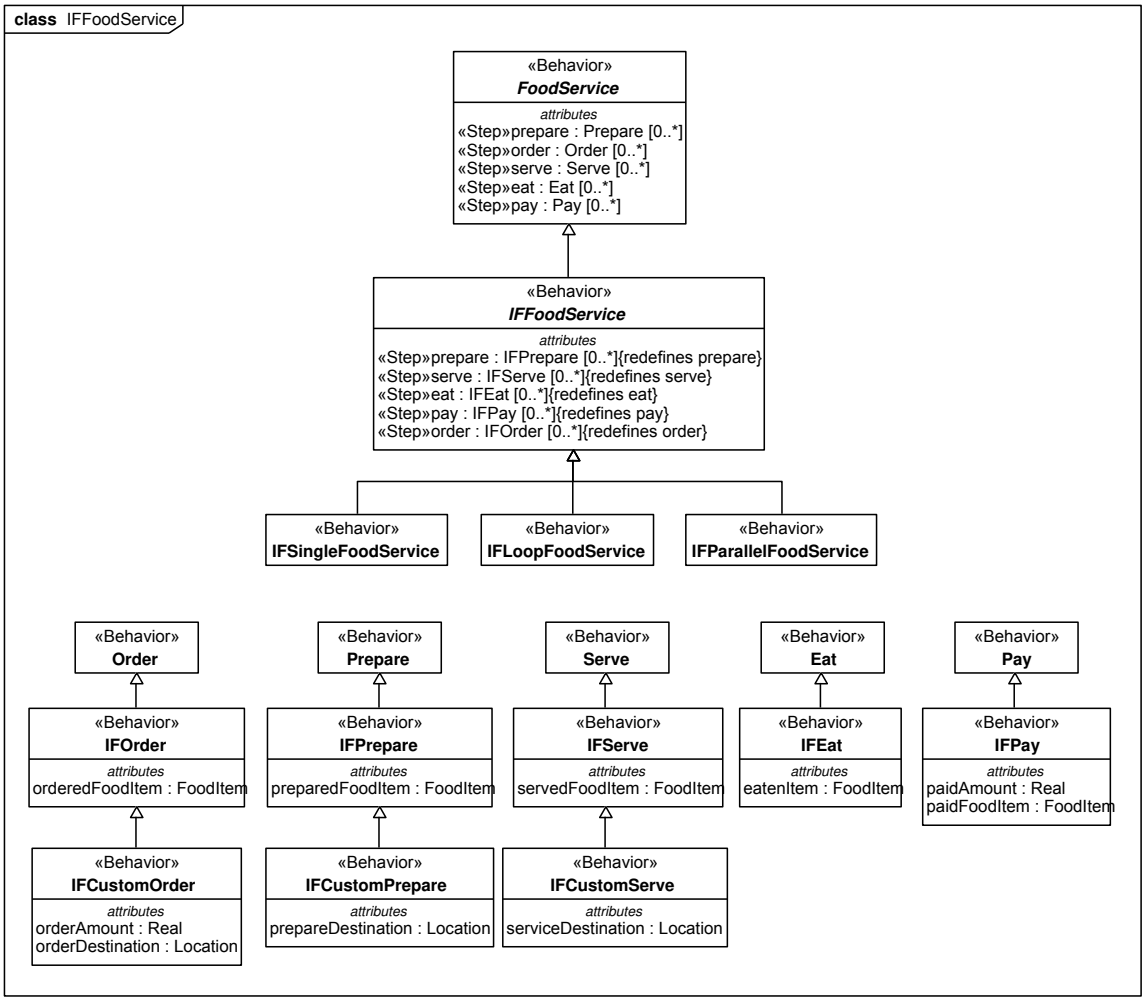


**Fig. 68.** Generic food item flow service

Figure 69 shows the generic composition of item flow food services. IFFoodService has connectors that are common to all food services: the flow of food from prepare to serve, and the flow of food from serve to eat. A connector from order to serve also shows the flow of food. These item flows redefine the corresponding HappensBefore relations defined in FoodService.

