Withdrawn NIST Technical Series Publication

Warning Notice

The attached publication has been withdrawn (archived), and is provided solely for historical purposes. It may have been superseded by another publication (indicated below).

Withdrawn Publication	
Series/Number	NIST IR 8388
Title	Verifying Executability of SysML Behavior Models Using Alloy Analyzer
Publication Date(s)	February 24, 2022
Withdrawal Date	October 31, 2024
Withdrawal Note	Superseded by updated version

Superseding Publication(s) *(if applicable)*

The attached publication has been **superseded by** the following publication(s):

Series/Number	NIST IR 8388-upd1
Title	Verifying Executability of SysML Behavior Models Using Alloy Analyzer
Author(s)	Jeremy Doerr; Conrad Bock; Raphael Barbau
Publication Date(s)	October 31, 2024
URL/DOI	https://doi.org/10.6028/NIST.IR.8388-upd1

Additional Information *(if applicable)*

Contact	
Latest revision of the attached publication	
Related Information	
Withdrawal Announcement Link	

Date updated: October 31, 2024

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NISTIR 8388

Verifying Executability of SysML Behavior Models Using Alloy Analyzer

Jeremy Doerr
Conrad Bock
Raphael Barbau

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8388

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

NISTIR 8388

Verifying Executability of SysML Behavior Models Using Alloy Analyzer

Jeremy Doeer
Georgia Tech Research Institute
Altanta, Georgia

Conrad Bock
Systems Integration Division
Engineering Laboratory

Raphael Barbau
Associate, Systems Integration Division
Engineering Laboratory
Engisis LLC
Bethesda, MD

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8388

February 2022

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
James K. Olthoff, Performing the Non-Exclusive Functions and Duties of the Under Secretary of Commerce for Standards and Technology & Director, National Institute of Standards and Technology

National Institute of Standards and Technology Interagency or Internal Report 8388
Natl. Inst. Stand. Technol. Interag. Intern. Rep. 8388, 111 pages (February 2022)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8388

Verifying executability of SysML behavior models using Alloy Analyzer

Jeremy Doerr*[1], Conrad Bock†[2] and Raphael Barbau‡[3,4]

[1]Georgia Tech Research Institute, Altanta, GA, USA
[2]National Institute of Standards and Technology, Gaithersburg, MD, USA
[3]Associate, National Institute of Standards and Technology, Gaithersburg, MD, USA
[4]Engisis LLC, Bethesda, MD, USA

July 2021

Abstract

This report presents an approach to verifying executability of system behavior models by treating them as logical constraint problems solved using Alloy Analyzer, a non-proprietary software tool supporting a textual language for logical constraints and underlying solvers. With behavior models interpreted as constraints on execution order, Alloy can determine whether the models are executable by attempting to find executions that meet those constraints. The approach relies on a logical interpretation of behavior modeling in the Systems Modeling Language (SysML) by Ontological Behavior Modeling method that unifies SysML behavior modeling based on its classification elements. This paper proposes translation between logical versions of SysML behavioral models and logical constructs in Alloy. Finally, the approach is 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

*jeremy.doerr@gtri.gatech.edu
†conrad.bock@nist.gov
‡barbau@nist.gov

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 document reviews.

The Systems Modeling Language 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 classification, which are typically only applied system structure, 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, the Ontological Behavior Modeling (OBM) method was developed to centralize aspects common to these ways of modeling behaviors [3][4]. It uses the classification elements of SysML to model behaviors in these three ways, capturing what they have in common, and building on this to reflect their differences.

From the perspective of logic and constraint solvers, behavior models can be viewed as constraints on their possible executions, imposed by the model and its language semantics, with executions as solutions to those constraints. The logical approach to executability is to translate models into logical statements, then use automated solvers to find solutions that are consistent with the statements. If a solution is found, the model is executable, otherwise if the search is complete, the model is not executable, or if the search is incomplete, it might or might not be executable. Solutions are said to satisfy the model, which becomes satisfiable, or if no solutions are possible, the model is unsatisfiable (over-constrained).[2]

This report describes a method for transforming SysML behavior models into Alloy, a textual language for specifying logical constraints, supported by Alloy Analyzer, a non-proprietary software tool for checking satisfiability using its solvers [5].[3] Common patterns in SysML behavior modeling are translated to SysML OBM diagrams, then to Alloy, and solved (or not).[4] Section 2 reviews the ways of modeling behavior in SysML and the OBM method for unifying them. Section 3 introduces Alloy by showing how

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

[2] In contrast to software execution, logical solvers produce results all at once, covering the entire duration of execution, rather than producing "snapshots" of execution incrementally over time. Logical solvers can still be applied to behavior models, however, with results including things (occurrences) that are ordered or nested in time.

[3] The rest of the paper will refer to the tool as Alloy also.

[4] The patterns are taken from [6], which translates the same examples for other solvers.

to translate SysML classification modeling to it, then gives the translation of a SysML model that OBM uses for behavior. Section 4 gives translations of example behaviors based on this, as well as solutions (executions) for the satisfiable ones. Finally, Section 5 summarizes the paper and outlines future work.

2. Behavior modeling in SysML

This section briefly describes the three behavior modeling techniques in SysML (Section 2.1), as well as the OBM method for unifying them (Section 2.2).

2.1 Behavior modeling in SysML

This subsection describes the main styles of SysML behavior modeling. Behaviors in SysML can be specified in three ways:

- Activities describe sequences of actions in a process and how things flow between them
- State machines also describes sequences of actions, but as combined into states of a behavior, and transitions between them in reaction to external stimuli
- Interactions describe messages exchanged between participants.

Each technique has its own terminology with corresponding diagrammatic notation. The following paragraphs will briefly present their overlapping capabilities.

Composition All three SysML behavior modeling techniques 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 other executions that compose it happen during the original execution.

Time ordering All three techniques 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 techniques 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 techniques can specify objects that participate in behaviors. Activities have partitions, while interactions have lifelines.[5]

Transfers Two techniques can specify flow of items between behaviors and structures, all of which can specify the kind of item flowing and the participant at the beginning and end of the flow (sender and receiver). Activities have object flows between object nodes, which can be pins on actions, or stand on their own between actions (central buffer nodes), or be on the boundary of activities (parameter nodes). Interactions messages between lifelines have arguments, which are also typed.

2.2 OBM method

OBM is a way of modeling behavior with SysML classification capabilities that usually are only applied to structure. Section 2.2.1 reviews SysML classification modeling, while Section 2.2.2 applies it to the behavior modeling techniques introduced in Section 2.1.

2.2.1 SysML classification modeling

Classification in SysML identifies kinds of things, their parts (whole-part relationships), and how these are interconnected (part-part relationships), using the terms shown in Figure 1, adapted from SysML's metamodel.[6] SysML treats system models as classes (sets) of the systems to be built (instances). SysML uses the term "block" instead of UML's "class", but this report sometimes uses "class" to emphasize logical classification, and omits the stereotype label for blocks from diagrams, for brevity.

[5]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.

[6]SysML's metamodel is a subset of UML's, which is specified in a subset of UML [7].

Classes can form taxonomies by specializing other classes, where instances of specialized classes are also instances of general ones (the ones being specialized). Classes can have properties that each instance can give values to. These values must be instances of the type specified by 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). Properties can form taxonomies also, by subsetting or redefining other properties, where values of the specialized properties are also values of general ones, or have exactly the same values in redefinition.

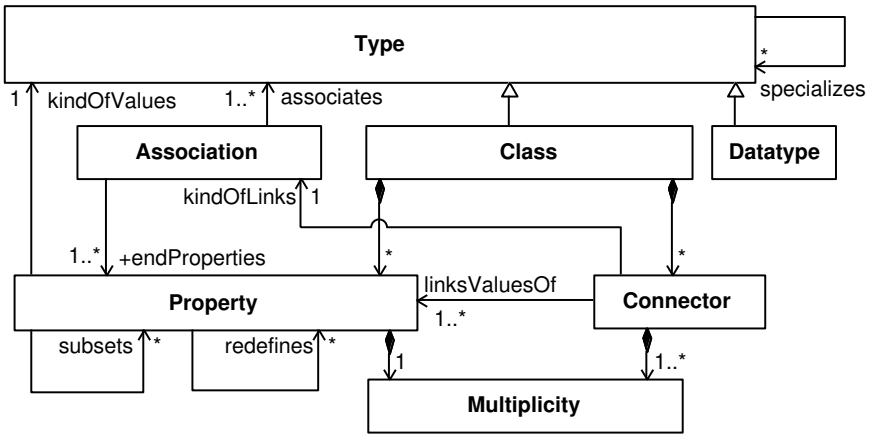


Fig. 1. SysML classification concepts

An association is a relationship between classes that specifies a property of each class for identifying 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. Multiplicities of these properties appear at the ends of associations lines in diagrams, indicating how many links the association specifies between each instance of the class at the other end to instances of the class at the end where the multiplicity appears. For example, here are two common association end multiplicities:

- 1..$*$ (or $*$): at least one link per instance of the class at the other end of the association
- 0..1: zero or one link per instance at the other end of the class at the other end of the association

Associations are typically between two classes, linking their instances, but also can associate a class to itself (same class at both ends), with links between instances of that class.

Classes can have connectors (part-part relationships), which are equivalent to properties typed by associations that connect other properties (whole-part relationships). They specify links (of the association typing the connector) between property values on the same object. Connectors are typically between two properties, linking their values, but also can connect a property to itself, with links between multiple values of that property. Connectors have multiplicities at each end, indicating how many links the connector specifies between each value of the connected property at the other end to values of the property at the end where the multiplicity appears. Two common connector end multiplicities are:

- 1..1 (or 1): exactly one link per value at the other end of the connector
- 0..1: zero or one connector link per value at the other end of the connector

See Sections 2.2.2 and 3.1.1 for examples of SysML classification modeling.

2.2.2 Applying classification modeling to behavior

The three ways of modeling behaviors in SysML have a lot in common, as explained in the previous subsection. They also overlap classification modeling, because these behaviors are also classes (blocks in SysML), with their instances being executions of the behaviors. OBM brings out more commonalities between behavior and classification 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 sequences

Although models specify real or virtual things and how they should perform or act, SysML does not have a model of these things or what they do, and consequently cannot capture the intended effect that models have on the real or virtual things being specified. For example, a SysML manufacturing activity might have two actions, one for painting an object, followed by another for drying it. This activity might be performed many times in many places, with the intention that in each case, an execution of painting is followed by an execution of drying. It is not intended that painting in one execution of the manufacturing activity should be followed by drying in another execution of the activity. The activity is the same, with the same actions and a single control flow between them, but it is executed many times, along with its actions, and requires many timing relationships between the action executions, limited to each manufacturing execution separately. SysML does not have a model of executions or their timing relationships to capture these intentions.

OBM addresses this by introducing a model (library) of executions and their temporal relations, expressed in SysML, as shown in Figure 2. The library is reused when defining specific behaviors. It has classes and associations, like any SysML classification model, starting with the most general class Anything, covering everything, physical or virtual. All other classes specialize it, including those created by system modelers. One is BinaryLink, which classifies things that identify two things being linked, via source and target properties, corresponding to SysML participant properties on associations. All associations specialize it, including SelfLink, which classifies all and only links between each thing and itself. Another specialization of Anything is Occurrence, for things that exist or happen in time, including performances (executions) of behavior models. All SysML behaviors specialize Performance, even though they can be diagrammed in multiple ways.

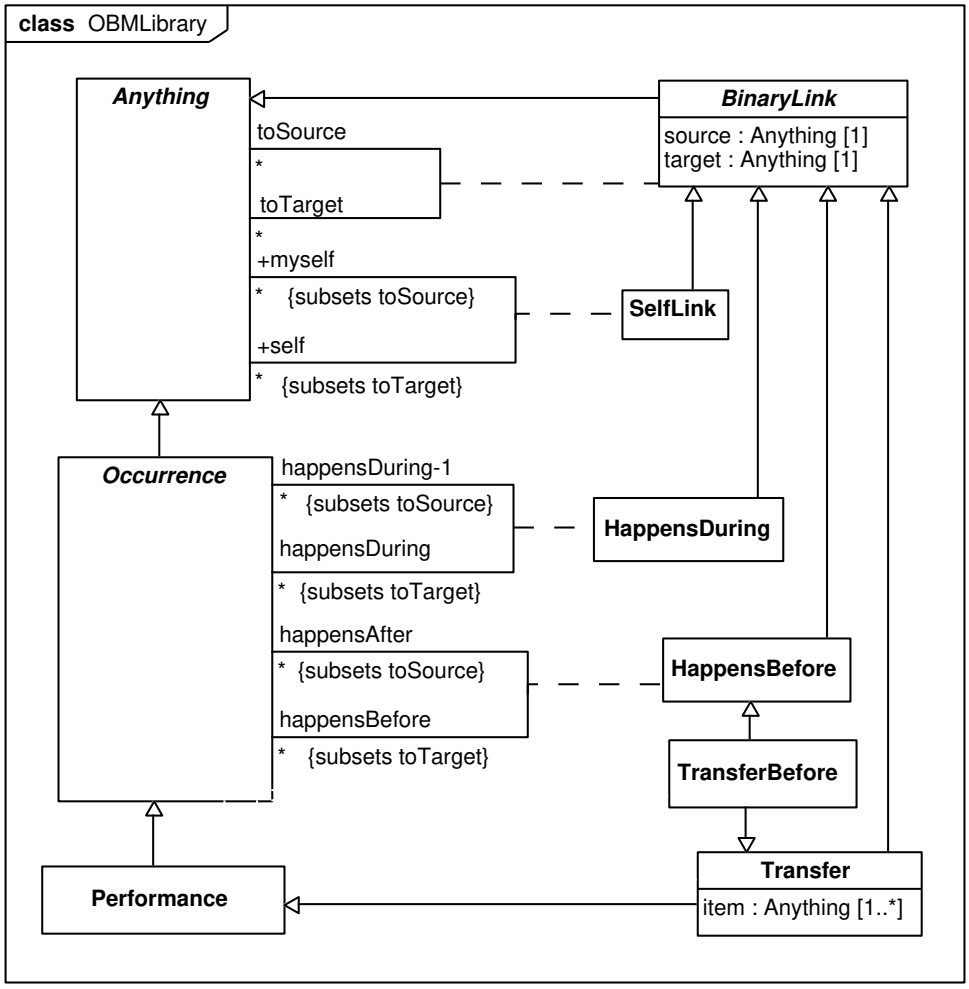


Fig. 2. OBM Library

Performances The Performance class specializes Occurrence for executions of behaviors.[7] Specializations of Performance can have properties typed by Performance or a specialization of it, in which case the properties are called *steps*. Occurrences identified by steps happen during the occurrence they are a step of, see temporal relations below. The multiplicity of steps indicates how many times a step must or might be executed during the owner's execution. Steps with multiplicity 0..∗ might occur multiple times, once, or not at all, depending on other constraints on them, including those due to temporal relations between steps in same behavior. Initial steps occur exactly once per execution of the behavior (multiplicity 1).[8]

Temporal relations OBM provides two temporal relations between Occurrences, modeled as associations (specializations of BinaryLink):

- HappensBefore links occurrences that are separate in time, one happening before the other.
- HappensDuring links occurrences that are completely overlapping in time, one happening completely during the other.

Logical characteristics of these relations are:

1. Both temporal relations are transitive.
2. HappensBefore is asymmetric, which includes being irreflexive.
3. HappensDuring is symmetric between occurrences that happen at the same time, including between each occurrence and itself, which means it is reflexive (every occurrence happens at the same time as itself). HappensDuring is asymmetric in all other cases.
4. When occurrences are related by HappensDuring, all HappensBefore relations involving the occurrence of longer or equal duration also apply to the other one (of shorter or equal duration).

Characteristics 2 and 4 combined imply the relations are disjoint (occurrences cannot be related by both HappensBefore and HappensDuring).

[7] [6] calls these BehaviorOccurrences.

[8] Non-initial steps often have multiplicity 0..∗, leaving reasoners to determine the number of executions in each case, but other multiplicities are useful, for example lower bounds greater than zero to check reachability.

These associations are adapted from Allen's interval logic [8]:

- HappensBefore is equivalent to Allen's before interval relation, which is for intervals that are completely separate in time.[9]
- HappensDuring is equivalent to the union of Allen's starts, during, finishes, and equals interval relations.

Connectors typed by HappensBefore and HappensDuring specify temporal relations between values of the connected properties, which are expected to be steps (typed by Performance or one of its specializations). Connectors typed by HappensBefore (successions) specify links where the source of the first connected property occurs before the target of the second connected property. Connectors typed by HappensDuring specify links where the source occurs during the target. Steps always imply a HappensDuring. Section 4 gives examples of connectors typed by temporal relations.

Transfers Transfers are treated as links (instances of associations) that happen over time (occurrences) between participants, which are the source and target of each link, which are also occurrences. Transfer is an association between Occurrence and itself, specializing BinaryLink and Performance. Transfers support a property item for the things being transferred. Each transfer picks up items from its source then drops them off at its target, which are values of particular properties on the source and target. The item and pickup-drop-off properties can be constrained in behaviors that use them, see Sections 4.1.4 and 4.2.2 for examples. TransferBefore covers transfers that happen after the source participant (ceases to exist) and before the target (comes into existence), as is common for flows between performances that are ordered in time. Interactions between participants are modeled as connectors (item flows) typed by Transfer, or one of its specializations.

[9] [6] also includes Allen's meets in HappensBefore, in which intervals overlap at a single start/end time point. This makes invariants on the intervals potentially inconsistent and complicates the explanation of temporal order in terms of sets of time points. In Allen and [6], including meets in before/HappensBefore also rules out zero duration intervals (which would be equivalent to time points), to preserve asymmetry of temporal order.

3. Translating SysML behaviors to Alloy

Section 3.1 introduces Alloy by describing translation of basic SysML classification features (as needed by OBM) to Alloy's textual language for expressing logical constraints. Section 3.2 applies this to the OBM behavior library for translating SysML behaviors for translating SysML behaviors to Alloy.

3.1 Translating SysML classification models to Alloy

Section 3.1.1 introduces Alloy by translation to SysML classification models. Section 3.1.2 covers Alloy constraint syntax and its application to SysML connectors. Section 3.1.3 describes logical predicates on relations in Alloy needed in this paper. Section 3.1.4 covers how Alloy presents solver results.

3.1.1 Basic Alloy and SysML

Alloy includes

- A textual language and user interface for defining logical constraints.
- A customizable graphical interface for viewing results, including repeated solving that prioritizes a wide variety of solutions, as well as multiple presentation styles (graphs, tables, and trees).
- Solving capability provided by other tools based on satisfiability modulo theories [9].

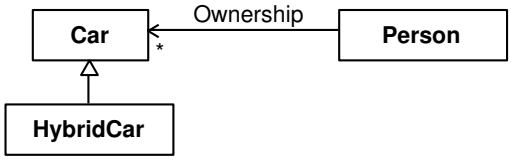
Table 1 gives correspondences between SysML classification terms and Alloy's. Alloy signatures classify atoms in the same way SysML blocks classify instances. Extensions indicate that the atoms of one signature are atoms of another, as SysML generalization does for instances of blocks. Relations classify tuples in the same way SysML associations classify links, with an atom from the relation's domain first, then an atom from its co-domain.[10]

SysML	Alloy
Block/Class	Signature
Instance	Atom
Generalization	Extension
Association	Relation
Link	Tuple

Table 1. SysML and Alloy terms

[10] Associations can specify links between more than two things, and Alloy tuples can have more than two elements in them, but this paper only covers associations linking two things (or one linking to itself), and Alloy tuples of exactly two atoms, which might be the same.

Figure 3 shows an example SysML block definition diagram (BDD, class diagram in UML) and corresponding Alloy for declaring logical classes and relations. The first Alloy statement introduces a signature named `Car`, corresponding to the SysML block (UML class) of the same name. The second statement adds another one named `HybridCar`, and constrains all its atoms (instances) to also be atoms of `Car`, as the SysML generalization does for the corresponding blocks. The third statement adds a `Person` signature and an `Ownership` relation between it and `Car`, as the SysML association does.[11] SysML associations depend on properties of their associated classes for identifying objects at the other end of links (see end properties in Figure 1 in Section 2.2.1), whereas Alloy uses just the relation do this (see dot notation in Section 3.1.2). The `set` keyword prevents the tuples of `Ownership` from pairing a particular person (atom) with the same car more than once, but places no other limitations on tuples, as SysML association with end property multiplicity $*$ does by default. The Alloy notation for logical relations is similar to fields/members in programming classes/structures, as well as UML's textual syntax for properties of classes [10], but is not analogous to software pointers or mathematical functions as those are. Connectors, the last part of SysML classification modeling, translate to Alloy as constraints, see Figure 5 in Section 3.1.2.



(a) SysML

```
sig Car {}
sig HybridCar extends Car {}
sig Person {
  Ownership: set Car }
```

(b) Alloy

Fig. 3. Example model in SysML and Alloy

3.1.2 Alloy constraints and SysML connectors

Alloy includes three styles of defining constraints, in addition to the ones implied by the syntax above, as shown below. All three examples restrict cars (as defined in Figure 3b) to having no more than one owner.

- Augmented predicate calculus

```
all p1,p2: Person, c:Car |
  p1->c in Ownership and p2->c in Ownership
    => p1=p2
```

[11] The SysML association arrow points to the class that is translated as the co-domain of the Alloy relation, without implying anything about efficient navigation as SysML does.

A variant of first-order logic (FOL) notation [11] that includes shorthands for:

  - Quantified variables with values (atoms) restricted by type (signature), notated with a colon between the variable name and signature, such as `c:Car` restricting values of `c` to be classified by `Car`. Conventional FOL notation requires a separate conjunction for variable typing.
  - Sets of tuples specified from sets of atoms, notated with `->` between them, such as `p1->c` being all tuples with a value of `p1` and of `c`, in that order (cartesian product of the sets). For variables with a single atom as value, such as quantified variables, the notation specifies a set of one tuple. Tuples are classified with the `in` keyword, such as `p1->c in Ownership`, indicating the set of tuples on the left is a subset of all the `Ownership` tuples. Conventional FOL notation for tuples is merged with its notation for classification and only applies to one tuple at a time.

- Navigation

  ```
  all c: Car | lone Ownership.c
  ```

  Adds dot (join) notation to predicate calculus for identifying atoms in tuples, such as `Ownership.c` for the set of cars in all `Ownership` tuples (a dot appearing before the relation would identify the set of people in the tuples).[12] The `lone` keyword restricts this set to have no more one member.

- Relational calculus

  ```
  no Ownership.~Ownership − iden
  ```

  Extends the dot notation to relations on both sides (relational composition), such as `Ownership.~Ownership`, which are all tuples where the first atom is the first atom in a `Ownership` tuple and the second is the second atom in a tuple of its inverse (~, reversing the atoms of all tuples) and the two tuples have the same second and first atom, respectively,[13] resulting in tuples that pair car owners with each owner of the same car. The notation also provides additional set operators, such as subtraction (−), as well as predefined sets, such as `iden`, the set of all tuples that have same atom in both positions. The `no` keyword requires a set to have no elements, in this case the set of tuples resulting from removing all `iden` tuples from the result of the dot expression, which produces all pairs of car owners with other owners of the same car. This is the most compact syntax, departing from FOL syntax by eliminating variables, and focusing on set operations, including sets of tuples.

[12] This is the translation of accessing the values of properties on specific objects in SysML, including association end properties.

[13] This is analogous to database join operations.

Constraints (signature facts) can be added to signatures within braces just after, which apply to every atom individually, available as a `this` variable identifying an atom of the signature. In Figure 4, the signature fact for `Car` requires them to be owned by at most one person, applying the dot notation for navigation (see above) with `this` to identify each car atom separately. It is often unnecessary (and incorrect) to use it, however, because relations defined in (or inherited to) a signature are treated in its signature facts as if they were prepended with "`this.`" identifying the set of atoms in the second position of the relation tuples that have the value of `this` in the first position. Applying `this` again to the relation would join an atom with a set of atoms, which is not possible, unless the relation name is prepended by "`@`" to prevent the implicit "`this.`". For example, the signature facts for `Person` in Figure 4 have the same effect as the one in `Car`, but treat `Ownership` as all the cars owned by each person separately and `@Ownership` as all the tuples of the relation, regardless of person. The first fact quantifies over cars owned by each person separately (`c in Ownership`), requiring that the cars be owned by at least one person (`lone @Ownership.c`). The other two facts have the same effect. The convention of implicit "`this.`" for relations does not apply when they are not defined in or inherited to the same signature as the fact, such as `Ownership` in the `Car` signature fact.

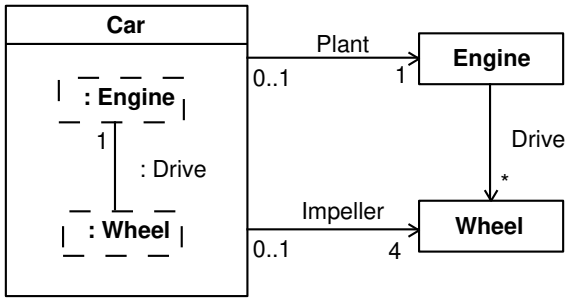
```
sig Car { }
  { lone Ownership.this }
sig Person {
  Ownership: set Car }
{ all c:Car | c in Ownership implies lone @Ownership.c
  lone @Ownership.Ownership
  no @Ownership.~@Ownership - iden }
```

Fig. 4. Example signature facts (constraints on each atom of a signature separately)

Connectors translate to Alloy as signature facts that constrain tuples between atoms related to each atom of that signature, as shown in Figure 5. The diagram nested in Car in SysML is an internal block diagram (IBD, internal structure diagram in UML), which shows properties as rectangles and connectors as lines between them. The SysML blocks translate to Alloy signatures similarly to those in Figure 3, using the Alloy keyword `one` to require exactly one `Plant` tuple for each `Car` atom in the first position. The first signature fact for `Car` requires every car to be impelled by exactly four wheels, using Alloy's `#` operator on sets to specify the number of elements, corresponding to the SysML's association multiplicity.[14] The second fact ensures all impellers in each car are in a `Drive` tuple with the engine in that car, as the SysML connector in Car does for links of the Drive association.[15]

[14] `Impeller` is treated as the set of wheels in each car, see above about relations signature facts.

[15] `Plant` is treated as a set of one `Engine`, and quantified variables have one value, making `(Plant->i)` a set of one tuple for each car, see above about Alloy's predicate calculus notation.



(a) SysML

```
sig Car {
  Plant : one Engine
  Impeller : set Wheel }
{ #Impeller = 4
  all i : Impeller |
    (Plant->i) in Drive }
sig Engine { Drive : set Wheel }
{ lone Plant.this }
sig Wheel { }
{ lone Impeller.this }
```

(b) Alloy

Fig. 5. Example connector in SysML and Alloy

3.1.3 Alloy logical predicates

Alloy provides predicates for some logical characteristics of relations, and additional ones are defined to support the translation in this paper, as shown in Listings 1 and 2 [12]. Predicate definitions include variable names between square brackets to identify the things being predicated, the first of which in these is a relation (usually a proper subset of all tuples, specified as pairs of atoms from `univ`, the set of all atoms), and sometimes another variable for a set of atoms that limits which tuples the predicate applies to. The predicates in Listing 1 are characteristics of relations treated as a "network" with atoms as nodes and their tuples as edges, mostly defined with Alloy's relational syntax:

1. *Reflexivity* requires all tuples with the same atom in both positions to be classified by the relation (every atom is related to itself). The requirement applies to all tuples with a first atom in the set `s`, the second variable of the predicate, typically the domain of the relation.[16] The `<:` notation limits `iden` (the set of all tuples that have same atom in both positions) to those with their first atom in `s`.

2. *Asymmetry* prevents two tuples with the same atoms in reverse order (`~`) from both being classified by the relation (atoms cannot be related in both "directions", including a single atom in both positions of a tuple, which is *irreflexivity*).

3. *Transitivity* ensures tuples are classified by the relation when a) their first atom is the also the first atom in another tuple, and b) the second is the second atom in a third tuple, and c) these other tuples have the same second and first atom, respectively (tuple "chaining", atoms are related to each other when they

[16] Domains cannot be derived from the first variable, because they are restrictions on the first atom of tuples in the relation specified elsewhere.

are "related through" other tuples of other atoms). The Alloy definition uses composition of a relation with itself, which identifies tuples corresponding to the first two tuples a chain. With these classified by the relation, self-composition identifies tuples corresponding to another "link" in the chain, and so on to tuples corresponding to chains of all lengths (see Section 3.1.2 about Alloy's dot notation).

4. *Acyclic* relations prevent their transitive closures from including any tuples from `iden`. The prohibition applies to all tuples with a first atom in the set `s`, the second variable of the predicate, typically the domain of the relation.[16]

```
1  pred reflexive [r: univ -> univ, s: set univ]
2    {s<:iden in r}
3
4  pred asymmetric [r: univ -> univ]
5    {no r & ~r}
6
7  pred transitive [r: univ -> univ]
8    {r.r in r}
9
10 pred acyclic[r: univ->univ, s: set univ]
11   { all x: s | x not in x.^r }
```

Listing 1. Predefined "network" predicates

The predicates in Listing 2 are characteristics of relations concerning how many atoms each atom in the (co)domain can be related to, defined with Alloy's augmented predicate calculus syntax:

1. *Function* relations must classify exactly one tuple for each atom in the set `s`, the second variable of the predicate, typically the domain of the relation,[16] with that atom in the first position (Alloy functions are total).

2. *Inverse function* relations must classify exactly one tuple for each atom in the set `s`, the second variable of the predicate, typically the co-domain of the relation,[17] with that atom in the second position.[18]

3. *Bijections* are functions and inverse functions over the sets `d` and `c`, respectively, typically the domain and co-domain of the relation,[16,17] which must have the same number of atoms (each atom in the domain appears in the first position of exactly one tuple, and likewise for codomain atoms in the second position). These relations are sometimes called "one-to-one", because they relate each atom in the domain to exactly one in the codomain and vice-versa.

[17] Codomains cannot be derived from the first variable, because they are restrictions on the second atom of tuples in the relation.

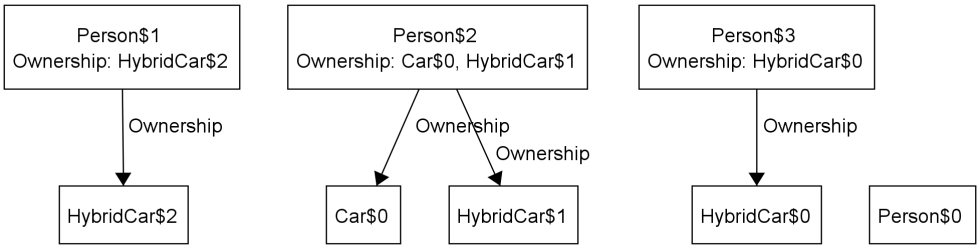
[18] Alloy calls these "bijective", which usually is the adjective for bijection. This paper uses a less misleading term.

```
1 pred function [r: univ->univ, s: set univ]
2   { all x: s | one x.r }
3
4 pred inverseFunction[r: univ->univ, s: set univ]
5   { all x: s | one r.x }
6
7 pred bijection[r: univ->univ, d, c: set univ]
8   { function[r, d] && inverseFunction[r, c] }
```

Listing 2. Predefined "(co)domain" predicates

3.1.4 Alloy solutions

Alloy displays solutions in three ways, as shown in Figure 6. Directed graphs (Figure 6a) use rectangles for atoms and arrows for tuples of them, pointing towards the atom in the second position, with the arrow labelled by a relation name classifying the tuple (atoms can appear without arrows from or to them when there are in no tuples). In addition to the arrow notation, tuples can appear as inside atoms (rectangles) that are in the first position of the tuple ("attribute" notation). These give a relation name and the second atom in the tuple, if any, separated by a colon. Figure 6a shows both displays for each tuple, with `Ownership` tuples appearing as edges in the graph and as attributes inside atoms. Each atom is identified by a "$" followed by a number unique to that atom's signature. The numbers in atoms identifiers do not imply ordering.



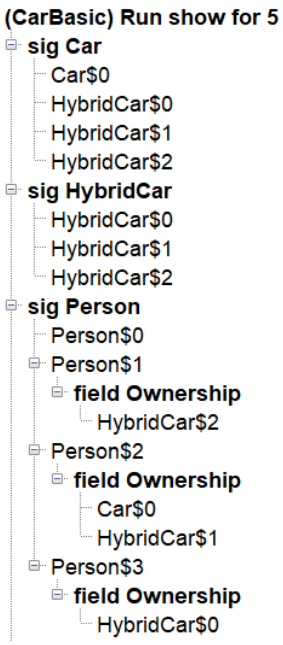
(a) Directed graph

this/Car	Car$0	HybridCar$0	HybridCar$1	HybridCar$2

this/HybridCar	HybridCar$0	HybridCar$1	HybridCar$2

this/Person	Ownership
Person$0	
Person$1	HybridCar$2
Person$2	Car$0
	HybridCar$1
Person$3	HybridCar$0

(b) Table



(c) Tree

Fig. 6. Three Alloy presentation styles for constraint solutions

Tables have a signature name in the top left cell, as shown in Figure 6b. All atoms classified by them are listed in cells below or to the right. The lower table is for tuples, with a second column headed by a relation name, and tuples classified by the relation in the rest of the rows. Atoms in a tuple's first position are under the signature, and second position under the relation (which is empty when the atom to the left is not in the first position of any tuple).

Trees views, shown in Figure 6c, list signatures left-justified, with atoms they classify indented under them. Indented under each atom are the relations that classify tuples with that atom in their first positions. Indented under each relation are the atoms in the second position of all tuples classified by the relation that have the atom classified by the signature in their first positions.

3.2 Translating the OBM library to Alloy

This section describes translation of the OBM library (Figure 2 in Section 2.2.2) to Alloy classification models. The current translation omits Anything, uses BinaryLink only for Transfer, and treats SysML behaviors as specialized Occurrences, rather than Performances, to avoid depending on metaclassification of model elements to distinguish them from structures. Additional relations specifically for translating SysML behaviors are defined with the examples in Section 4.

