Softw Syst Model (2004) / Digital Object Identifier (DOI) 10.1007/s10270-004-0066-x

PSL: A semantic domain for flow models

Conrad Bock[1], **Michael Gruninger**[2]

[1] U.S. National Institute of Standards and Technology, 100 Bureau Drive, Stop 8263, Gaithersburg, MD 20899-8263, USA
e-mail: conrad.bock@nist.gov
[2] Institute for Systems Research, University of Maryland, College Park, MD 20742, USA
e-mail: michael.gruninger@nist.gov

Received: 25 June 2003/Accepted: 22 November 2003
Published online: 10 November 2004 – © Springer-Verlag 2004

Abstract. Flow models underlie popular programming languages and many graphical behavior specification tools. However, their semantics is typically ambiguous, causing miscommunication between modelers and unexpected implementation results. This article introduces a way to disambiguate common flow modeling constructs, by expressing their semantics as constraints on runtime sequences of behavior execution. It also shows that reduced ambiguity enables more powerful modeling abstractions, such as partial behavior specifications. The runtime representation considered in this paper uses the Process Specification Language (PSL), which is defined in first-order logic, making it amenable to automated reasoning. The activity diagrams of the Unified Modeling Language are used for example flow models.

Keywords: Flow model – Flow semantics – PSL – Process specification – Control flow – Data flow – Concurrency – UML – Activity model

Communicated by Steve Cook

Abbreviations:

CL	= Common Logic
KIF	= Knowledge Interchange Format
OCL	= Object Constraint Language
PSL	= Process Specification Language
UML	= Unified Modeling Language

Flow models are the most common form of behavior specification. They underlie popular programming languages and many graphical behavior specification tools. However, their semantics is typically given in natural language or in varied implementations, leading to unexpected effects in the final system. This article gives a way to disambiguate common flow modeling constructs in terms of constraints on runtime sequences of behavior execution. Runtime effects are represented in the most concrete way, to cover all possible execution traces. Desired behavior is specified by constraining which of the possible executions are allowed. Reducing ambiguity enables more powerful abstractions, such as partial specifications that incrementally add constraints in behavior taxonomies. The runtime representation considered in this paper is the Process Specification Language (PSL) [12, 23], which is defined in first-order logic. Constraints on runtime effects are also stated this way, making behavior specifications in PSL amenable to automated reasoning with widely available inference engines.

The article begins with a short discussion relating ambiguity, abstraction, and expressiveness in languages. It identifies several unclear aspects of a typical flow model that will be addressed in the paper. Section 2 gives some background on PSL, the approach it takes to semantics, and how it is presented in this article. Section 3 introduces the basic PSL concepts for representing runtime execution, how these are composed, and how constraints are written on them. Section 4 covers the ordering of steps in an execution. Section 5 covers specifications where the steps are unordered. Section 6 applies the techniques of Sect. 4 to create behavior taxonomies using partial flow specifications. Section 7 addresses the issue of controlling what an inference engine can add to a specification (closure). Section 8 briefly examines other approaches to process semantics.

Examples are given in activity diagrams of the Unified Modeling Language, version 2 (UML 2) [2, 27], but could be other flow diagrams, or even programming languages, most of which are textual forms of flow models. Some of the examples happen to require hardware or humans, but implementations would often include software to coordinate the other two, and in any case no restriction is implied by UML. For lack of more general terms, we use "execution" and "runtime" from software to mean the actual playing out of a specified activity, whether it is in a computer, robot, or human organization.

1 Ambiguity, abstraction, and expressiveness

Ambiguity and abstraction have the common characteristic of omitting information. However, in abstraction the omission is intentional, clearly identified, and only contains information that users do not need. In ambiguity the omission is inadvertent, unidentified, and contains useful information. For example, readers of in Fig. 1 often take the arrow to mean message passing, so the diagram would be incorrectly interpreted as saying that the behavior Paint sends a message to the behavior Dry. Or those familiar with rule-based techniques might take it to mean that drying must always happen after painting whenever the behavior Paint is performed. It is actually intended to say the runtime effect of an execution of the ChangeColor behavior is that an execution of the behavior Paint will occur, and after that is completed, an execution of the behavior Dry will occur. However, the specification does not say that it means this and only this.

Figure 1 is ambiguous not because it is graphical, textual languages have the same problem, but because it is specifying execution with constructs that only implicitly refer to runtime, rather than explicitly. For example, the nodes labeled ChangeColor, Paint, and Dry will be executed many times in many situations, and the diagram does not clarify which executions are referred to, or how the graphical nesting and arcs constrain them. The exact runtime effect is only given in documentation, examples, or other human communication, and in implementations, which may or may not conform to the original specification. The one-to-many mapping between language elements and runtime effects makes the specification more concise, but also more imprecise.

Other open questions about Fig. 1 include:

1. Are other behaviors allowed to be introduced in ChangeColor? For example, can an inspection of the paint occur after painting and before drying?
2. Is there any constraint on how long after painting that drying must occur?
3. Is it possible under exceptional conditions to not do any drying at all, for example, if the paint job is accidentally ruined for some reason?
4. Is there any constraint on what behaviors may happen concurrently with painting, for example operating leaf blowers nearby?

Resolving ambiguities results in abstractions that add expressiveness. For example, ambiguity 1 above is addressed by a construct that clarifies whether additional behavior occurrences can be introduced between Paint and Dry. The new construct increases expressiveness by supporting partial specification of processes, which in turn enables behavior classification (see Sect. 6). In general, abstraction not only reduces ambiguity, but also increases the power of languages.

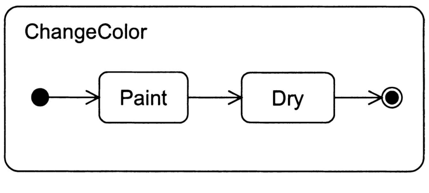

Fig. 1. Example UML 2 flow model

2 PSL and semantics

The approach taken in this paper is to use language elements referring directly to runtime execution, for example the executions of Paint and Dry, sometimes called a semantic domain. In particular, the idea is to define a simple set of concepts that cover all possible runtime execution traces, then use these concepts to specify which execution traces are allowed. For example, if the arrow between Paint and Dry under ChangeColor rules out any other behaviors occurring in between, then a statement can be written to only accept execution traces where that is the case. At runtime, exactly one of the allowed traces will actually happen for each execution of ChangeColor, and it will conform to the ChangeColor specified in this way.

The runtime representation in this paper is PSL, which is designed to facilitate correct and complete exchange of process information. It is based on a long period of research stemming from the situation calculus [21, 28] and enterprise modeling [5, 6]. It has been applied in scheduling, process modeling, process planning, production planning, simulation, project management, workflow, and business process reengineering. PSL is project 18629 at the International Organisation of Standardization, and part of the work is a Draft International Standard.

PSL has a rigorously-developed semantics using first-order logic, and is based on a three-step methodology: identifying intuitions, refining them in mathematical structures, and then defining a logical language for the intuitions [9, 12]. Specifically, we first chose intuitions about executing processes arising from applications and existing languages [29]. These restricted the scope of the effort, and served as informal requirements. Next, we mapped each intuition to an element of some mathematical structure. These are defined either by specifying some class of algebraic or combinatorial structures, or by extending classes of structures defined for other parts of PSL. Examples include graphs, linear orderings, partial orderings, groups, and vector spaces [9, 11]. Once we were satisfied that the class of structures reflected our intuitions, we wrote axioms and definitions in first order logic to formalize the original intuitions, guided by the mathematical representation [10]. Finally, we proved that the structures were isomorphic to the extensions of the predicates in the logical language [11]. This process of using well-understood mathematical structures in the translation of intuitions to logic validates that the language does what we expect.

For the purposes of exposition, this article expresses PSL concepts in UML class diagrams. These are intended as a guide to the logical definition and mathematical structures, rather than a replacement. They sometimes introduce classes that are implied by PSL, but not explicitly represented in the logic. These could be added to PSL if needed. The diagrams always use binary associations, even when the corresponding PSL concept is ternary, to simplify presentation. For those wishing to read the logical definition, it should be noted that it happens to be in the Knowledge Interchange Format (KIF) [8, 13], but it could be any first-order logic notation, for example as used traditionally in mathematics, or UML's Object Constraint Language (OCL) [26]. It could also be expressed in a notation-independent model as in the second version of OCL [25], or Common Logic (CL) [4].

When PSL concepts are used to specify particular behaviors, such as ChangeColor in Fig. 1, the format is also KIF, since these specifications are first-order constraints on runtime execution using the predicates defined by PSL. However, they could also be expressed in OCL or other first-order logic language. This article uses KIF for behavior specification, to give process modelers a sense of how their intentions are rigorously specified using the format in which PSL is currently defined.

3 Basic PSL concepts

This section introduces the most basic PSL concepts. They are simple and general so they can be used repeatedly as the basis for disambiguation of many kinds of behavior model. In fact, most of PSL consists of patterns expressed in terms of the core concepts. Section 3.1 covers activities and their execution. Section 3.2 shows how they are composed.

3.1 Occurrences and activities

Since the primary purpose of behavior specification is to constrain runtime execution, we begin with these two as the most basic concepts: ACTIVITY (a behavior specification) and OCCURRENCE (a runtime execution of a behavior specification). They are shown as a UML model in Fig. 2.