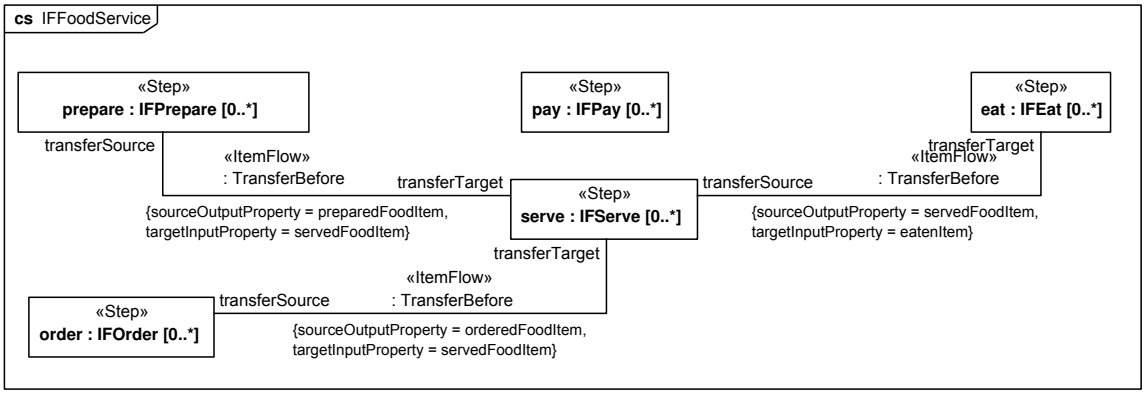


**Fig. 69.** Composition of generic food item flow service

**Single item flow food service** Figures 70 and 71 show SysML activity and OBM representations, respectively, of a food service in which items flow once. In the OBM representation, IFSingleFoodService is a specialization of IFFoodService that redefines its properties to use IFCustomOrder, IFPay, IFCustomPrepare, IFCustomServe, and IFEat. The amount of money is flowing from order to pay, and the destination is flowing from order to prepare, and from prepare to serve. Also, the food item is flowing from pay to eat, and from serve to eat.