Occurrences The OBM method treats behaviors as classes (of their occurrences), specialized from Occurrence in the library, which translate to signatures in Alloy, see Listing 3. Steps in the OBM method (properties of occurrences typed by other occurrences) translate to Alloy as relations between occurrences. The first signature fact (constraint, line 3) ensures occurrences are not steps of themselves, including steps of step occurrences to any depth.[19] The second (line 4) prevents occurrences from being steps of more than one occurrence, while the last fact (line 5) ensures that step occurrences happen during the occurrence they are a step of, see temporal relations below for HappensDuring.[20]

```
1 sig Occurrence {
2   Step: set Occurrence }
3 { r/acyclic[Step]
4   lone @Step.this
5   one @Step.this => (this->@Step.this) in HappensDuring }
```

Listing 3. Occurrences in Alloy

Temporal relations The OBM library defines two associations to model temporal relations between performances (HappensBefore and HappensDuring), along with their logical characteristics, see Temporal relations in Section 2.2.2. Listing 4 shows an Alloy translation for these associations and their logical characteristics. Logical predicates (see Section 3.1.3) are available for all but the last two (lines 8-14), which are needed to further distinguish HappensDuring and HappensBefore.

[19] This does not prevent behaviors from having steps typed by themselves (recursion), but they need to be separate occurrence of the behavior.

[20] Step is the only element of OBM syntax (a metamodel extension of SysML) this paper translates to Alloy. The rest are translations of the OBM library (Figure 2 in Section 2.2.2).

```
sig Occurrence {
  HappensDuring, HappensBefore: set Occurrence }
{ r/transitive[HappensDuring]
  r/reflexive[HappensDuring]
  r/transitive[HappensBefore]
  r/asymmetric[HappensBefore]

  all x,y,z: Occurrence |
    (x->y) in HappensBefore and (z->y) in HappensDuring
      => (x->z) in HappensBefore

  all x,y,z: Occurrence |
    (y->x) in HappensBefore and (z->y) in HappensDuring
      => (z->x) in HappensBefore }
```

Listing 4. Temporal relations in Alloy

Transfers Transfers pick up items from their source occurrence and drop them off at their target, see Transfers in Section 2.2.2. The OBM library models them as both binary links and occurrences, enabling transfers to take time, as well as classes of them (Transfer and specializations of it) to type connectors in OBM models. The current translation to Alloy treats all binary links as occurrences, and they only link occurrences, with transfers as kinds of binary links, as shown in Listing 5. In `BinaryLink`, the `Source` and `Target` relations identify exactly one occurrence each. `Transfer` inherits these as participants of the transfer (sources being "senders" and targets being "receivers", respectively). The `Item` relation in `Transfer` identifies at least one (`some`) thing to be transferred from source to target, which is treated as an occurrence.

```
sig BinaryLink extends Occurrence {
   Source, Target: one Occurrence }

sig Transfer extends BinaryLink {
  Item : some Occurrence }
{ r/acyclic[Item + Source + Target]

  all i in Item | (this->i) in HappensDuring

  this in TransferBefore <=>
    (Source->this) in HappensBefore and
    (this->Target) in HappensBefore }

sig TransferBefore extends Transfer {}
```

Listing 5. Transfers in Alloy

The first `Transfer` signature fact in Listing 5 (line 6) ensures transfers are not items, sources, or targets of themselves, including via those relations to any depth or combination, by applying `acyclic` (lines 10-11 in Listing 1 in Section 3.1.3) to the union of `Item`, `Source`, and `Target` tuples.[21] The second fact (line 8) requires items to exist at least while the transfers carrying them do. See next about the last fact.

The `TransferBefore` signature (line 14) is for transfers that "happen between" their participants, where the source ends (ceases to exist) before the transfer starts, and the target starts (comes into existence) when the transfer ends, as is common for flows between step performances. The last `Transfer` fact (lines 10-12) specifies this with a birectional implication, to ensure `TransferBefore` classifies all and only these kind of transfers. It prevents transfers from an occurrence to or from its steps from being a `TransferBefore`.

Occurrences include relations to identify items for transfers to pick up and drop off, as shown in Listing 6. For transfers between step occurrences, items for pickup are identified by the (`Output`) relation of one step occurrence, while items dropped off are identified by the (`Input`) relation of another step occurrence. For transfers from an occurrence to one of its step occurrences, items for pickup and dropoff are identified by the (`Input`) relation of both occurrence. For transfers in the opposite direction, from a step occurrence to the one it is a step of, items for pickup and dropoff are identified by the (`Output`) relation of both occurrences. Because of this variation in determining items to be transferred, the Transfer signature facts in Listing 5 does not require their `Items` to be identified by `Output`) and (`Input`), leaving this to the occurrence classes that use them. See example translations of SysML object flows in Sections 4.1.4, 4.1.5, and 4.2.2.

```
1 sig Occurrence {
2   Input, Output: set Occurrence }
3 { r/acyclic[Input + Output] }
```

Listing 6. Inputs and outputs in Alloy

The signature fact in Listing 6 (line 3) ensures occurrences are not inputs or outputs of themselves, including via those relations to any depth, via the union of `Input` and `Output` tuples, notated by "`+`" in Alloy.[22]

[21] This does not prevent transfers from transferring other transfers.

[22] This does not prevent behaviors from having input and output relations with themselves as codomain, but the tuples cannot have the same occurrence of the behavior in both positions.

4. Behavior translation examples

This section contains example translations of typical SysML behavior modeling patterns into Alloy. Each example shows a SysML activity diagram and equivalent SysML IBD (internal block diagram) using the OBM method. These are followed by an Alloy translation of the IBD. Finally, atoms and tuples found by Alloy's solvers are shown in directed graphs produced by Alloy, see Figure 6a in Section 3.1.4.[23] The diagrams take several views, mostly of the first below and the rest for solutions of models involving transfers (translations of SysML object flows):

- The most common view in the paper highlights tuples of the `Step` relation with dotted lines, which implies `HappensDuring`, and tuples of the `HappensBefore` relation with solid lines, showing temporal composition and ordering, respectively.
- In views of `Transfers` and their relations, `Source` tuples appear as dotted lines, `Target` tuples as dashed lines, and `Item` tuples inside the rectangles of `Transfer` atoms ("attribute" notation).
- Another view of transfer solutions highlight temporal ordering of steps, as well as their inputs and outputs, specifically `HappensBefore` tuples, as well as `Input` and `Output` tuples with first elements that are the second element of `Step` tuples. `HappensBefore` tuples are shown as solid lines, while `Input` and `Output` tuples appear as attributes of their `Transfer` atoms.
- The third view for transfer solutions combines the the ones above, but omits `Source` and `Target` tuples.

These translations of OBM introduce some predicates as shorthands for longer expressions, improving readability and consistency of the Alloy code. Predicates return a Boolean (`True` or `False`). The additional predicates are defined on first use in the following sections.

Section 4.1 covers examples of common behavior modeling patterns in SysML, while Section 4.2 shows more advanced examples related to behavior taxonomies. Complete Alloy source and other materials for these are available at [13].

4.1 Basic examples

This section translates examples of basic behavior modeling patterns in SysML. Section 4.1.1 covers basic time ordering, including parallel and alternative steps. Section 4.1.2 uses these in loops. Section 4.1.3 shows steps that are other ("nested") behaviors. Section 4.1.4 covers basic transfers. Section 4.1.5 has examples of previous non-loop patterns where the same step is taken more than once, and Section 4.1.6 shows examples that have no solutions (unsatisfiable).

[23] Solutions in this paper were found by Alloy Analyzer 5.1.0 running on a 1.8GHz/16GB laptop.

4.1.1 Time orderings

This section translates SysML activity models of simple sequences of actions. This includes parallel and alternative actions, which use activity control nodes (the same kind of translations apply to SysML state machine pseudostates and interaction fragments, see Section 2.1). Control nodes are translated as logical constraints between the actions they are connected to in the activity, without an additional step for control.

Simple sequence Figures 7 and 8 show SysML activity and OBM representations with two actions and a control flow between them, respectively. In the OBM representation, SimpleSequence is an occurrence class (specialization of Occurrence) with two properties p1 and p2, typed by two other occurrence classes P1 and P2, respectively. Both properties have multiplicity 1 because they will happen once each for every instance (execution) of SimpleSequence.

One connector typed by HappensBefore links p1 with p2. The multiplicities on the connector ends is 1, meaning that each occurrence in p1 must have a HappensBefore relationship to exactly one occurrence in p2, and vice-versa (each occurrence in p2 must have a HappensBefore relationship with exactly one occurrence in p1). This is a simple time ordering of p1 to p2, with occurrences in (values of) p1 in each occurrence of SimpleSequence happening (ending) before occurrences (starting) in p2 in the same occurrence of SimpleSequence.

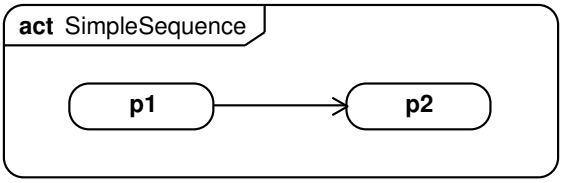


Fig. 7. Simple sequence behavior (Activity)

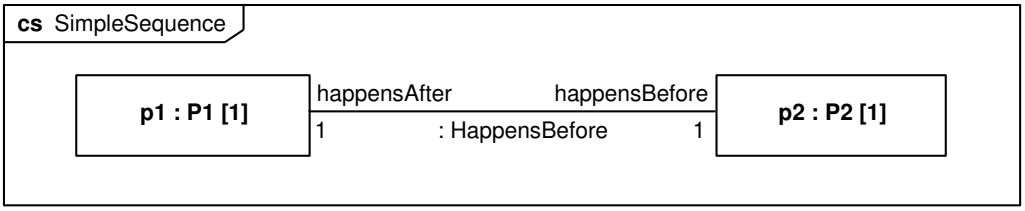


Fig. 8. Simple sequence behavior (OBM)

Listing 7 shows the Alloy translation of Figure 8. `P1` and `P2` are kinds (specializations) of `Occurrence` (line 1), providing the types used by steps `p1` and `p2`, respectively (lines 4 and 5). Line 7 makes tuples in `p1` and `p2` subset those of the `Step` relation for each atom of `SimpleSequence` as the first element (implicit "`this.`", see Figure 4 in 3.1.2), as well as subset in the other direction to constrain those atoms to have only those steps. For brevity, bidirectional `Step` subsets is assumed and omitted in the paper, except in Section 4.2.2. Line 8 is the translation of `p1`'s multiplicity in Figure 8, restricting the step to happen exactly once (to identify exactly one tuple for each atom of `SimpleSequence` as the first element). Step multiplicities are so easily translated that they will mostly be shown at the end of the code and not explained in the rest of the paper.

The connector end multiplicities in Figure 8 model the "one-to-one" semantics of the control flow in Figure 7). They translate to the predefined Alloy predicate `bijection` (lines 7-8 in Listing 2 in Section 3.1.3). However, applying the predicate to the full co(domain) of the `HappensBefore` relation between `p1` and `p2` would be unsatisfiable in most models (those with more than one control flow, see other examples in this section), even though it would work in this simple one. The predicate `bijectionFiltered`, introduced for this translation (line 10), applies `bijection` to subsets of the (co)domain of a relation (1st parameter), specifically, only values (occurrences) of the source step (2nd parameter) from the domain, and only values of the target step (3rd parameter) from the co-domain. The application on Line 10 "filters" `HappensBefore` tuples down to those that have occurrences of the source step as their first element and occurrences of the target step as their second. The definition of `bijectionFiltered` applies `bijection` to the "filtered" relation, as shown in Listing 8.

```
1  sig P1, P2 extends Occurrence{}
2
3  sig SimpleSequence extends Occurrence{
4      p1: set P1,
5      p2: set P2
6  }{
7      p1 + p2 in Step and Step in p1 + p2
8      #p1 = 1
9      #p2 = 1
10     bijectionFiltered[HappensBefore, p1, p2] }
```

Listing 7. Alloy translation of Figure 8

```
1  pred bijectionFiltered[relation: univ -> univ, src, tgt: set
       Occurrence] {
2  r/bijection[(src <: relation) & (relation :> tgt), src, tgt]}
```

Listing 8. Definition of filtered bijection

Figure 9 shows a solution to Listing 7 with a view highlighting `Step` and `Happens Before` tuples. As expected, it includes a single atom for `P1` and `P2`, with a single `HappensBefore` between them.

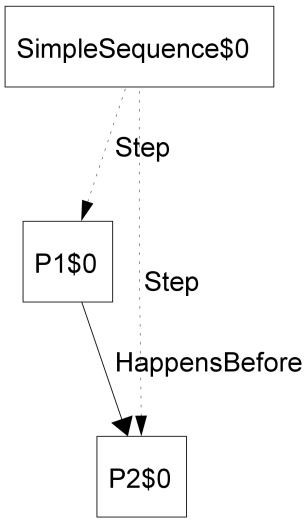


Fig. 9. Solution to Listing 7

Fork Figures 10 and 11 show SysML activity and OBM representations respectively of a behavior with actions that can happen in any order, including simultaneously (activity fork). In the OBM representation, Fork is an occurrence class with three properties p1, p2, and p3, typed by occurrence classes P1, P2, and P3, respectively. The multiplicity of p1 is 1, indicating that the step happens exactly once (like an activity initial node), and the multiplicity on the two other properties is 0..∗, indicating that the steps might happen any number of times (unrestricted actions). Reasoners determine the number of times these happen based on temporal constraints, reducing the burden on modelers to figure this out, especially in examples that have many downstream steps.

Two connectors typed by HappensBefore link p1 with p2 and p3 respectively. The last two steps are not constrained to happen in any particular order (or to happen in any order at all, such as overlapping or simultaneously). The multiplicities on all the ends of the connectors are 1, as in Figure 8, making this like two simple sequences acting independently, but starting with the same step. The connector end multiplicities ensure each occurrence in p1 has a HappensBefore relationship to exactly one occurrence in p2 and exactly one in p3, and vice versa (each occurrence in p2 or in p3 has a HappensBefore relationship with exactly one occurrence in p1). The time ordering is expected to be from p1 to both p2 and p3 independently, with occurrences in p1 happening (ending) before occurrences (starting) in p2 and p3.

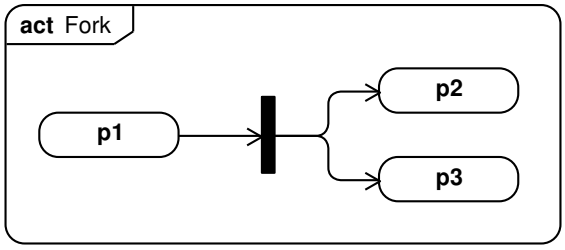


Fig. 10. Fork behavior (Activity)

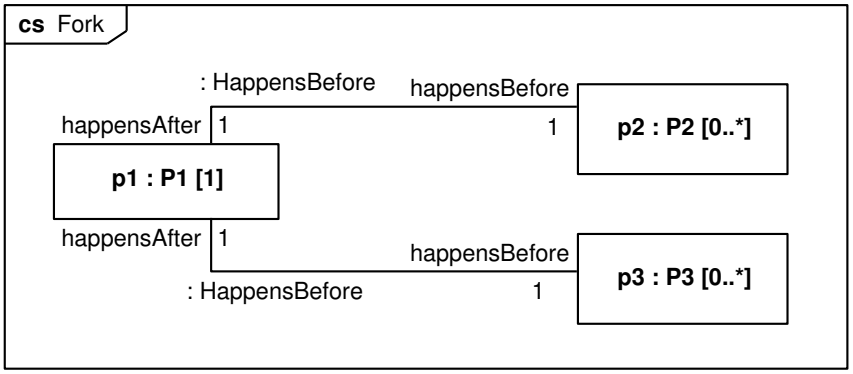


Fig. 11. Fork behavior (OBM)

Listing 9 shows the Alloy translation of Figure 11. `P1`, `P2`, and `P3` are kinds (specializations) of `Occurrence` (line 1), providing the types used by steps `p1`, `p2`, and `p3`, respectively (lines 4-6). The two connectors (fork) translate as two filtered bijections across `HappensBefore`, like two simple sequence translations, one from `p1` to `p2` and the other from `p1` to `p3`, each filtered by their respective steps (lines 8 and 9). The bijections are filtered down to the steps connected by control flows in Figure 10, because the same occurrence in `p1` will happen before two others, those in `p2` and `p3`, see solution in Figure 12.

```
1  sig P1, P2, P3 extends Occurrence{}
2
3  sig Fork extends Occurrence{
4      p1: set P1,
5      p2: set P2,
6      p3: set P3
7  }{
8      bijectionFiltered[HappensBefore, p1, p2]
9      bijectionFiltered[HappensBefore, p1, p3]
10     #p1 = 1 }
```

Listing 9. Alloy translation of Figure 11

Figure 12 shows an Alloy solution to Listing 9 highlighting `Step` and `HappensBefore` tuples. The atom of `P1` happens before the atoms of both `P2` and `P3`, as expected for a fork.

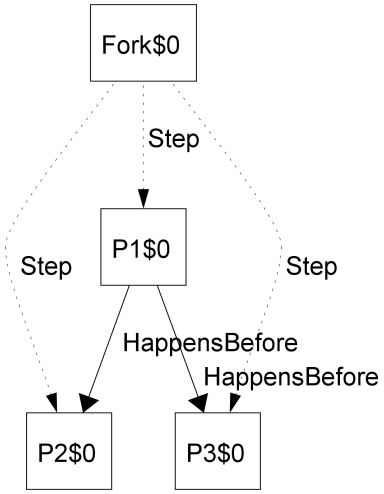


Fig. 12. Solution to Listing 9

Join Figures 13 and 14 show SysML activity and OBM representations respectively of a behavior with an action that happens after two others, which can happen in any order, including simultaneously (activity join). In the OBM representation, Join is an occurrence class with the same properties and occurrence types as Figure 11. Steps p1 and p2 start off the behavior with multiplicity 1, and p3 is unrestricted with $0..*$.

The HappensBefore connectors are the same as in Figure 11, but link the steps differently, p1 and p2 with p3 The multiplicity on all the ends of these connectors is 1, as in Figures 11 and 8, making this like two simple sequences that are dependent on each other by ending with the same step (compare to them being independent in forks). The connector end multiplicities ensure each occurrence in p1 and in p2 has a HappensBefore relationship to exactly one occurrence in p3, and vice-versa (each behavior in p3 has a HappensBefore relationship from exactly one occurrence in p1 and exactly one in p2). The time ordering is expected to be from both p1 and p2 to p3 independently with occurrences in p1 and p2 both happening (ending) before occurrences (starting) in p3.

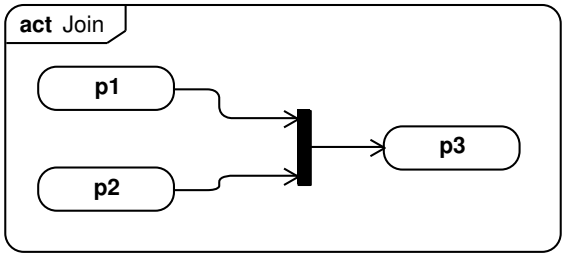


Fig. 13. Join behavior (Activty)

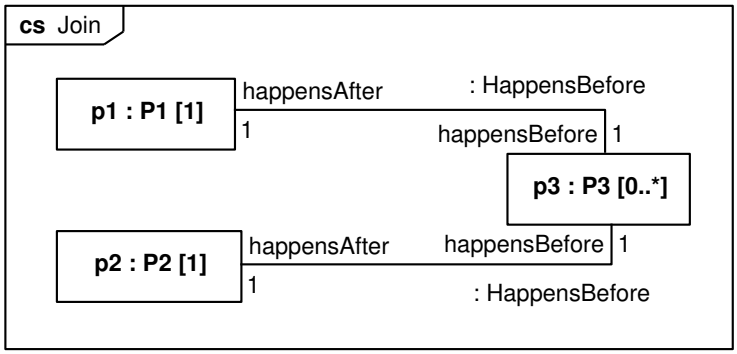


Fig. 14. Join behavior (OBM)

Listing 10 shows the Alloy translation of Figure 14, defining kinds of occurrences and uses of them by steps of `Join` (lines 1 and 4-6, respectively), the same as Listing 9. The two connectors (join) translate as two filtered bijections across `HappensBefore`, like the two simple sequences in Listing 9, but between different steps, one from `p1` to `p3` and the other from `p2` to `p3` (lines 8 and 9). The bijections are filtered down to the steps connected by control flows in Figure 13, because the same occurrence in `p3` will happen after two others, those in `p1` and `p2`, see solution in Figure 15.

```
1  sig P1, P2, P3 extends Occurrence{}
2
3  sig Join extends Occurrence {
4      p1: set P1,
5      p2: set P2,
6      p3: set P3
7  }{
8      bijectionFiltered[HappensBefore, p1, p3]
9      bijectionFiltered[HappensBefore, p2, p3]
10     #p1 = 1
11     #p2 = 1 }
```

Listing 10. Alloy translation of Figure 14

Figure 15 shows a solution to Listing 10 highlighting `Step` and `HappensBefore` tuples. The atoms `P1` and `P2` happen before the atom of `P3`, so the solution corresponds to a join.

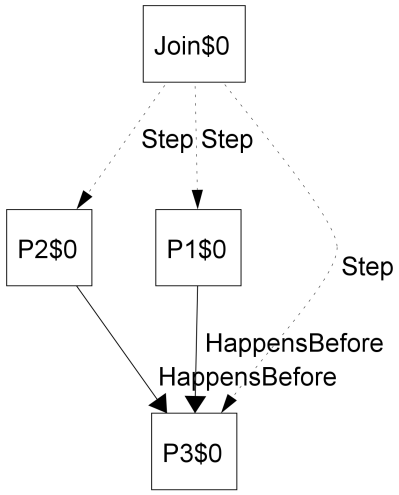


Fig. 15. Solution to Listing 10

Decision Figures 16 and 17 show SysML activity and OBM representations of a behavior with a decision, respectively. In the OBM representation, Decision is an occurrence class with the same properties as in Figures 11 and 14, and multiplicities the same as Figure 11.

Two HappensBefore connectors link p1 with p2 and p3, respectively, as in Figure 11. The multiplicities on the connector ends towards p1 are 1, also as in Figure 11, but the multiplicities on the ends towards p2 and p3 are 0..1 (optional), leaving the reasoner to determine which one will happen and which will not. However, the optional connector end multiplicities do not reflect the intention that exactly one of p2 and p3 will happen. It would be consistent with the model for them both to happen or neither to happen. A SysML constraint could be added to prevent this, but SysML is not currently defined in a logical enough manner to express the constraint. The connector end multiplies in Figure 17 as they are only require occurrences in p1 to have a HappensBefore relationship to no more than one occurrence each in p2 or p3, and each occurrence in p2 or in p3 to have a HappensBefore relationship from exactly one occurrence in p1. Alloy is expressive enough to define the decision constraint, see below.

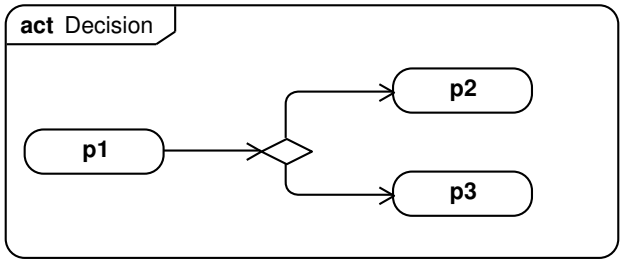


Fig. 16. Decision behavior (Activity)

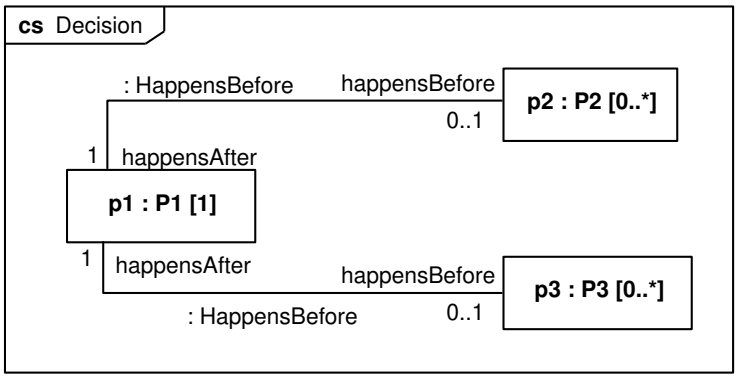


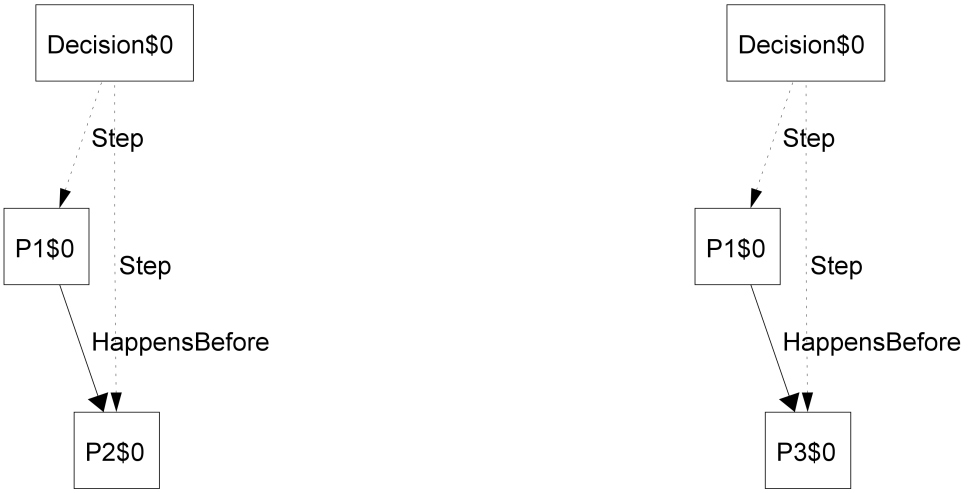
Fig. 17. Decision behavior (OBM)

Listing 11 shows the Alloy translation of Figure 17, defining kinds of occurrences and uses of them by steps of `Decision` (lines 1 and 4-6, respectively), the same as in Listings 9 and 10. The two connectors translate to a single filtered bijection across `HappensBefore` from `p1` to the union of `p2` and `p3` (line 8). Unioning the codomain of the bijection ensures that exactly one of the `p2` and `p3` steps will be taken (have a value), which is the constraint missing in Figure 17.

```
1 sig P1, P2, P3 extends Occurrence{}
2
3 sig Decision extends Occurrence {
4     p1: set P1,
5     p2: set P2,
6     p3: set P3
7 }{
8     bijectionFiltered[HappensBefore, p1, p2 + p3]
9     #p1 = 1 }
```

Listing 11. Alloy translation of Figure 17

Figure 18 shows solutions to Listing 11 highlighting `Step` and `HappensBefore` tuples. Figure 18a shows a solution in which `P1` happens before P2 and there are no atoms of P3, corresponding to a decision in which P2 is chosen. Figure 18b shows a solution in which a P1 atom happens before a P3 and there are no atoms of P2, corresponding to a decision in which P3 is chosen.



(a) Solution choosing P2 **(b)** Solution choosing P3

Fig. 18. Solutions to Listing 11

Merge Figures 19 and 20 show SysML activity and OBM representations of a behavior with a merge, respectively. In the OBM representation, Merge is an occurrence class with the same properties as Figures 11,14, and 17, and multiplicities the same as Figure 14.

Two HappensBefore connectors link p1 and p2 to p3, as in Figure 14. The multiplicities on the ends towards p1 and p2 are 1, also as in Figure 14, but the multiplicities on the ends towards p3 are 0..1 (optional), leaving the reasoner to determine which occurrences (values) of p3 will follow which occurrences of p1 and p2. However, the optional connector end multiplicities do not reflect the intention that p3 happen twice, once for each occurrence of p1 and p2, which is makes this a merge, rather than a join. It would be consistent with the model for p3 to happen only once, or not at all. A SysML constraint could be added to prevent this, but SysML is not currently defined in a logical enough manner to express the constraint. The connector end multiplies in Figure 20 as they are only require occurrences in p1 and p2 to have a HappensBefore relationship to no more than one occurrence each in p3, and each occurrence in p3 to have a HappensBefore relationship from exactly one occurrence each in p1 and p2. Alloy is expressive enough to define the merge constraint, see below.

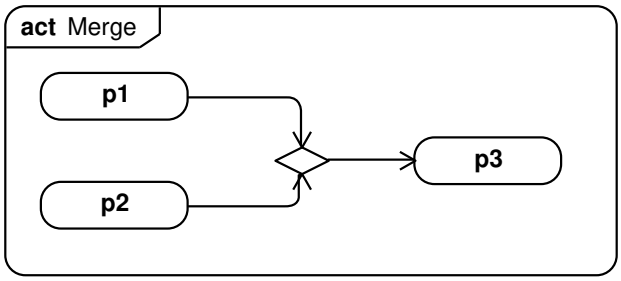


Fig. 19. Merge behavior (Activty)

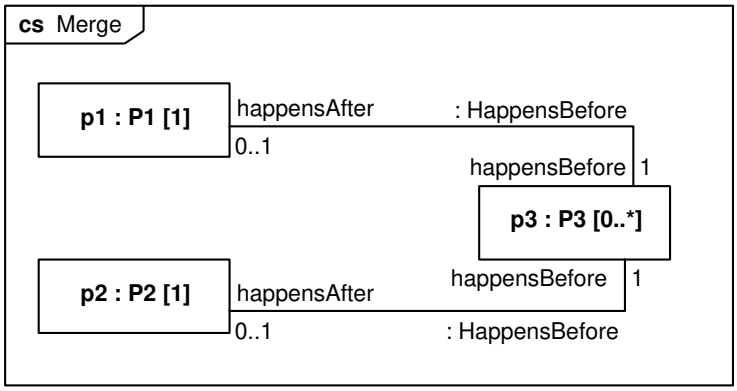


Fig. 20. Merge behavior (OBM)

Listing 12 shows the Alloy translation of Figure 20, defining kinds of occurrences and uses of them by steps of `Merge` (lines 1 and 4-6, respectively), the same as in Listings 9, 10, and 11. The two connectors translate to a single filtered bijection across `HappensBefore` from the union of `p1` and `p2` to `p3` (line 8). Unioning the domain of the bijection ensures that step `p3` will taken (have a value) exactly once each time `p1` or `p2` are, which is the constraint missing in Figure 20.

```
1  sig P1, P2, P3 extends Occurrence{}
2
3  sig Merge extends Occurrence {
4      p1: set P1,
5      p2: set P2,
6      p3: set P3
7  }{
8      bijectionFiltered[HappensBefore, p1 + p2, p3]
9      #p1 = 1
10     #p2 = 1
```

Listing 12. Alloy translation of Figure 20

Figure 21 shows a solution to Listing 12 highlighting `Step` and `HappensBefore` tuples. For each atom of `P3`, there is exactly one atom each of `P1` and `P2` related by `HappensBefore`, as required for merges.

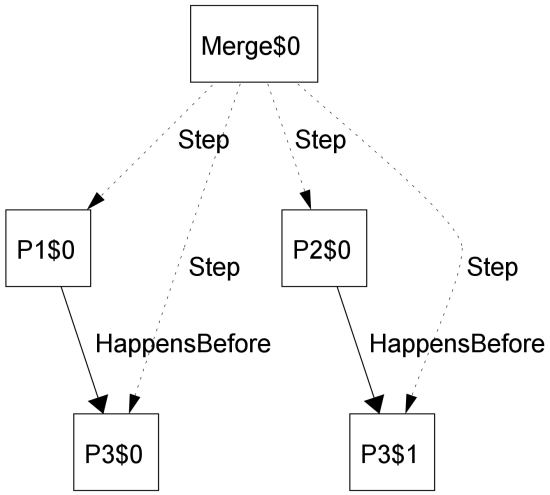


Fig. 21. Solution to Listing 12

Altogether Figures 22 and 23 show SysML activity and OBM representations respectively of a behavior that combines a fork, join, decision, and merge. In the OBM representation, AllControl is an occurrence class with seven occurrence properties (steps), linked by HappensBefore connectors to form the constructs detailed above:

- Fork: between p1, p2, and p3
- Join: between p2, p3, and p4
- Decision: between p4, p5, and p6
- Merge: between p5, p6, and p7

The steps are expected to be taken in order 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.