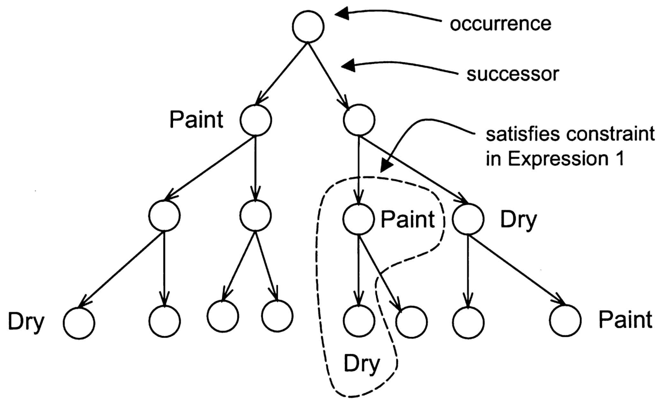

Fig. 3. PSL occurrence tree

A PSL activity is a reusable behavior, for example, ChangeColor or Paint, and is equivalent to the UML 2 concept called Behavior. A PSL occurrence is a runtime execution of an activity. It has no analog in UML, because UML does not have a direct model of runtime execution yet. The PSL SUCCESSOR relation associates occurrences with each other to represent all temporal orderings of runtime execution of activities whether they conform to a behavior specification or not, and even including orderings that are physically impossible. The relation forms a tree where every occurrence has exactly one successor for each activity, indicating the possibility of that activity happening next, so the branches represent possible execution traces. Figure 3 shows a partial tree with occurrences as small circles, and executions of Paint and Dry at various points. The occurrence tree contains all possible executions of Paint and Dry at any time. This includes branches where they are not executed in the order specified in ChangeColor, do not execute at all, execute multiple times, and where other activities are interposed between them.

A behavior specification in PSL identifies those parts of the occurrence tree that conform to the behavior. For example, one of the aforementioned interpretations of the ChangeColor specification in Fig. 1 is that all executions of Paint must be followed by an execution of Dry, even if they are not due to ChangeColor. The occurrences conforming to this semantics need to satisfy the constraint in Expression 1, expressed in KIF. The PSL function SUCCESSOR is defined to return the successor that is an

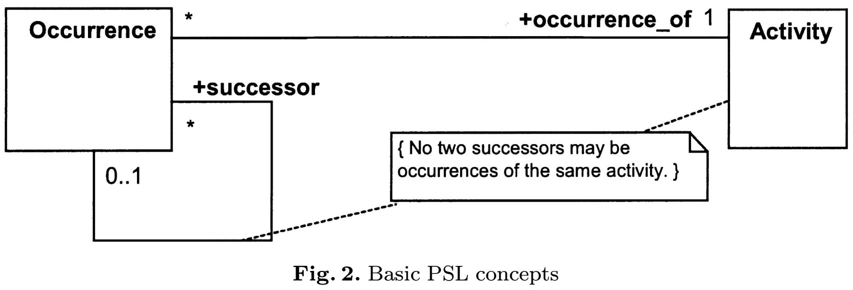
Occurrence
*
+occurrence_of 1
Activity
+successor
*
0..1
{ No two successors may be occurrences of the same activity. }

Fig. 2. Basic PSL concepts

occurrence of a given activity, Dry in this case.[1] The function LEGAL returns a Boolean telling if the occurrence is allowed under some behavior specification. The expression says that if an occurrence of Paint is allowed, then an occurrence of Dry immediately after it is allowed. The part of the occurrence tree in Fig. 3 in the dashed line is permitted under this constraint. The expression is not as strong as the original intention that Dry must always follow Paint, which requires additional constraint against other activities being legal after Paint. See Expression 26 in Sect. 7.

```
(activity Paint)
(activity Dry)
(forall (?occPaint)
  (implies (and (occurrence_of ?occPaint Paint)
                (legal ?occPaint))
  (legal (successor Dry ?occPaint))))
```

Expression 1: Constraint for Fig. 3

The original UML specification of ChangeColor Fig. 1, however, is only intended to constrain those occurrences of Paint and Dry that happen in executions of ChangeColor, not all occurrences of Paint and Dry that ever happen, as in Expression 1. PSL provides a representation of activity composition for this purpose, as addressed in the next section.

[1] The UML model in Fig. 2 defines successor as a binary association for simplicity. Using this model, the equivalent OCL for Expression 1 would be:

```
Context Occurrence inv: self.allinstances->forall( occ1 |
  occ1.occurrence_of = Paint and legal(occ1)
  implies successor->exists(occ2 | occ2.occurrence_of = Dry
                                   and legal(occ2)))
```

3.2 Subactivities

PSL activities may be composed of others, which means that the execution of one activity may involve the execution of another. This is represented by the SUBACTIVITY relation shown in the upper right of Fig. 4. Subactivities may be executed during an occurrence of the superactivity. ACTIVITY is factored into primitive and complex, which are activities that may have subactivities, and activities that may not, respectively. OCCURRENCE is subtyped the same way.[2] The SUCCESSOR relation is restricted to occurrences of primitive activities, following the PSL principle of representing runtime traces at the most complete and detailed level.

```
(activity ChangeColor)
(subactivity Paint ChangeColor)
(subactivity Dry ChangeColor)
```

Expression 2: Subactivities from Fig. 1

The ChangeColor subactivities are written in Expression 2. It means that the executions of ChangeColor may be composed of executions of Paint and Dry, but does not specify what order they occur in, or even that they occur

[2] Complex activities do not have an explicit relation in PSL, even though it is included in our conceptual model for explanatory purposes. It is implied by the SUBACTIVITY relation and could be added to PSL if needed.

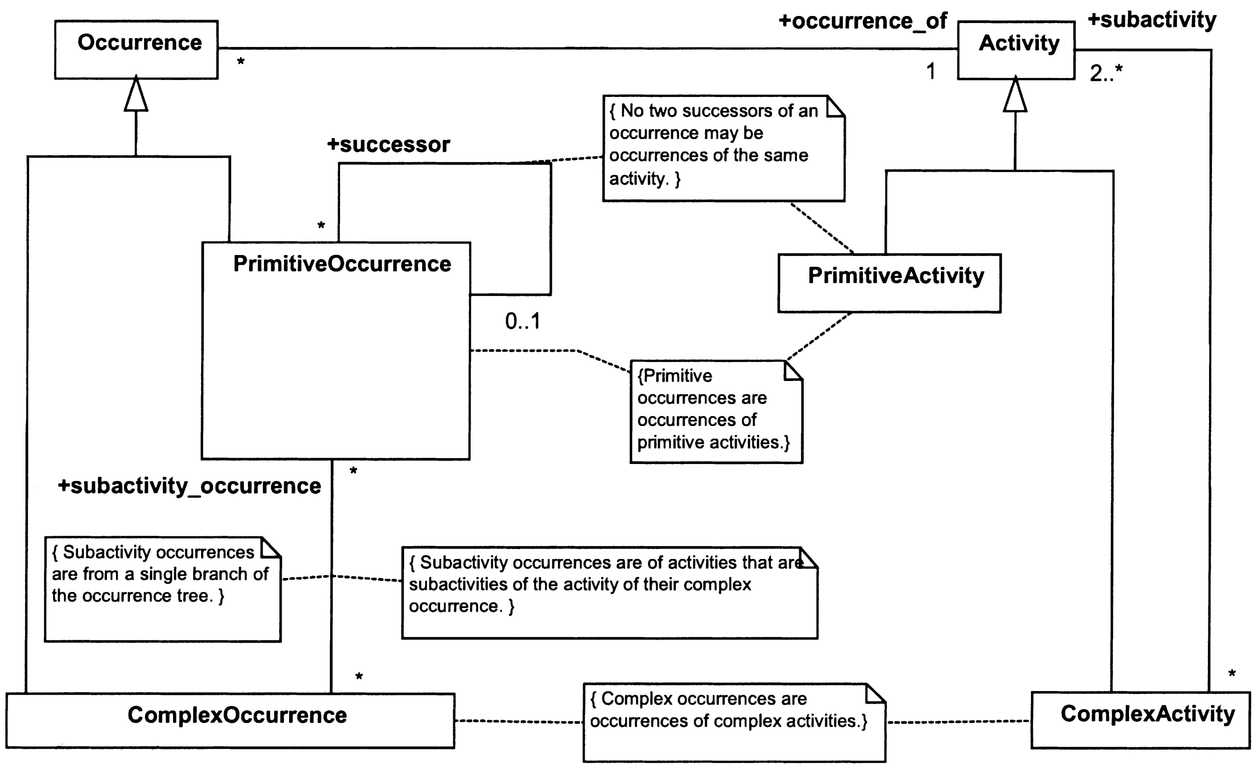
Occurrence
*
+occurrence_of
1
Activity
+subactivity
2..*
+successor
{ No two successors of an occurrence may be occurrences of the same activity. }
*
PrimitiveOccurrence
PrimitiveActivity
0..1
{Primitive occurrences are occurrences of primitive activities.}
+subactivity_occurrence
*
{ Subactivity occurrences are from a single branch of the occurrence tree. }
{ Subactivity occurrences are of activities that are subactivities of the activity of their complex occurrence. }
*
*
ComplexOccurrence
{ Complex occurrences are occurrences of complex activities.}
ComplexActivity

Fig. 4. PSL concepts for behavior composition

at all. An execution of ChangeColor conforming to the statements has the following possible traces:

– Paint, Dry
– Dry, Paint
– Paint
– Dry
– (traces with repetitions of above)
– (nothing)

Executions of complex activities, like ChangeColor, are called complex occurrences. A complex occurrence is represented by selecting primitive occurrences of subactivities from a single branch of the occurrence tree. The relation between complex and primitive occurrence is represented in the conceptual model as the SUBACTIVITY_OCCURRENCE relation between PRIMITIVEOCCURRENCE and COMPLEXOCCURRENCE, as shown in Fig. 4.[3,4] Figure 5 presents it graphically by encircling the subactivity occurrences of ChangeColor and ConstructHouse with dashed lines.