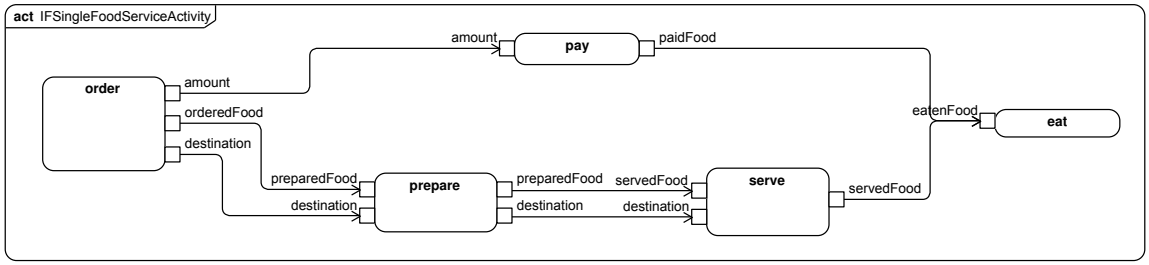


**Fig. 70.** Single food item flow service (activity)

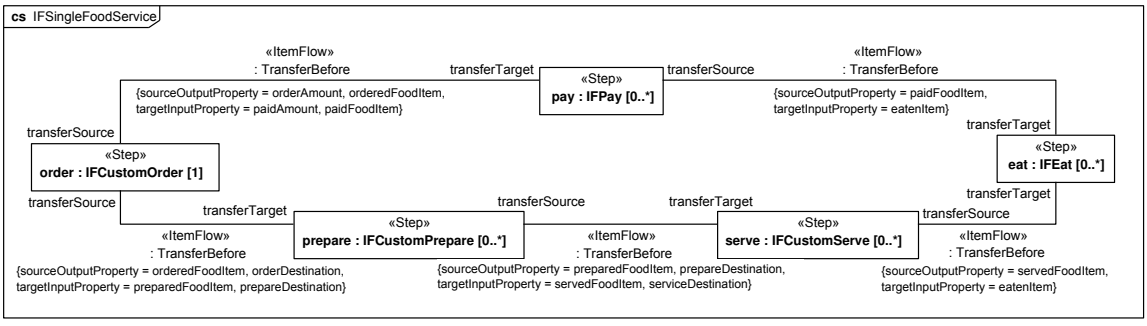


**Fig. 71.** Single food item flow service (OBM)

Figure 72 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of IFSingleFoodService, with the instances 2 as eat, #19 as pay, #1 as order, #15 as prepare, and #11 as serve. The ordered food corresponds to the prepared food, to the served food, to the paid food, and to the eaten food. The destination is also passed along these actions, and the amount during the order corresponds to the amount paid. #1 happens before #19 and #15, #15 happens before #11, and #19 and #11 happen before #2.