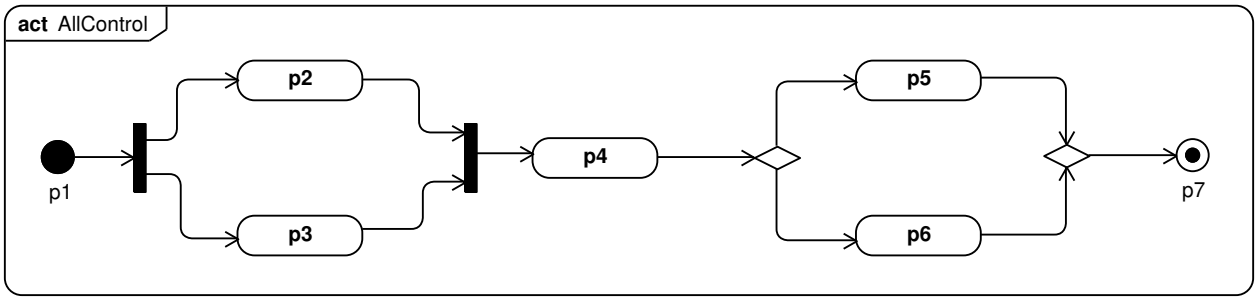


Fig. 22. All control nodes in one behavior (Activity)

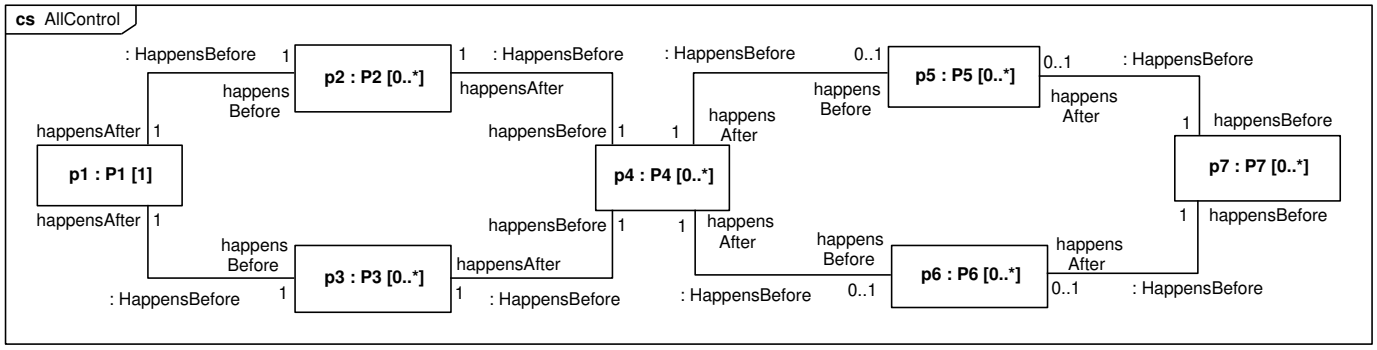


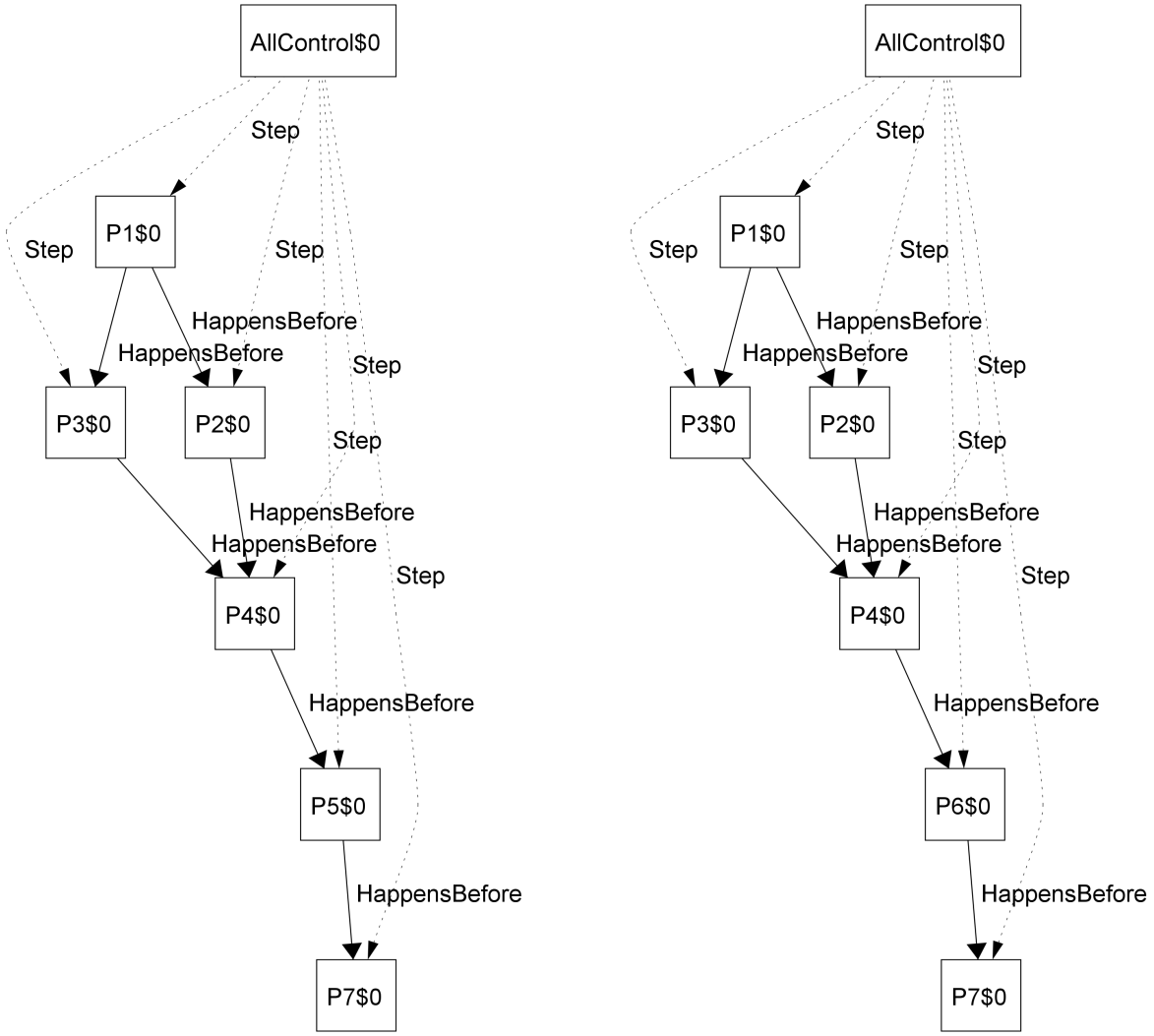
Fig. 23. All control nodes in one behavior (OBM)

Listing 13 shows the Alloy translation of Figure 23. The kinds of occurrences (`P1` through `P7`) and uses of them in `AllControl` (line 1 and 4-5, respectively) are similar to the examples above. The bijections specifying the control nodes are all across the `HappensBefore` relation, filtered by the steps they constrain, also as in the corresponding examples above.

```
1  sig P1, P2, P3, P4, P5, P6, P7 extends Occurrence{}
2
3  sig AllControl extends Occurrence{
4      p1: set P1, p2: set P2, p3: set P3,
5      p4: set P4, p5: set P5, p6: set P6, p7: set P7
6  }{  // Fork
7      bijectionFiltered[HappensBefore, p1, p2]
8      bijectionFiltered[HappensBefore, p1, p3]
9      // Join
10     bijectionFiltered[HappensBefore, p2, p4]
11     bijectionFiltered[HappensBefore, p3, p4]
12     // Decision
13     bijectionFiltered[HappensBefore, p4, p5 + p6]
14     // Merge
15     bijectionFiltered[HappensBefore, p5 + p6, p7]
16     #p1 = 1}
```

Listing 13. Alloy translation of Figure 23

Figure 24 shows solutions to Listing 13 highlighting `Step` and `HappensBefore` tuples. Figure 24a shows a solution in which a `P1` atoms happens before those of `P2` and `P3`, demonstrating the fork; `P2` and `P3` atoms happen before a `P4`, demonstrating the join; a `P4` happens before a `P5`, and there are no atom of `P6`, demonstrating the decision; and a `P5` happens before a `P7` with no atoms of `P6`, demonstrating the merge. Figure 24b shows a similar sequence choosing `P6` instead of `P5` in the decision and merge.



(a) Solution choosing P5 **(b)** Solution choosing P6

Fig. 24. Solutions to Listing 13

4.1.2 Loops

Figures 25 and 26 show SysML activity and OBM representations of a behavior with a loop, respectively. The OBM representation of Loop has the same three occurrence classes and uses of them as most of the examples in 4.1.1, except the multiplicity of p2 is 2..∗ to require at least two occurrences in the loop, and p3 has multiplicity 1..∗, leaving the reasoner to determine how many times it will happen.

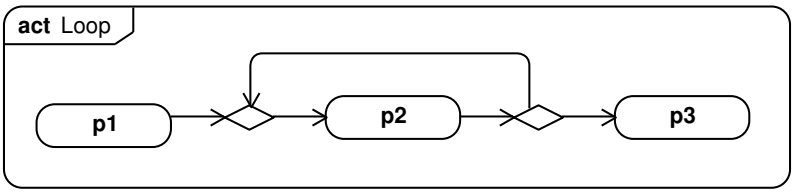


Fig. 25. Loop behavior (Activity)

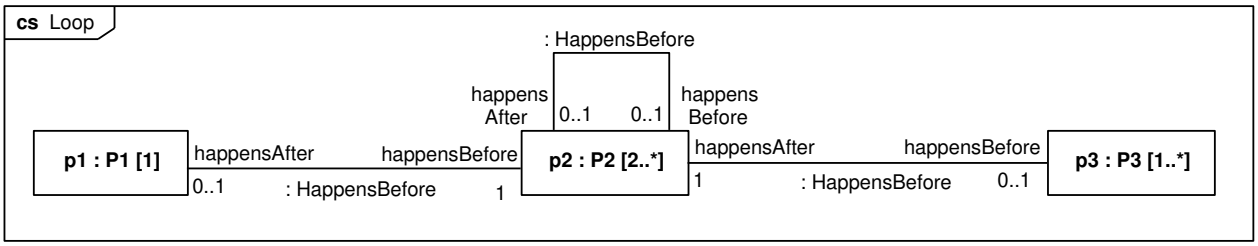


Fig. 26. Loop behavior (OBM)

Three HappensBefore connectors link

1. p1 to p2, with optional end multiplicity towards p1, as part of a merge to start the loop with this connector, or continue it with connector 3 (p2 to itself). See Section 4.1.1 about merges and decisions, Figures 20 and 17.

2. p2 to p3, with optional end multiplicity towards p2, as part of a decision to end the the loop via this connector, or continue it via connector 3.

3. p2 to itself, with optional end multiplicities at both ends. The happensBefore end multiplicity is part of a merge with connector 1 (p1 to p2), while the happensAfter end multiplicity is part of a decision with connector (p2 to p3).

The optional connector end multiplicities do not reflect the intention that p2 will happen once due to connector 1 (from p1), and loop more times due to connector 2 (from itself). It would be consistent with the model for p2 to happen any number of times (except zero), regardless of when p1 and p3 happen. A SysML constraint could be added to prevent this, but SysML is not currently defined in a logical enough

manner to express the constraint. The connector end multiplies in Figure 26 as they are only require occurrences in p2 to have a HappensBefore relationship to no more than one occurrence each in p1 or p3, and each occurrence in p1 or in p3 to have a HappensBefore relationship to/from exactly one occurrence in p2. Alloy is expressive enough to define these merge and decision constraints, see below.

Listing 14 shows the Alloy translation of Figure 26, defining kinds of occurrences and uses of them by steps of `Loop` (lines 1 and 4-6, respectively), the same as in most of the listings in Section 4.1.1.

Translating the connector end multiplicities in Figure 26 is dependent on which properties are connected and which is at the optional end, compared to translating them all to filtered bijections, as in the listings in Section 4.1.1. This requires filtered versions of predefined predicates `function` and `inverseFunction` (see lines 1-2 and 4-5, respectively, in Listing 2 in Section 3.1.3), as defined in Listing 15. The connector in Figure 26 from

1. p1 to p2 translates to `functionFiltered` (line 8), applying `function` to the subset of `HappensBefore` tuples pairing occurrences in p1 with occurrences in p2. This requires every occurrence in p1 to be followed by exactly one occurrence in p2, per the end multiplicity of the connector on its p2 end.

2. p2 to p3 translates to `inverseFunctionFiltered` (line 11), applying `inverse Function` to the subset of `HappensBefore` tuples pairing occurrences in p2 with occurrences in p3. This requires every occurrence in p3 to follow exactly one occurrence in p2, per the end multiplicity of the connector on its p2 end.

3. p2 to p2 translates to `inverseFunctionFiltered` and `functionFiltered`, reflecting the connector's dual role in a merge and decision, respectively:
   (a) `inverseFunctionFiltered` (line 9) applies `inverseFunction` to `Happens Before` tuples filtered down to pairing occurrences in p1 or p2 (union) with occurrences in p2. This requires every occurrence in p2 to follow exactly one occurrence in either p1 or p2, the merge constraint missing in Figure 26.
   (b) `functionFiltered` (line 10) applies `function` to `HappensBefore` tuples filtered down to pairing occurrences in p2 with occurrences in p2 or p3 (union). This requires every occurrence in p2 to be followed by exactly one occurrence in p2 or p3, the decision constraint missing in Figure 26.

The above combine to ensure that (only) occurrences in p1 start the loop and (only) occurrences in p3 end it.

```
sig P1, P2, P3 extends Occurrence{}

sig Loop extends Occurrence {
    p1: set P1,
    p2: set P2,
    p3: set P3
}{
    functionFiltered[HappensBefore, p1, p2]
    inverseFunctionFiltered[HappensBefore, p1 + p2, p2]
    functionFiltered[HappensBefore, p2, p2 + p3]
    inverseFunctionFiltered[HappensBefore, p2, p3]

    #p1 = 1
    #p2 >= 2
    #p3 >= 1}
```

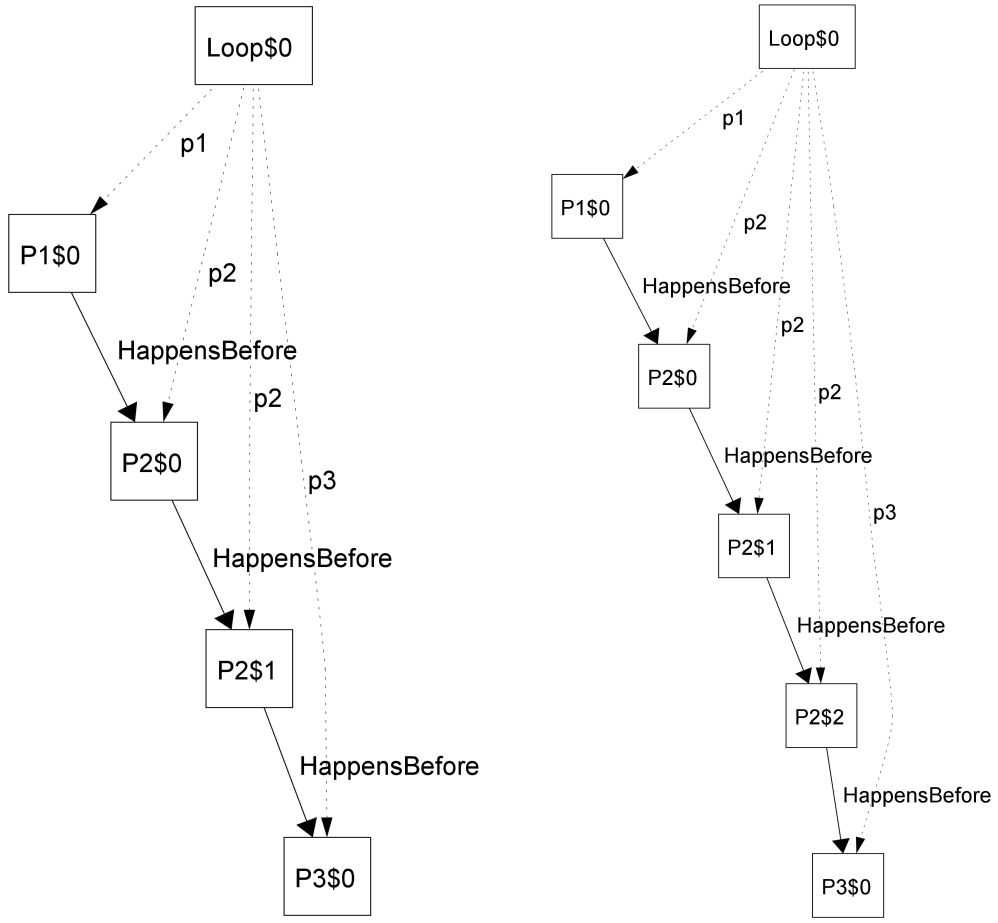
Listing 14. Alloy translation of Figure 26

Listing 15 shows the definitions of `function Filtered` and `inverseFunction Filtered` introduced for this translation. These apply the predefined predicates `function` and `inverseFunction`, respectively (see Section 3.1.3), to subsets of the (co)domain of a relation (1st parameter), specifically, only values (occurrences) of the source step (2nd parameter) from the domain, and only values of the target step (3rd parameter) from the co-domain, similarly to `bijectionFiltered`, see Listing 8 in Section 4.1.1.

```
pred functionFiltered[relation: univ -> univ, src, tgt: set
   Occurrence] {
r/function[(src <: relation) & (relation :> tgt), src]}

pred inverseFunctionFiltered[relation: univ -> univ, src, tgt:
    set Occurrence] {
r/inverseFunction[(src <: relation) & (relation :> tgt), tgt]}
```

Listing 15. Definition of filtered function and inverseFunction predicates

Figure 27 shows solutions to Listing 14 highlighting `Step` and `HappensBefore` tuples. Figure 27a shows a solution with the minimum number of `P2` atoms, both for step `p2`. The temporal flow goes from `p1` to `p2`. The flow goes a second time to `p2` before `p3`, forming a loop. Generating more solutions in Alloy produces solutions with more visits to `p2` in the loop, as shown in Figure 27b. See the last example in Section 4.2.2 for object flows in loops.



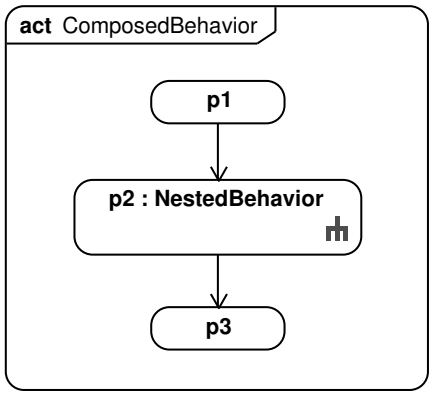
(a) Solution with 2 atoms of P2 **(b)** Solution with 3 atoms of P2

Fig. 27. Solutions to Listing 14

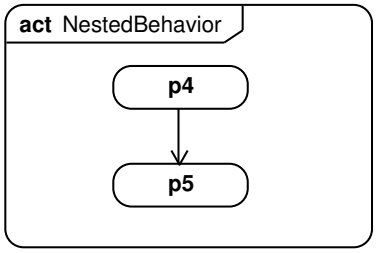
4.1.3 Using behaviors that have steps

Figures 28 and 29 show SysML activity and OBM representations, respectively, of a behavior using ("calling") another behavior that has steps of its own. The activity in Figure 28a has an action p2 that uses the activity in Figure 28b, which has its own actions p4 and p5. In the OBM representation, ComposedBehavior is a block with a property p2 typed by NestedBehavior, which has its own steps p4 and p5, typed by P4 and P5, respectively (shown nested in its property rectangle, rather than as a separate diagram). Before and after p2 are steps typed by behaviors that do not have steps, as in previous examples.

Two HappensBefore connectors link p2 to the steps before and after it, and another one links the steps in ComposedBehavior. All connector end multiplicities are 1. The steps are expected to be taken in order from p1 to p2, during which p4 are p5 are taken in order, and when those are done, p3.



(a) Behavior with an action using Figure 28b



(b) Behavior with used by Figure 28a

Fig. 28. Behavior (on left) with an action using another behavior (on right) (Activity)

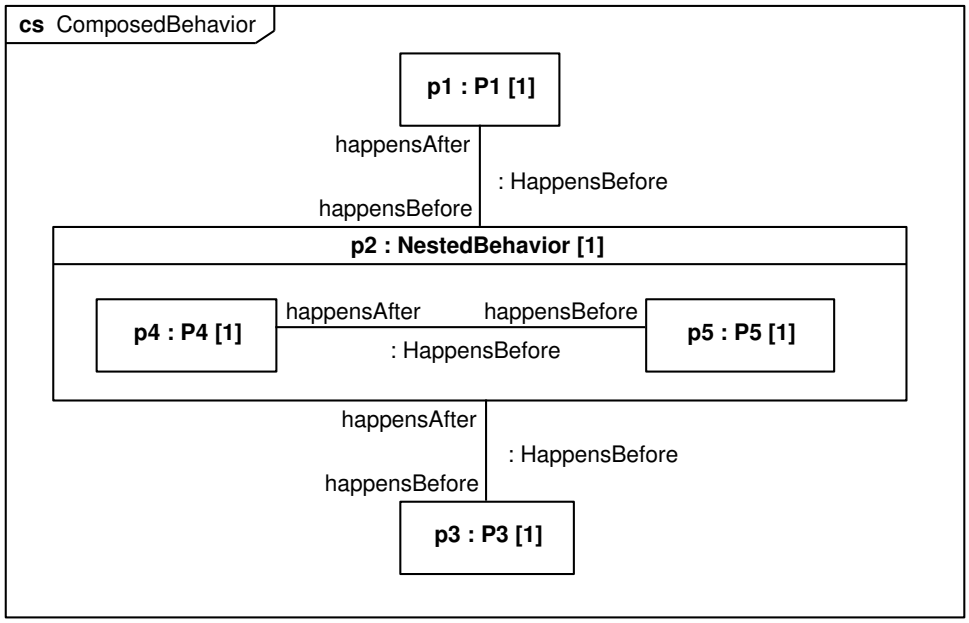


Fig. 29. Behavior with a step typed by another behavior, shown nested (OBM)

Listing 16 shows the Alloy translation of Figure 29, which is similar to the simple sequence in Listing 7 in Section 4.1.1, with an additional step relation for another occurrence signature that has its own step relations. `P1` through `P5` specialize `Occurrence` without defining any steps for them (line 1). `NestedBehavior` specializes `Occurrence` also (line 3), but includes steps for `P4` and `P5` (lines 4 and 5). `ComposedBehavior` pulls these all together with steps for `P1`, `NestedBehavior`, and `P3` (lines 11-13).

Control flows in Figure 29 translate to filtered bijections over `HappensBefore` (lines 7, 15, and 16), as in simple sequences. The multiplicity of the initial steps are `1`

(lines 8 and 17). The series of bijections imply the steps following the initial ones also happen exactly once, even though these have no multiplicities (unrestricted).

```
1  sig P1, P3, P4, P5 extends Occurrence{}
2
3  sig NestedBehavior extends Occurrence {
4      p4: set P4,
5      p5: set P5
6  }{
7      bijectionFiltered[HappensBefore, p4, p5]
8      #p4 = 1     }
9
10 sig ComposedBehavior extends Occurrence{
11     p1: set P1,
12     p2: set NestedBehavior,
13     p3: set P3
14 }{
15     bijectionFiltered[HappensBefore, p1, p2]
16     bijectionFiltered[HappensBefore, p2, p3]
17     #p1 = 1     }
```

Listing 16. Alloy translation of Figure 29

Figure 30 shows a solution to Listing 16 highlighting `HappensBefore` tuples and subsets of `Step` tuples (`p1`, `p2`, and so on). It shows the occurrences happening during ("composed into") an atom of `ComposedBehavior`, which has an atom of `P1` as the first step, `NestedBehavior` as the second step, and `P3` as the third step. The `NestedBehavior` atom further decomposes into atoms of `P4` and `P5`, as expected for a called behavior. Because atoms of `P4` and `P5` happen during that of `NestedBehavior`, and a `P1` happens before the `NestedBehavior` atom, the `P1` atom should happen before those of `P4` and `P5`. Similarly, because `NestedBehavior` happens before `P3`, both `P4` and `P5` should happen before `P3`. Figure 30 shows that both of these expectations are met.

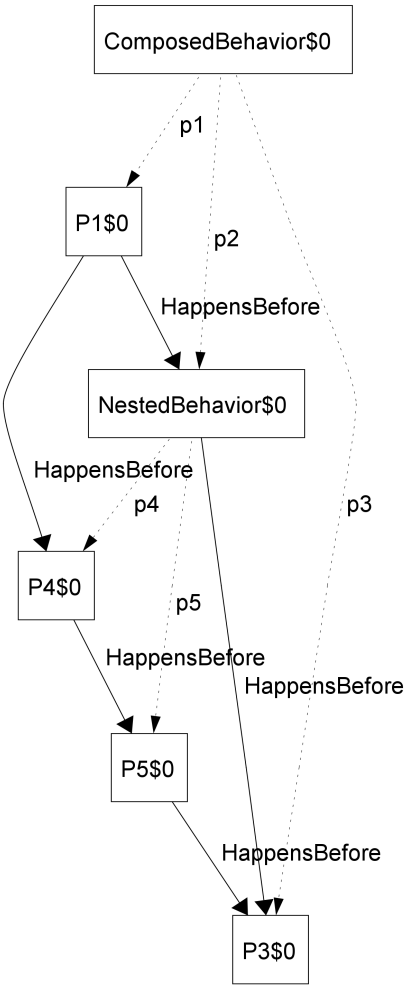


Fig. 30. Solution to Listing 16

4.1.4 Transfers and parameters

This section translates examples of SysML item and object flows and behavior parameters, for passing things between behaviors.

Item flows Figures 31 and 32 shows a SysML internal block diagram and an OBM representation, respectively, of transferring something between two interacting participants. The SysML diagram includes an item flow (filled triangle) on the connector between properties for the participants, supplier of type Supplier, and customer of type Customer. The item flow indicates that Products move from one participant to the other, in the direction the triangle is pointing. The participant types have flow properties that provide or accept the things flowing, suppliedProduct and receivedProduct, respectively, both of type Product (shown in nested property rectangles, dashed to indicate the outputs and inputs are not part of the participants,

just flowing out of and into them). The OBM representation is an occurrence class defining the same properties for the participants, and properties on their types for output and input.[24] The participant properties are linked by a connector of type Transfer. Solutions are expected to show that the product (instance) output from supplier is the same in the one input to customer.[25]

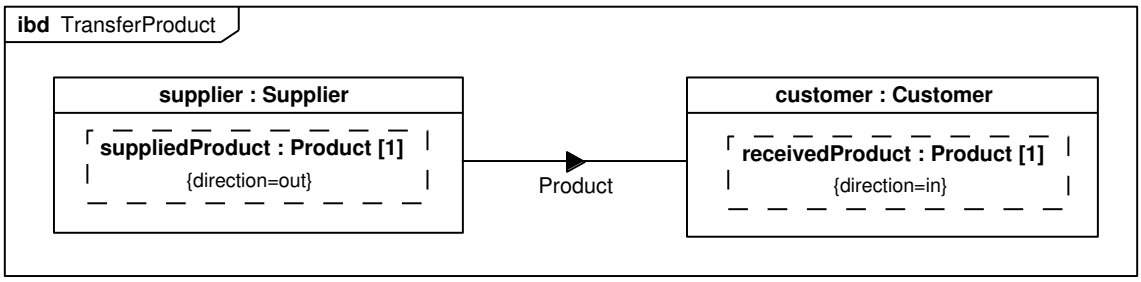


Fig. 31. Item flow behavior (Internal Block Diagram)

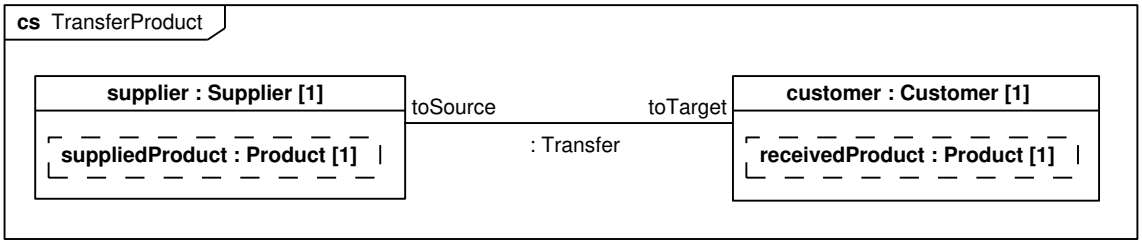


Fig. 32. Item flow behavior (OBM)

Listing 17 shows the Alloy translation of Figure 32. Signature facts for `Supplier` and `Customer` subset `suppliedProduct` and `receivedProduct` from `Output` and `Input`, respectively (lines 5 and 9), see Listing 6 in Section 3.2. Output and input multiplicities are translated in the declaration of those relations (lines 4, 8, and 12-14).

The `Source` and `Target` relations from `Transfer` are constrained by filtered bijections (lines 16 and 17), because in `TransferProduct` a `Product` can only flow from the `Supplier` to the `Customer` via a `Transfer`. Each occurrence (value) of `transferSupplierCustomer` must link (only) the `supplier` and `customer`. The translation introduces predicates `subsettingItemRuleForSources` and `subsettingItem RuleForTargets` to specify that the transfer identified by `transferSupplierCustomer` picks up its `Item` from the `Output` of the `Source` and drops it off at the `Input` of the

[24]The OBM models for item flows do not link them to the output and input properties of their end types, which also do not give direction, to avoid depending on metaclassification of model elements, as in [6], for simplicity. The translations to Alloy reflect this information, however.

[25]The translation does not currently support changing objects, so the solver cannot produce atoms of the two participants during the time when they do not output and input a product.

Target, respectively, see Transfers in Section 3.2 (taking the participants as steps) and the description of Listing 19 in the next example, last bullet.

```
sig Product extends Occurrence{}

sig Supplier extends Occurrence{
    suppliedProduct: one Product
}{ suppliedProduct in Output and Output in suppliedProduct}

sig Customer extends Occurrence{
    receivedProduct: one Product
}{ receivedProduct in Input and Input in receivedProduct}

sig TransferProduct extends Occurrence{
    supplier: one Supplier,
    customer: one Customer,
    transferSupplierCustomer: set Transfer
}{
    bijectionFiltered[Source, transferSupplierCustomer,
       supplier]
    bijectionFiltered[Target, transferSupplierCustomer,
       customer]
    subsettingItemRuleForSources[transferSupplierCustomer]
    subsettingItemRuleForTargets[transferSupplierCustomer]}
```

Listing 17. Alloy translation of Figure 32

Figures 33 through 35 show multiple views of the same solution to Listing 17. The first and last figures are slightly different than the views described at the beginning of Section 4. Figure 33 shows `Step` tuples as attributes of `TransferProduct`, and the specific subsets of `Step` as edges on the graph, `supplier`, `transferSupplierCustomer`, and `customer`. The figures do not show `HappensBefore` tuples because the model does not require these structural components to happen (exist) in any particular order, although they *might* in some solutions.

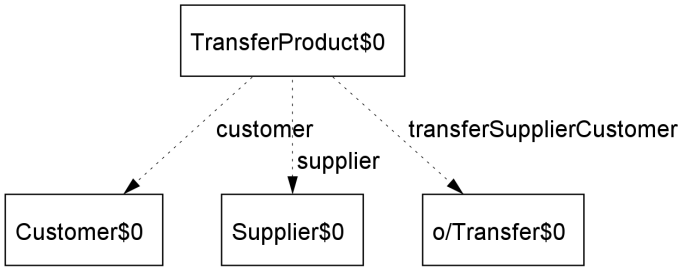


Fig. 33. View of solution to Listing 17 showing subsets of Step

Figure 34 shows a view highlighting `Source` and `Target` tuples (atoms of `Transfer` as their first element). The atom of `Product` is the `Item` being transferred from `Supplier` to `Customer`.

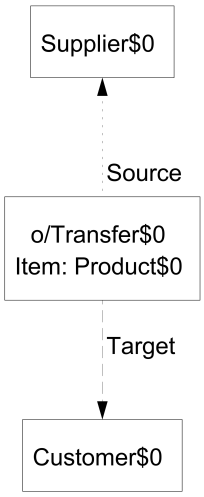


Fig. 34. Another view of the solution to Listing 17, showing Transfer relations

Figure 35 highlights `suppliedProduct` and `receivedProduct` tuples, shown as edges on the graph, as subsets of `Output` and `Input`, respectively.

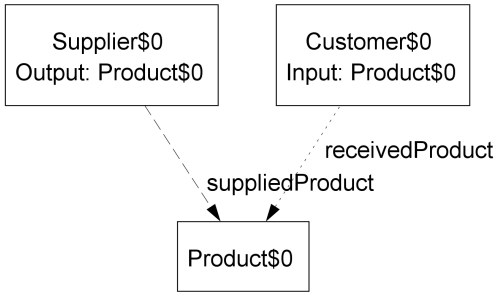
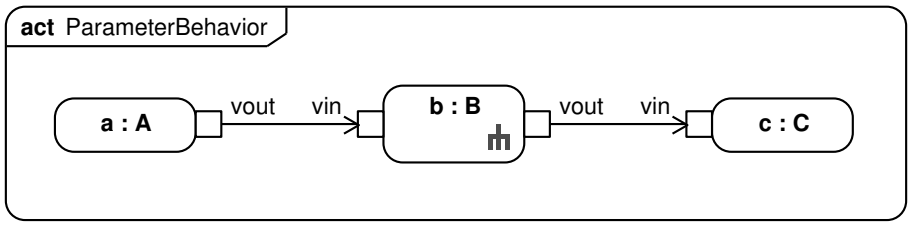


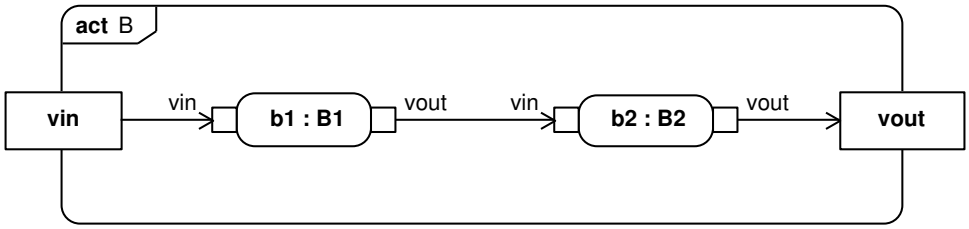
Fig. 35. A third view of the solution to Listing 17, showing subsets of Input and Output

Object flows and behavior parameters Figures 36 and 37 (using occurrence classes in Figure 38) show SysML activity and OBM representations of a behavior with object flows carrying values of behavior parameters between steps, respectively. The activities are similar to Figure 28 in Section 4.1.3, with different names, having an action using an activity that has its own actions, but with object flows between parameter nodes and pins (appearing as small rectangles on activities and actions, respectively), rather than control flows between actions. The OBM representation follows the pattern of Figure 29, with different names, and Transfer or TransferBefore connectors instead of HappensBefore, see Listing 5 in Section 3.2. Object flows between

pins translate to TransferBefore, connectors because these transfers happen after their source step ends and before their target step starts, while those from or to activity parameter nodes translate to Transfer connectors, because the transfers happen during their source or target when those are occurrences of their activity. Parameter nodes translate to input and output properties of occurrence classes, vin as input and vout as output from the occurrence classes (some having only input or only output), all with multiplicity 1.[26] SelfLink connectors in the OBM representation equate values of input and output properties for B1 and B2, like SysML binding connectors. This has the same effect as transferring input values directly to outputs, except equating values does not take time, like transfers do.



(a) Behavior with an action using Figure 36b



(b) Behavior used by Figure 36a

Fig. 36. Behavior using another behavior that has parameters (Activity)

[26]See footnote 24 in the previous example, about translating item flows, inputs, and outputs to OBM. The same applies to object flows, inputs, and outputs in behaviors also.