Complex occurrences are limited to a single branch of the occurrence tree because any given execution of a complex activity will perform a completely defined set of steps in a completely defined order. If a complex activity has some indeterminacy, as happens if there is concurrency in the flow model, then each branch is a separate occurrence of the complex activity. See Figs. 14 and 16 in Sect. 5.1. The branching shown in Fig. 5, for example after Dry on the right, represents interrupted activities. These will be discussed in a later article.

[3] Complex occurrences do not have an explicit relation in PSL, even though it is included in our conceptual model for explanatory purposes. It is implied by the SUBACTIVITY_OCCURRENCE relation and could be added to PSL if needed.

[4] The subactivity relations for ChangeColor are equivalent to the constraint:

```
(forall (?occChangeColor)
   (implies (occurrence_of ?occChangeColor ChangeColor)
            (forall (?subOcc)
               (implies (subactivity_occurrence ?subOcc
                                                ?occChangeColor)
                        (or (occurrence_of ?subOcc Paint)
                            (occurrence_of ?subOcc Dry))))))
```

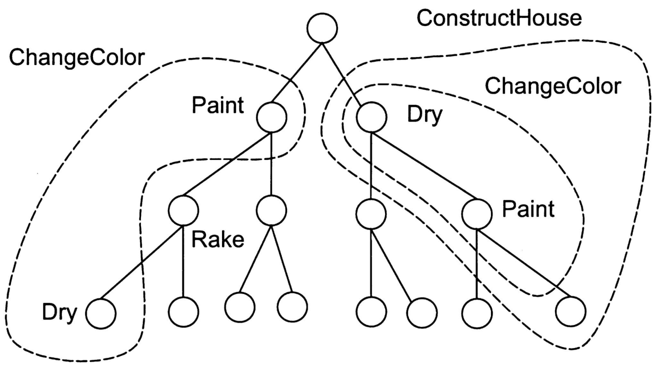

Fig. 5. Occurrence tree with complex occurrences

Occurrences that are not part of the activity may happen between subactivity occurrences. For example, there may be other occurrences between painting and drying that are not part of ChangeColor, even though painting and drying are part of a single ChangeColor execution, as shown on the left side of Fig. 5. This reflects the principle that the occurrence tree represents everything that can possibly occur at runtime, which might include other activities happening while ChangeColor is executing.

A primitive occurrence may belong to more than one complex occurrence, in part because complex activities can be broken down into other complex activities. Figure 5 shows this with Dry on the right under both ChangeColor and the larger ConstructHouse activity occurrences.

The UML diagram for ChangeColor in Fig. 1 does not commit to whether Paint and Dry break down further into other behaviors, but PSL requires that some activities are chosen as primitive. This is because primitive activities give a complete picture of runtime effects of the specification. Each sequence of primitive activity occurrences is a possible execution trace, and all sequences together represent everything of concern that can possibly happen at runtime.

However, the choice of what level of detail to take as primitive depends on the application. For example, a planning system for robots may treat the activity of moving a robot arm as primitive, but a mechanical system for the robot will break this down into finer-grained electro-mechanical behaviors. PSL does not dictate what level of detail should be primitive, and it is an area of future work in PSL to support multiple levels of abstraction at once. In this example, we assume Paint and Dry are primitive, shown in Expression 3.

```
(primitive Paint)
(primitive Dry)
```

Expression 3: Primitive activities from Fig. 1

4 Ordered processes

This section addresses constraints on sequences of subactivity execution, building on the specification of activities, subactivities, and occurrences described in the previous sections. Section 4.1 describes the strong and weak forms of sequences supported in PSL, a distinction that adds expressiveness applied to behavior classification in Sect. 6. Section 4.2 proposes an extension to PSL for object flow and shows how to parameterize activities to use it. Section 4.3 covers alternative flows and Sect. 4.4 shows how timing constraints on sequences are expressed in PSL.

4.1 Complete and partial sequencing

A common interpretation of Fig. 1 is that ChangeColor will result in an execution of Paint and then immediately an execution of Dry, with no other execution in between. This is one answer to the first question in Sect. 1. The interpretation is represented in PSL using the NEXT_SUBOCC relation between primitive occurrences, shown in Fig. 6, an extension of the model in Fig. 4. It sequences subactivity occurrences under a complex activity occurrence.

The NEXT_SUBOCC relation is applied with quantifications over all the complex occurrences of the activity being constrained, as shown in Expression 4.[5] It says that all occurrences of ChangeColor will have occurrences of Paint and Dry that happen one after the other, with no other subactivities of ChangeColor in between. For example, the right side of Fig. 7 shows a complex occurrence of ChangeColor that is illegal under Expression 4. However, the NEXT_SUBOCC relation allows other activities between Paint and Dry as long as they are not executing under ChangeColor, as shown on the left side of Fig. 7.

Another interpretation of the UML control flow link is that additional activities could occur between Paint and Dry under ChangeColor. In this case the specification is only intended as a general constraint on ChangeColor

[5] The LEGAL predicate is implied by NEXT_SUBOCC in other axioms.

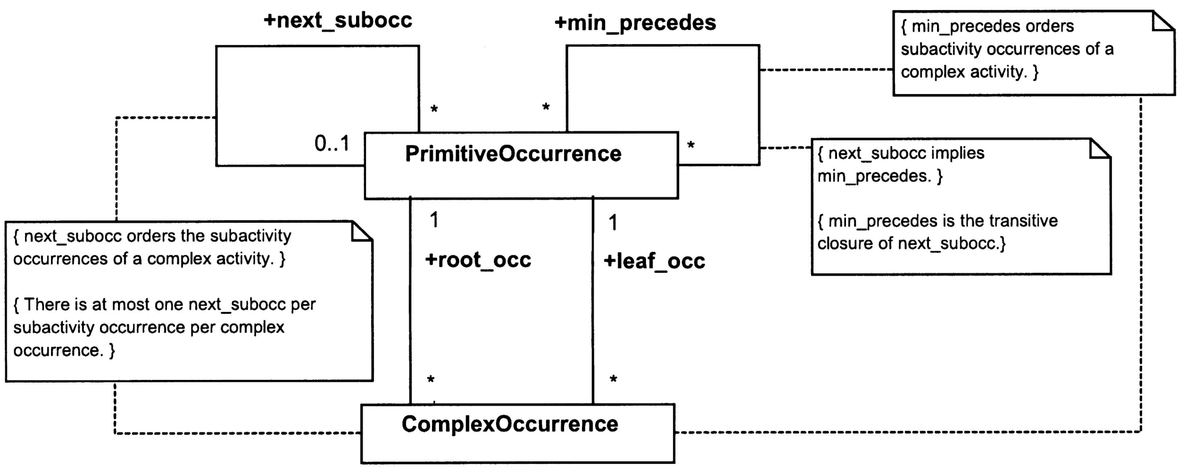
+next_subocc
+min_precedes
{ min_precedes orders subactivity occurrences of a complex activity. }
*
*
0..1
PrimitiveOccurrence
*
{ next_subocc implies min_precedes. }
{ min_precedes is the transitive closure of next_subocc.}
1
1
+root_occ
+leaf_occ
{ next_subocc orders the subactivity occurrences of a complex activity. }
{ There is at most one next_subocc per subactivity occurrence per complex occurrence. }
*
*
ComplexOccurrence

Fig. 6. PSL flow concepts

```
(forall (?occChangeColor)
  (implies
    (occurrence_of ?occChangeColor ChangeColor)
    (exists (?occPaint ?occDry)
       (and (occurrence_of ?occPaint Paint)
            (occurrence_of ?occDry Dry)
            (subactivity_occurrence ?occPaint ?occChangeColor)
            (subactivity_occurrence ?occDry ?occChangeColor)
            (next_subocc ?occPaint ?occDry ChangeColor)))))
```

Expression 4: Control flow from Fig. 1 using NEXT_SUBOCC

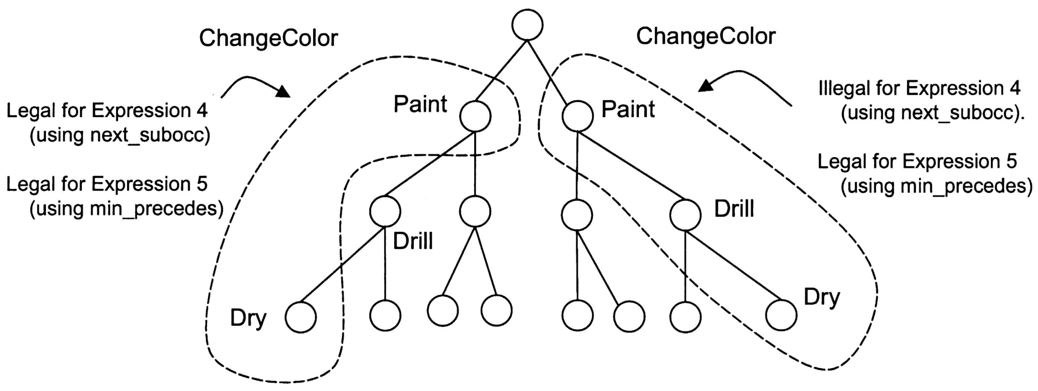

Fig. 7. Conformance to NEXT_SUBOCC and MIN_PRECEDES

```
(forall (?occChangeColor)
  (implies
    (occurrence_of ?occChangeColor ChangeColor)
    (exists (?occPaint ?occDry)
       (and (occurrence_of ?occPaint Paint)
            (occurrence_of ?occDry Dry)
            (subactivity_occurrence ?occPaint ?occChangeColor)
            (subactivity_occurrence ?occDry ?occChangeColor)
            (min_precedes ?occPaint ?occDry ChangeColor)))))
```