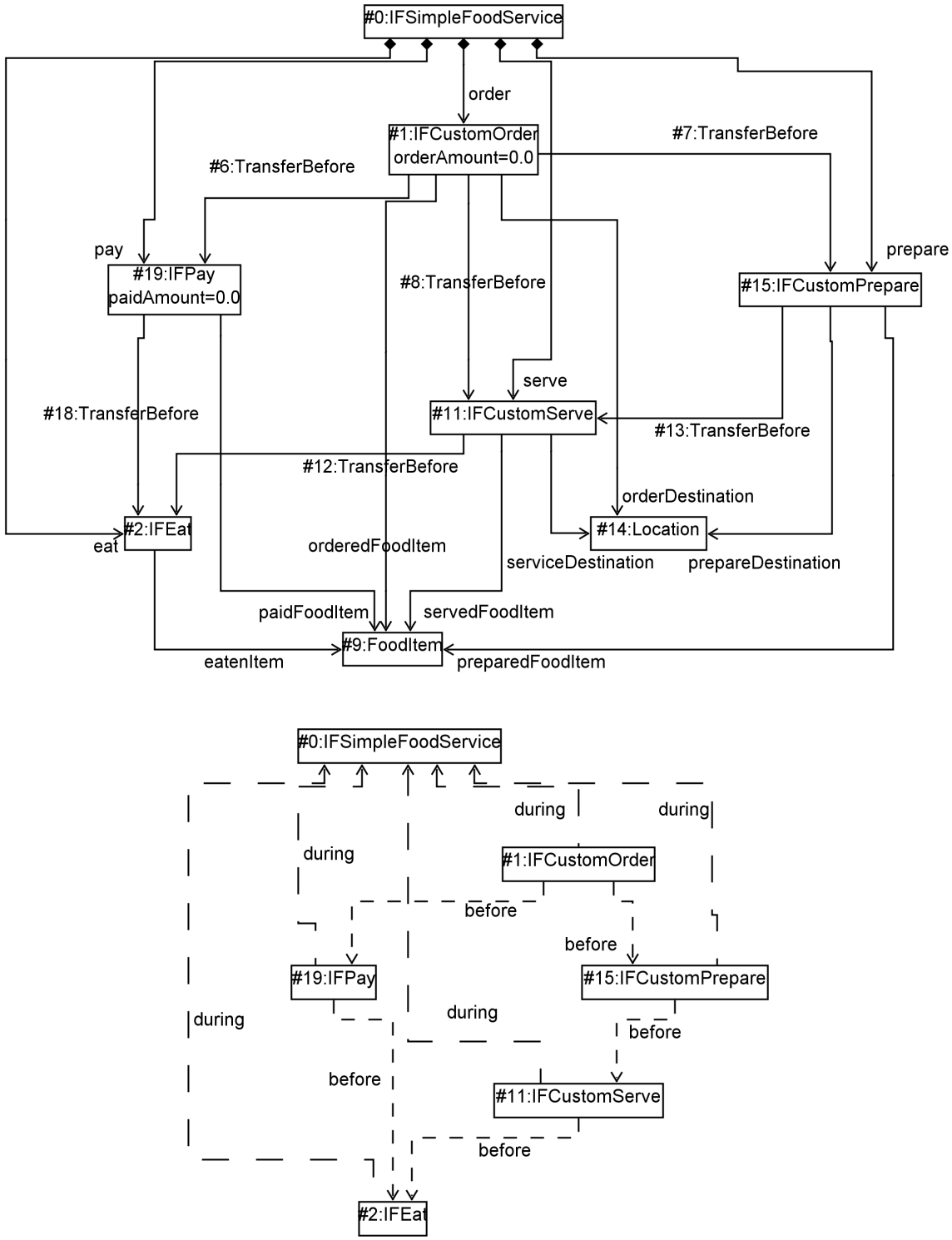


**Fig. 72.** Single food item flow instance model

**Food item flow with loop** Figures 73 and 74 show SysML activity and OBM representations, respectively, of a food service in which items flow twice, one after the other. In the OBM representation, IFLoopFoodService is a specialization of IFFoodService that is similar to the previous example with the following modifications: a property start is introduced before order, a property end is introduced after eat, the multiplicity of Order is set to 2. A new connector goes from eat to order. The properties start, eat, and order form a Merge, while the properties eat, order and end form a Decision.