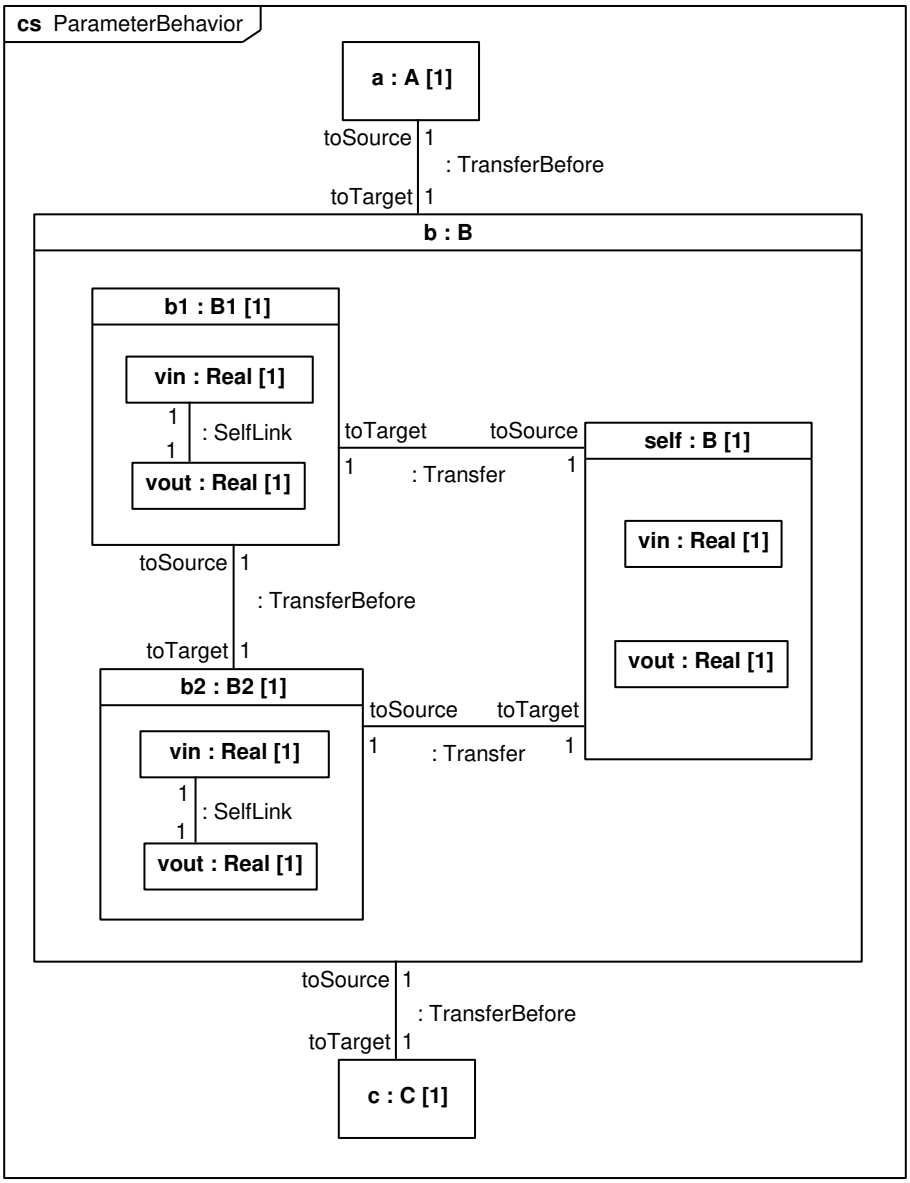


Fig. 37. Behavior using behavior that has parameters, shown nested (OBM)

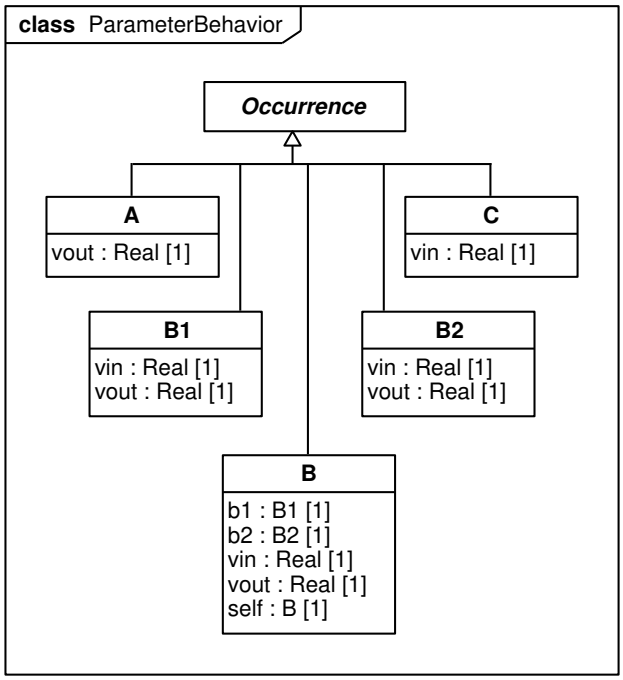


Fig. 38. Occurrence classes used in Figure 37

The Alloy translation of Figure 37 is presented below in several listings discussed separately, building up from the specifications for the lowest level steps to the highest. Listing 18 shows translations of occurrence classes B1 and B2. `Real` classifies atoms for real numbers, also treated as occurrences, for simplicity, but constrained to act just as distinguishable things, rather than full occurrences (lines 1-3). Input and outputs `vin` and `vout` subset the `Input` and `Output` relations, respectively (lines 8-9 and 17-18), as in the translation of object flows above. SelfLink connectors in Figure 37 between inputs and outputs of B1 and B2 translate to Alloy's equality operator (lines 10 and 19).

```
1  sig Real extends Occurrence{}{
2    no HappensBefore && no @HappensBefore.this &&
3      no Step && no Input && no Output}
4
5  sig B1 extends Occurrence
6  {   vin, vout: set Real
7  }{
8      vin in Input and Input in vin
9      vout in Output and Output in vout
10     vin = vout
11     #vin = 1
12     #vout = 1}
13
```

```
14 sig B2 extends Occurrence
15 {   vin, vout: set Real
16 }{
17     vin in Input and Input in vin
18     vout in Output and Output in vout
19     vin = vout
20     #vin = 1
21     #vout = 1}
```

Listing 18. Alloy translation of Real and occurrence classes B1 and B2 in Figure 37

Listing 19 shows the Alloy translation of occurrence class B in Figure 37. Steps, inputs, and outputs translate in the same way as above. The Transfer connectors require several constraints, some introducing predicates for this translation to capture commonly needed constraints, improving readability and reliability of the Alloy code:

- These connectors never have the same values (line 5).[27]
- The values of these connectors (transfers) sometimes happen after their source, before their target, or both, as specified with the predicates `isAfterSource` and `isBeforeTarget`, see Listing 20. Connectors translated from object flows linking
  - parameter nodes to pins require their transfers to happen before their target, `isBeforeTarget` (line 17).
  - pins to parameter nodes require their transfers to happen after their source, `isAfterSource` (line 38).
  - pins to pins require their transfers to happen in the time between their source and target, `isBeforeTarget` and `isAfterSource` (lines 27-28).

  The first above are transfers with sources of which they are also a step (the source and the transfer are a tuple in the `Step` relation), while the second is likewise for targets. These transfers HappenDuring each occurrences of B, like the other steps, rather than before or after it, and cannot be TransferBefores. The third is for transfers with sources and targets that are steps of the same occurrence as the transfer (the source or target and the transfer are the second elements in tuples of the `Step` relation that have the same first element). These are equivalent to TransferBefores and the corresponding connectors can be typed that way, as in Figure 37.

[27] This is also implied by the translation of connector multiplicities, see next bullets, but is included here and in other object flow translations to reflect the intent and simplify future automation. It is not necessary to do the same for `b1` and `b2`, because Alloy treats their signatures `B1` and `B2` as disjoint due to extending the same signature, `Occurrence` (signatures extending the same other signature never classify the same atoms).

- Sources and targets of the connector values (transfers) are constrained to be occurrences of B or (values) of one of its steps, depending on whether the corresponding end of the object flow being translated is an activity parameter node of B or a pin of one of its actions, respectively. For ends of connectors translated from object flow ends that are
  - parameter nodes, the constraint is a filtered function (lines 13 and 35) rather than a bijection, to accommodate translations of similar models where the containing occurrence is the source or target for more than one transfer from steps with multiple executions.
  - pins, the corresponding constraint is a filtered bijection (lines 14, 23-24, and 34), similar to control flows in simple sequences, see Listing 7 in Section 4.1.1.

- The items carried by values of these connectors (transfers) are constrained to be either the outputs or inputs of their source or target, depending on whether the corresponding end of the object flow being translated is an activity parameter node or a pin. The difference is that items for connectors between
  - parameter nodes and pins are either picked up at input parameter nodes and dropped off at input pins or picked up at output pins and dropped of at output parameter nodes.
  - pins and pins are always picked up at output pins and dropped off at input pins.

  The predicates `subsettingItemRuleForSources` and `subsettingItemRuleForTargets` applied to transfers requires their items to be among the source or target inputs or outputs according to which one of the above they are (lines 15-16, 25-26, 36-37), see Listing 20.[28]

```
1  sig B extends Occurrence {
2      vin, vout: set Real,
3      b1: set B1,
4      b2: set B2,
5      disj transferBB1, transferB1B2, transferB2B: set Transfer
6  }{
7      vin in Input and Input in vin
8      vout in Output and Output in vout
9      #vin = 1
10     #vout = 1
11
```

[28] In this example, subsetting is equivalent to items being the same as inputs/outputs, as if it were bidirectional (see Listing 41 in the second example in Section 4.2.2), because transfers must carry at least one item and their sources and targets accept/produce exactly one thing.

```
12      /** Constraints on transfer from B input to B1 input */
13      functionFiltered[Source, transferBB1, this]
14      bijectionFiltered[Target, transferBB1, b1]
15      subsettingItemRuleForSources[transferBB1]
16      subsettingItemRuleForTargets[transferBB1]
17      isBeforeTarget[transferBB1]
18
19      /** Constraints on b1: B1 */
20      #b1 = 1
21
22      /** Constraints on transfer from B1 output to B2 input */
23      bijectionFiltered[Source, transferB1B2, b1]
24      bijectionFiltered[Target, transferB1B2, b2]
25      subsettingItemRuleForSources[transferBB1]
26      subsettingItemRuleForTargets[transferBB1]
27      isAfterSource[transferB1B2]
28      isBeforeTarget[transferB1B2]
29
30      /** Constraints on b2: B2 */
31      // None
32
33      /** Constraints on transfer from B2 output to B output */
34      bijectionFiltered[Source, transferB2B, b2]
35      functionFiltered[Target, transferB2B, this]
36      subsettingItemRuleForSources[transferB2B]
37      subsettingItemRuleForTargets[transferB2B]
38      isAfterSource[transferB2B]}
```

Listing 19. Alloy translation of occurrence class B in Figure 37

Listing 20 shows definitions of the predicates introduced for the translation in Listing 19. The first two, `isAfterSource` and `isBeforeTarget`, ensure the transfers they are applied to are in the second or first position of `HappensBefore` tuples, respectively. The third, `subsettingItemRuleForSources`, requires transfers that are (values of) steps of their source to carry (have items) that are among the inputs of their source, otherwise the items must be among the outputs of their source (typically an occurrence of a step in the same behavior as the transfer step). The last, `subsettingItemRuleForTargets`, requires transfers that are (values of) steps of their target to carry (have items) that are among the outputs of their target, otherwise the items must be among the inputs of their target (typically an occurrence of a step in the same behavior as the transfer step).

```
pred isAfterSource [transfers: set Transfer]{all t: transfers
    | t.Source -> t in HappensBefore}

pred isBeforeTarget [transfers: set Transfer]{all t: transfers
    | t -> t.Target in HappensBefore}

pred subsettingItemRuleForSources [transfers: set Transfer]{
all t: transfers | all src: t.Source | t in src.Step =>
    t.Item in src.Input else t.Item in src.Output}

pred subsettingItemRuleForTargets [transfers: set Transfer]{
all t: transfers | all tgt: t.Target | t in tgt.Step =>
    t.Item in tgt.Output else t.Item in tgt.Input}
```

Listing 20. Alloy code for predicates regarding Transfer

Listing 21 shows the Alloy translation of occurrence classes A and C in Figure 37. Steps, inputs, and outputs translate in the same way as above.

```
sig A extends Occurrence{vout: set Real}{
    no Input
    vout in Output and Output in vout
    #vout = 1}

sig C extends Occurrence{vin: set Real}{
    vin in Input and Input in vin
    #vin = 1
    no Output}
```

Listing 21. Alloy translation of occurrence classes A and C in Figure 37

Listing 22 shows the Alloy translation of occurrence class ParameterBehavior in Figure 37. Steps, inputs, and outputs translate in the same way as above. The activity object flows being translated are all between pins, rather than between pins and parameter nodes, reflected in the OBM representation as the Transfer connectors being between steps. These translate to filtered bijections, rather than filtered functions, see Transfer connector translations above.

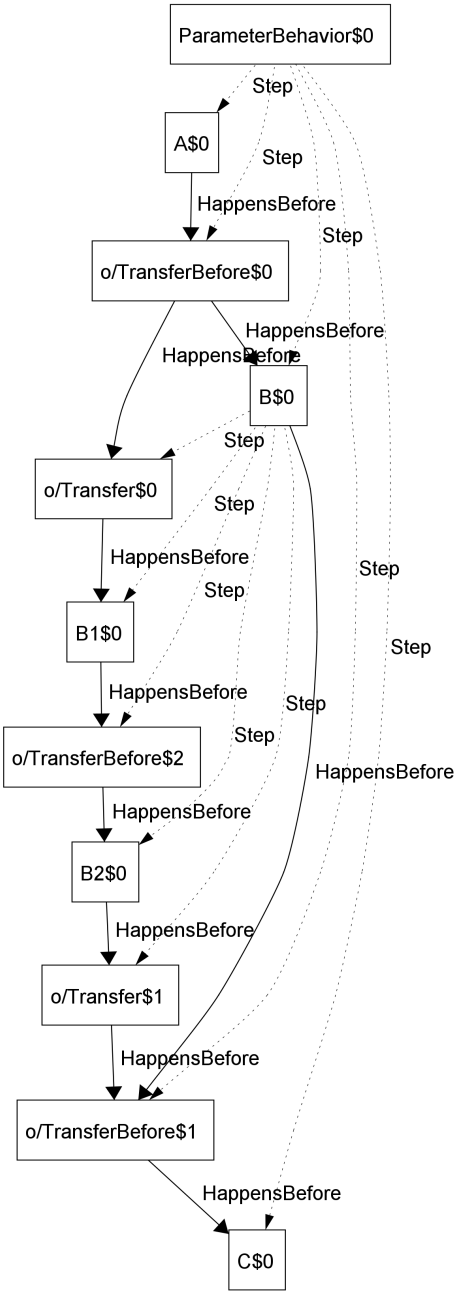
```
sig ParameterBehavior extends Occurrence{
    a: set A,
    b: set B,
    c: set C,
    disj transferAB, transferBC: set TransferBefore
}{
    no Input

    /** Constraints on a: A */
    #a = 1
```

```
11
12      /** Constraints on transfer from output of A to input of B
            */
13      bijectionFiltered[Source, transferAB, a]
14      bijectionFiltered[Target, transferAB, b]
15      subsettingItemRuleForSources[transferAB]
16      subsettingItemRuleForTargets[transferAB]
17      isAfterSource[transferB1B2]
18      isBeforeTarget[transferB1B2]
19
20      /** Constraints on b: B */
21      // None
22
23      /** Constraints on transfer from output of B to input of C
            */
24      bijectionFiltered[Source, transferBC, b]
25      bijectionFiltered[Target, transferBC, c]
26      subsettingItemRuleForSources[transferBC]
27      subsettingItemRuleForTargets[transferBC]
28      isAfterSource[transferBC]
29      isBeforeTarget[transferBC]
30
31      /** Constraints on c: C */
32      // None
33
34      no Output}
```

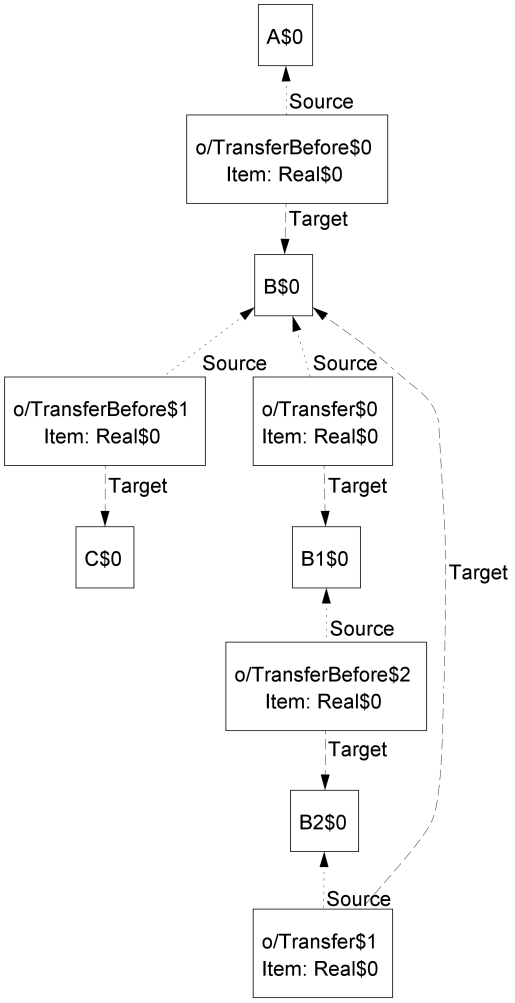
Listing 22. Alloy translation of occurrence class ParameterBehavior in Figure 37

Figure 39a shows a solution to Listings 18 through 22 highlighting `Step` and `HappensBefore` tuples. The figure shows atoms of `A`, `B`, `C`, which are all steps of `ParameterBehavior`, as are two atoms of `TransferBefore`. The remaining atoms are in steps of `B`. Because step `a` happens before step `b`, the `A` atom also happens before those in the steps of `B`. Similarly, because `b` happens before `c`, all the atoms in `B` steps happen before the one in `c`.

Figure 39b shows the same solution as Figure 39a, but highlights tuples with `Transfer` atoms as their first element, including `TransferBefore` tuples. The figure shows that every transfer has as a source, target, and item, the latter shown in attribute notation, rather than with graph edges. The solution confirms that the item is the same atom of `Real` for all the transfers. The transfers carry the item from (atoms of) `A`, to `B`, to `C`. Internal to the `B` atom, the transfers carry the item from the `B` atom, through the atoms of its steps, and back to the `B` atom.



(a) View of solution showing steps



(b) View of same solution, showing transfers

Fig. 39. Views of a solution to Listings 18 through 22

Figure 40 shows a third view of the same solution, highlighting `HappensBefore` tuples as graph edges, and inputs and outputs as attributes. As expected, the same atom of `Real` is the input and output of every step, confirming that the transfers passed it correctly between all of them.

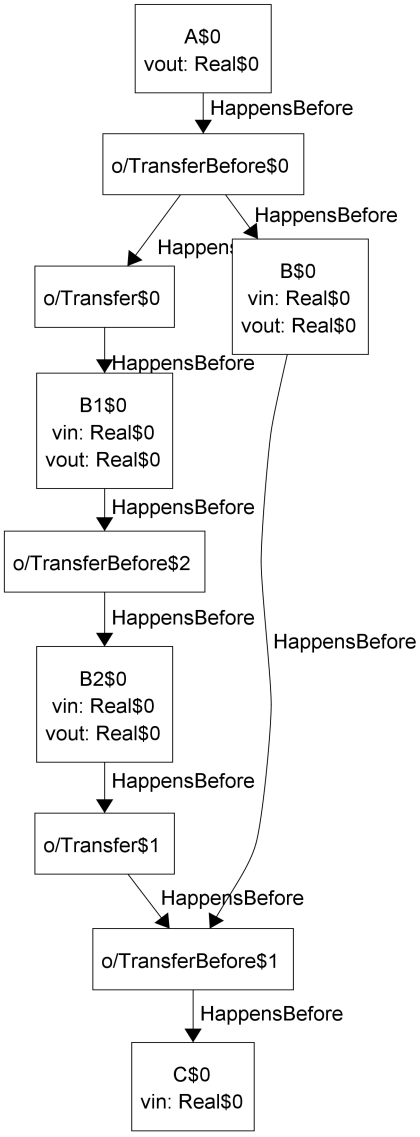


Fig. 40. Another view of the solution to Listings 18 through 22, showing input and output tuples from non-transfer steps

4.1.5 Steps with multiple executions

The examples in this section have steps that happen multiple times in each occurrence of their behavior, modeled only with step multiplicities greater than 1, rather than including loops, see Section 4.1.2. This means multiple occurrences of the same step are not required to happen in order, as they are in loops. The examples in this section use control flows, then object flows, between the same actions ordered in the same way.

Control flows Figures 41 and 42 show SysML activity and OBM representations respectively of a behavior with a fork and a join in a series of control flows. The intention is for the first action to happen twice, which is not expressible in SysML currently, but is specified in the OBM representation as the multiplicity of the first step (compare to multiplicity 1 for these in the examples in Section 4.1.1). The other aspects of the OBM representation are similar to the fork and join examples in Section 4.1.1, Figures 11 and 14.

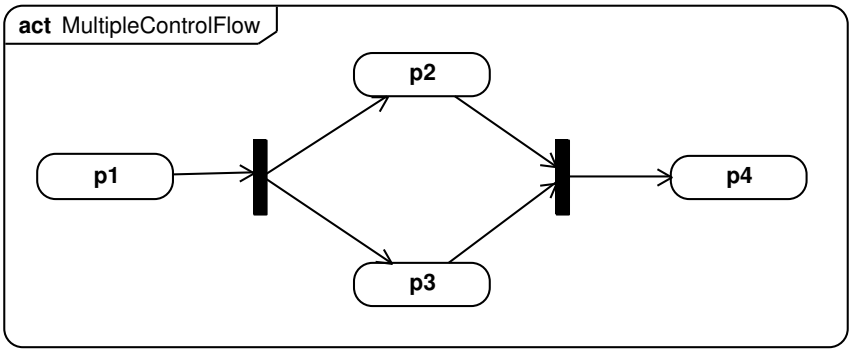


Fig. 41. Behavior with actions intended to be executed multiple times (Activity)

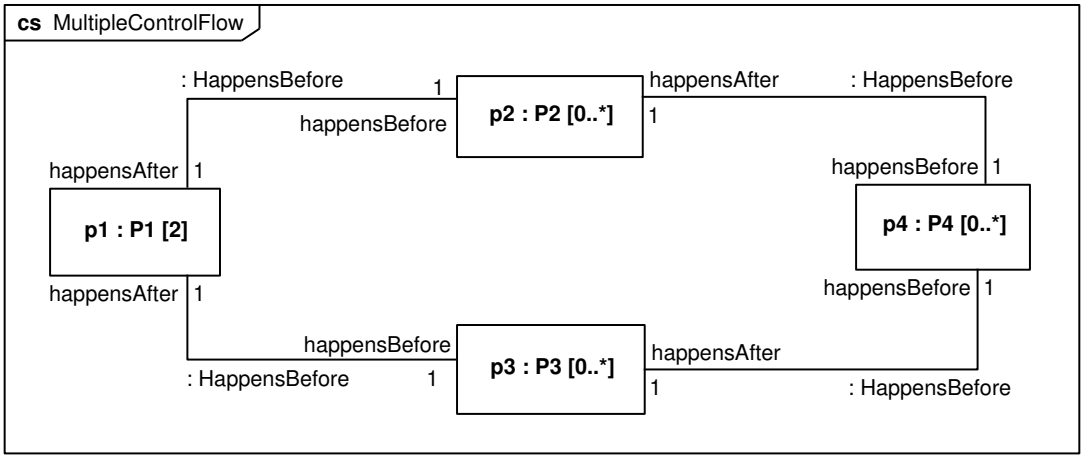


Fig. 42. Behavior with steps that identify multiple occurrences (OBM)

Listing 23 shows the Alloy translation of Figure 42. The number of values of `p1` is 2, but in other ways is similar to the translations of the fork and join examples in Section 4.1.1, Listings 9 and 10.

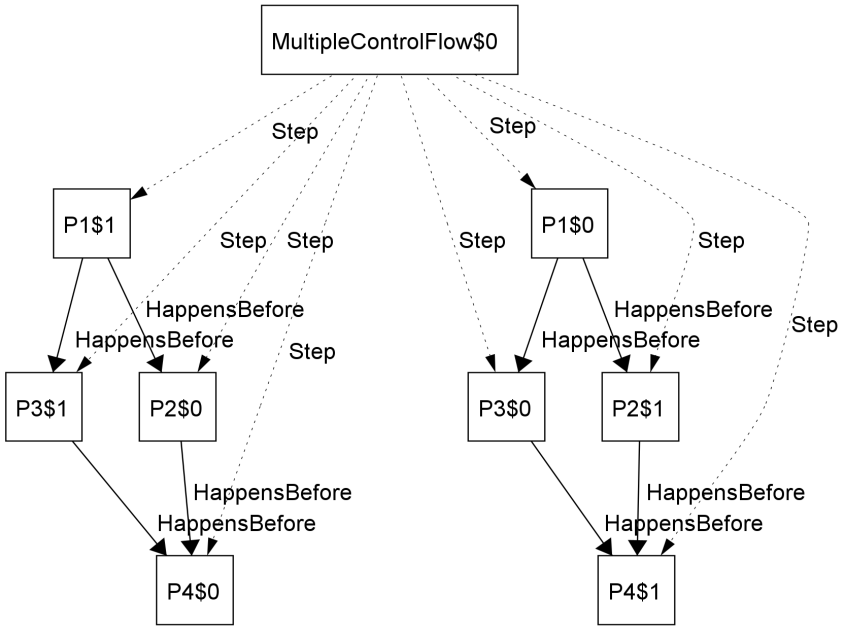
```
1  sig P1, P2, P3, P4 extends Occurrence{}
2
3  sig MultipleControlFlow extends Occurrence {
4      p1: set P1, p2: set P2,
5      p3: set P3, p4: set P4
6  }{
7      #p1 = 2
8
9      // Fork from p1 to p2 and p3
10     bijectionFiltered[HappensBefore, p1, p2]
11     bijectionFiltered[HappensBefore, p1, p3]
12
13     // Join from p2 and p3 to p4
14     bijectionFiltered[HappensBefore, p2, p4]
15     bijectionFiltered[HappensBefore, p3, p4]}
```

Listing 23. Alloy translation of Figure 42

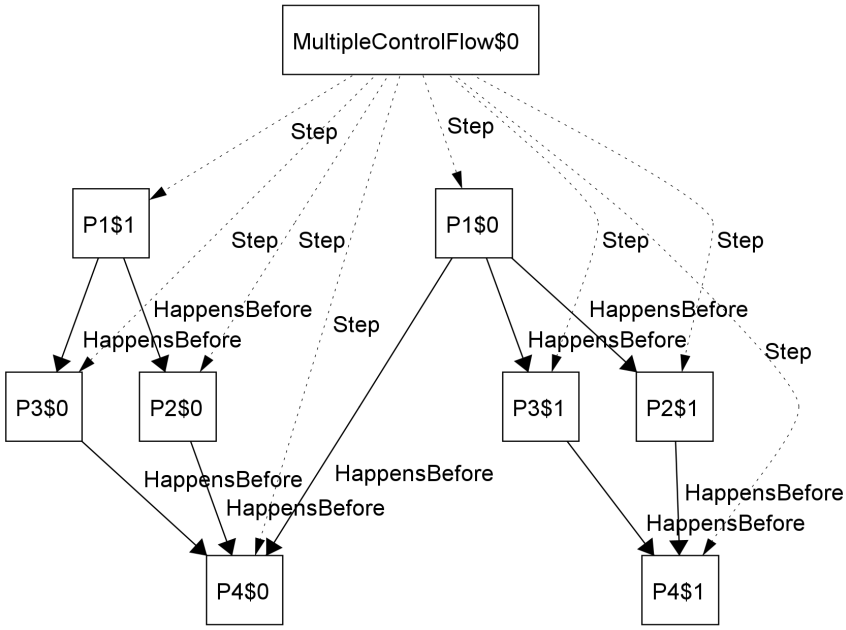
Figure 43 shows three solutions to Listing 23, with all steps happening twice in both (two values each), due to the multiplicity of step `p1` and the fork and join constraints. The solutions highlight that the occurrences happening sometime after the those in step `p1` (`P1$0` and `P1$1`) are not coordinated based on which of those two they happen after:

- Figure 43a has no particular ordering between the occurrences on the left (that happen sometime after `P1$1`) and the occurrences on the right (happening sometime after the `P1$0`. Such time orderings probably exist, or would in a real system, but they cannot be inferred just from Listing 23.
- Figure 43b includes a time ordering between an occurrence on the right, `P1$0`, and one on the left, `P4$0`. However, the ones happening sometime after `P1$0` on right are not in any particular time order with the occurrences on the left.
- Figure 43c shows that the time orderings between occurrences of step `P4` and those of steps `P2` and `P3` are not dependent on which occurrences of `P1` the latter happen after. In this solution, `P4$0` happens after `P3$1`, which happens after `P1$0`, but `P4$0` also happens after P2$0, which happens after `P1$1`, not necessarily `P1$0`. Similarly for `P4$1`.[29]

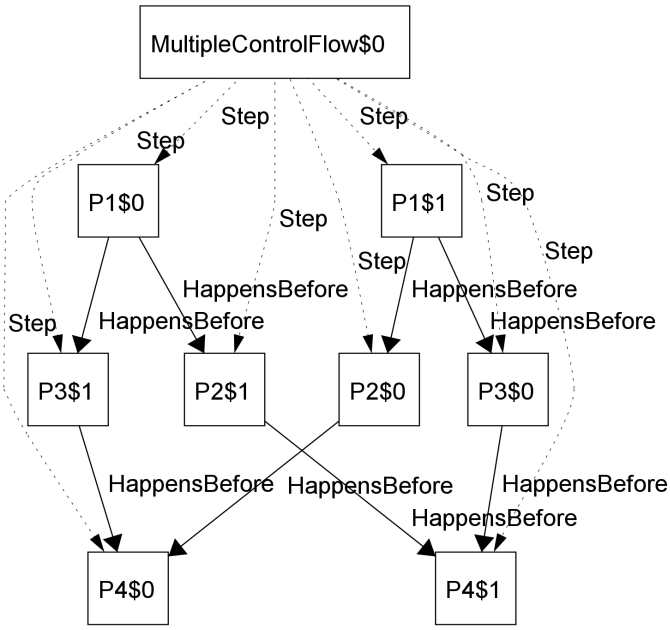
[29] This case could be ruled out with items transferred between steps to keep track of which occurrence of the first step "leads to" to which occurrences in later steps, see the food service with object flow and parallelism in Section 4.2.2



(a) Solution with no time ordering between the two occurrences of step `P1` (`P1$1` and `P1$0`) or those happening sometime after them



(b) Solution with an occurrence of `P1` happening before another that happens sometime after the other occurrence of `P1`



(c) Solution with two occurrences of `P4`, each happening after occurrences that happen after different occurrences of `P1`

Fig. 43. Solutions to Listing 23

Object flows Figures 44 and 45 show SysML activity and OBM representations, respectively, of a behavior similar to Figure 41, but with object flows between output and input pins, rather than control flows between actions. Forks and joins have a similar effect on object flows as they do on control flows. The things flowing are duplicated by forks and reunited by joins.

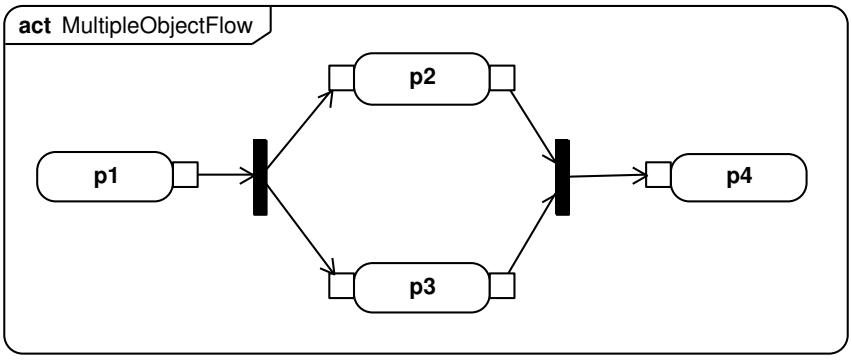


Fig. 44. Behavior with actions and object flows intended to be executed multiple times (Activity)

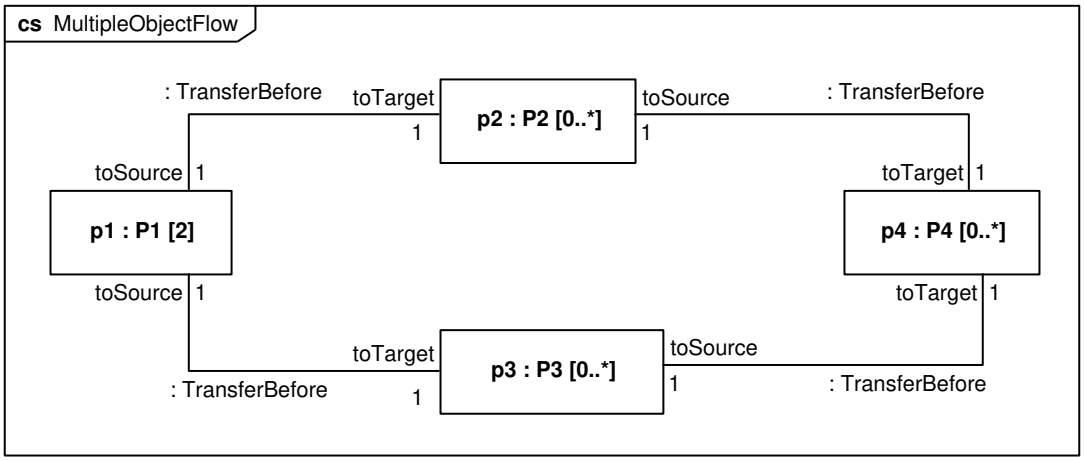


Fig. 45. Behavior with steps and transfer connectors that identify multiple occurrences (OBM)

Listing 24 shows the Alloy translation of Figure 45. The predicate `eachOccInputsEqualOutputs` is introduced for this translation to specify that the `Input` of a (occurrence in a) step is equal to its `Output`, `p2` and `p3` in this example (lines 28 and 33), see Listing 25. The rest of the object flow translation is similar to Listing 19 under Object flows and behavior parameters in Section 4.1.4. Forks and joins of object flows translate the same as multiple separate object flows, similar to the translation of forks and joins of control flows, see Forks and Joins in Section 4.1.1.

```
1  sig P1, P2, P3, P4, Real extends Occurrence{}
2
3  sig MultipleObjectFlow extends Occurrence {
4      p1: set P1, p2: set P2,
5        p3: set P3, p4: set P4,
6      disj transferP1P2, transferP1P3, transferP2P4,
          transferP3P4: set TransferBefore}
7    { no Input
8
9      /** Constraints on p1: P1 */
10     #p1 = 2
11     p1.@Output in Real
12
13     // Two transfers specified to act as a fork
14     /** Constraints on transfer from p1 to p2 */
15     bijectionFiltered[Source, transferP1P2, p1]
16     bijectionFiltered[Target, transferP1P2, p2]
17     transferP1P2.Item in Real
18     subsettingItemRuleForSources[transferP1P2]
19     subsettingItemRuleForTargets[transferP1P2]
```

```
20      /** Constraints on transfer from p1 to p3 */
21      bijectionFiltered[Source, transferP1P3, p1]
22      bijectionFiltered[Target, transferP1P3, p3]
23      transferP1P3.Item in Real
24      subsettingItemRuleForSources[transferP1P3]
25      subsettingItemRuleForTargets[transferP1P3]
26
27      /** Constraints on p2: P2 */
28      eachOccInputsEqualOutputs[p2]
29      p2.@Input in Real
30      p2.@Output in Real
31
32      /** Constraints on p3: P3 */
33      eachOccInputsEqualOutputs[p3]
34      p3.@Input in Real
35      p3.@Output in Real
36
37      // Two transfers specified to act as a join
38      /** Constraints on transfer from p2 to p4 */
39      bijectionFiltered[Source, transferP2P4, p2]
40      bijectionFiltered[Target, transferP2P4, p4]
41      transferP2P4.Item in Real
42      subsettingItemRuleForSources[transferP2P4]
43      subsettingItemRuleForTargets[transferP2P4]
44      /** Constraints on transfer from p3 to p4 */
45      bijectionFiltered[Source, transferP3P4, p3]
46      bijectionFiltered[Target, transferP3P4, p4]
47      transferP3P4.Item in Real
48      subsettingItemRuleForSources[transferP3P4]
49      subsettingItemRuleForTargets[transferP3P4]
50
51      /** Constraints on p4: P4 */
52      p4.@Input in Real
53
54      no Output}
```

Listing 24. Alloy translation of Figure 45

Listing 25 shows the definition of `eachOccInputsEqualOutputs` predicate introduced for this translation. It requires the (values of) `Inputs` and `Outputs` to be the same for each occurrence (separately) in the set it is applied to.

```
1 pred eachOccInputsEqualOutputs [occSet: set Occurrence] {
2 all x: occSet | x.Input = x.Output}
```

Listing 25. Alloy code for the eachOccInputsEqualOutputs predicate

Figure 46 shows a view of a solution to Listing 24, highlighting `Step` and `Happens Before` tuples reflecting the composition and temporal ordering of steps. `Input`, `Output`, and `Item` tuples appear as attributes of their respective step type and connector atoms. The figure shows that `Real$0` and `Real$1` are passed along by `Transfers` (as their `Items`) between separate atoms of `P1`, `P2`, `P3`, and `P4` on each side of the figure. The constraints on the flow of these two atoms keep the lines of execution from mixing, as in the previous example.