Expression 5: Control flow from Fig. 1 using MIN_PRECEDES

```
(forall (?occChangeColor)
  (implies
   (occurrence_of ?occChangeColor ChangeColor)
   (exists (?occPaint ?occDry)
      (and (occurrence_of ?occPaint Paint)
           (occurrence_of ?occDry Dry)
           (subactivity_occurrence ?occPaint ?occChangeColor)
           (subactivity_occurrence ?occDry ?occChangeColor)
           (min_precedes ?occPaint ?occDry ChangeColor)
           (root_occ ?occPaint ?occChangeColor)
           (leaf_occ ?occDry ?occChangeColor)))))
```

Expression 6: Roots and leaves for Fig. 1

and not a complete definition. It is represented in PSL using the MIN_PRECEDES relation between primitive occurrences, as shown in Expression 5. It says that Paint must happen before Dry under ChangeColor, but allows other behaviors under ChangeColor to occur in between. Both sides of Fig. 7 are legal under this constraint. In logical terms, MIN_PRECEDES is the transitive closure of NEXT_SUBOCC. Both order subactivity occurrences under a complex occurrence, so are restricted to a single branch of the occurrence tree.

The actual interpretation in UML 2 is ambiguous regarding NEXT_SUBOCC versus MIN_PRECEDES, because UML 2 supports behavior redefinition, in which an activity can be changed arbitrarily, either by inheritance through a class hierarchy, or by direct redefinition. However, the semantics of NEXT_SUBOCC and MIN_PRECEDES can be used to control arbitrary redefinition for particular behaviors. See Sect. 6.

The UML diagram for ChangeColor in Fig. 1 shows control links between an initial node and Paint, where the initial node is shown as a dot, and also between Dry and a final node, shown as a bullseye. Initial and final nodes are called control nodes in UML 2, and do not support behaviors. They only mark the beginning and end of the flows. The PSL versions of these focus on runtime execution by using the relations ROOT_OCC and LEAF_OCC to indicate which subactivity occurrences are at the beginning and end of a complex occurrence, shown in Expression 6. The root and leaf occurrences are unique for each complex occurrence, as indicated by the multiplicities in Fig. 6.

Since roots and leaves are at the extreme beginning and end of complex occurrences, Expression 6 does not allow any other activity to be introduced before Paint or after Dry under ChangeColor, as shown on the left side of Fig. 8. Specifying roots and leaves is similar to using the NEXT_SUBOCC relation in the sense that they prevent introducing new behaviors at particular points in the flow.

It is beneficial if the specification of complex activities such as ChangeColor are independent of whether the subactivities are complex or not. Then Paint and Dry can be changed to be complex without affecting the constraints for ChangeColor. Expressions 4, 5, and 6 assume Paint and Dry are primitive, because it links their occurrences with NEXT_SUBOCC. We can loosen this constraint by using NEXT_SUBOCC on the ROOT_OCC and LEAF_OCC of occurrences of Paint and Dry. These relations identify the primitive roots and leaves of complex activities, even if they are deeply nested. For example, in Fig. 5 the root occurrence of the ConstructHouse execution is the same as the root of ChangeColor under it. Also, the root and leaf of a primitive activity occurrence is just itself. Using these properties of roots and leaves, the quantification over ChangeColor occurrences can be made independent of whether the subactivities are complex or not, as shown in Expression 7.

The last three statements in Expression 7 say:

1. The first occurrence under the Paint occurrence (which is the Paint occurrence itself if Paint is primitive) is the same as the first occurrence under Change-

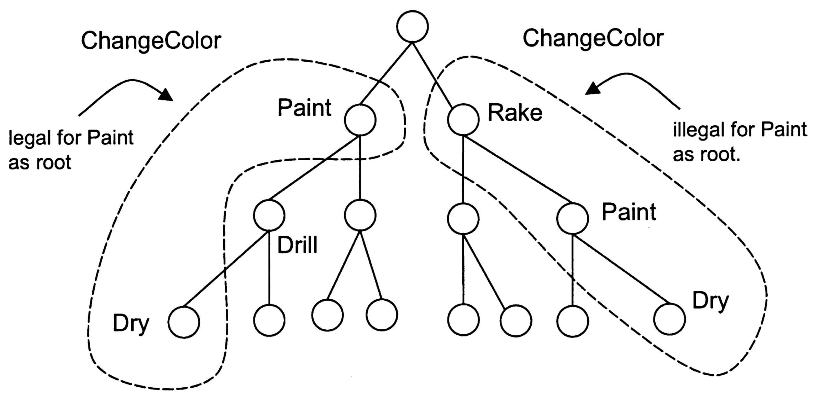

Fig. 8. Conformance to ROOT_OCC

```
(forall (?occChangeColor)
  (implies
    (occurrence_of ?occChangeColor ChangeColor)
    (exists (?occPaint ?occDry
             ?rootPaint ?leafPaint
             ?rootDry ?leafDry)
       (and (occurrence_of ?occPaint Paint)
            (occurrence_of ?occDry Dry)
            (subactivity_occurrence ?occPaint ?occChangeColor)
            (subactivity_occurrence ?occDry ?occChangeColor)

            (root_occ ?rootPaint ?occPaint)
            (leaf_occ ?leafPaint ?occPaint)
            (root_occ ?rootDry ?occDry)
            (leaf_occ ?leafDry ?occDry)

            (root_occ ?rootPaint ?occChangeColor)
            (next_subocc ?leafPaint ?rootDry ChangeColor)
            (leaf_occ ?leafDry ?occChangeColor)))))
```

Expression 7: Using roots and leaves for primitive and complex subactivities

Color. This means painting is the first step in ChangeColor.

2. The last occurrence under Paint is just before the first occurrence under Dry. This means drying happens after painting.
3. The last occurrence under Dry is the same as the last occurrence under ChangeColor. This means drying is the last step in ChangeColor.

Expression 7 has the same meaning as Expression 6, but works whether Paint and Dry are primitive or complex. Any expression constraining the execution order of primitive activities can be generalized in this way by using roots and leaves instead.

4.2 Object flow and parameterized activities

Most process models support the notions of input and output, which are data or objects provided to a behavior execution before it starts, and data produced when it finishes, respectively. Figure 9 shows an example using one of the UML 2 notations for object flow, where cars flow between a factory and trucking process. PSL supports the concept of objects participating in an activity

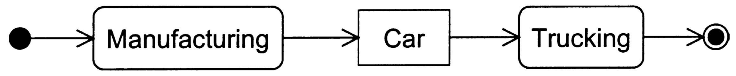

Fig. 9. Example UML object flow

```
(forall (?flowObject ?occ2)
  (implies (input ?flowObject ?occ2)
           (or (exists (?occ1 ?activity)
                  (and (output ?flowObject ?occ1)
                       (min_precedes ?occ1 ?occ2 ?activity)))
               (exists (?occ)
                  (and (input ?flowObject ?occ)
                       (subactivity_occurrence ?occ2 ?occ))))))
```

Expression 8: General constraint on inputs and outputs

execution, but not specifically as inputs or outputs. This section suggests that they are independent of other PSL concepts and proposes an extension. It also describes parameterized activities so inputs and outputs can vary by each occurrence of an activity.

Although inputs and outputs are taken as basic in process modeling, it is less obvious from the point of view of runtime execution that these are primitive concepts, rather than derivations from existing ones. At first glance it seems like an input is any object participating in an activity occurrence that also participates in some other occurrence earlier in time, and the reverse for output. However, such an object is not always an input or an output. For example, suppose a factory and trucking company use the same forklift rental service and happen to use the same individual forklift to bring the car to the loading dock and remove it from the loading dock, with the forklift returned to the rental company in between. Both processes use the same forklift, one after the other, but the factory is not providing the forklift to the truckers as in input.

Another straightforward approach is to use preconditions and postconditions, which PSL supports on activity executions. Input can be defined as a kind of precondition on an occurrence requiring a particular object to be available to the execution in some specified way, and to define an output as a postcondition that a particular object is available in some specified way from the execution to participate in other executions. For example, a factory might output a car by putting it on a dock, which is picked up as input by a trucking activity. The postcondition for the factory activity is that cars are on the dock, which is also the precondition for the trucking activity. However, the particular way an output is provided to an input, the dock in this example, is too specific a constraint for input and output. The factory and trucking processes may be changed to use the front driveway as the point of exchange, but this does not change the inputs and outputs of the factory and trucking activities, which are still cars. Inputs and outputs must be independent of the definition of the behavior taking the inputs and providing the outputs.

These examples suggest that the notions of input and output are justifiably primitive for a language describing runtime execution, in particular they cannot be reduced to existing PSL concepts such as ordering and pre/postconditions of activity occurrences. The relation to these existing concepts provides constraints we can use to make the formalization more specific. For example, activities requiring input objects must be executed after others providing those objects as outputs, or to be due to execution of a complex activity having the object as in input, as defined in Expression 8. It uses the weaker MIN_PRECEDES ordering, but the stronger NEXT_SUBOCC relation can be stated separately as needed.[6,7] With these definitions, inputs and outputs of activities can be defined in terms of their occurrences, for example in Expression 9, and used in object flow specifications as in Expression 10, which defines the process of Fig. 9 as a complex activity called CarIndustry.