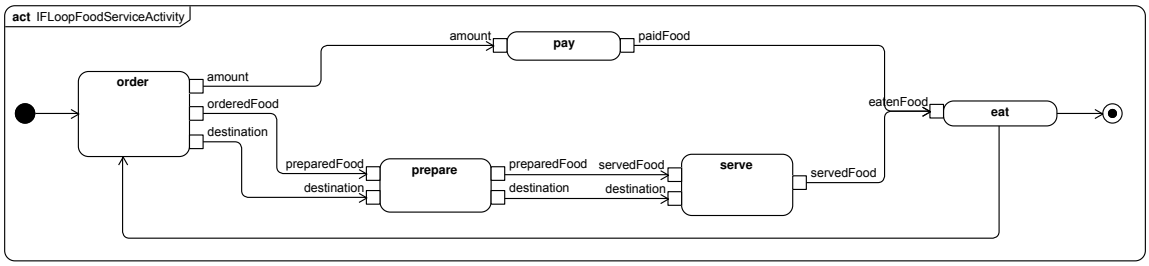


**Fig. 73.** Food item flow service with loop (activity)

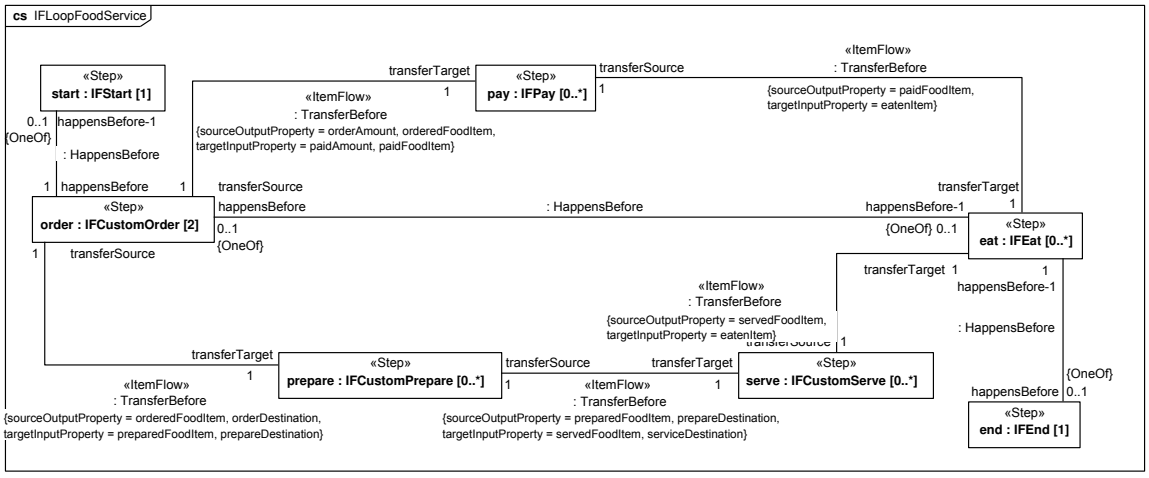


**Fig. 74.** Food item flow service with loop (OBM)

Figure 75 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of IFLoopFoodService, with the instances #1 as start, #25 and #34 as eat, #18 and #3 as pay, #15 and #5 as order, #23 and #28 as prepare, and #24 and #33 as serve, and #2 as end. There are two distinct set of steps, happening one after the other: first #15, #18, #23, #24 and #25 are executed, then #5, #25, #3, #33 and #34 are executed. Each set has a same location, food item, and amount.