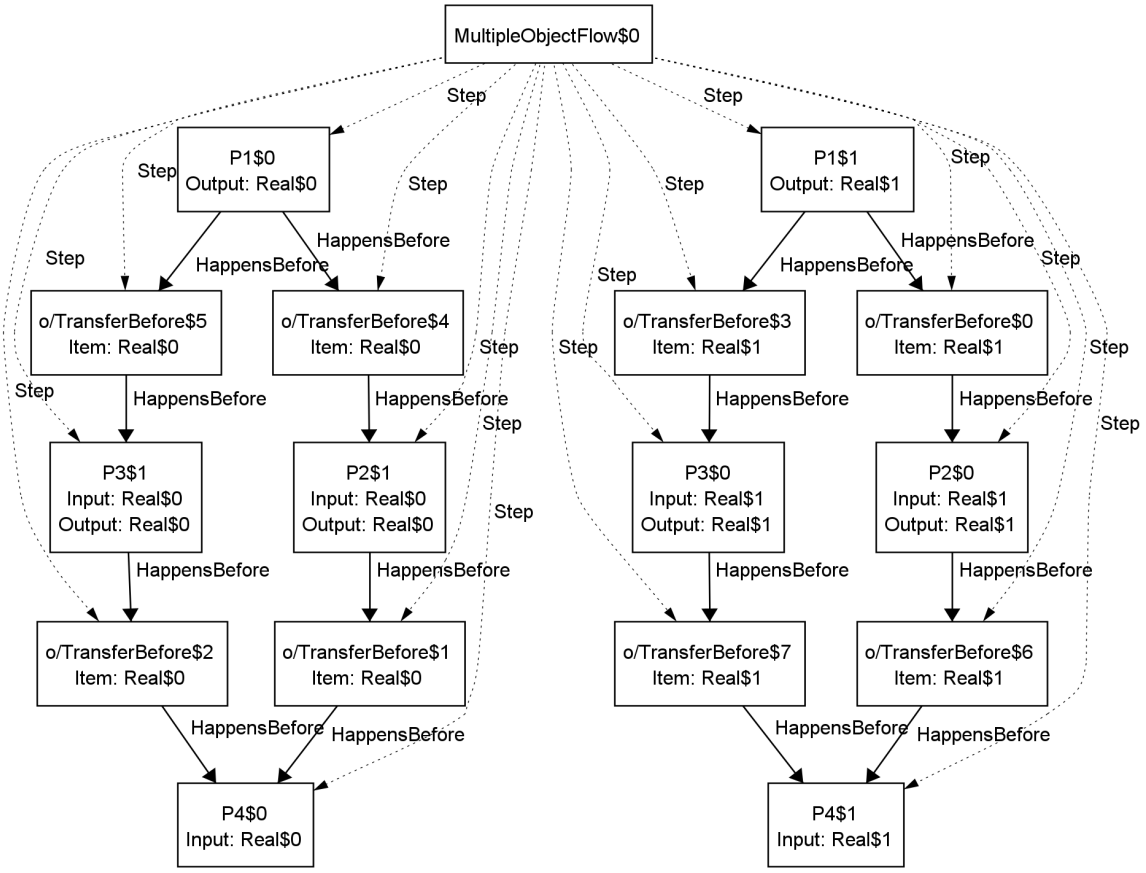


Fig. 46. Summary view of a solution to Listing 24

Figure 47 shows another view of the same solution, highlighting `Source` and `Target`, and `Item` tuples pairing `TransferBefore` atoms with `Occurrence` atoms. Because the specification calls for `TransferBefore` tuples of `Occurrences`, the temporal and object flows act as a forks and joins.

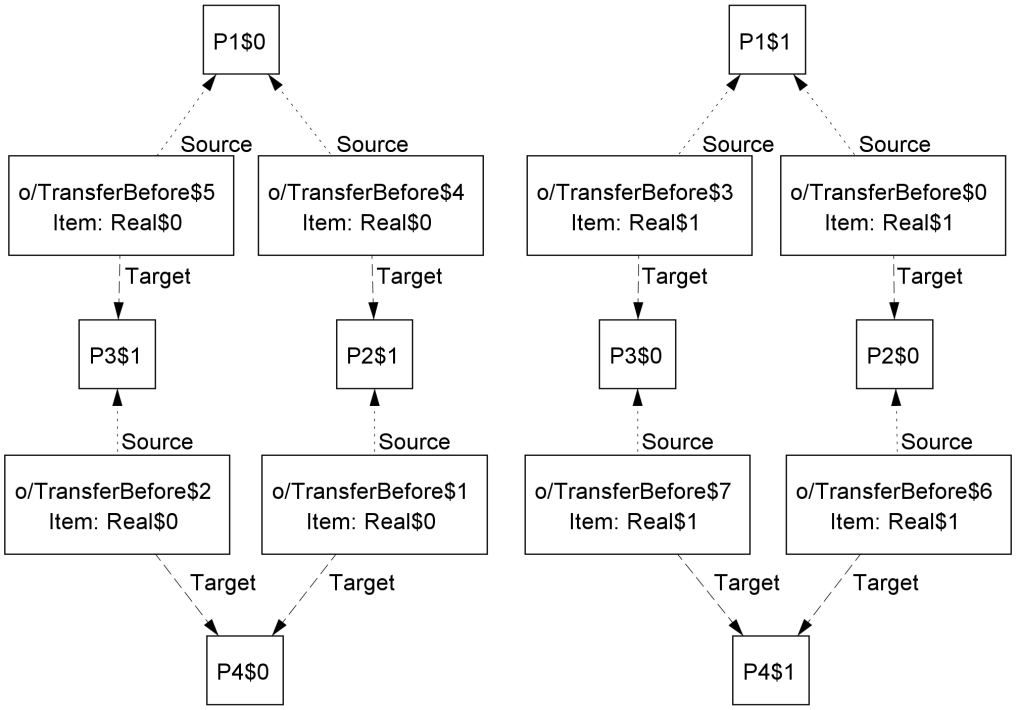


Fig. 47. Another view of the same solution to Listing 24, showing transfer sources and targets

4.1.6 Unsatisfiable behaviors

The section presents some unsatisfiable SysML behavior models, due to conflicting constraints implied by their elements. Alloy could not find solutions to these, even when searching all possibilities that include many more atoms than should be needed.[30]

Connector end multiplicities Figures 48 and 49 show SysML activity and OBM representations, respectively, of a behavior with a decision followed by a join, which is unsatisfiable due to the multiplicities on connector ends. The OBM representation combines those of decision and join in Section 4.1.1, Figures 17 and 14. The intention is for either step `p2` or `p3` to be taken, but not both (assuming the missing decision constraint), and for `p4` to be taken only after `p2` or `p3`. This is not possible because the decision prevents `p2` and `p3` from both having values to complete the join.

[30] Alloy searches for possible solutions up to a maximum number of atoms specified by the user (*scope*), which does not prove unsatisfiability (compare to the solver used in [6]). However, the scope in this section's examples was set to three times larger than should be needed to solve them, giving some confidence that they are provably unsatisfiable. The examples also include solutions when some constraints are removed to further reinforcing this.

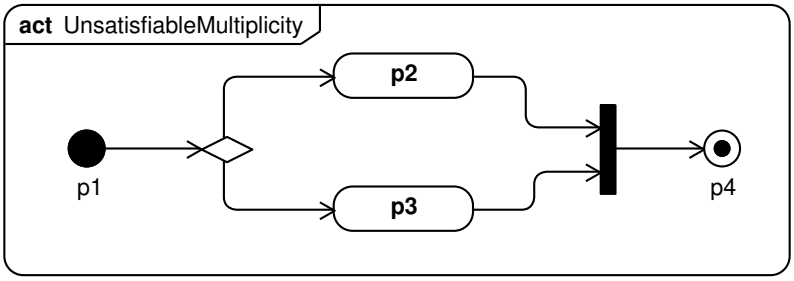


Fig. 48. Unsatisfiable model, connector end multiplicities (Activity)

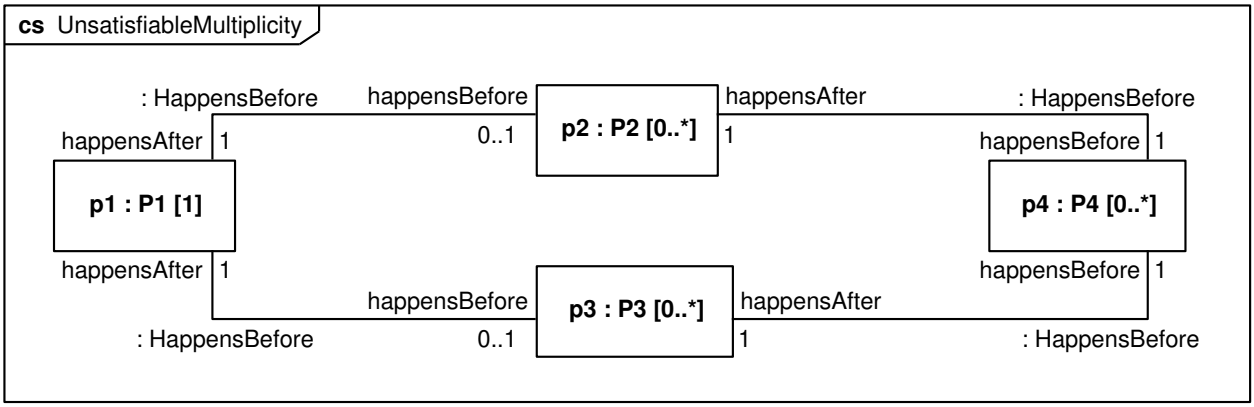


Fig. 49. Unsatisfiable model, connector end multiplicities (OBM)

Listing 26 shows the Alloy translation of Figure 49, combining the translations of decision and join in Section 4.1.1, Listings 11 and 10. Alloy does not find a solution among possibilities up to many more atoms than should be needed.[30]

```
1  sig P1, P2, P3, P4 extends Occurrence{}
2
3  sig UnsatisfiableMultiplicity extends Occurrence {
4      p1: set P1,
5      p2: set P2,
6      p3: set P3,
7      p4: set P4
8  }{  // Decision from p1 to p2 and p3
9      bijectionFiltered[HappensBefore, p1, p2+p3]
10
11     // Join from p2 and p3 to p4
12     bijectionFiltered[HappensBefore, p2, p4]
13     bijectionFiltered[HappensBefore, p3, p4]
14     #p1 = 1}
```

Listing 26. Alloy translation of Figure 49

Figure 50 shows a solution to Listing 26 when the multiplicity of p1 (line 16) is increased to 2. This step is taken twice, with a decision after one those occurrences (P1$0) leading to p3 being taken, and the decision after the other (P1$1) leading to p2. This satisfies the join (the two HappensBefore connectors to p4 have values) and step p4 is taken, per the connector end multiplicities. This gives some confidence that the unsatisfiability of Listing 26 is due to bijections translated from the 1-1 connector end multiplicities.

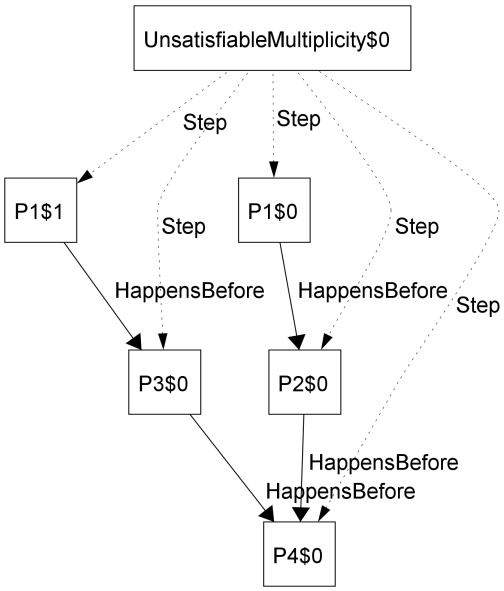


Fig. 50. Solution to Listing 26 with p1 multiplicity = 2

Asymmetry of HappensBefore Figures 51 and 52 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 (line 6 in Listing 4 in Section 3.2). In the OBM representation, UnsatisfiableAsymmetry is an occurrence class with two properties of multiplicity 1 typed by P1 and P2: p1 and p2. They are linked by HappensBefore connectors in a cycle, so they happen before each other, which is not allowed by asymmetry.

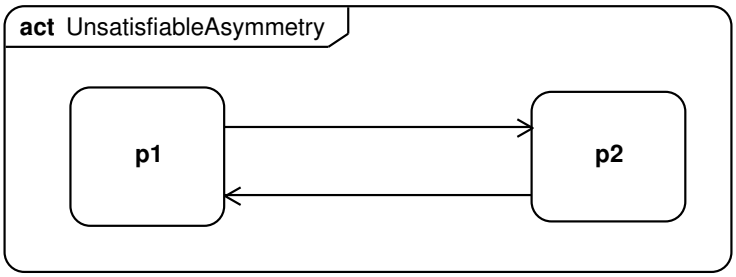


Fig. 51. Unsatisfiable model, asymmetry of HappensBefore (Activity)

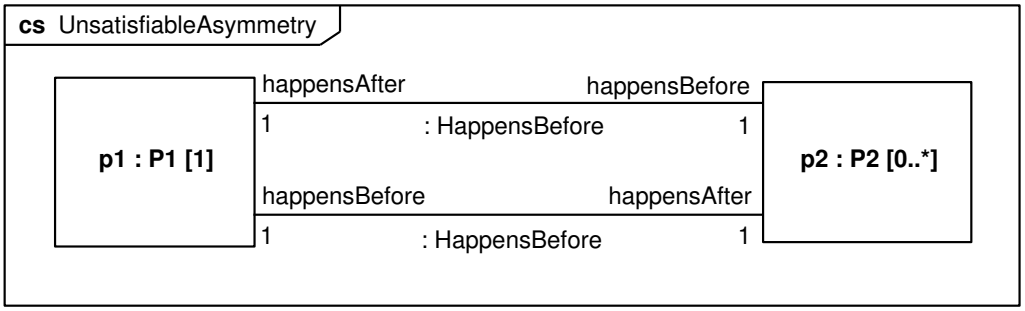


Fig. 52. Unsatisfiable model, asymmetry of HappensBefore (OBM)

Listing 27 shows the Alloy translation of Figure 52. Alloy does not find a solution among possibilities up to many more atoms than should be needed.[30]

```
sig P1, P2 extends Occurrence{}

sig UnsatisfiableAsymmetry extends Occurrence {
    p1: set P1,
    p2: set P2
}{
    #p1 = 1
    #p2 = 1

    bijectionFiltered[HappensBefore, p1, p2]
    bijectionFiltered[HappensBefore, p2, p1]}
```

Listing 27. Alloy translation of Figure 52

Figure 53 shows a solution to Listing 27 after temporarily removing the asymmetry constraint on `HappensBefore`. The bidirectional HappensBefore edges (tuples) in the graph implies that `P1` and `P2` happen before each other, which is normally not possible. This gives some confidence that the unsatisfiability of Listing 27 is attributable to the asymmetry constraint.

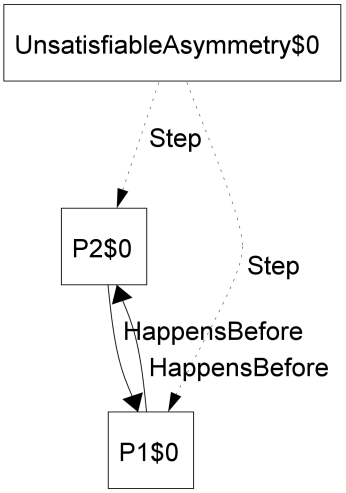


Fig. 53. Solution to Listing 27 without asymmetry of HappensBefore

Transitivity of HappensBefore Figures 54 and 55 show SysML activity and OBM representations, respectively, of a behavior with three occurrences happening before each other in a cycle, which is unsatisfiable due to HappensBefore being transitive and asymmetric (lines 5-6 in Listing 4 in Section 3.2). In the OBM representation, the steps of UnsatisfiableTransitivity are linked by a cycle of HappensBefore connectors, implying by transitivity that they all happen before themselves and each other bidirectionally, which is not allowed by asymmetry.

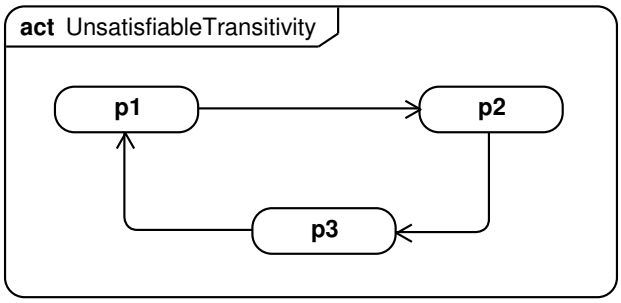


Fig. 54. Unsatisfiable model, transitivity and asymmetry of HappensBefore (Activity)

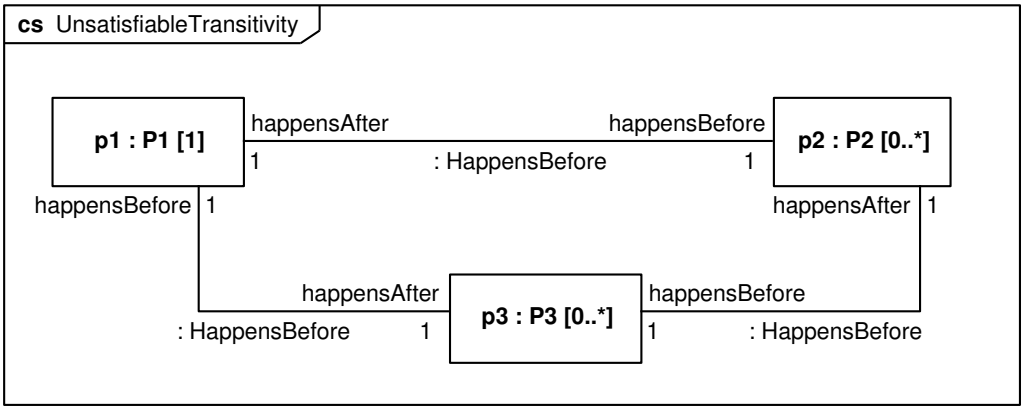


Fig. 55. Unsatisfiable model, transitivity and asymmetry of HappensBefore (OBM)

Listing 28 shows the Alloy translation of Figure 55. Alloy does not find a solution among possibilities up to many more atoms than should be needed.[30]

```
sig P1, P2, P3 extends Occurrence{}

sig UnsatisfiableTransitivity extends Occurrence {
    p1: set P1,
    p2: set P2,
    p3: set P3
}{
    #p1 = 1

    bijectionFiltered[HappensBefore, p1, p2]
    bijectionFiltered[HappensBefore, p2, p3]
    bijectionFiltered[HappensBefore, p3, p1]}
```

Listing 28. Alloy translation of Figure 55

Figure 56 shows a solution to Listing 28 after temporarily removing transitivity from `HappensBefore`. It has a cycle of `HappensBefore` tuples, implying that all the occurrences happen before each other and themselves, which is normally not possible. This gives confidence that the unsatisfiability of Listing 28 is due to the transitivity of `HappensBefore`.

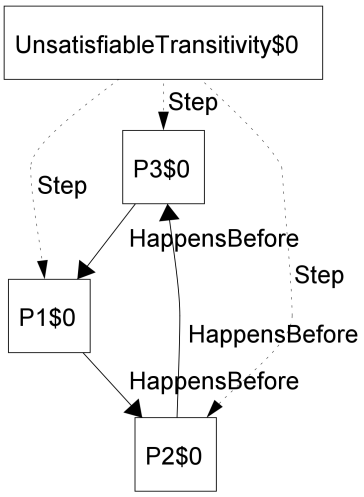


Fig. 56. Solution to Listing 28 without transitivity of `HappensBefore`

Logical interaction of temporal relations (1) Figures 57 and 58 show SysML activity and OBM representations, respectively, of a behavior with three steps linked by temporal connectors, which is unsatisfiable due to the first constraint between HappensBefore and HappensDuring (lines 8-10 in Listing 4 in Section 3.2), and the asymmetry of HappensBefore (line 6). In the OBM representation, UnsatisfiableComposition1 is an occurrence class with three steps, p3 happening during p2 (translated from the SysML structured node) and before p1, which happens before p2. The two temporal relations imply p1 and p3 happen before each other, because steps happening before p2 also happen before its steps. This is not allowed by the asymmetry of HappensBefore.

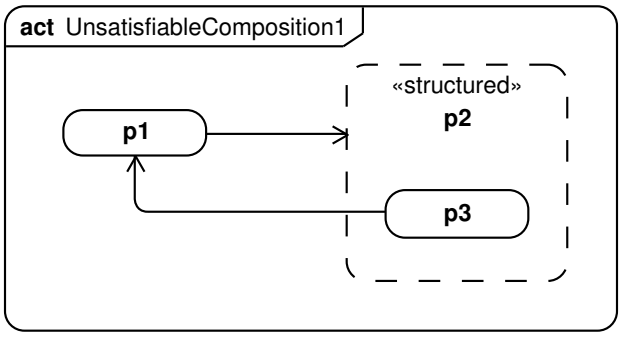


Fig. 57. Unsatisfiable model, logical interaction of HappensBefore and HappensDuring 1 (Activity)

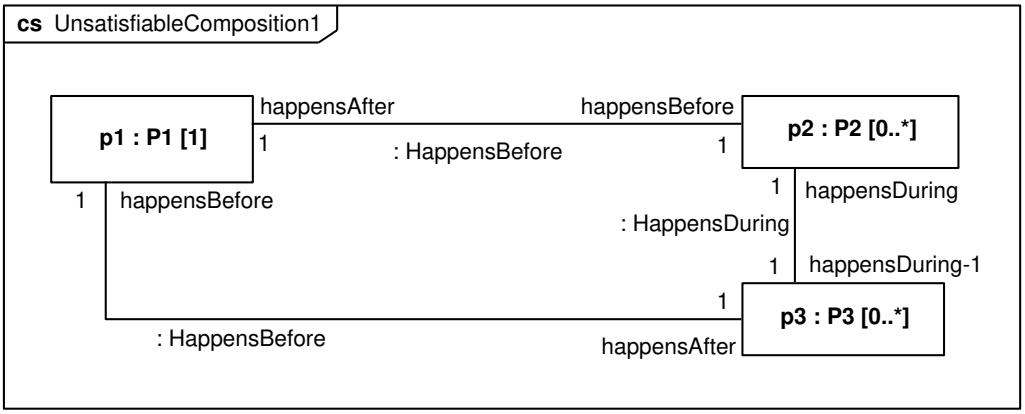


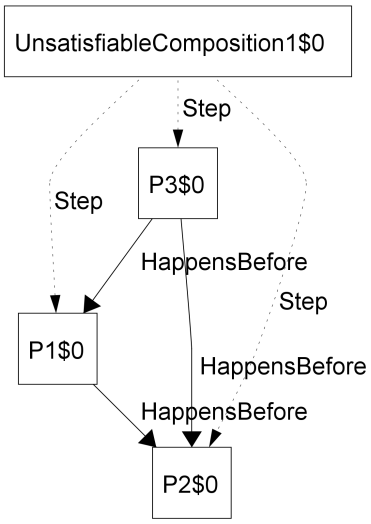
Fig. 58. Unsatisfiable model, logical interaction of HappensBefore and HappensDuring 1 (OBM)

Listing 29 shows the Alloy translation of Figure 58. Alloy does not find a solution among possibilities up to many more atoms than should be needed.[30]

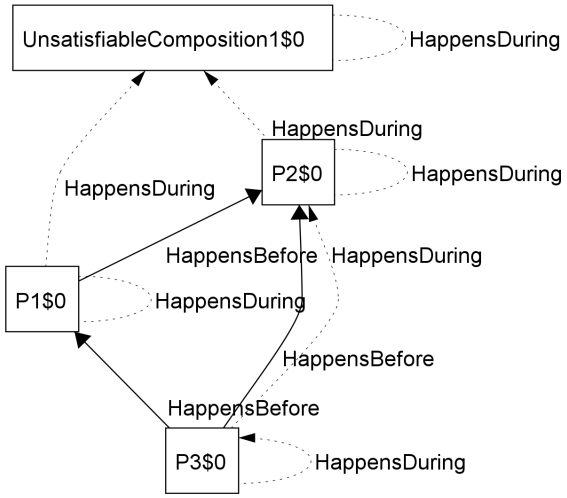
```
1  sig P1, P2, P3 extends Occurrence{}
2
3  sig UnsatisfiableComposition1 extends Occurrence {
4      p1: set P1,
5      p2: set P2,
6      p3: set P3
7  }{
8      #p1 = 1
9
10     bijectionFiltered[HappensBefore, p1, p2]
11     bijectionFiltered[HappensDuring, p3, p2]
12     bijectionFiltered[HappensBefore, p3, p1]}
```

Listing 29. Alloy translation of Figure 58

Figure 59 shows views of the same solution to Listing 29 after temporarily removing the first constraint between `HappensBefore` and `HappensDuring`. Figure 59a highlights `Step` and `HappensBefore` tuples. The `HappensBefore` cycle between `P3$0` and `P1$0` is only possible with the asymmetry fact removed. Figure 59b highlights both temporal tuples and shows why the constraints on interaction of `HappensBefore` and `HappensDuring` are needed. The `P3$0` occurrence happens during and before `P1$0`, which is normally not possible. This gives some confidence that the unsatisfiability of Listing 29 is due to the logical interaction of the temporal relations and asymmetry of `HappensBefore`.



(a) Solution showing steps and `HappensBefore` tuples



(b) Solution showing `HappensBefore` and `HappensDuring` tuples

Fig. 59. Solutions to Listing 29 without the first constraint between `HappensBefore` and `HappensDuring` in Listing 4, Section 3.2

Logical interaction of temporal relations (2) Figures 60 and 61 show SysML activity and OBM representations, respectively, of a behavior with three steps linked by temporal connectors, which is unsatisfiable due to the second constraint between HappensBefore and HappensDuring (lines 12-14 in Listing 4 in Section 3.2), and the asymmetry of HappensBefore (line 6). In the OBM representation, UnsatisfiableComposition2 is an occurrence class with three steps, p3 happening during p1 (translated from the SysML structured node) and after p2, which happens after p1. The two temporal relations combined imply p3 and p2 happen after each other, because steps happening after p1 also happen after its steps. This is not allowed by the asymmetry of HappensBefore.

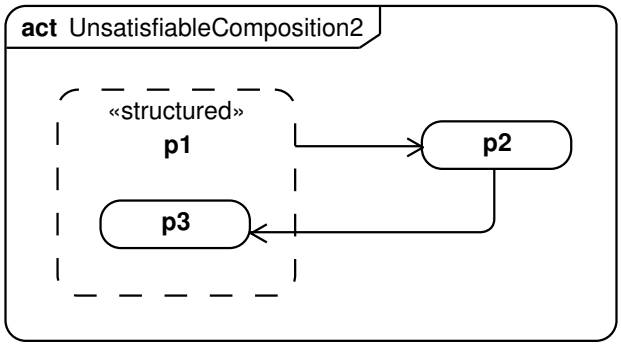


Fig. 60. Unsatisfiable model, logical interaction of HappensBefore and HappensDuring 2 (Activity)

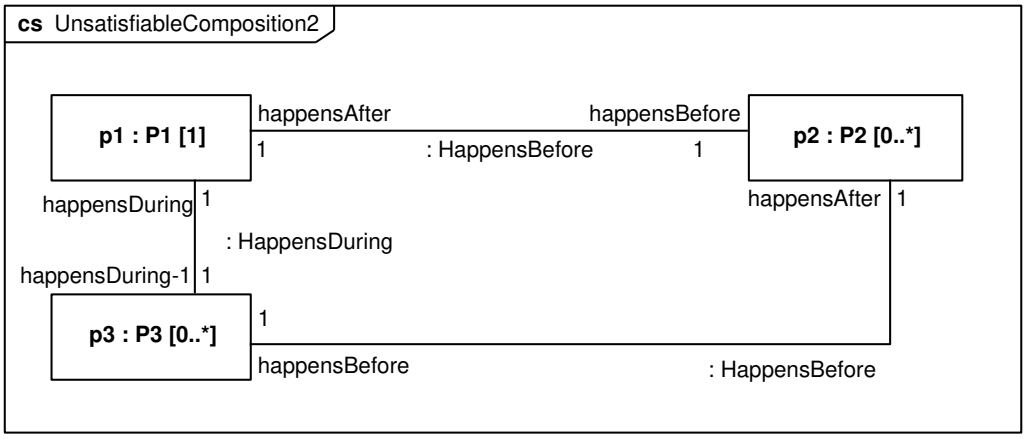


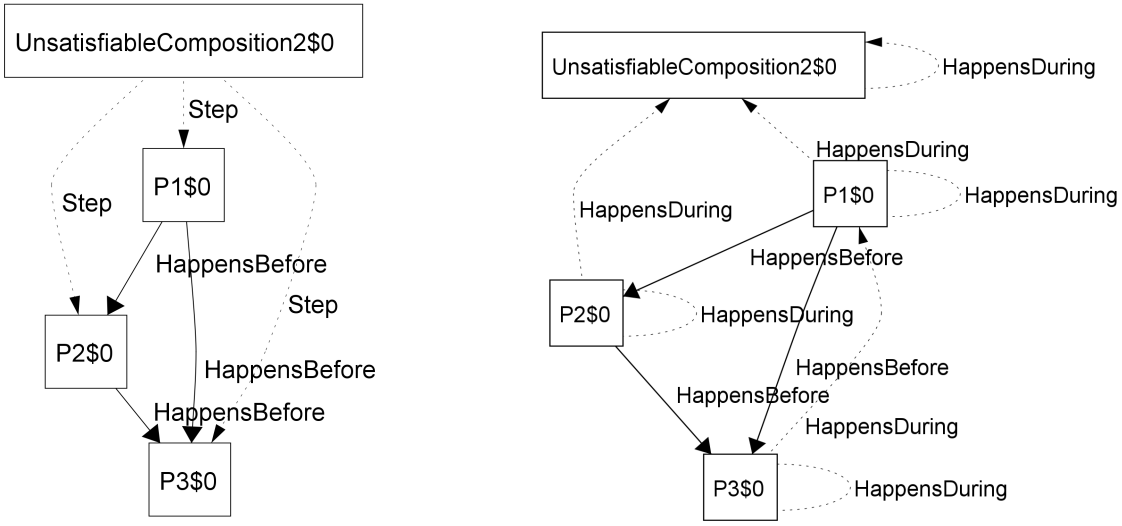
Fig. 61. Unsatisfiable model, logical interaction of HappensBefore and HappensDuring 2 (OBM)

Listing 30 shows the Alloy translation of Figure 61. Alloy does not find a solution among possibilities up to many more atoms than should be needed.[30]

```
sig P1, P2, P3 extends Occurrence{}

sig UnsatisfiableComposition2 extends Occurrence {
    p1: set P1,
    p2: set P2,
    p3: set P3
}{
    #p1 = 1

    bijectionFiltered[HappensBefore, p1, p2]
    bijectionFiltered[HappensDuring, p3, p1]
    bijectionFiltered[HappensBefore, p2, p3]}
```

Listing 30. Alloy translation of Figure 61

Figure 62 shows views of the same solution to Listing 30 after temporarily removing the second constraint between `HappensBefore` and `HappensDuring`. Figure 62a highlights `Step` and `HappensBefore` tuples. The `HappensBefore` cycle between `P1$0` and `P2$0` is only possible with the asymmetry fact removed. Figure 62b highlights both temporal tuples and shows why the constraints on interaction of `HappensBefore` and `HappensDuring` are needed. The `P3$3` occurrence happens during and before `P1$1`, which is normally not possible. This gives some confidence that the unsatisfiability of Listing 30 is due to the logical interaction of the temporal relations and asymmetry of `HappensBefore`.