More work is in progress on formalizing inputs and outputs, for example to relate them to pre/postconditions, goals, implementation, and partial behavior specification. There are also other application areas from which to draw examples, such as embedded real-time. These will be reported in a separate article.

In usual PSL fashion, the expressions so far in this section refer to particular runtime objects rather than classes. The PSL core supports only "ground" activities, that is, those with specific objects as inputs and outputs, rather than classes of objects. A ground activity operates the same way each time it occurs. For example, a ground activity for picking up an object would either pick up the same object each time it occurred, or go through the same decision process about which object to pick up. In the fac-

[6] The axiom allows flows across activities, because `?activity` could be a wider complex activity than the one immediately containing the source and target of the flow, or even one that partially overlaps. Most flow models do not allow these, so the axiom can be tightened to prevent it. This is another unstated characteristic of flow models that PSL can disambiguate and use for addtional expressiveness. This will be addressed in future work.

[7] Other constraints are sometimes applied to data flow, for example, to prevent two inputs from using the data or objects from the same output. These relate to issues of resource contention that are not addressed by the above expressions. For example, a phone number may be output from one activity and input to multiple others, whereas a depletable object like fuel may not. PSL has additional relations to address resource contention and consumption.

```
(forall (?occFactory)
   (implies (occurrence_of ?occFactory Factory)
            (exists (?car)
               (and (Car ?car)
                    (output ?car ?occFactory)))))

(forall (?occTrucking)
   (implies (occurrence_of ?occTrucking Trucking)
            (exists (?car)
               (and (Car ?car)
                    (input ?car ?occTrucking)))))
```

Expression 9: Constraints on input and output for specific activities

```
(forall (?occCarIndustry)
   (implies
     (occurrence_of ?occCarIndustry CarIndustry)
     (exists (?occFactory ?occTrucking ?car)
        (and (occurrence_of ?occFactory Factory)
             (occurrence_of ?occTrucking Trucking)
             (subactivity_occurrence ?occFactory
                                     ?occCarIndustry)
             (subactivity_occurrence ?occTrucking
                                     ?occCarIndustry)
             (output ?car ?occFactory)
             (input ?car ?occTrucking)
             (next_subocc ?occFactory ?occTrucking
                          CarIndustry)))))
```

Expression 10: Object flow constraints using input and output

tory example above, the same car would be output at each execution of Factory.

Most process models define activities that operate differently at each occurrence, usually based on the inputs they receive, and consequently also provide varying outputs. In PSL these are represented as ground activities, one for each pair of input and output instances. Rather than explicitly enumerate ground activities for all pairs of input and output instances, a function can be defined that produces them. Expression 11 defines a function Trucking that yields a trucking activity parameterized by a car to move and a receipt for the cost.

```
(forall (?x ?y)
   (iff (and (Car ?x)
             (Receipt ?y))
        (activity (Trucking ?x ?y))))

(forall (?occ)
   (implies (occurrence_of ?occ (Trucking ?x
                                          ?y))
            (and (input ?x ?occ)
                 (output ?y ?occ))))
```

Expression 11: Parameterized activities

With similar axioms for Factory to define its output, the OCCURRENCE_OF statements in Expression 10 for CarIndustry are replaced with Expression 12.

```
(occurrence_of ?occFactory (Factory ?car))
(occurrence_of ?occTrucking (Trucking ?car
                                      ?receipt))
```

Expression 12: Using parameterized activities in Expression 10

where the variable `?receipt` is also existentially quantified.[8]

The definitions of Factory and Trucking use the input and output relations to refer to the particular objects required and provided by executions. The question of how exactly objects are exchanged as inputs and outputs, for example whether the cars are on the dock or driveway, is a matter of system design, and is not restricted by the input and output relations. This is analogous to choosing a network protocol to exchange information after the direction and kind of information flow has been determined. One design might reduce coordination overhead between

[8] The input/output axiom does not require that the `?receipt` output is used as an input, so it does not affect Expression 10.

the factory and trucks, for example, by using the cars' GPS systems to tell where they should be picked up. Another design might reduce cost by choosing a constant location. The particulars of the exchange are detail added under the abstraction of input and output.

4.3 Decision points and merges

UML activity models, like all flow languages, support flows that split and come together. For example, the runtime effect of UML decision points is to choose between alternative flow directions, and UML forks to initiate concurrent flows. These are called control nodes in UML 2 because they coordinate the execution of behaviors. Figure 10 shows a decision point, notated as the diamond on the left. Exactly one path from painting to cleanup will be taken by any particular execution of the model, depending on the result of inspection.[9] The diamond on the right is called a merge. Any control or data arriving on its incoming edges is passed to its outgoing edge. Decision points and merges are distinguished notationally by the number of incoming and outgoing edges.

To support flow choices, PSL has constructs for representing aspects of the state of the world before and after each activity execution in the occurrence tree. These states are more general than UML states, which only apply to specific objects. The PSL representation of decisions uses states to express which branches of the occurrence tree are allowed under the specification. The relation HOLDS tells what state of the world is the case after a specific activity execution, that is, at one point in the occurrence tree. Expression 13 below uses it to specify what executions will occur following painting. Only one of the two implications will constrain the occurrences in each execution of ChangeColor, because the antecedent of the implications are written to exclude each other in Expression 13.

It is important to distinguish splits in a flow model from branching in the PSL occurrence tree. In the decision point example above, each execution of ChangeColor will result in a sequence of activity occurrences that has no alternatives, because at each execution the guard will choose exactly one path. Even though the flow model merges flows, the branches of the occurrence tree never join back together. Each point in the execution has a unique series of previous activity executions leading to it, as shown in Fig. 11.

4.4 Timing constraints

There is no restriction so far on how much time can elapse between the executions of Paint and Dry under ChangeColor (question 2 from Sect. 1). This is true in UML also, but the common interpretation is that there will be some limit to how much time passes, even if this limit is not given. Regardless of the specification language, these expectations should be explicitly added if the implementation is required to fulfill them. PSL has functions for the begin and end time of behavior executions, see example in Expression 14.

This means that the amount of time between the ending of the Paint occurrence and the beginning of the Dry

9 UML does not define semantics for models that have no guards, or for guards that are not mutually exclusive.

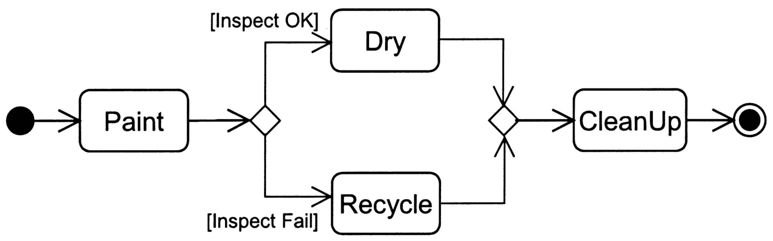

Fig. 10. UML decision point and merge

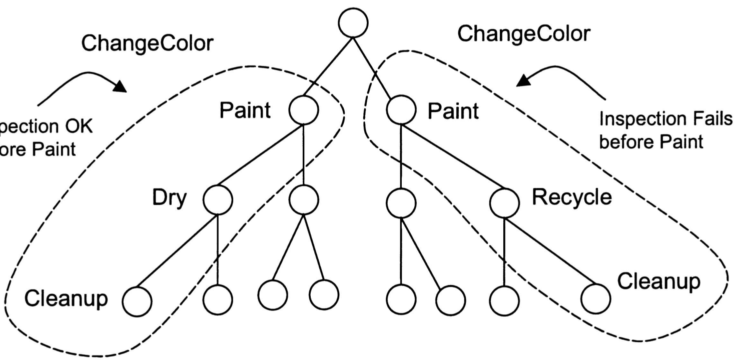

Fig. 11. Occurrences for Fig. 10 and Expression 13

```
(state InspectionOK)

(state InspectionFailed)

(forall (?occChangeColor)
  (implies
    (occurrence_of ?occChangeColor ChangeColor)
    (exists (?occPaint ?occCleanUp)
       (and (occurrence_of ?occPaint Paint)
            (subactivity_occurrence ?occPaint ?occChangeColor)
            (occurrence_of ?occCleanUp CleanUp)
            (subactivity_occurrence ?occCleanUp
                                    ?occChangeColor)

            (root_occ ?occPaint ?occChangeColor)

            (implies
               (and (holds InspectionOK ?occPaint)
                    (not (holds InspectionFailed ?occPaint)))
               (exists (?occDry)
                   (and (occurrence_of ?occDry Paint)
                       (subactivity_occurrence ?occDry
                                           ?occChangeColor)
                        (next_subocc ?occPaint ?occDry
                                     ChangeColor)
                        (next_subocc ?occDry ?occCleanUp
                                     ChangeColor))))

               (implies
                   (and (holds InspectionFailed ?occPaint)
                       (not (holds InspectionOK ?occPaint)))
                   (exists (?occRecycle)
                      (and (occurrence_of ?occRecycle Recycle)
                       (subactivity_occurrence ?occRecycle
                                          ?occChangeColor)
                           (next_subocc ?occPaint ?occRecycle
                                        ChangeColor)
                           (next_subocc ?occRecycle ?occCleanUp
                                        ChangeColor))))

               (leaf_occ ?occCleanUp ?occChangeColor)))))
```

Expression 13: Flow constraints for Fig. 10

```
(forall (?occChangeColor)
   (implies
      (occurrence_of ?occChangeColor ChangeColor)
      (exists (?occPaint ?occDry)
         (and (occurrence_of ?occPaint Paint)
              (occurrence_of ?occDry Dry)
              (subactivity_occurrence ?occPaint ?occChangeColor)
              (subactivity_occurrence ?occDry ?occChangeColor)
              (next_subocc ?occPaint ?occDry ChangeColor)
              (lesser (timeduration (endof ?occPaint)
                                    (beginof ?occDry))
                      10)))))
```

Expression 14: Time constraint

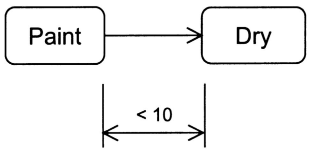

Fig. 12. UML time constraint

occurrence must be less than 10, but does not specify the unit of measure. PSL leaves units of measure open for other extensions to define and add to Expression 14. UML 2 also has a time model and a way to refer to the begin and end time of the execution. The notation for this is not standardized, but might be shown as in Fig. 12.