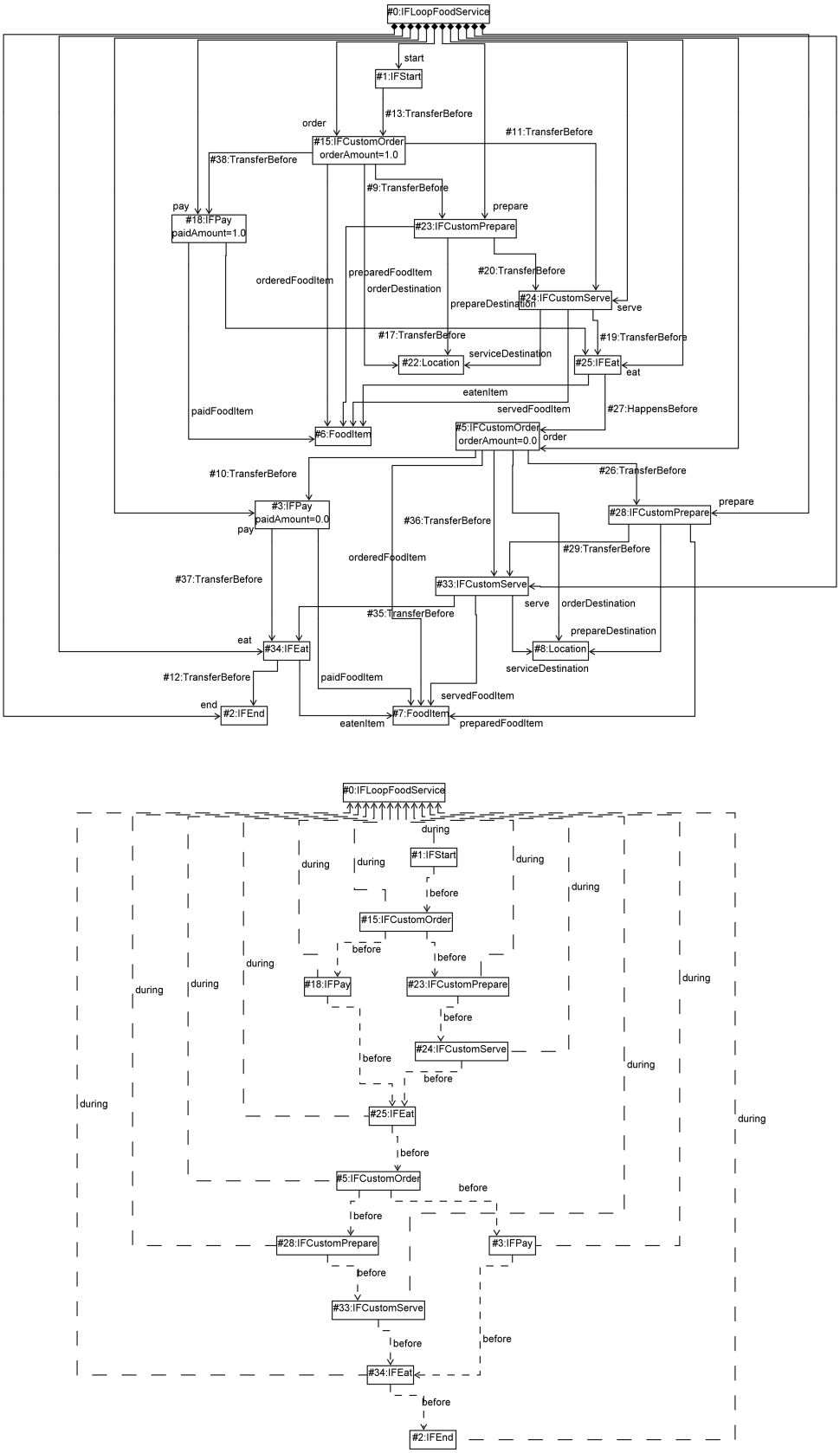


**Fig. 75.** Loop food item flow instance model

**Food item flow with parallelism** Figure 76 shows an OBM representation of a food service in which items flow twice, with no particular temporal relationship between the two flows. The SysML activity representation is the same as in Figure 70, since the only change is the multiplicity of a step and SysML actions do not have multiplicities. IFParallelFoodService is a specialization of IFFoodService that is similar to IFSingleFoodService, except that the multiplicity of order is 2.

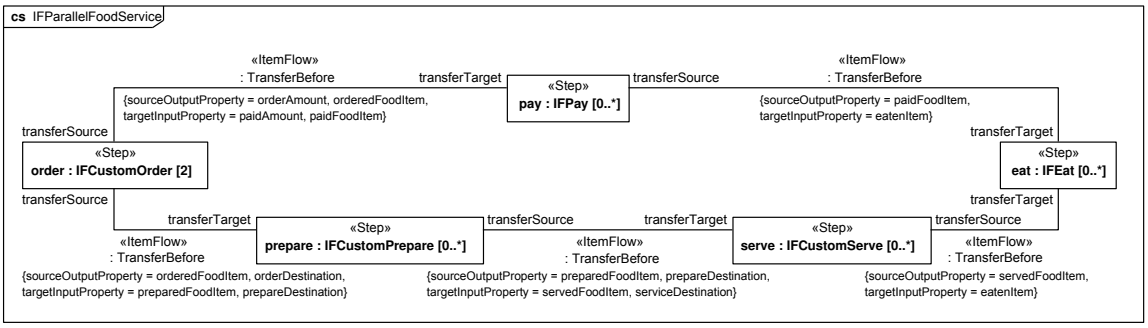


**Fig. 76.** Food item flow service with parallelism (OBM)

Figure 77 shows an instance model produced by the reasoner, with the links in the upper part and the temporal relations shown in the lower part. #0 is an instance of IFParallelFoodService, with the instances #12 and #24 as eat, #17 and #21 as pay, #16 and #20 as order, #25 and #14 as prepare, and #11 and #1 as serve. For each step, the former instance belongs to the first execution and the later instance belongs to the second execution. All instances in each execution refer to the same food item, destination, and amount. There are two distinct set of steps: #12, #17, #16, #25 and #11, and #24, #21, #20, #14 and #1. Each set has a same location, food item, and amount.