(a) Showing steps and `HappensBefore` tuples

(b) Showing steps and both temporal tuples

Fig. 62. Solutions to Listing 30 without the second constraint between `HappensBefore` and `HappensDuring` in Listing 4, Section 3.2

4.2 Advanced examples

This section shows examples adapted from [14], illustrating how to model taxonomies of behaviors using OBM. Section 4.2.1 shows examples with control flows between actions and a single token (occurrence) per action, while Section 4.2.2 shows examples with object flows between pins and multiple tokens (occurrences) per action.

4.2.1 Control flow examples

Food services Figure 63 shows a definition of food service occurrence class FoodService, intended to be specialized into various kinds of food services. It has five steps (properties typed by occurrence classes): prepare, order, serve, eat, and pay, all with multiplicity 0..∗, typed byOrder, Prepare, Serve, Eat, and Pay, respectively.

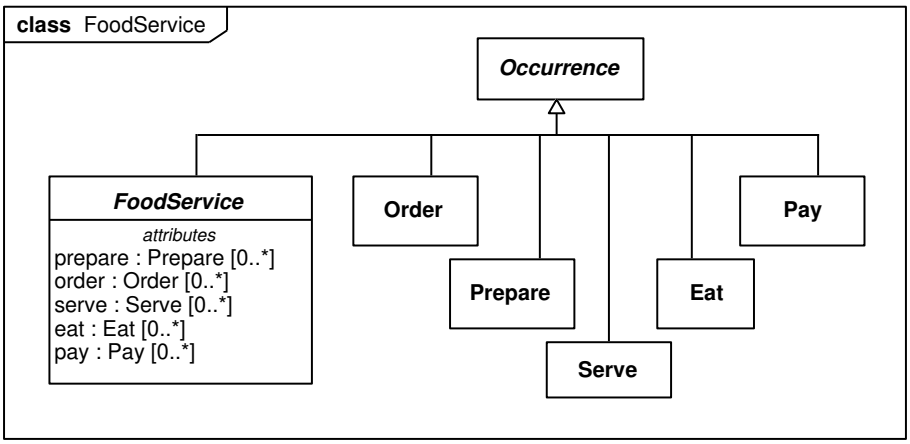


Fig. 63. Food service occurrence classes

Figures 64 and 65 show SysML activity and OBM representations of FoodService time orderings, respectively. In the OBM representation, these are HappensBefore connectors for all food services: prepare and order must happen before serve, serve must happen before eat. Nothing is specified about when pay happens, except that it is during FoodService. The connectors will be inherited and possibly redefined by specializations of FoodService. Multiplicites are unconstrained, for specializations to restrict as needed.

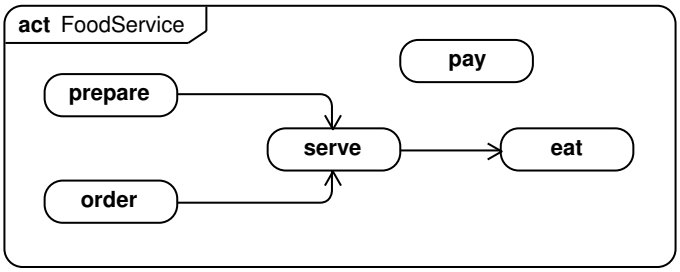


Fig. 64. Food service behavior (Activity)

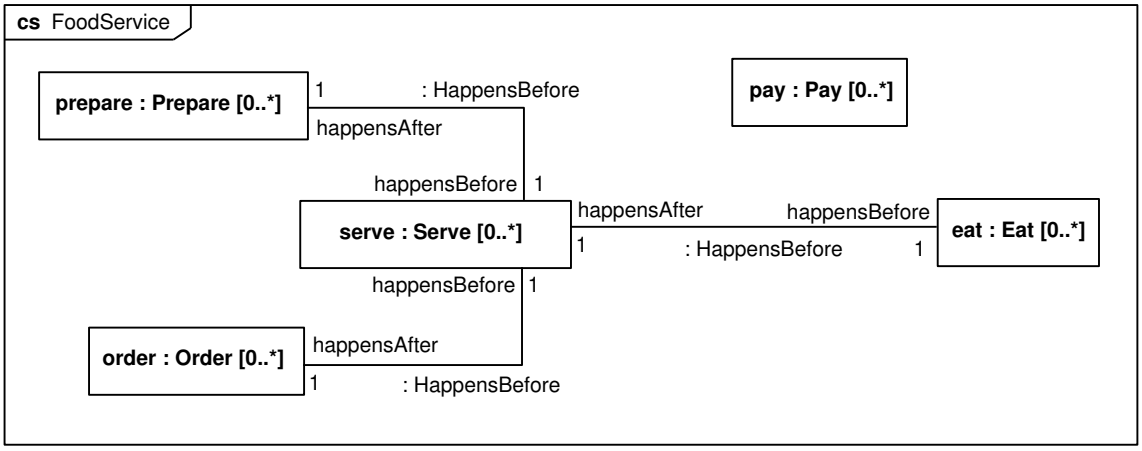


Fig. 65. Food service behavior (OBM)

Listing 31 shows the Alloy translation of Figure 65. It translates the connectors similarly to Listing 7 in the simple sequence example in Section 4.1.1. The other examples in this section specialize `FoodService`.

```
1  sig Order, Prepare, Serve, Eat, Pay extends Occurrence {}
2
3  sig FoodService extends Occurrence {
4      order: set Order,
5      prepare: set Prepare,
6      pay: set Pay,
7      eat: set Eat,
8      serve: set Serve
9  }{
10     bijectionFiltered[HappensBefore, order, serve]
11     bijectionFiltered[HappensBefore, prepare, serve]
12     bijectionFiltered[HappensBefore, serve, eat]}
```

Listing 31. Alloy translation of Figure 65

Single food services Figure 66 introduces a specialized FoodService that redefines its steps to have a multiplicity of exactly one (happen exactly once), SingleFoodService. This occurrence class is specialized further to add more time ordering between the steps in various ways: BuffetService, ChurchSupperService, FastFoodService, and RestaurantService. The subsections after this describe these specializations.

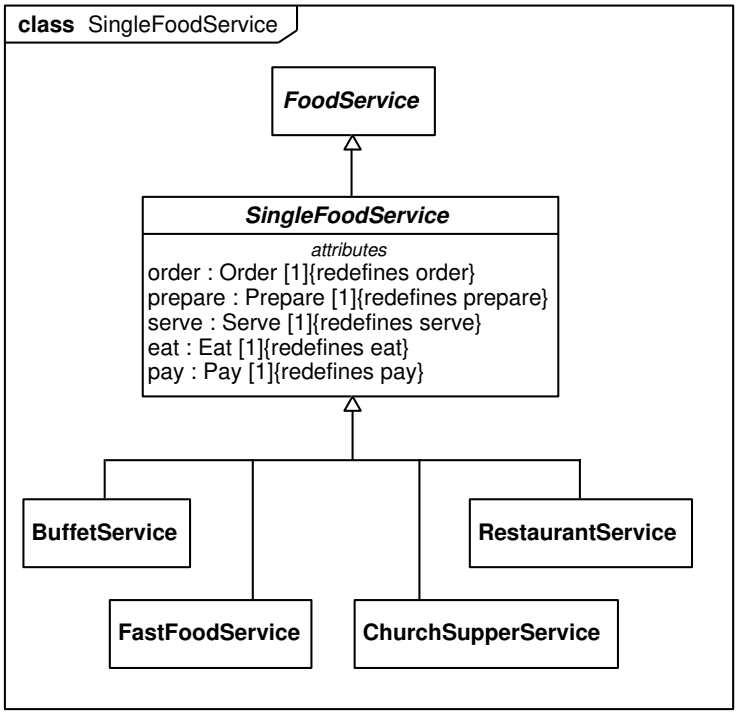


Fig. 66. Single food service occurrence classes

Listing 32 shows the Alloy translation of Figure 66, which extends `FoodService` and narrows the multiplicity of each step to `1`.

```
1 sig SingleFoodService extends FoodService{}{
2     #order = 1
3     #prepare = 1
4     #pay = 1
5     #eat = 1
6     #serve = 1}
```

Listing 32. Alloy translation of Figure 66

Figure 67 shows a solution to Listing 32 highlighting `Step` and `HappensBefore` tuples. The solution satisfies the constraints of `SingleFoodService` with each step happening exactly once, and satisfies `FoodService` with `Order` and `Prepare` atoms happening before a `Serve` atom, which happens before an `Eat` atom.

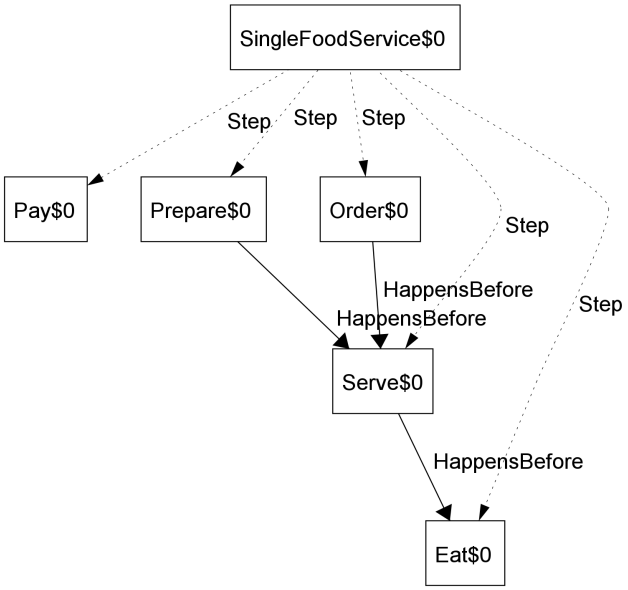


Fig. 67. Solution to Listing 32

Buffet Figures 68 and 69 show SysML activity and OBM representations of a buffet food service, respectively. In the OBM representation, BuffetService introduces HappensBefore connectors from prepare to order, and eat to pay, adding to the ones inherited from FoodService via SingleFoodService.

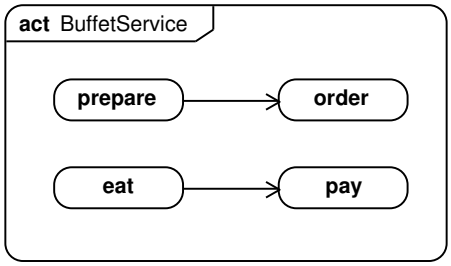


Fig. 68. Buffet service (Activity)

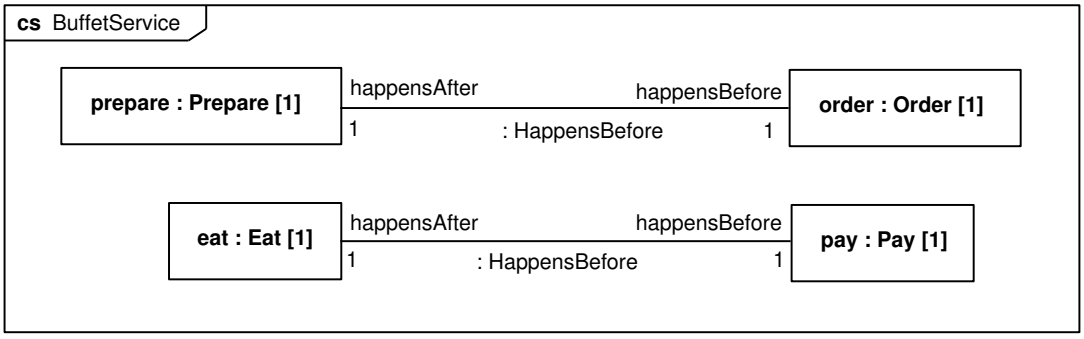


Fig. 69. Buffet service (OBM)

Listing 33 shows the Alloy translation of Figure 69.

```
1 sig BuffetService extends SingleFoodService{}{
2     bijectionFiltered[HappensBefore, prepare, order]
3     bijectionFiltered[HappensBefore, eat, pay]}
```

Listing 33. Alloy translation of Figure 69

Figure 70 shows a solution to Listing 33 highlighting `Step` and `HappensBefore` tuples. A `Prepare` atom happens before an `Order` atom, and an `Eat` atom happens before a `Pay` atom, as specified by `BuffetService`. The `Order` and `Prepare` atoms happen before a `Serve` atom, which happens before the `Eat` atom, as required by `FoodService`, which applies to `BuffetService` as its specialization via `SingleFoodService`.

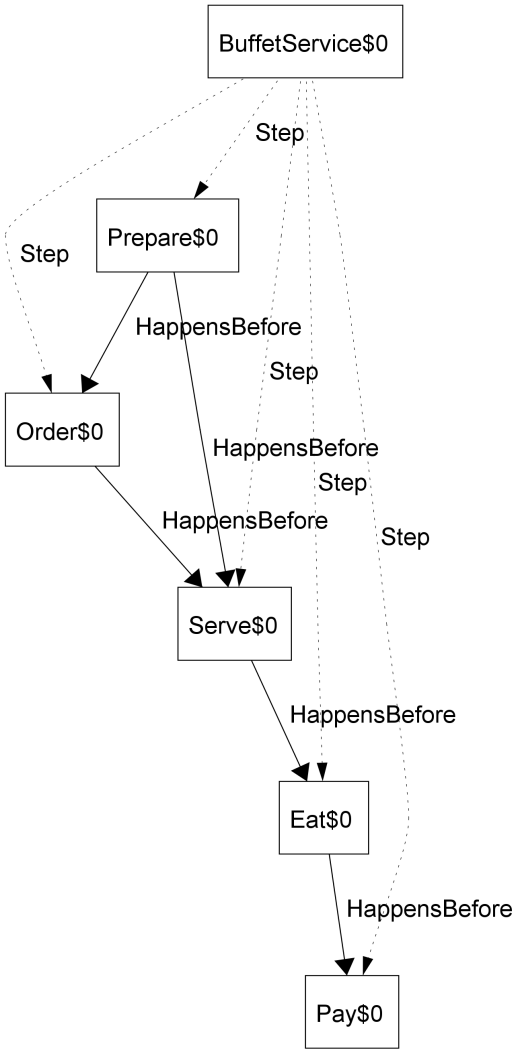


Fig. 70. Solution to Listing 33

Church supper Figures 71 and 72 show SysML activity and OBM representations of a church supper, respectively. In the OBM representation, `ChurchSupperService` introduces HappensBefore connectors from pay to prepare and order, adding to the ones inherited from FoodService via SingleFoodService.

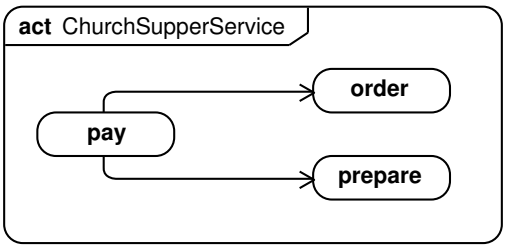


Fig. 71. Church supper (Activity)

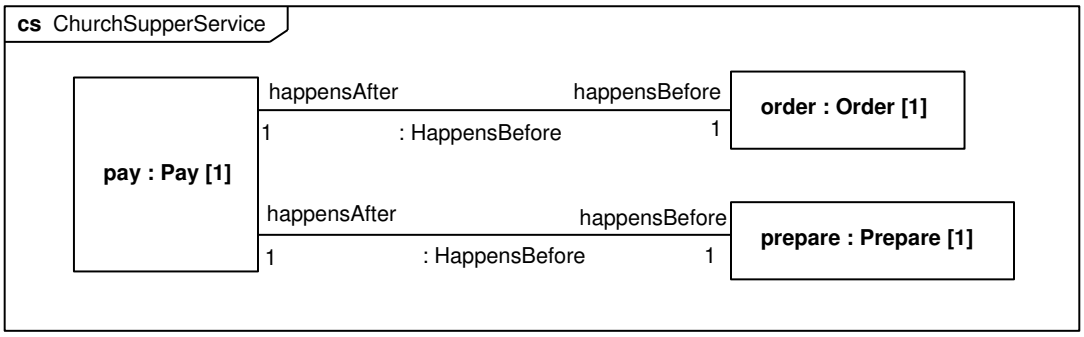


Fig. 72. Church supper (OBM)

Listing 34 shows the Alloy translation of Figure 72.

```
1 sig ChurchSupperService extends SingleFoodService{}{
2     bijectionFiltered[HappensBefore, pay, prepare]
3     bijectionFiltered[HappensBefore, pay, order]}
```

Listing 34. Alloy translation of Figure 72

Figure 73 shows a solution to Listing 34, highlighting `Step` and `HappensBefore` tuples. A `Pay` atom happens before `Prepare` and `Order` atoms, as specified by `ChurchSupperService`. The `Order` and `Prepare` atoms happen before a `Serve` atom, which happens before an `Eat` atom, as required by `FoodService`, which applies to `BuffetService` as its specialization via `SingleFoodService`.

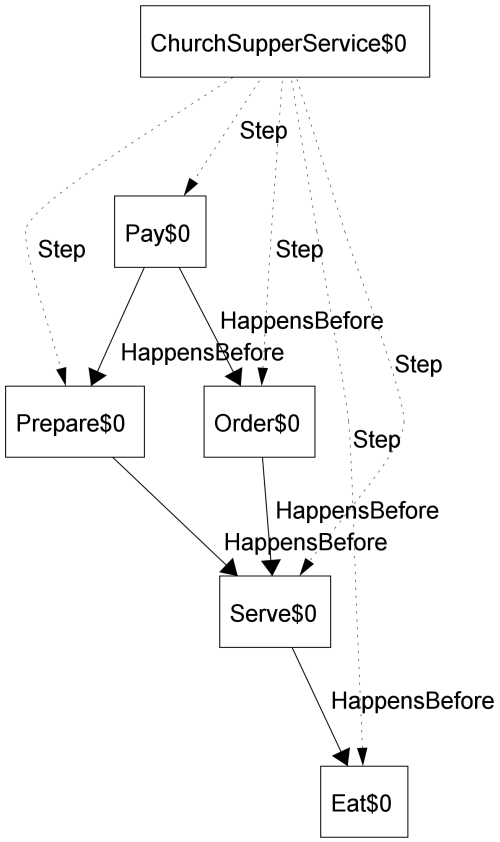


Fig. 73. ChurchSupperService solution

Fast food Figures 74 and 75 show SysML activity and OBM representations of a fast food service, respectively. In the OBM representation, FastFoodService introduces HappensBefore connectors from order to pay to eat, adding to the ones inherited from FoodService via SingleFoodService.

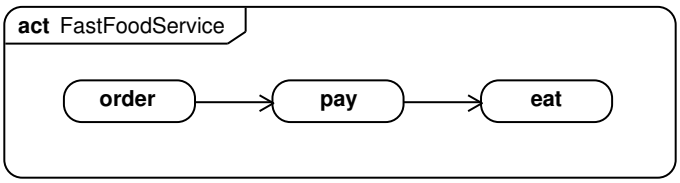


Fig. 74. Fast food service (Activity)

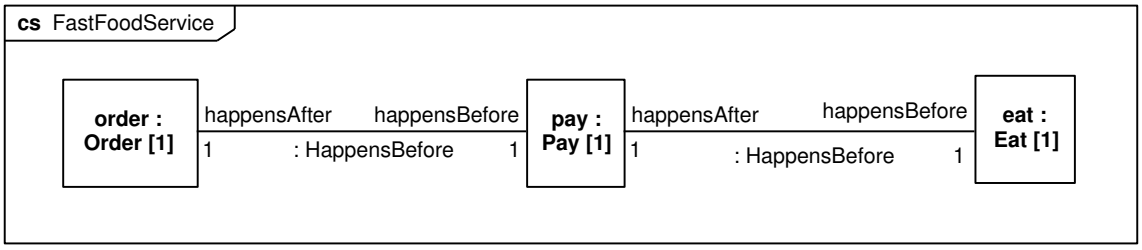


Fig. 75. Fast food service (OBM)

Listing 35 shows the Alloy translation of Figure 75.

```
1 sig FastFoodService extends SingleFoodService{}{
2     bijectionFiltered[HappensBefore, order, pay]
3     bijectionFiltered[HappensBefore, pay, eat]}
```

Listing 35. Alloy translation of Figure 75

Figure 76 shows a solution, highlighting `Step` and `HappensBefore` tuples. An `Order` happens before a `Pay` atom, which happens before an `Eat` atom, as specified by `FastFoodService`. The `Order` and `Prepare` atoms happen before a `Serve` atom, which happens before the `Eat` atom, as required by `FoodService`, which applies to `BuffetService` as its specialization via `SingleFoodService`.

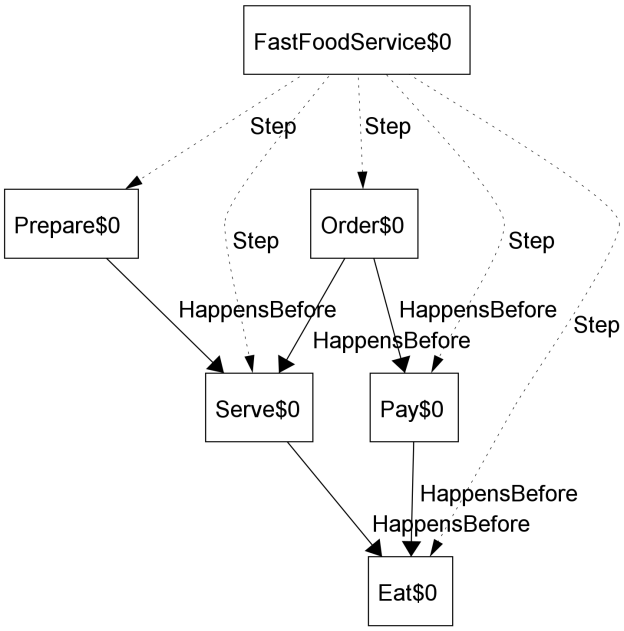


Fig. 76. FastFoodService solution

Restaurant The Figures 77 and 78 show SysML activity and OBM representations of a restaurant service, respectively. In the OBM representation, FastFoodService introduces a HappensBefore connector from eat to pay, adding to the ones inherited from FoodService via SingleFoodService.

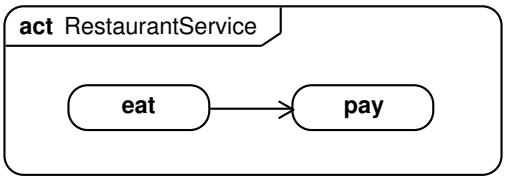


Fig. 77. Restaurant service (Activity)

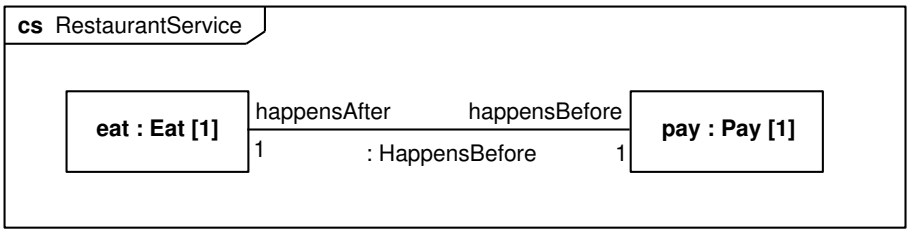


Fig. 78. Restaurant service (OBM)

Listing 36 shows the Alloy translation of Figure 78.

```
1 sig RestaurantService extends SingleFoodService{}{
2     bijectionFiltered[HappensBefore, eat, pay]}
```

Listing 36. Alloy translation of Figure 78

Figure 79 shows a solution, highlighting `Step` and `HappensBefore` tuples. An `Eat` atom happens before a `Pay` atom, as specified by `RestaurantService`. The `Order` and `Prepare` atoms happen before a `Serve` atom, which happens before the `Eat` atom, as required by `FoodService`, which applies to `BuffetService` as its specialization via `SingleFoodService`.

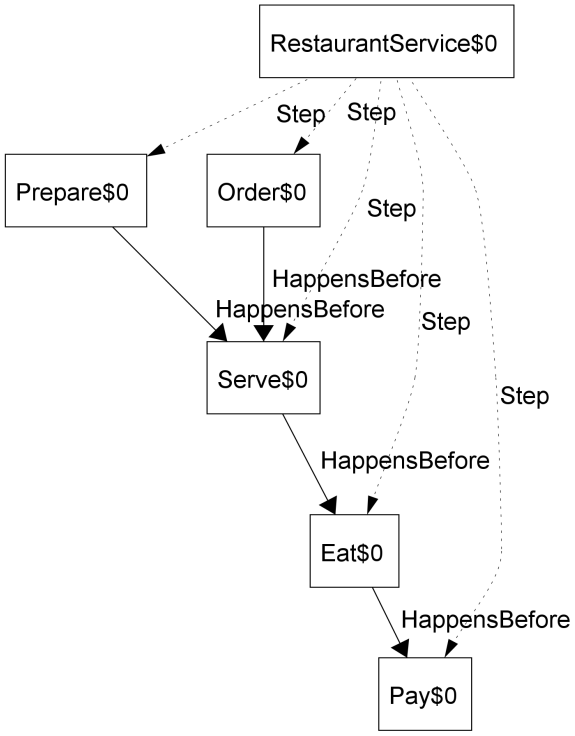


Fig. 79. Restaurant service solution

Unsatisfiable Figures 80 and 81 show SysML activity and OBM representations, respectively, of a food service that cannot happen, at least not following the constraints on food service occurrences. In the OBM representation, UnsatisfiableFoodService introduces a HappensBefore connectors from eat to pay to prepare, conflicting with the one from prepare to eat inherited from FoodService via SingleFoodService, due to the asymmetry and transitivity of HappensBefore, see lines 5-6 in Listing 4 in Section 3.2, and the second and third examples in Section 4.1.6.

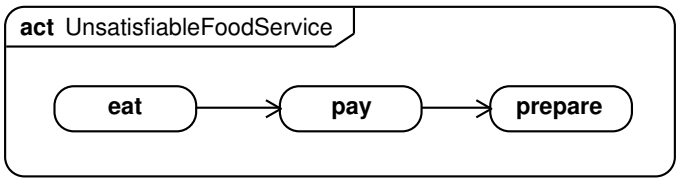


Fig. 80. Unsatisfiable food service (Activity)

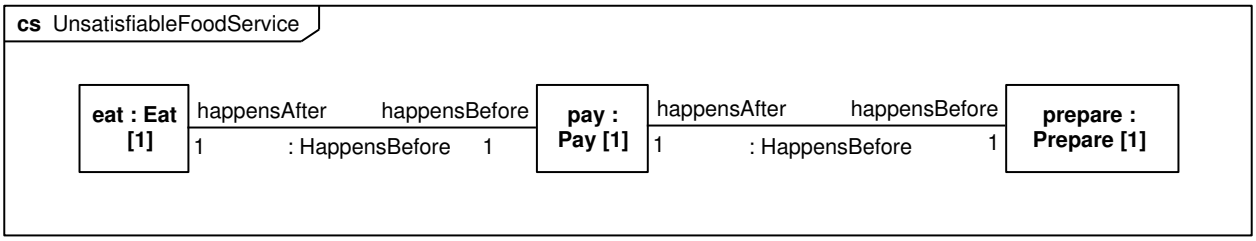


Fig. 81. Unsatisfiable food service (OBM)

Listing 37 shows the Alloy translation of Figure 81.

```
1 sig UnsatisfiableFoodService extends SingleFoodService{}{
2     bijectionFiltered[HappensBefore, eat, pay]
3     bijectionFiltered[HappensBefore, pay, prepare]}
```

Listing 37. Alloy translation of Figure 81

Alloy does not find a solution among possibilities up to many more atoms than should be needed.[31] Figure 82 shows a solution after temporarily removing the asymmetry constraint om HappensBefore. The figure shows that the `Serve`, `Eat`, `Pay`, and `Prepare` atoms form a `HappensBefore` cycle, which is normally not possible.

[31] See footnote 30 in Section 4.1.6.

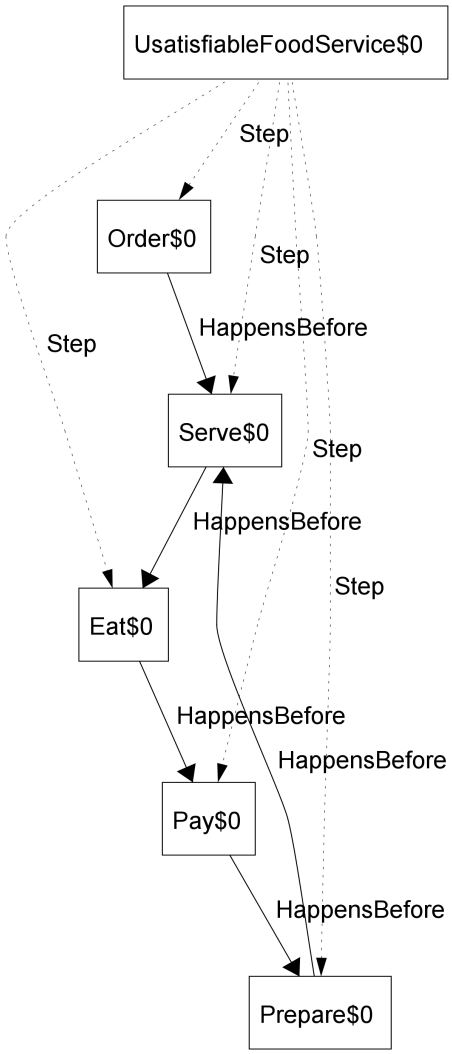


Fig. 82. Solution to Listing 37 with asymmetry constraint removed

4.2.2 Object flow examples

Object flow food service Figure 83 introduces OFFoodService, a specialized FoodService with steps redefined to add properties to their types that are used as inputs or outputs by transfers between the steps (SysML object flows between pins), defined later. These "OF" behaviors add a property with values unique to the food involved in each occurrence of a service, typed by FoodItem.[32] OFPay adds a property for the amount being paid.

[32]See Listing 43 in the third example in this section for constraints enforcing uniqueness of FoodItem under multiple executions.

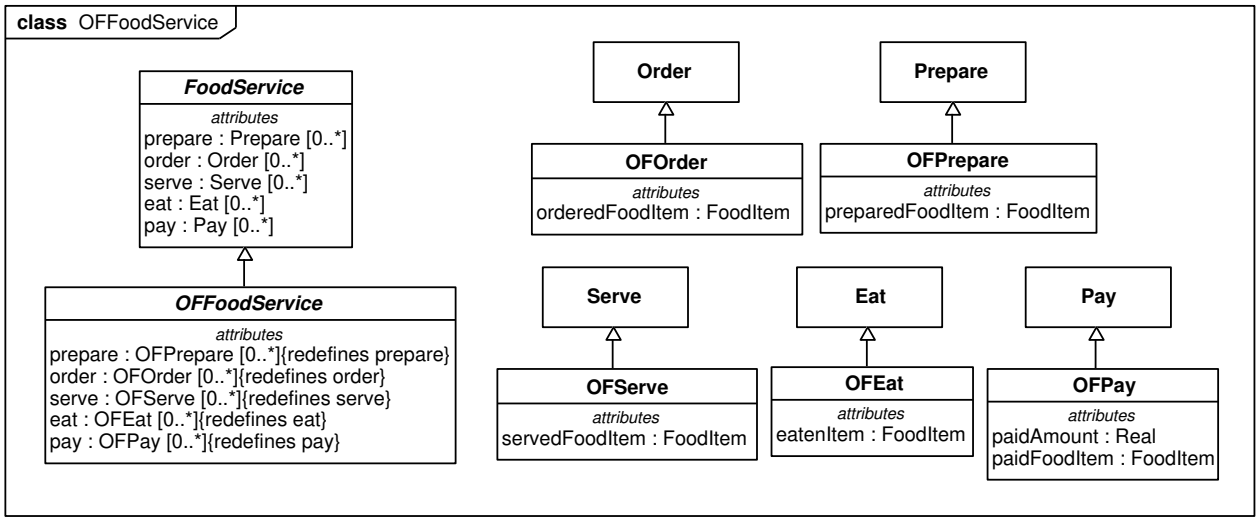


Fig. 83. Object flow food service occurrence classes

Listing 38 shows the Alloy translation of the specialized food service step types on the right in Figure 83. The things being output and input (flowing between steps) are classified by `FoodItem`, `Location`, or `Real`. They are occurrences, for simplicity, as in Listing 18 in the second example of Section 4.1.4 (lines 1-3). They are used by the specialized `OF` step types as inputs and outputs, with multiplicity `1` (the default for SysML properties). These are only constrained to subset the signature's `Input` and `Output`, not vice-versa, to provide flexibility when they are used by the examples in this section.

```
1  sig FoodItem, Location, Real extends Occurrence {}{
2    no HappensBefore && no @HappensBefore.this &&
3      no Step && no Input && no Output}
4
5  sig OFOrder extends Order {
6      orderedFoodItem: one FoodItem
7  }{  no Input
8      orderedFoodItem in Output}
9
10 sig OFPrepare extends Prepare {
11     preparedFoodItem: one FoodItem
12 }{  preparedFoodItem in Input
13     preparedFoodItem in Output}
14
15 sig OFServe extends Serve {
16    servedFoodItem: one FoodItem
17 }{  servedFoodItem in Input
18     servedFoodItem in Output}
19
```

```
20 sig OFEat extends Eat {
21     eatenItem: one FoodItem
22 }{  eatenItem in Input
23     no Output}
24
25 sig OFPay extends Pay {
26     paidAmount: one Real,
27     paidFoodItem: one FoodItem}
28 {   paidAmount in Input
29     paidFoodItem in Input
30     paidFoodItem in Output}
```

Listing 38. Alloy translation of specialized food service step types in Figure 83

Figures 84 and 85 show SysML activity and OBM representations, respectively, of TransferBefore connectors between steps of OFFoodService, consistent with the corresponding HappensBefore connectors in FoodService, see Figure 65 in Section 4.2.1.