5 Unordered processes

This section addresses unordered activity execution. It is a benefit of the PSL occurrence tree that the intentional absence of ordering constraints is represented as just another kind of constraint, because unordered execution creates more legal branches to account for. PSL distinguishes two kinds of unordered execution, the weaker form described in Sect. 5.1 and the stronger form in Sect. 5.2.

5.1 Forks and joins

UML forks are another kind of split in flow, but unlike decisions, forks initiate multiple concurrent flows, rather than choose among them. Figure 13 shows a fork, notated as a vertical bar on left. The executions of Dry and Cleanup can occur in any order, or can overlap in time. However, Paint must complete execution before Dry or Cleanup start. The vertical bar on right is called a join. Control or data must arrive on both incoming edges to be passed to the outgoing edge. This means Notify can only start after Dry and Cleanup are finished. Forks and joins are distinguished notationally by the number of incoming and outgoing edges.

In the example above, the fact that there is no constraint on the ordering of execution between Dry and Cleanup means that when ChangeColor is executed there must be multiple possible complex occurrences of ChangeColor to cover the various ordering combinations. This covers multiple branches of the occurrence tree, as shown in Fig. 14. It illustrates the weak form of unordered execution, which prevents Dry and Cleanup from overlapping in time (compare to concurrent activities in Sect. 5.2). Specifically, NEXT_SUBOCC yields multiple occurrences under the same complex activity, one for each complex occurrence. For example, the Paint occurrence in Fig. 14 has two NEXT_SUBOCC occurrences under the ChangeColor activity under separate complex occurrences. Complex occurrences that cover the possible orderings of a single executions are collectively called an activity tree.

To represent this in PSL, we could use the techniques already introduced for decision points, quantifying over all occurrences of ChangeColor with a disjunction to require all combinations of runtime execution order as possibilities. This is fine for a simple flow model, but the number of orderings becomes very large if there are many steps on the tines of a fork. We could define a "macro" facility that expanded out to all the combinations. However, the explicit representation of constraints or absence of them would disappear when the macros were expanded.

Another approach is to define relations expressing patterns in the tree corresponding to the constraints. These patterns appear most clearly when using control constructs individually, as in the example of Fig. 15. This simple use of fork and join corresponds to the PSL tree shown in Fig. 16. Notice that every occurrence has occurrences below it (NEXT_SUBOCC) that are the same activities as the ones next to it at the same level of the

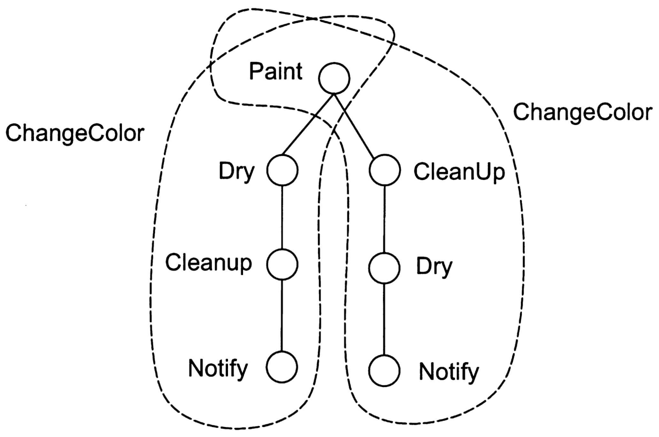

Fig. 14. Activity tree

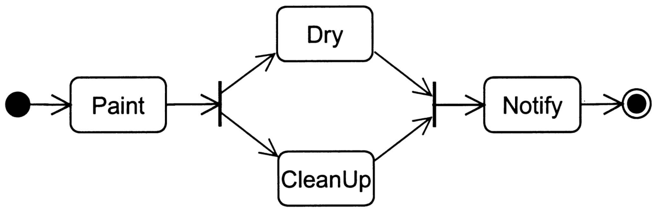

Fig. 13. UML fork and join

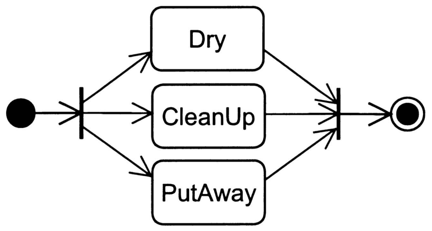

Fig. 15. Simple UML fork and join

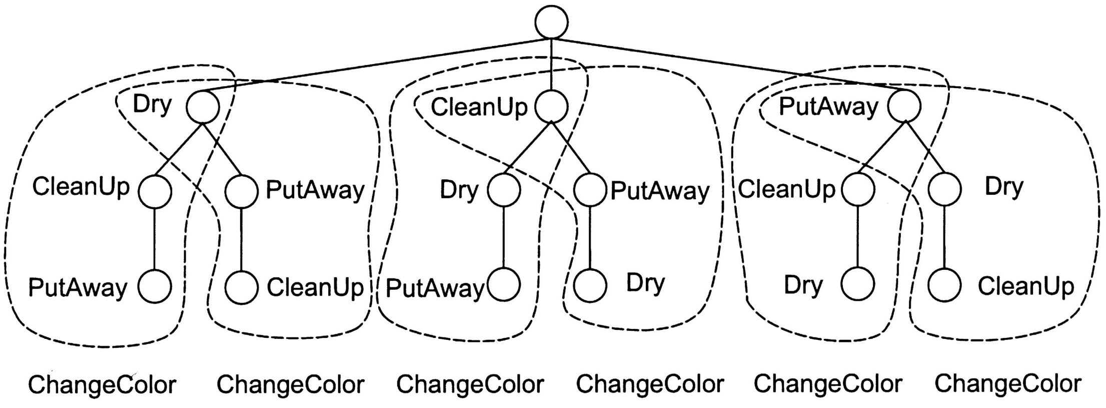

Fig. 16. Activity trees for Fig. 15

```
(forall (?occ ?activity)
   (iff (sibling_iso_next_subocc ?occ ?activity)
        (forall (?occ1)
           (iff (next_subocc ?occ ?occ1 ?activity)
                (exists (?occ2)
                   (and (sibling ?occ ?occ2 ?activity)
                        (iso_occ ?occ1 ?occ2)))))))

(forall (?occ1 ?occ2 ?activity)
   (iff (sibling ?occ1 ?occ2 ?activity)
        (or (exists (?occ3)
               (and (next_subocc ?occ3 ?occ1 ?activity)
                    (next_subocc ?occ3 ?occ2 ?activity)))
            (and (root ?occ1 ?activity)
                 (root ?occ2 ?activity)
                 (exists (?occ4 ?act1 ?act2)
                    (and (= ?occ1 (successor ?act1 ?occ4))
                         (= ?occ2 (successor ?act2 ?occ4))))))))

(forall (?occ1 ?occ2)
   (iff (iso_occ ?occ1 ?occ2)
        (exists (?activity)
           (and (occurrence_of ?occ1 ?activity))
                (occurrence_of ?occ2 ?activity))))
```

Expression 15: Relations for identifying patterns in Fig. 16

tree (SIBLING). For example, Dry on the left has Cleanup and PutAway under it, which are also the siblings of Dry. This local characteristic of the tree is a pattern created by the constraints implicit in Fig. 15. It can be expressed as a relation SIBLING_ISO_NEXT_SUBOCC that forces NEXT_SUBOCC activities to be the same as the SIBLING activities, as shown in Expression 15.

The relations in Expression 15 all refer to the activity under which the pattern is being defined through the variable `?activity`. However, the occurrences of these activities must be in the same "grove" of activity trees, as in Fig. 16, so the options for executing the complex activity are described at the same point in the occurrence tree. We define the SAME_GROVE relation using SIBLING to identify these complex occurrences, as shown in Expression 16.

The above relations are used in Expression 17 to define the pattern in Fig. 16.