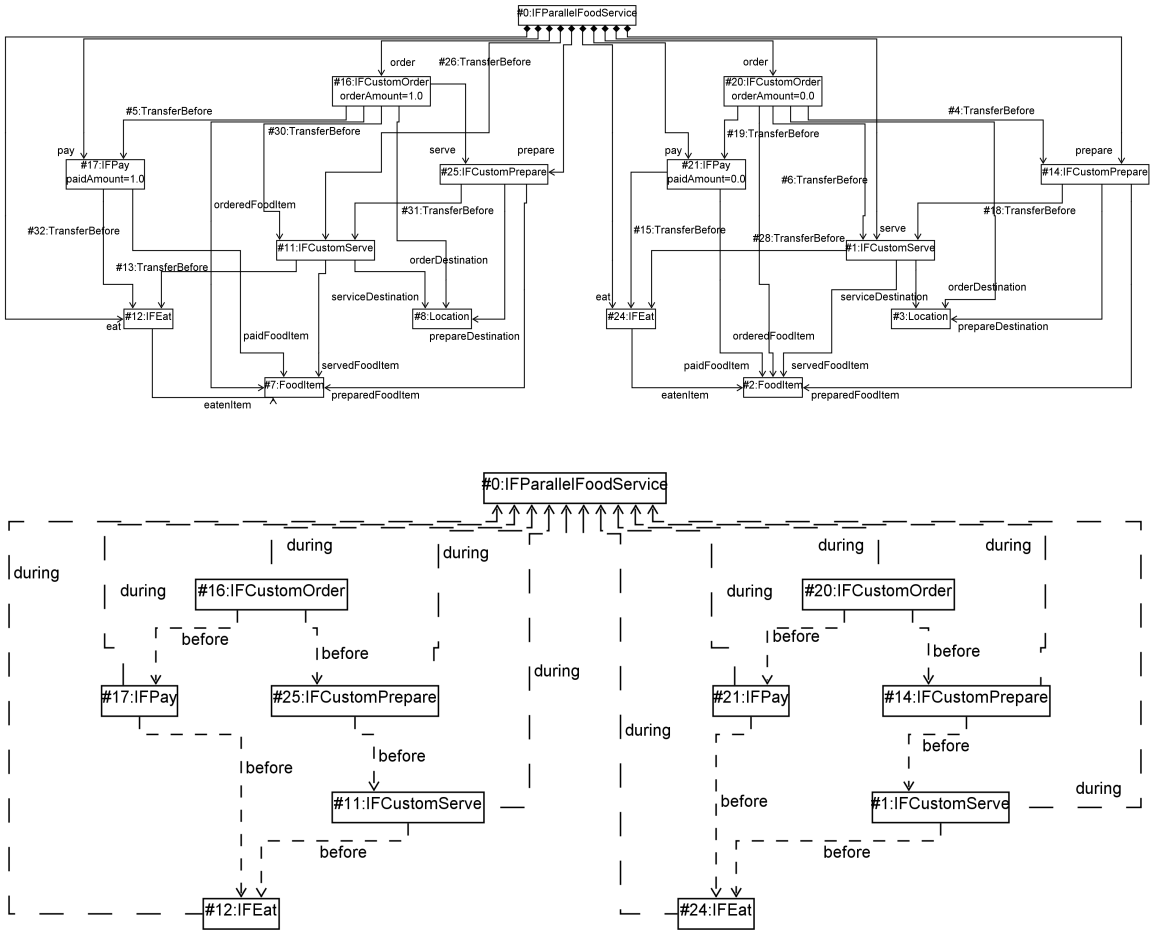


**Fig. 77.** Parallel Food item flow service instance model

## 5. Conclusion and Future work

This report presents an implementation of Ontological Behavior Modeling (OBM) in SysML, as well as an approach for automatically verifying executability of these models with Satisfiability Modulo Theories (SMT) solvers. OBM unifies similar concepts from the three SysML behavior modeling techniques (activities, interactions, state machines). The report describes a translation from OBM to SMT-LIB, a de-facto standard language for SMT solvers. A software implementation automatically translates OBM models to SMT-LIB, calls an SMT solver to verify executability by finding a satisfying set of instances (execution), and displays the instances graphically. Examples covering various SysML behaviors are presented, as well as their automatic verification using the software.

Regarding the next steps following this work, we identified several directions:

- Modeling the start and finish of occurrences as zero-duration occurrences, to enable temporal precedence beyond finish to start. This requires looser asymmetry

in the temporal relations.

- Support changing values of objects and behavior properties, which requires more advanced techniques such as 4D modeling to break single instances into time slices that differ in their property values [17].
- Support for OCL to model, for example, the OneOf multiplicity constraint used in decision and merge nodes.
- Mapping and translator from UML/SysML behavior modeling techniques to OBM models, for the overlapping concepts identified in this report. This will enable applicaiton of our approach to existing behavior models.
- Supporting features that are specific to only one UML/SysML behavior modeling technique.