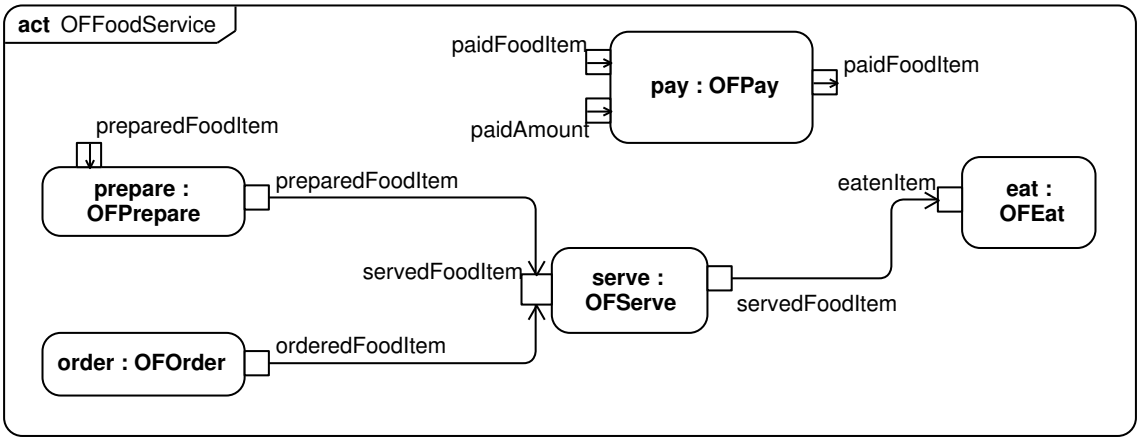


Fig. 84. Object flow food service (Activity)

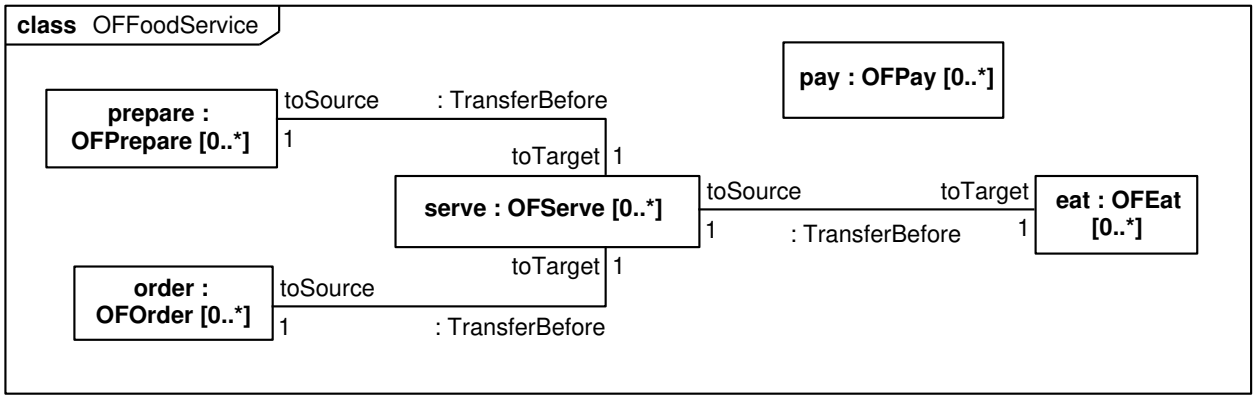


Fig. 85. Object flow food service (OBM)

Listing 39 shows the Alloy translation of Figure 85. It assumes the translation of FoodService (Listing 31 in Section 4.2.1) only subsets `Step`:

```
order + prepare + pay + eat + serve in Step
```

rather than also supersetting it, as assumed in previous examples in the paper (see description of Listing 7 in Section 4.1.1). This enables specializations of `FoodService` in this section to add object flows, which OBM treats as steps typed by `Transfer` or one of its specializations.

The action (non-transfer) steps are constrained to be typed by `OF` specializations (lines 5-9), to support the specialized `Input` and `Output` relations defined in Listing 38. The `TransferBefore` connectors in Figure 85 translate to steps of that type (lines 2 and 11), as in Listing 19 in Section 4.1.4, except they subset in only one direction, to enable other transfer steps to be added in specializations.

The rest of the translation follows Listing 19, including to pick up and drop off its items from `Output` or `Input`, respectively, specified by `subsettingItemRuleForSources` and `subsettingItemRuleForTarget`, respectively. The pickup and dropoff relations are constrained further in the next example.[33]

```
1  sig OFFoodService extends FoodService {
2      disj transferPrepareServe, transferOrderServe,
          transferServeEat: set TransferBefore
3  }{
4  /** Constraints on OFFoodService */
5      order in OFOrder
6      prepare in OFPrepare
7      pay in OFPay
8      eat in OFEat
9      serve in OFServe
10
11     transferPrepareServe + transferOrderServe +
          transferServeEat in Step
12
13 /** Constraints on transfers */
14     /** Constraints on the Transfer from prepare to serve */
15     bijectionFiltered[Source, transferPrepareServe, prepare]
16     bijectionFiltered[Target, transferPrepareServe, serve]
17     subsettingItemRuleForSources[transferPrepareServe]
18     subsettingItemRuleForTargets[transferPrepareServe]
19
```

[33]This translation uses `TransferBefore` connectors, rather than applying `isAfterSource` and `isBeforeTarget` as in Listing 19, which is equivalent.

```
20      /** Constraints on the Transfer from serve to eat */
21      bijectionFiltered[Source, transferServeEat, serve]
22      bijectionFiltered[Target, transferServeEat, eat]
23      subsettingItemRuleForSources[transferServeEat]
24      subsettingItemRuleForTargets[transferServeEat]
25
26      /** Constraints on the Transfer from order to serve */
27      bijectionFiltered[Source, transferOrderServe, order]
28      bijectionFiltered[Target, transferOrderServe, serve]
29      subsettingItemRuleForSources[transferOrderServe]
30      subsettingItemRuleForTargets[transferOrderServe]}
```

Listing 39. Alloy translation of Figure 85

Solutions to Listing 39 are found next for a specialization of `OFFoodService`.

Single object flow food service Figure 86 introduces OFSingleFoodService, a specialized OFFoodService (Figure 83) with its order step constrained to happen exactly once each time. The other step multiplicities are unconstrained, letting the reasoner determine how many times they happen, based on TransferBefores between them. The steps are also redefined to further specialize their types with more properties used as inputs or outputs by transfers between the steps (SysML object flows between pins), defined later. OFCustomOrder adds a property for the cost of the order, typed by Real. All the Custom specializations add a property for the place the food is ultimately to be served.

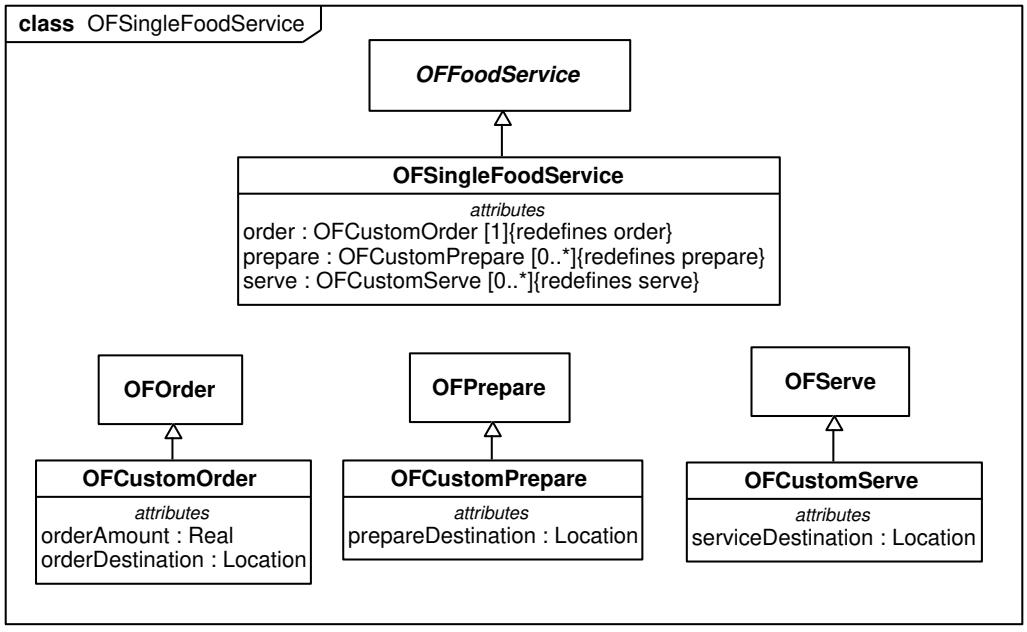


Fig. 86. Single object flow food service occurrence classes

Listing 40 shows the Alloy translation of the specialized food service step types at the bottom of Figure 86. As in Listing 38, the properties are only constrained to subset the signature's `Input` and `Output`, not vice-versa, to provide flexibility when they are used by the examples in this section.

```
1  sig OFCustomOrder extends OFOrder {
2      orderAmount: one Real,
3      orderDestination: one Location
4  }{  orderAmount in Output
5      orderDestination in Output}
6
7  sig OFCustomPrepare extends OFPrepare {
8      prepareDestination: one Location
9  }{  prepareDestination in Input
10     prepareDestination in Output}
11
12 sig OFCustomServe extends OFServe {
13     serviceDestination: one Location
14 }{serviceDestination in Input}
```

Listing 40. Alloy translation of specialized food service step types in Figure 86

Figures 87 and 88 show SysML activity and OBM representations, respectively, of OFSingleFoodService, including transfers between steps (SysML object flows between pins), those it introduces along with those it inherits. In the OBM representation, OFSingleFoodService specializes OFFoodService, adding TransferBefore connectors to those inherited from OFFoodService (Figure 85). Multiple object flows between the same steps translate to a single transfer connector, because transfer can carry multiple kinds of items, whereas object flows are always between pins, which have no more than one type. The additional connectors (object flows) make OFSingleFoodService a kind of FastFoodService (Figures 74 and 75 in Section 4.2.1), because they require paying after ordering and before eating, and use TransferBefores, which imply HappensBefore, see Listing 5 in Section 3.2 (lines 10-14).

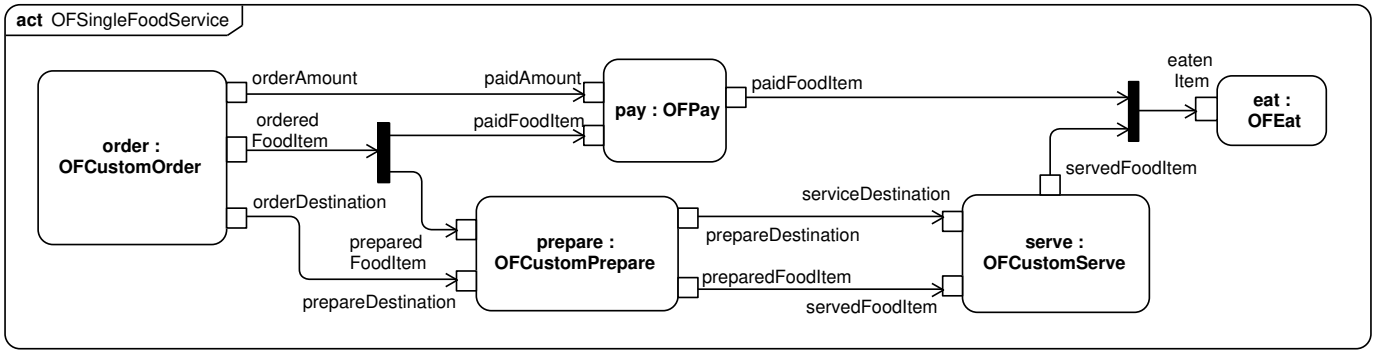


Fig. 87. Single object flow food service (Activity)

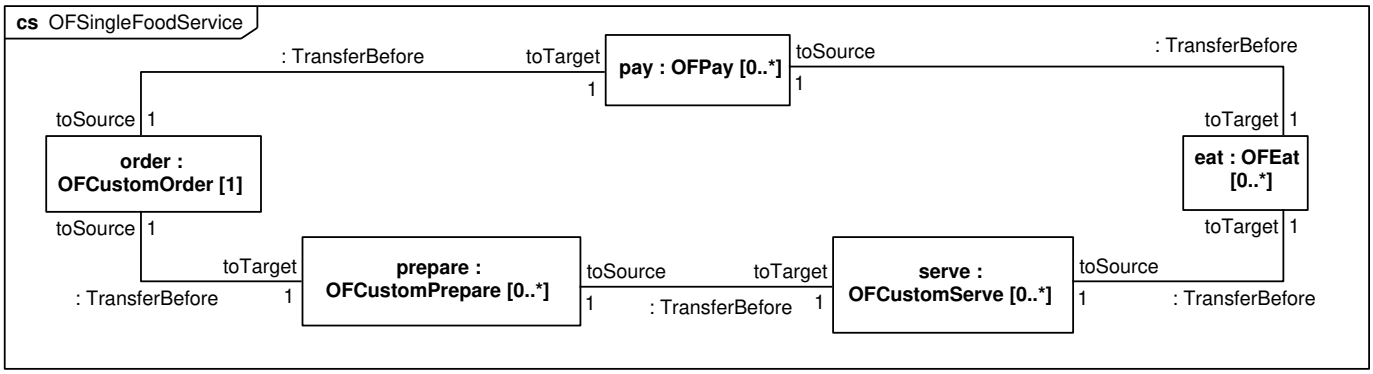


Fig. 88. Single object flow food service (OBM)

Listing 41 shows the Alloy translation of Figure 88. It defines the additional `TransferBefore` relations needed (line 2), constrains `order` to happen exactly once (line 15), as well as `order`, `prepare`, and `serve` to use the `OFCustom` signatures defined in Listing 38 (lines 5-7). The listing ensures the new `TransferBefore` relations are disjoint with the inherited ones (line 9) and that the transfers they identify for each atom of `OFSingleFoodService` are subset from those identified by the `Step` relation, and vice-versa, for all the specialized step relations, including inherited ones (line 10-11).[34] Similar bidirectional subsetting applies to the `Input` and `Output` relations and their specializations for each occurrence class (lines 16, 19-20, 27-28, and 31 add the inverse subset directions of those inherited from Listing 38 and assumed in Listing 39).

The rest of the translation constrains the `TransferBefore` steps introduced by `OFSingleFoodService` (lines 35-38, 45-48, 61-64) in the same way as Listing 39 does for the ones inherited to `OFSingleFoodService`, except Listing 41 additionally ensures the items in all the transfers are the exactly the same as the specialized outputs of their source, and the inputs of their target (lines 39-42, 49-52, 55-58, 65-80). These lines subset the items of (the values of) each `TransferBefore` step from the union of the outputs and inputs of the transfer's source and target, and vice versa. For example, the items of the `TransferBefore` between `order` and `pay` (`transferOrderPay.Item`) subset the union of the food item and amount of the order (line 39), as well as the union of the same attributes on `pay` (line 41), and vice-versa (lines 40 and 42).[34]

[34]The `TransferBefore` steps, such as `transferOrderPay`, are treated as sets of `TransferBefore` atoms of each `OFSingleFoodService`, whereas relations not defined in this signature, such as `Item`, `Source`, and `orderedFoodItem`, are treated as sets of tuples (see signature facts, Figure 4 in Section 3.1.2).

```
1  sig OFSingleFoodService extends OFFoodService{
2      disj transferOrderPrepare, transferOrderPay,
          transferPayEat: set TransferBefore
3  }{
4  /** Constraints on OFSingleFoodService */
5      order in OFCustomOrder
6      prepare in OFCustomPrepare
7      serve in OFCustomServe
8
9      no (transferOrderPrepare + transferOrderPay +
          transferPayEat) & (transferPrepareServe +
          transferOrderServe + transferServeEat)
10     transferOrderPrepare + transferOrderPay + transferPayEat
          in Step
11     Step in order + prepare + pay + serve + eat +
          transferPrepareServe + transferOrderServe +
          transferServeEat + transferOrderPrepare +
          transferOrderPay + transferPayEat
12
13 /** Constraints on process steps */
14     /** Constraints on order: OFCustomOrder */
15     #order = 1
16     order.@Output in order.@orderedFoodItem + order.
          @orderAmount + order.@orderDestination
17
18     /** Constraints on pay: IFPay */
19     pay.@Input in pay.paidAmount + pay.paidFoodItem
20     pay.@Output in pay.paidFoodItem
21
22     /** Constraints on prepare: OFCustomPrepare */
23     prepare.@Input in prepare.preparedFoodItem + prepare.
          prepareDestination
24     prepare.@Output in prepare.preparedFoodItem + prepare.
          prepareDestination
25
26     /** Constraints on serve: OFCustomServe */
27     serve.@Input in serve.servedFoodItem + serve.
          serviceDestination
28     serve.@Output in serve.servedFoodItem + serve.
          serviceDestination
29
30     /** Constraints on eat: OFEat */
31     eat.@Input in eat.eatenItem
```

```
32
33 /** Constraints on transfers */
34     /** Constraints on the Transfer from order to pay*/
35     bijectionFiltered[Source, transferOrderPay, order]
36     bijectionFiltered[Target, transferOrderPay, pay]
37     subsettingItemRuleForSources[transferOrderPay]
38     subsettingItemRuleForTargets[transferOrderPay]
39     transferOrderPay.Item in transferOrderPay.Source.
          orderedFoodItem + transferOrderPay.Source.orderAmount
40     transferOrderPay.Source.orderedFoodItem + transferOrderPay
          .Source.orderAmount in transferOrderPay.Item
41     transferOrderPay.Item in transferOrderPay.Target.
          paidFoodItem + transferOrderPay.Target.paidAmount
42     transferOrderPay.Target.paidFoodItem + transferOrderPay.
          Target.paidAmount in transferOrderPay.Item
43
44     /** Constraints on the Transfer from pay to eat */
45     bijectionFiltered[Source, transferPayEat, pay]
46     bijectionFiltered[Target, transferPayEat, eat]
47     subsettingItemRuleForSources[transferPayEat]
48     subsettingItemRuleForTargets[transferPayEat]
49     transferPayEat.Item in transferPayEat.Source.paidFoodItem
50     transferPayEat.Item in transferPayEat.Target.eatenItem
51     transferPayEat.Source.paidFoodItem in transferPayEat.Item
52     transferPayEat.Target.eatenItem in transferPayEat.Item
53
54     /** Constraints on the Transfer from order to serve */
55     transferOrderServe.Item in transferOrderServe.Source.
          orderedFoodItem
56     transferOrderServe.Item in transferOrderServe.Target.
          servedFoodItem
57     transferOrderServe.Source.orderedFoodItem in
          transferOrderServe.Item
58     transferOrderServe.Target.servedFoodItem in
          transferOrderServe.Item
59
60     /** Constraints on the Transfer from order to prepare*/
61     bijectionFiltered[Source, transferOrderPrepare, order]
62     bijectionFiltered[Target, transferOrderPrepare, prepare]
63     subsettingItemRuleForSources[transferOrderPrepare]
64     subsettingItemRuleForTargets[transferOrderPrepare]
65     transferOrderPrepare.Item in transferOrderPrepare.Source.
          orderedFoodItem + transferOrderPrepare.Source.
          orderDestination
```

```
66      transferOrderPrepare.Item in transferOrderPrepare.Target.
           preparedFoodItem + transferOrderPrepare.Target.
           prepareDestination
67      transferOrderPrepare.Source.orderedFoodItem +
           transferOrderPrepare.Source.orderDestination in
           transferOrderPrepare.Item
68      transferOrderPrepare.Target.preparedFoodItem +
           transferOrderPrepare.Target.prepareDestination in
           transferOrderPrepare.Item
69
70      /** Constraints on the Transfer from prepare to serve */
71      transferPrepareServe.Item in transferPrepareServe.Source.
           preparedFoodItem + transferPrepareServe.Source.
           prepareDestination
72      transferPrepareServe.Source.preparedFoodItem +
           transferPrepareServe.Source.prepareDestination in
           transferPrepareServe.Item
73      transferPrepareServe.Item in transferPrepareServe.Target.
           servedFoodItem + transferPrepareServe.Target.
           serviceDestination
74      transferPrepareServe.Target.servedFoodItem +
           transferPrepareServe.Target.serviceDestination in
           transferPrepareServe.Item
75
76      /** Constraints on the Transfer from serve to eat */
77      transferServeEat.Item in transferServeEat.Source.
           servedFoodItem
78      transferServeEat.Item in transferServeEat.Target.eatenItem
79      transferServeEat.Source.servedFoodItem in transferServeEat
           .Item
80      transferServeEat.Target.eatenItem in transferServeEat.Item
           }
```

Listing 41. Alloy translation of Figure 88

Figures 89 through 92 show views of the same solution to Listing 41. Figure 89 is the usual summary view, highlighting `Step` and `HappensBefore` tuples. `Input` and `Output` of each step atom are shown as attributes, as are the `Item` of each `TransferBefore`. The figure shows a solution to the timing and transfer constraints in `OFFoodService` and `OFSingleFoodService` implied by their TransferBefore connectors in Figure 88. The series of atoms down the left side of the graph are ordered in time as they should be, and the same `FoodItem` and `Location` atoms are related to the occurrences of `Order`, `Prepare`, and `Serve`, as well as the `FoodItem` of `Eat` being the same as those, due to the `TransferBefore` atoms between the step occurrences.

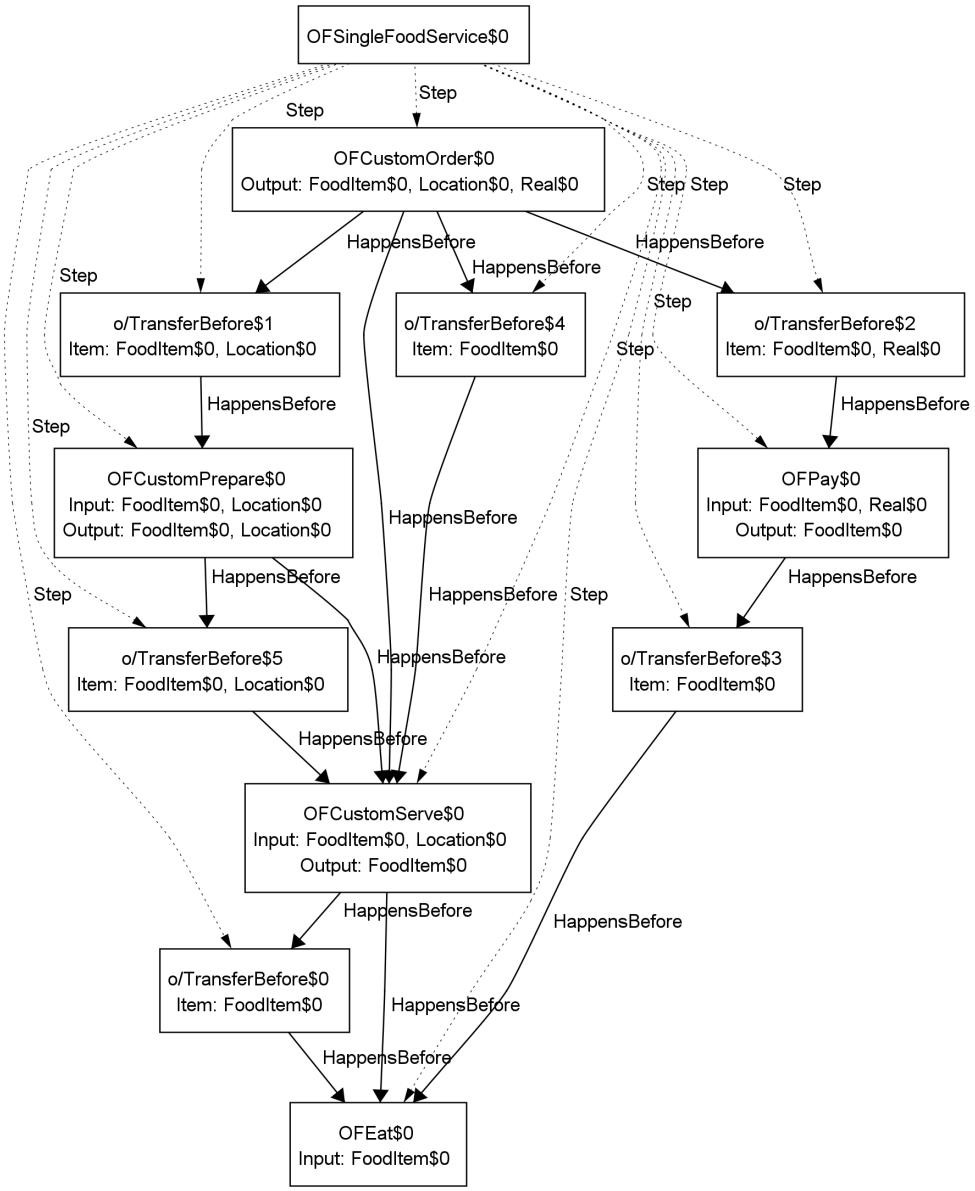


Fig. 89. Summary view of a solution to Listing 41

Figure 90 shows a view of the same solution as above, but including the names of action steps, and omitting the `TransferBefore` step edges (tuples).

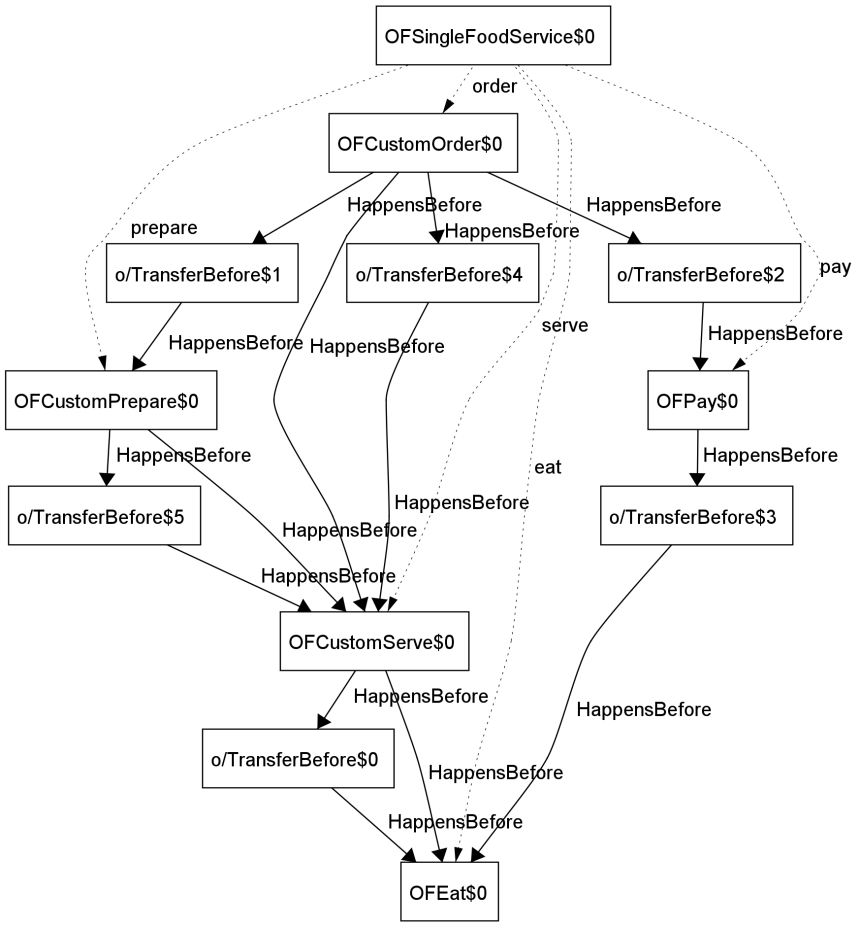


Fig. 90. Another view of the same solution to Listing 41, showing action step names

Figure 91 shows a view highlighting the same `Input` and `Output` tuples in two ways, as edges and as attributes with parameter names. It shows that the `FoodItem`, `Location`, and `Real` atoms are in the `Input` and `Output` tuples of all the step occurrences they should be.

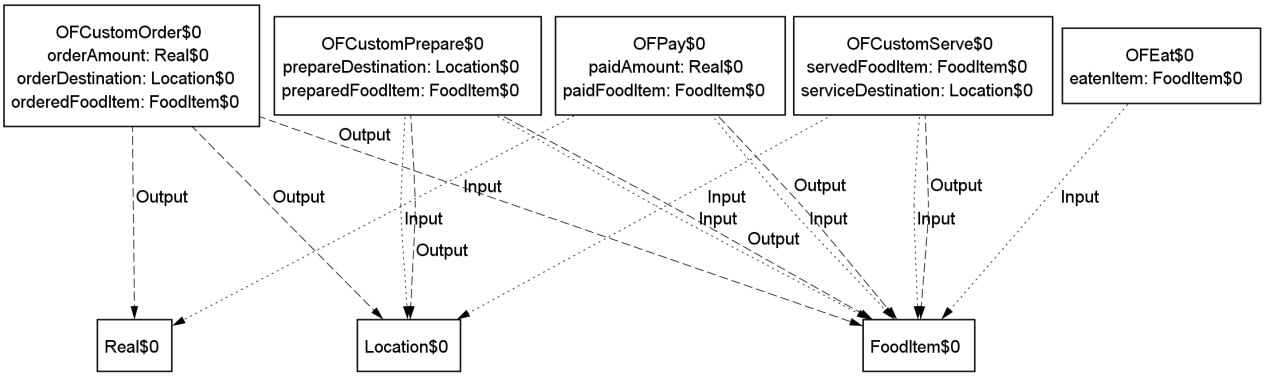


Fig. 91. Another view of the same solution to Listing 41, showing inputs, outputs, and their specific parameter names

Figure 92 shows a view highlighting tuples with `TransferBefore` atoms as first element. It confirms that they carry multiple types of items correctly.

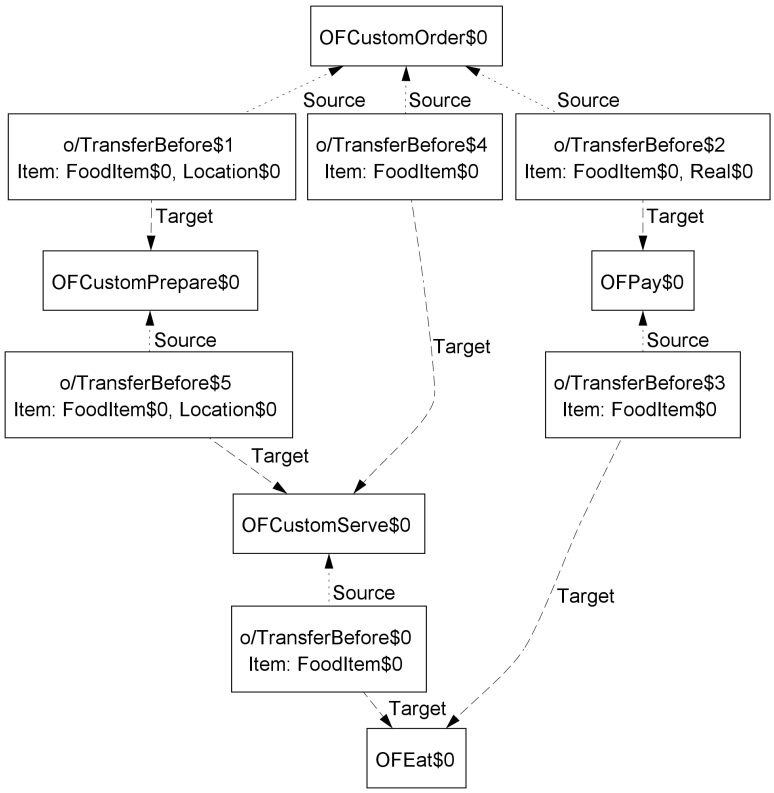


Fig. 92. Another view of the same solution to Listing 41, showing transfer sources and targets

Object flow food service with parallelism The intention of the last two examples in this section was to show solutions to specialized object flow food services for multiple executions, as in the object flow example in Section 4.1.5 and in the last two examples in [6]. These would have been specializations of OFFoodService (Figure 83) similar to OFSingleFoodService (Figure 86) that redefine the order step to happen exactly twice. These specializations were solved in [6] by another reasoner.

The "Alt" examples in Figures 93 and 94 are presented here instead of the above specializations, because Alloy was still searching for a solution to those after one workday, as compared to around two minutes to solve most of the other examples in this paper.[35] The alternatives simplify Figures 87 and 88, respectively, by combining preparing and serving into one step, typed by an occurrence class that has no steps defined.[36] The intention is for order to happen twice, which is not expressible in SysML currently, but is specified in the OBM representation as its multiplicity.

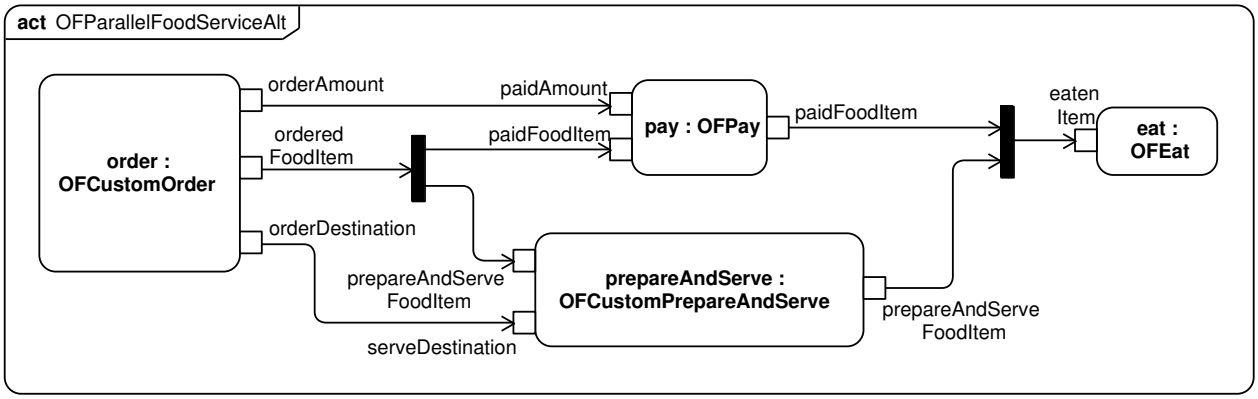


Fig. 93. Food service with actions and object flows intended to be executed multiple times (Activity)

[35] The reasoner used in [6] found solutions to the original examples in a few seconds on a slower machine.

[36] The time Alloy requires to find solutions increases with the number of atoms needed per signature, including the atoms of signatures extending them. Merging these two steps reduces the number of `Occurrence` atoms enough for Alloy to find a solution in reasonable time to this and the next example (compare to unsatisfiable examples in Sections 4.1.6 and 4.2.1). The simplified examples cannot specialize OFSingleFoodService because they remove step and transfer relations it requires.