Expression 17 ensures that all occurrences under groves of ChangeColor executions satisfy SIBLING_ISO_NEXT_SUBOCC. This technique produces more concise expressions by identifying patterns that can be defined locally at each occurrence in groves of ChangeColor occurrences, SIBLING_ISO_NEXT_SUBOCC in this case. Previously we achieved the same thing by constraining each complex occurrence of ChangeColor individually with a potentially large disjunction for alternative paths.

```
(forall (?occ1 ?occ2)
   (iff (same_grove ?occ1 ?occ2)
        (exists (?activity)
           (and (occurrence_of ?occ1 ?activity)
                (occurrence_of ?occ2 ?activity)
                (sibling (root_occ ?occ1) (root_occ ?occ2)
                         ?activity)))))
```

Expression 16: Constraint for "grove" of activity trees

```
(forall (?occChangeColor)
  (implies
    (occurrence_of ?occChangeColor ChangeColor)
    (forall (?a ?s ?occ1)
       (implies (and (same_grove ?occ1 ?occChangeColor)
                     (occurrence_of ?occ1 ChangeColor)
                     (subactivity_occurrence ?s ?occ1))
                (sibling_iso_next_subocc ?s ChangeColor)))))
```

Expression 17: Constraint for pattern in Fig. 16

The pattern for Fig. 13 is more complex, because the linear orderings, for example between Paint and Dry, mean that SIBLING_ISO_NEXT_SUBOCC does not always apply. The same is the case for conditional splits, as in Fig. 10, which cause some of the executions of ChangeColor occurrences to have different suboccurrences than other executions of ChangeColor. These require a new relation similar to MIN_PRECEDES to allow the linear orderings to be inserted in the SIBLING_ISO_NEXT_SUBOCC pattern, and to allow some of those orderings to be exclusive of each other. These patterns will be described in a future article.

5.2 Concurrency

As described in Sect. 3.1, each branch of the occurrence tree represents a possible runtime trace, and each step in the execution does not overlap in time with other steps before or after it on the branch. This guarantees that effects of the primitive activities are composable. If any steps overlapped, they might have unpredictable interactions that would cause the branch to have equally unpredictable results. For example, the behaviors of lifting one side of a book and lifting the other side will have different results when done separately than together, such as whether an object on the book will fall off. The effects of concurrently executing activities cannot be mechanically composed the way effects of subactivities in a complex activity can.

To represent concurrent activities, PSL aggregates them into one activity. This way the interactions between them can be captured as the effect of a single behavior separate from any of the concurrent activities. This is the strong form of unordered execution in PSL. The aggregated activity is no longer primitive, since it is created from others, but is also not complex, because the effects are not composable. A new class of ACTIVITY called ATOMICACTIVITY[10] is introduced for it, as shown in Fig. 17, a refined version of Fig. 4.

Only atomic occurrences can participate in the SUCCESSOR relation (the occurrence tree), and they cannot have subactivity occurrences, since the concurrent activities aggregated under an atomic no longer produce distinct occurrences with separate effects. Concurrent activities are treated as one for the purposes of representing runtime execution. This is shown in Fig. 17 by replacing PRIMITIVEOCCURRENCE from Fig. 4 with ATOMICOCCURRENCE. For simplicity, PSL represents concurrent activities as subactivities of an atomic activity. This is shown in Fig. 17 by promoting the SUBACTIVITY association to ACTIVITY. Primitive activities are atomic, and consequently are always concurrent with themselves.

A simple application of concurrency is to alter the PSL expression for Fig. 1 to allow other activities to execute during painting that are external to the process of changing color, but that could happen at the same time. These "external" subactivities can be introduced by loosening the constraint in Expression 4 to allow the possibility of another activity executing concurrently with Paint. This is an answer to question 4 in Sect. 1. A convenience function CONC is defined in PSL that produces

[10] This is not "atomic" in the sense of a database transaction, which allows interruption and rollback. When atomic PSL activities occur, they always occur completely, because they are a single node in the occurrence tree. There is a separate PSL extension for interruptable activities, which are represented as a kind of complex activity. See discussion of Fig. 5 in Sect. 3.2.

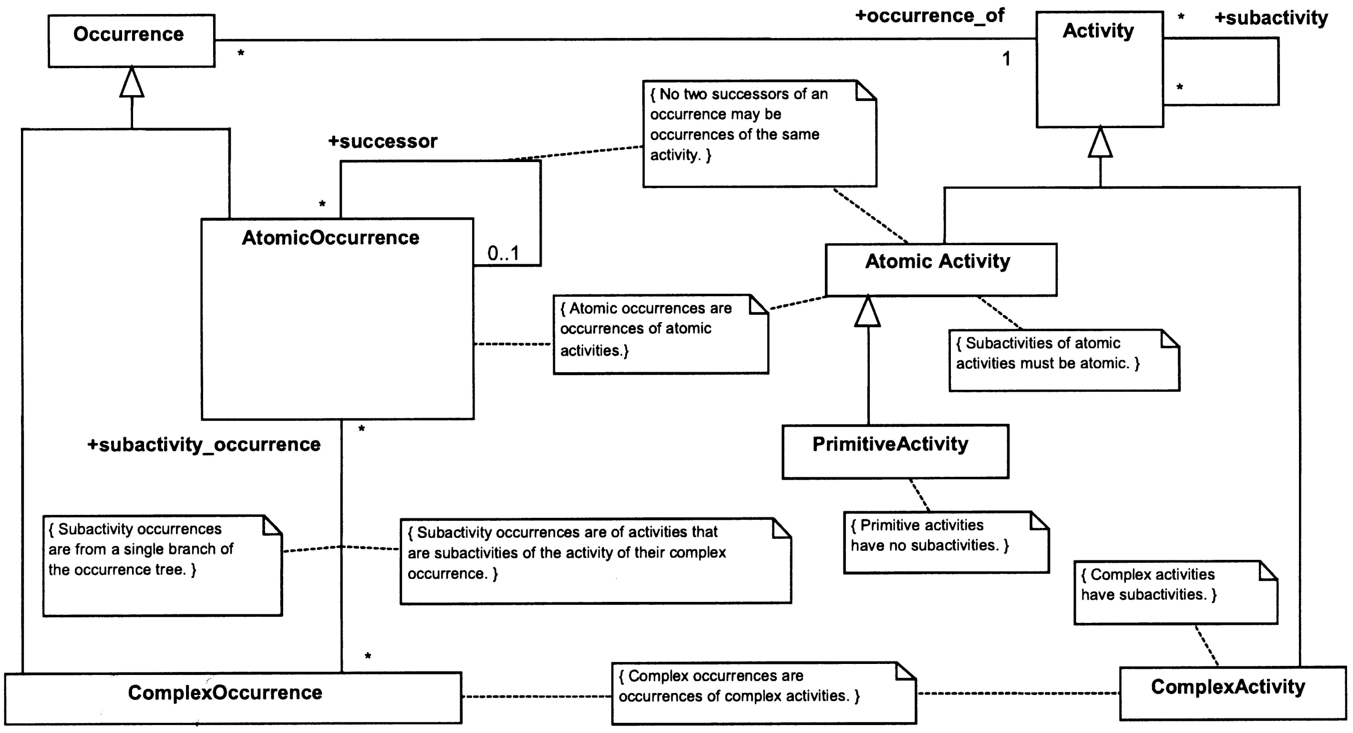
Occurrence
+occurrence_of
Activity
+subactivity
*
1
*
*
{ No two successors of an occurrence may be occurrences of the same activity. }
+successor
*
AtomicOccurrence
0..1
Atomic Activity
{ Atomic occurrences are occurrences of atomic activities.}
{ Subactivities of atomic activities must be atomic. }
+subactivity_occurrence
*
PrimitiveActivity
{ Subactivity occurrences are from a single branch of the occurrence tree. }
{ Subactivity occurrences are of activities that are subactivities of the activity of their complex occurrence. }
{ Primitive activities have no subactivities. }
{ Complex activities have subactivities. }
*
ComplexOccurrence
{ Complex occurrences are occurrences of complex activities. }
ComplexActivity

Fig. 17. PSL concepts with concurrency

an activity that is the concurrent aggregation of two others. We can use it to replace the occurrence statement for Paint in Expression 4 with the fragment in Expression 18.[11] The complex occurrences shown in Fig. 18 satisfy the revised constraint. Since the subactivities of an atomic activity are concurrent, the use of CONC can be replaced with an explicitly defined concurrent aggregation using subactivities. However the CONC function enables quantification over aggregated activities, as in Expression 18, rather than being restricted to a predefined set of subactivities.

[11] If Paint or the external activity is complex, then concurrent aggregations must be created for every pair of the primitive activities underneath them, since CONC only applies to primitives. This will be shown in future work.

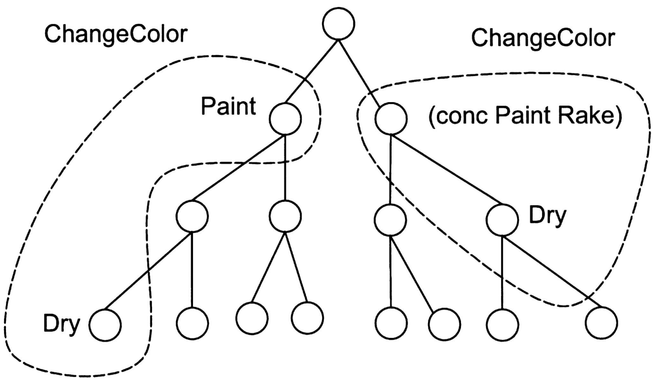

Fig. 18. ChangeColor occurrences with a concurrent external activity

```
(or (occurrence_of ?occPaint Paint)
    (exists (?activity)
       (occurrence_of ?occPaint (conc ?activity
                                           Paint))))
```

Expression 18:
External concurrent activities in Expression 4

Concurrent activities can be applied to UML fork and joins, for example in Fig. 13 to support Dry and Clean executions that overlap in time. The occurrence tree can be constrained to include both overlapping and nonoverlapping executions, but the expressions are rather complicated and will be addressed in future work. Another topic for the future is to apply the two kinds of unordered execution to formalize the semantics of UML state machines. The semantics of orthogonal regions is a combination of overlapping and nonoverlapping unordered execution, due to partial synchronization under the run-to-completion principle. This little-known fact about the execution of UML state machines could be made explicit using PSL.