We plan to apply this approach to more realistic use cases, including examples related to manufacturing systems. This will test scalability of the approach, because these cases will have more complex behaviors, number of occurrences, deep nesting, and so on. We might also study alternative approaches and tools for logical verification, such as TLA+ (finite-state automata)[18] or the Alloy analyzer[19].


## Acknowlegdements

The authors thank Jeremy Doerr and Karen Ryan for their useful feedback.

This material is based in part on work supported by U.S. National Institute of Standards grant awards 70NANB18H200 and 70NANB16H172 to Engisis, LLC.


## Disclaimer

Identification of any commercial equipment and materials is only to adequately specify certain procedures. It is not intended to imply recommendation or endorsement by the U.S. National Institute of Standards and Technology, nor does it imply that the materials or equipment are necessarily the best available for the purpose.

## References


[1] Object Management Group (2015) Systems Modeling Language, version 1.5. Available at https://www.omg.org/spec/SysML/1.5/.

[2] Object Management Group (2017) Unified Modeling Language, version 2.5.1. Available at https://www.omg.org/spec/UML/2.5.1/.

[3] Bock C, Odell J (2011) Ontological behavior modeling. *Journal of Object Technology* 10(3):1–36. https://doi.org/10.5381/jot.2011.10.1.a3

[4] Barrett C, Sebastiani R, Seshia SA, Tinelli C (2009) Satisfiability modulo theories. *Handbook of Satisfiability*, pp 825–885.

[5] Barrett C, Stump A, Tinelli C (2010) The SMT-LIB Standard: Version 2.0. *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*.

[6] C Barrett PF, Tinelli C (2018) The SMT-LIB Standard: Version 2.6. Available at http://SMT-LIB.cs.uiowa.edu/language.shtml.

[7] Havelund K, Kumar R, Delp C, Clement B (2016) K: A wide spectrum language for modeling, programming and analysis. *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*.

[8] Flatscher RG (2002) Metamodeling in EIA/CDIF—meta-metamodel and metamodels. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 12(4):322–342. https://doi.org/10.1145/643120.643124

[9] Object Management Group (2010) Unified Modeling Language, v2.3 – Infrastructure. Available at http://doc.omg.org/formal/2010-05-03.

[10] Mellor SJ, Scott K, Uhl A, Weise D (2004) *MDA distilled: principles of model-driven architecture* (Addison-Wesley).

[11] Allen JF (1983) Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843. https://doi.org/10.1145/182.358434

[12] Bansal K, Reynolds A, Barrett C, Tinelli C (2016) A new decision procedure for finite sets and cardinality constraints in smt. *International Joint Conference on Automated Reasoning* (Springer). https://doi.org/10.1007/978-3-319-40229-1

[13] Object Management Group (2014) Object Constraint Language (OCL), version 2.4. Available at https://www.omg.org/spec/OCL/2.4/.

[14] Barbau R (2020) Automated translator from OBM to SMT. Available at https://github.com/usnistgov/mbsdaism/releases/download/obmsmttrans/obmsmttrans.zip.

[15] De Moura L, Bjørner N (2008) Z3: An efficient SMT solver. *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (Springer). https://doi.org/10.1007/978-3-540-78800-3_24

[16] Wyner GM, Lee J (2003) Defining specialization for process models. *Organizing Business Knowledge: The MIT Process Handbook*, pp 131–174.

[17] Bock C, Galey C (2019) Integrating four-dimensional ontology and systems requirements modelling. *Journal of Engineering Design* 30(10-12):477–522. https://doi.org/10.1007/978-3-319-40229-1_7

[18] Yu Y, Manolios P, Lamport L (1999) Model checking TLA+ specifications. *Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (Springer). https://doi.org/10.1007/3-540-48153-2

[19] Jackson D, Schechter I, Shlyahter H (2000) Alcoa: the alloy constraint analyzer. *Proceedings of the 22nd international conference on Software engineering* (ACM). https://doi.org/10.1145/337180.337616