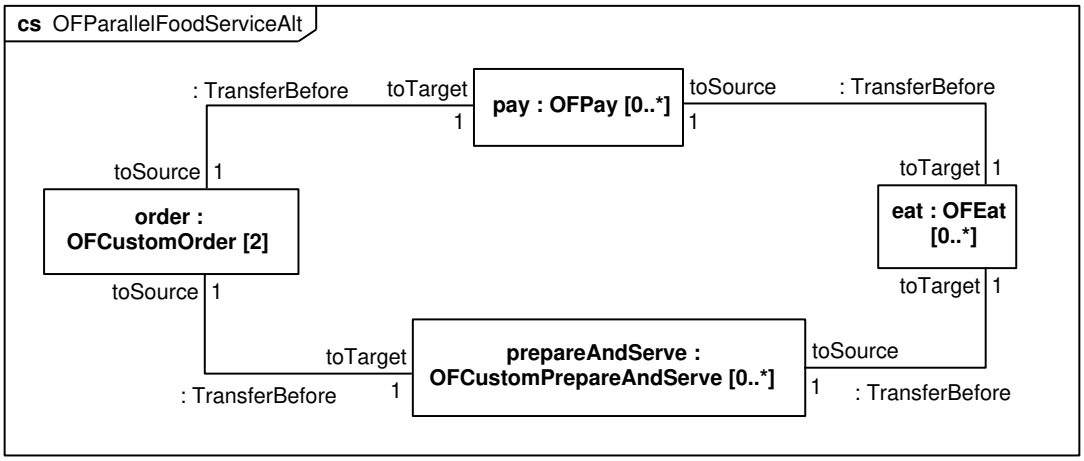


Fig. 94. Food service with steps and transfer connectors that identify multiple occurrences (OBM)

Listing 42 shows Alloy for the combined step type, similar to Listing 40. Specific parameter names are declared and included in `Inputs` and `Outputs`.

```
1  sig PrepareAndServe extends Occurrence {}
2
3  sig OFPrepareAndServe extends PrepareAndServe {
4      prepareAndServeFoodItem: one FoodItem
5  }{  prepareAndServeFoodItem in Input
6      prepareAndServeFoodItem in Output}
7
8  sig OFCustomPrepareAndServe extends OFPrepareAndServe {
9      serveDestination: one Location
10 }{  serveDestination in Input}
```

Listing 42. Alloy for merged step type used in Figure 94

Listing 43 shows the translation of Figure 94, following Listing 41 and earlier object flow examples. It differs from those by requiring each flowing atom to go through exactly one atom of each step type, by filtered bijections between the steps and their inputs and outputs (lines 26-28, 33-35, 41-43, and 47). This ensures `FoodItem` is unique per order.[37]

```
1  sig OFParallelFoodService extends Occurrence{
2      order: set OFCustomOrder,
3      pay: set OFPay,
4      prepareAndServe: set OFCustomPrepareAndServe,
5      eat: set OFEat,
```

[37]It also prevents orders from having the same locations and amounts, even though they normally would in a restaurant, just to avoid solutions where every order is delivered to the same location and costs the same.

```
6       disj transferOrderPrepareAndServe,
7         transferPrepareAndServeEat,
8         transferOrderPay, transferPayEat: set TransferBefore
9   }{
10  /** Constraints on OFParallelFoodService */
11      no Input and no @Input.this
12      no Output and no @Output.this
13
14      order + pay + prepareAndServe + eat +
15        transferOrderPrepareAndServe +
16        transferPrepareAndServeEat + transferOrderPay +
              transferPayEat in Step
17      Step in order + pay + prepareAndServe + eat +
18        transferOrderPrepareAndServe +
19        transferPrepareAndServeEat + transferOrderPay +
              transferPayEat
20
21  /** Constraints on process steps */
22      /** Constraints on order: OFCustomOrder */
23      #order = 2
24      order.@Output in order.orderedFoodItem + order.orderAmount
25        + order.orderDestination
26      bijectionFiltered[Output, order, FoodItem]
27      bijectionFiltered[Output, order, Real]
28      bijectionFiltered[Output, order, Location]
29
30      /** Constraints on pay: OFPay */
31      pay.@Input in pay.paidAmount + pay.paidFoodItem
32      pay.@Output in pay.paidFoodItem
33      bijectionFiltered[Input, pay, Real]
34      bijectionFiltered[Input, pay, FoodItem]
35      bijectionFiltered[Output, pay, FoodItem]
36
37      /** Constraints on prepare: OFCustomPrepareAndServe */
38      prepareAndServe.@Input in prepareAndServe.
            prepareAndServeFoodItem +
39        prepareAndServe.serveDestination
40      prepareAndServe.@Output in prepareAndServe.
            prepareAndServeFoodItem
41      bijectionFiltered[Input, prepareAndServe, FoodItem]
42      bijectionFiltered[Input, prepareAndServe, Location]
43      bijectionFiltered[Output, prepareAndServe, FoodItem]
44
45      /** Constraints on eat: OFEat */
46      eat.@Input in eat.eatenItem
```

```
47      bijectionFiltered[Input, eat, FoodItem]
48
49  /** Constraints on transfers */
50      /** Constraints on the Transfer from order to pay*/
51      bijectionFiltered[Source, transferOrderPay, order]
52      bijectionFiltered[Target, transferOrderPay, pay]
53      subsettingItemRuleForSources[transferOrderPay]
54      subsettingItemRuleForTargets[transferOrderPay]
55      transferOrderPay.Item in transferOrderPay.Source.
56        orderedFoodItem + transferOrderPay.Source.orderAmount
57      transferOrderPay.Source.orderedFoodItem + transferOrderPay
           .Source.orderAmount in transferOrderPay.Item
58      transferOrderPay.Item in transferOrderPay.Target.
59        paidFoodItem + transferOrderPay.Target.paidAmount
60      transferOrderPay.Target.paidFoodItem + transferOrderPay.
           Target.paidAmount in transferOrderPay.Item
61
62      /** Constraints on the Transfer from pay to eat */
63      bijectionFiltered[Source, transferPayEat, pay]
64      bijectionFiltered[Target, transferPayEat, eat]
65      subsettingItemRuleForSources[transferPayEat]
66      subsettingItemRuleForTargets[transferPayEat]
67      transferPayEat.Item in transferPayEat.Source.paidFoodItem
68      transferPayEat.Item in transferPayEat.Target.eatenItem
69      transferPayEat.Source.paidFoodItem in transferPayEat.Item
70      transferPayEat.Target.eatenItem in transferPayEat.Item
71
72      /** Constraints on the Transfer from order to
           prepareAndServe*/
73      bijectionFiltered[Source, transferOrderPrepareAndServe,
           order]
74      bijectionFiltered[Target, transferOrderPrepareAndServe,
           prepareAndServe]
75      subsettingItemRuleForSources[transferOrderPrepareAndServe]
76      subsettingItemRuleForTargets[transferOrderPrepareAndServe]
77      transferOrderPrepareAndServe.Item in
78        transferOrderPrepareAndServe.Source.orderedFoodItem +
79        transferOrderPrepareAndServe.Source.orderDestination
80      transferOrderPrepareAndServe.Item in
81        transferOrderPrepareAndServe.Target.
82        prepareAndServeFoodItem +
83        transferOrderPrepareAndServe.Target.serveDestination
84      transferOrderPrepareAndServe.Source.orderedFoodItem +
85        transferOrderPrepareAndServe.Source.orderDestination
86        in transferOrderPrepareAndServe.Item
```

```
87      transferOrderPrepareAndServe.Target.
88        prepareAndServeFoodItem +
89        transferOrderPrepareAndServe.Target.serveDestination
90        in transferOrderPrepareAndServe.Item
91
92      /** Constraints on the Transfer from prepareAndServe to
            eat */
93      bijectionFiltered[Source, transferPrepareAndServeEat,
            prepareAndServe]
94      bijectionFiltered[Target, transferPrepareAndServeEat, eat]
95      subsettingItemRuleForSources[transferPrepareAndServeEat]
96      subsettingItemRuleForTargets[transferPrepareAndServeEat]
97      transferPrepareAndServeEat.Item in
            transferPrepareAndServeEat.Source.
            prepareAndServeFoodItem
98      transferPrepareAndServeEat.Item in
            transferPrepareAndServeEat.Target.eatenItem
99      transferPrepareAndServeEat.Source.prepareAndServeFoodItem
            in transferPrepareAndServeEat.Item
100     transferPrepareAndServeEat.Target.eatenItem in
            transferPrepareAndServeEat.Item}
```

Listing 43. Alloy translation of Figure 94

Figures 95 through 97 show views of the same solution to Listing 43. Figure 95 highlights `Step` and `HappensBefore` tuples. `Input`, `Output`, and `Item` relations appear as attributes of their respective step type and transfer atoms. The figure shows that `FoodItem$0` and `FoodItem$1` are passed along by `Transfers` (as their `Items`) between separate atoms of `OFCustomOrder`, `OFCustomOrderOFPrepareAndServe`, `OFPay`, and `OFEat` on each side of the figure.

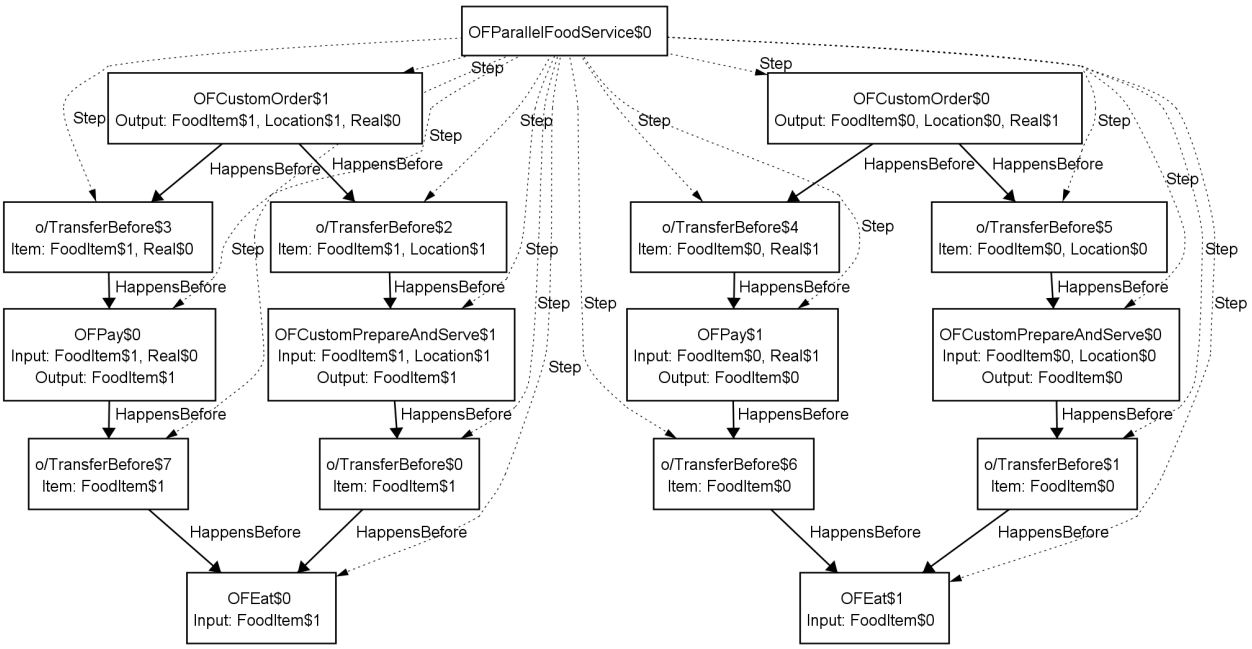
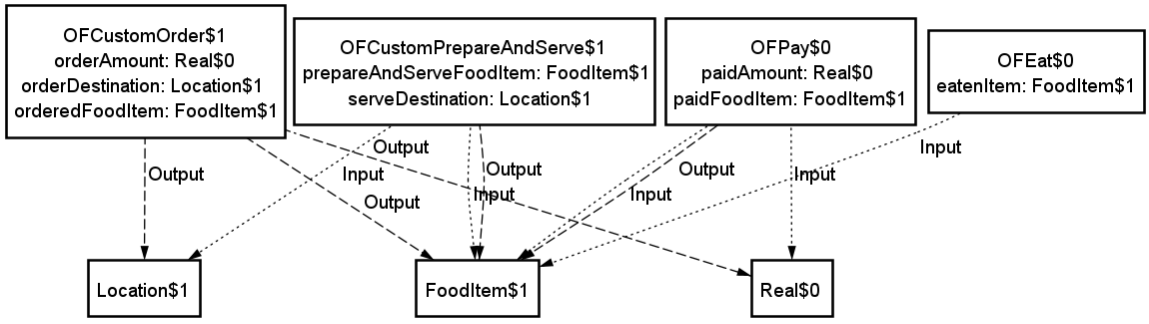
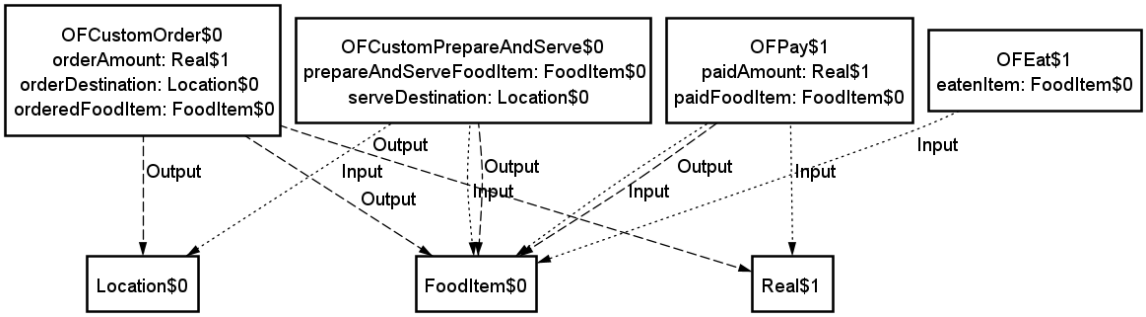


Fig. 95. Summary view of a solution to Listing 43

Figure 96 shows views of the solution above highlighting `Input` and `Output` tuples in two ways, as edges and as attributes with parameter names, for two branches of the solution in Figure 95. Each branch has its own `FoodItem`, `Location`, and `Real` atoms in `Input` and `Output` tuples of the step atoms.



(a) Left branch



(b) Right branch

Fig. 96. Views of the two branches of the solution in Figure 95, showing inputs and outputs relations their specific parameter names

Figure 97 shows a view highlighting tuples with `TransferBefore` atoms as first element. It confirms that they carry multiple types of items correctly.

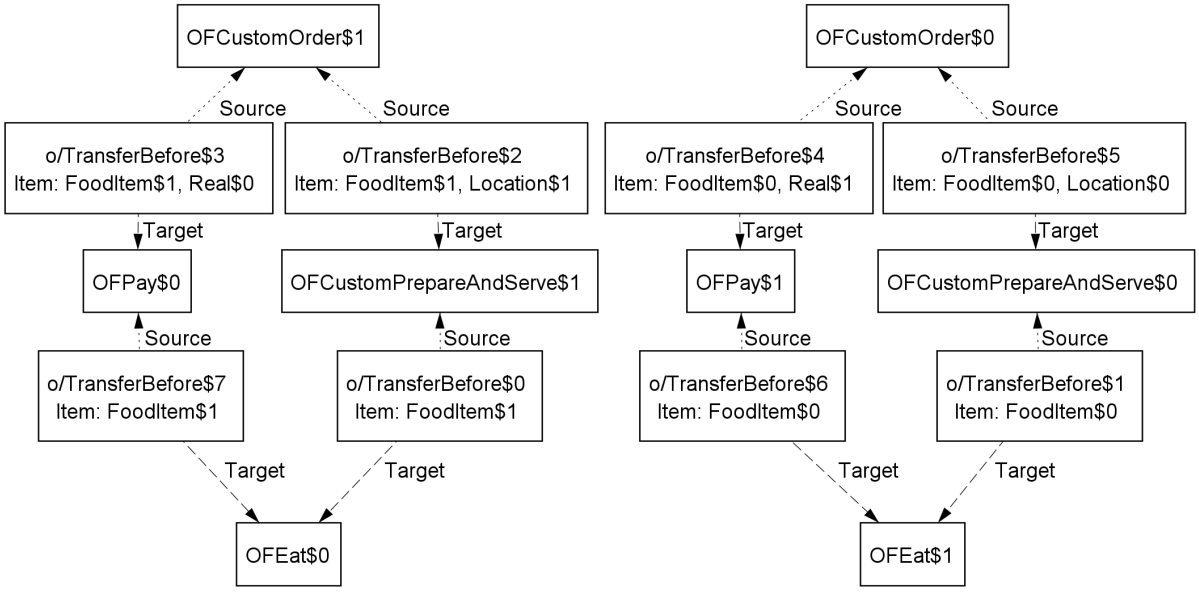


Fig. 97. Another view of the same solution to Listing 43, showing transfer sources and targets

Object flow food service with loop Figures 98 and 99 modify Figures 93 and 94, respectively, to constrain the two branches of Figure 95 to happen in order, rather than parallel, similar to the example in Section 4.1.2. The figures add SysML start and final nodes, treated in OBM as steps start and end, to happen before the first occurrence of order and after the last occurrence of eat, respectively. The steps start, eat, and order form a merge (as in Figure 20 in Section 4.1.1), while eat, order and end form a decision (Figure 17). A control flow is added for iteration, corresponding to a HappensBefore connector in OBM.

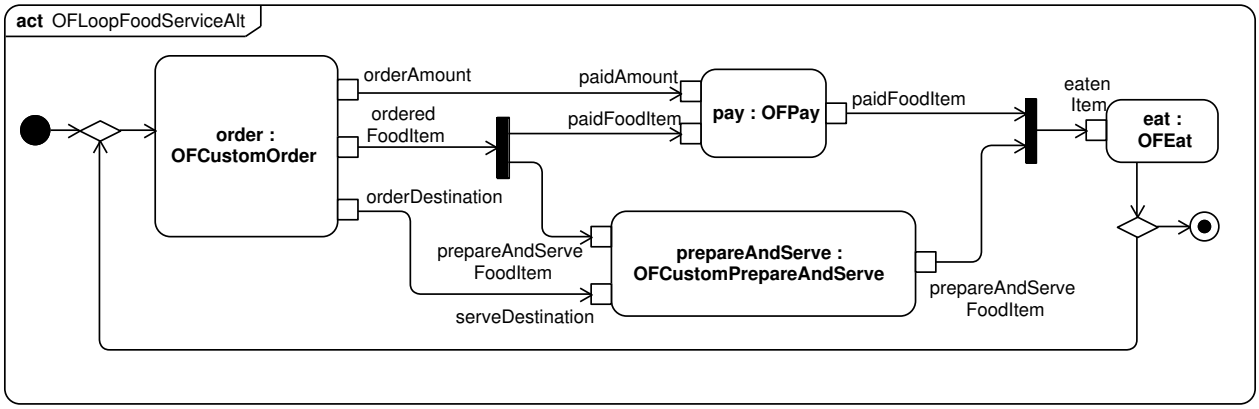


Fig. 98. Food service with actions and object flows intended to be executed multiple times in order (Activity)

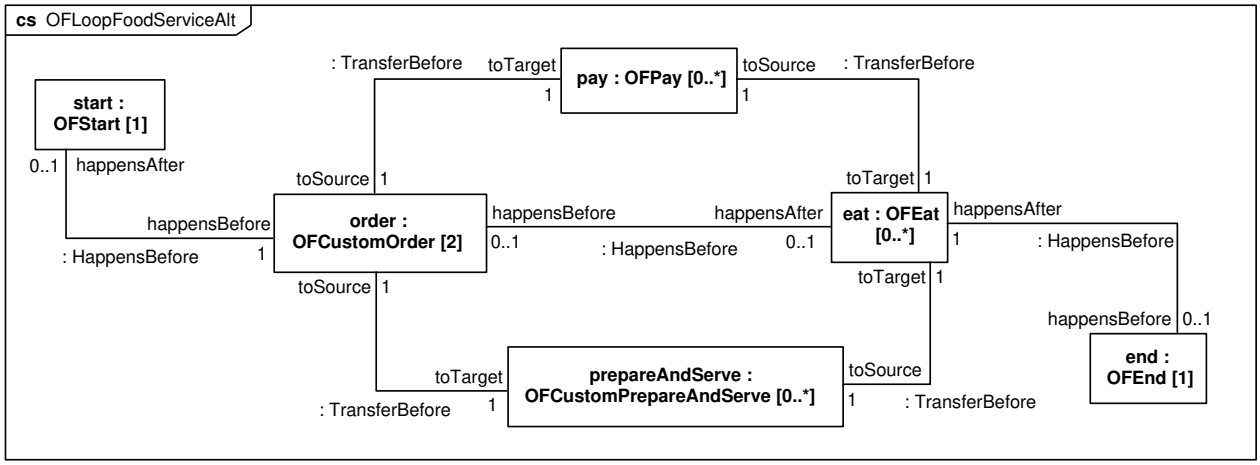


Fig. 99. Food service with steps and transfer connectors that identify multiple occurrences that happen in order (OBM)

Listing 44 shows the lines in translation of Figure 99 added to Listing 43. Lines 9 and 13 modify lines 14 and 17 in Listing 43 to include `start` and `end`.[38] Lines 20, 23, 26, and 30 are additional temporal constraints for the merge and decision, following Listings 12 and 11 in Section 4.1.1.

```
1  sig OFStart, OFEnd extends Occurrence {}{
2    no Inputs && @Inputs.this && no Outputs && no @Outputs.this
       && no Items.this}
3
4  sig OFLoopFoodService extends Occurrence {
5      start: one OFStart,
6      end: one OFEnd
7  {
8      /** Constraints on OFLoopFoodService */
9      start + end + order + pay + prepareAndServe + eat +
10       transferOrderPrepareAndServe +
11       transferPrepareAndServeEat + transferOrderPay +
12       transferPayEat in Step
13     Step in start + end + order + pay + prepareAndServe +
14       eat + transferOrderPrepareAndServe +
15       transferPrepareAndServeEat + transferOrderPay +
16       transferPayEat
17
18     /** Constraints on start: OFStart */
19     #start = 1
20     functionFiltered[HappensBefore, start, order]
21
22     /** Constraints on order: OFCustomOrder */
23     inverseFunctionFiltered[HappensBefore, start + eat, order]
24
25     /** Constraints on eat: OFEat */
26     functionFiltered[HappensBefore, eat, end + order]
27
28     /** Constraints on end: OFEnd */
29     #end = 1
30     inverseFunctionFiltered[HappensBefore, eat, end]
```

Listing 44. Alloy translation of Figure 99 showing only differences from Listing 43

Figures 100 through 102 show views of the same solution to Listing 44. They are the same kind of views as in the previous example (Figures 95 through 97).[39]

[38] The second of these prevents `OFLoopFoodService` from extending `OFParallelFoodService`. Translation could omit "closure" constraints like this, adding them only as needed for reasoning.

[39] Figure 100 includes an unspecified but consistent `HappensBefore` edge, in the second loop iteration, between `OFPay$0` and `OFCustomPrepareAndServe$1`. These steps are not required to happen in any particular order, but are not prevented either (see Forks in Section 4.1.1).

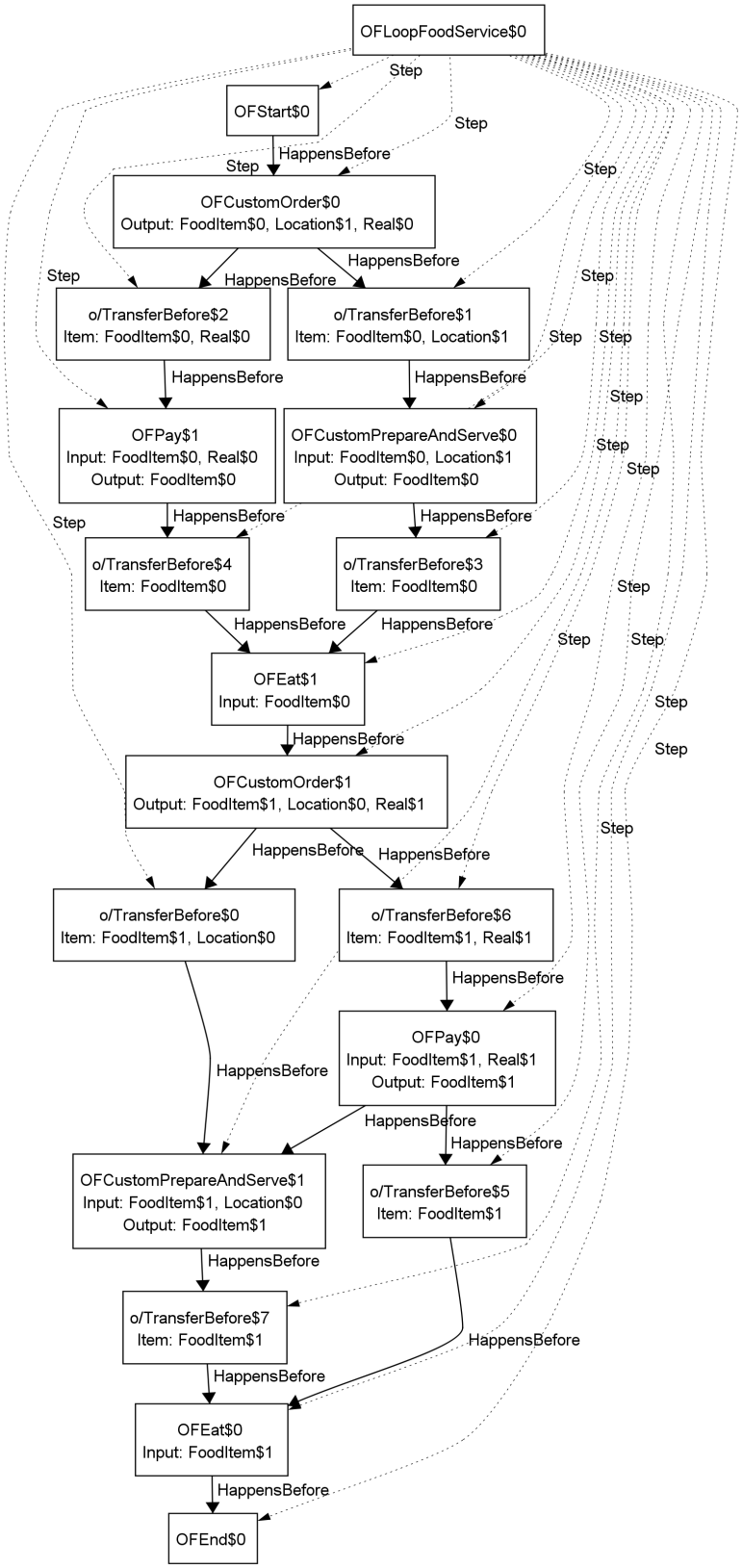
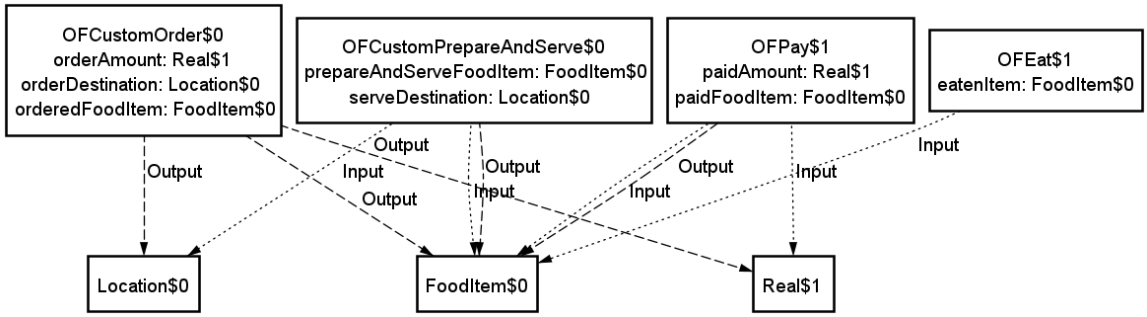
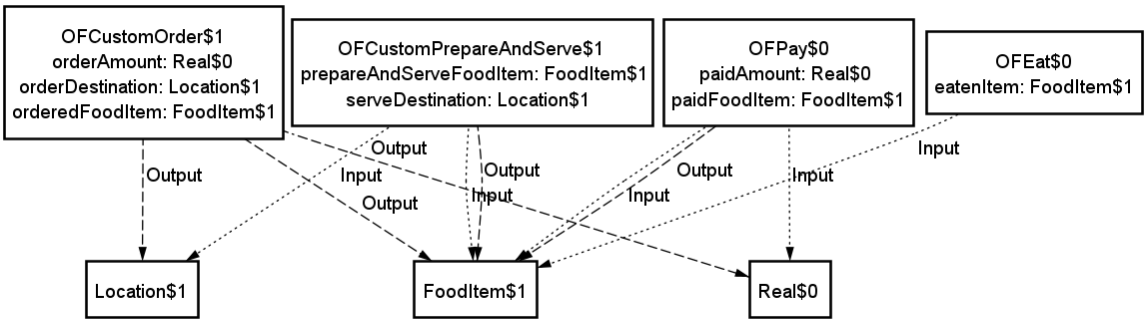


Fig. 100. Summary view of solution to Listing 44



(a) First iteration



(b) Second iteration

Fig. 101. Views of the two iterations of the solution in Figure 100, showing inputs and outputs relations their specific parameter names

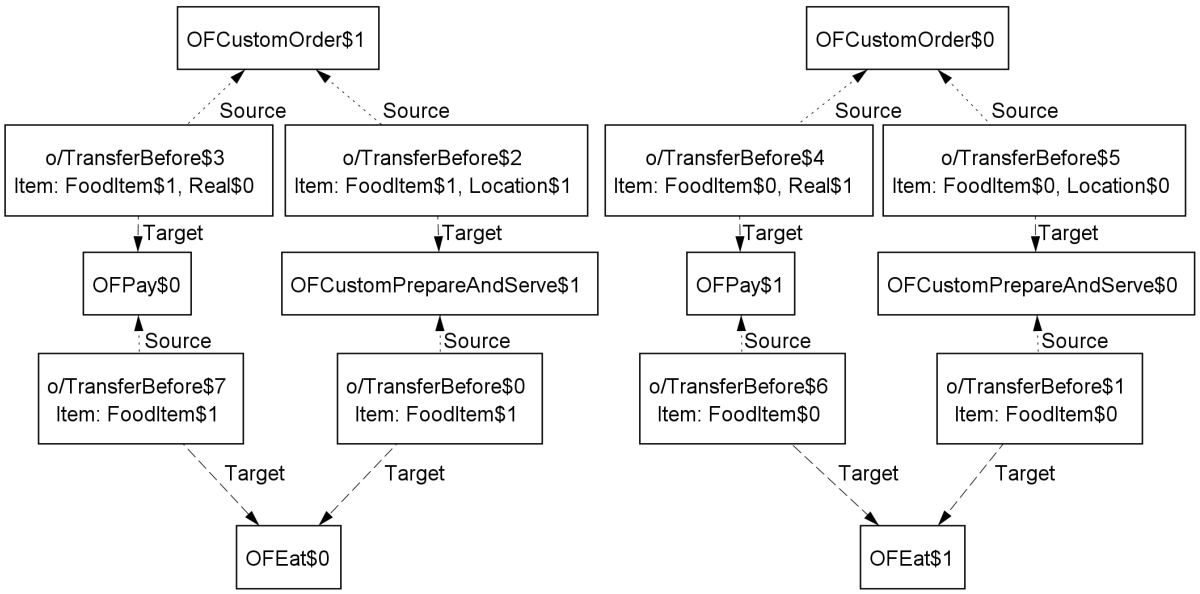


Fig. 102. Another view of the same solution to Listing 44, showing transfer sources and targets

5. Conclusion and Future work

This report presents an approach to verifying executability of SysML models under the OBM method by translating them to Alloy Analyzer's logical constraint language, and applying its underlying solvers to find executions. OBM unifies similar concepts from the three SysML behavior modeling techniques (activities, interactions, state machines). The paper translates SysML behavior models to Alloy, which supports SMT solvers and graphical presentation of solutions. Examples covering many common SysML behavior modeling patterns are presented and analyzed with Alloy.

We identified several directions for future work:

- Modeling start and finish of occurrences as zero-duration occurrences, to enable temporal precedence beyond finish to start.
- Support changing values of objects and behavior properties, which requires more advanced techniques such as 4D modeling to break single occurrences into time slices that differ in their property values [4].
- Solving object flow loops with iterations on the same object. The last example in Section 4.2.2 is an object flow loop, but each iteration involves different objects.
- Automated translation from SysML OBM diagrams to Alloy, based on the approach of this paper, as well as automated translation from SysML behavior models to SysML OBM diagrams.
- Supporting features that are not available in all SysML behavior modeling techniques.
- Separating threads of related occurrences in control flow models with multiple control nodes, as in 4.1.5. Compare to the last example in 4.2.2, which includes a unique item flowing through each thread.
- Addressing tractability of more complex models, including the last two examples in [6], which had to be simplified for Alloy in the last two examples in Section 4.2.2.

We plan to apply this approach to more realistic use cases, including examples related to manufacturing systems, to test scalability. These cases will have more complex behaviors, more occurrences, deep nesting, and so on.

Acknowledgements

The authors thank Bjorn Cole and Vincenzo Ferraro for their useful feedback.

This material is based in part on work supported by U.S. National Institute of Standards grant awards 70NANB19H066 and 70NANB20H177 to Georgia Tech Applied Research Corporation, and 70NANB18H200 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 (2019) Systems Modeling Language, version 1.6. Available at https://www.omg.org/spec/SysML/1.6/.

[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] 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

[5] Jackson D (2011) Alloy: A language and tool for exploring software designs. *Communications of the ACM* 62(9):66–76. https://doi.org/10.5381/jot.2011.10.1.a3

[6] Barbau R, Bock C (2020) Verifying executability of sysml behavior models using satisfiability modulo theory solvers (U.S. National Institute of Standards and Technology), Interagency Report 8283. https://doi.org/10.6028/NIST.IR.8283

[7] Object Management Group (2016) Meta Object Facility, version 2.5.1. Available at https://www.omg.org/spec/MOF/2.5.1/.

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

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

[10] Object Management Group (2017) Action Language for Foundational UML, version 1.1. Available at https://www.omg.org/spec/ALF/1.1.

[11] Smullyan R (1995) *First-Order Logic* (Dover Publications), .

[12] Pinter C (2014) *A Book of Set Theory* (Dover Publications), .

[13] Doerr J (2021) Translations from obm library and sysml behavior examples to alloy. Available at https://github.com/usnistgov/mbsdaism/releases/download/obmalloytrans/obmalloytrans.zip.

[14] Wyner G, Lee J (2003) Defining specialization for process models. *Organizing Business Knowledge: The MIT Process Handbook* (Massachusetts Institute of Technology), , pp 131–174.