6 Behavior classification

This section gives an example of how reducing ambiguity increases expressiveness and power in a language. It applies the PSL concepts from Sect. 4.1, which were used to disambiguate UML control flow, to enable classification of behavior. Partial execution sequences are used to incrementally add constraints in a behavior taxonomy. The section first covers existing UML capabilities for behavior classification. It concludes with some other applications of partial and complete sequencing.

Behaviors in UML 2 are classes and their executions are instances. For example, ChangeColor is a class and each time it is executed a new instance is created, and when it is finished executing, the instance is destroyed. Like all classes, behaviors can have attributes, for example for how long the instance has been executing, and operations for suspending it. Applications such as workflow and operating systems treat processes as objects, to manipulate and monitor them [31].

Also like classes, behaviors constrain their instances and can form subclass hierarchies by incrementally adding constraints. For example, we can create a hierarchy of food service behaviors, as shown in Fig. 19, adapted from [33]. The food service process gives general constraints, such as preparing food must happen before eating it, while specializations like fast food service add others, such as the food is eaten after it is paid for. In a behavior taxonomy, the same execution is an occurrence of multiple activities at once. For example, an execution of a fast food service process is also an execution of food service process.

PSL facilitates behavior specialization because it provides for declaring partial constraints on runtime execution, which can be incrementally combined in a behavior class hierarchy. In particular, the MIN_PRECEDES relation can be used to constrain a process execution without specifying every step and flow link in the process. For example, we might specify that the food service process above must include ordering, preparing, serving, eating, and paying, but not necessarily in exactly that order. The constraints might be:

– Ordering, preparing, and serving always happen before eating.
– Serving happens after preparing and ordering.
– Paying can happen anytime in the process.

In PSL, this is expressed as shown in Expression 19.

Expression 19 is more complicated than those in Sect. 4.1 because it is not only declaring constraints on fast food processes, but also other processes that might be specialized from it. That is why it has an existential quantification, to ensure that the appropriate subactivities occur in those specializations, and a universal quantification, to ensure that all occurrences of the subactivities obey the ordering constraints. Without the existential the specializations may fail to perform some steps. For example, the restaurant food service could omit eating. Without the universal the constraints in specializations might refer to different occurrences of the steps referred to by food service in general. For example, the constraint in restaurants that paying happen after eating could be satisfied by an eating step that happens before ordering.[12,13]

Once the general food service process is specified, we define the additional sequencing constraint on eating and paying for fast food service in Expression 20.

The process for a full service restaurant requires paying to happen after eating, as shown in Expression 21.

And payment happens first at church suppers, as shown in Expression 22.

Finally, to specify that the constraints on food services apply to its specializations, we require all occurrences under FastFoodService to also be under FoodService, and so on. This is facilitated by defining a relation ACTIVITY_SPECIALIZATION for specializing PSL activities that requires all occurrences under one complex activity to also be occurrences under another, as shown in Expression 23.[14] The ACTIVITY_SPECIALIZATION relation is used in the example as shown in Expression 24.

UML 2 does not support the semantics described above, but it provides for behavior redefinition, that is,

[12] If some steps should be optional, for example, a buffet might not have the ordering step or the serving step, then those can be omitted from the existential quantification. The expressions using the SUBACTIVITY relation are fine as is, because they only define activities that might be executed, rather than those that must be executed.

[13] Another approach is to constrain the suboccurrences to be of the expected kinds:

```
(forall (?s)
   (implies
      (subactivity_occurrence ?s ?occFoodService)
      (or (= ?s ?occOrder)
          (= ?s ?occPrepare)
          (= ?s ?occServe)
          (= ?s ?occEat)
          (= ?s ?occPay))))
```

This restricts the occurrences in all food services, forcing the occurrences of eating, ordering, and so on to be the same in all specializations. This rather large hammer prevents using the same subactivity twice in food service activities, or adding other activities not listed in the constraint.

[14] An alternative is to classify activities themselves, rather than complex occurrences. This will be explored in future work. The approach used in the paper is chosen because it corresponds to the classificiation of executions in UML.

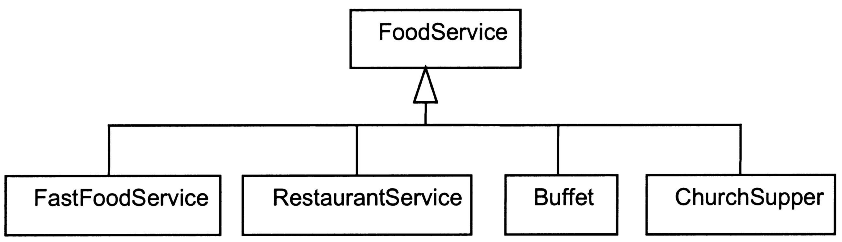

Fig. 19. Behavior taxonomy

```
(activity Order)
(activity Prepare)
(activity Serve)
(activity Eat)
(activity Pay)

(subactivity Order FoodService)
(subactivity Prepare FoodService)
(subactivity Serve FoodService)
(subactivity Eat FoodService)
(subactivity Pay FoodService)

(forall (?occFS)
 (implies
   (occurrence_of ?occFS FoodService)
   (and
     (exists (?occOrder ?occPrepare ?occServe ?occEat ?occPay)
       (and (occurrence_of ?occOrder Order)
            (occurrence_of ?occPrepare Prepare)
            (occurrence_of ?occServe Serve)
            (occurrence_of ?occEat Eat)
            (occurrence_of ?occPay Pay)

            (subactivity_occurrence ?occOrder ?occFS)
            (subactivity_occurrence ?occPrepare ?occFS)
            (subactivity_occurrence ?occServe ?occFS)
            (subactivity_occurrence ?occEat ?occFS)
            (subactivity_occurrence ?occPay ?occFS)))
     (forall (?occOrder ?occPrepare ?occServe ?occEat ?occPay)
        (implies
           (and (occurrence_of ?occOrder Order)
                (occurrence_of ?occPrepare Prepare)
                (occurrence_of ?occServe Serve)
                (occurrence_of ?occEat Eat)
                (occurrence_of ?occPay Pay)
                (subactivity_occurrence ?occOrder ?occFS)
                (subactivity_occurrence ?occPrepare ?occFS)
                (subactivity_occurrence ?occServe ?occFS)
                (subactivity_occurrence ?occEat ?occFS)
                (subactivity_occurrence ?occPay ?occFS))
           (and (min_precedes ?occServe ?occEat FoodService)
                (min_precedes ?occPrepare ?occServe
                              FoodService)
                (min_precedes ?occOrder ?occServe
                              FoodService)))))))
```

Expression 19: Flow constraints for FoodService

```
(forall (?occFastFoodService ?occEat ?occPay)
   (implies
      (and (occurrence_of ?occFastFoodService FastFoodService)
           (occurrence_of ?occEat Eat)
           (occurrence_of ?occPay Pay)
           (subactivity_occurrence ?occEat ?occFastFoodService)
           (subactivity_occurrence ?occPay ?occFastFoodService))
           (min_precedes ?occPay ?occEat FastFoodService)))
```

Expression 20: Flow constraints for FastFoodService

```
(forall (?occRestaurantService ?occEat ?occPay)
   (implies
      (and (occurrence_of ?occRestaurantService
                          RestaurantService)
           (occurrence_of ?occEat Eat)
           (occurrence_of ?occPay Pay)
           (subactivity_occurrence ?occEat
                                   ?occRestaurantService)
           (subactivity_occurrence ?occPay
                                   ?occRestaurantService))
      (min_precedes ?occEat ?occPay RestaurantService)))
```

Expression 21: Flow constraints for RestaurantService

```
(forall (?occChurchService ?occPay)
   (implies
      (and (occurrence_of ?occChurchService ChurchService)
           (occurrence_of ?occPay Pay)
           (subactivity_occurrence ?occPay ?occChurchService))
           (root_occ ?occPay ?occChurchService)))
```

Expression 22: Flow constraints for ChurchSupper

```
(forall (?aSub ?aSuper)
  (iff (activity_specialization ?aSub ?aSuper)
       (forall (?occSub)
         (implies
           (occurrence_of ?occSub ?aSub)
           (exists (?occSuper)
              (and (occurrence_of ?occSuper ?aSuper)
                   (forall (?s)
                     (implies
                       (subactivity_occurrence ?s ?occSub)
                       (subactivity_occurrence ?s
                                               ?occSuper)))))))))
```

Expression 23: Definition of ACTIVITY_SPECIALIZATION

```
(activity_specialization FastFoodService FoodService)
(activity_specialization RestaurantFoodService FoodService)
(activity_specialization Buffet FoodService)
(activity_specialization ChurchSupper FoodService)
```

Expression 24: Activity specializations for FoodService

adding and removing elements from the behavior specification, such as adding or removing a step in a flow model. This could be used for behavior specialization, if it is restricted to the semantics of ACTIVITY_SPECIALIZATION, because behavior redefinition is more general than specialization. For example, a behavior can be redefined to add a step between two others, even if the link between those two does not allow it (NEXT_SUBOCC). The techniques of this section can be used to control behavior redefinition and give it a semantic basis.

The PSL semantics presented in this section is helpful in other areas of modeling, such as constraining polymorphism. Many behavior specification languages support runtime decisions about how a task is to be carried out. For example, a machine might support a polishing operation, but processes using the machine probably do not want to dictate how the polishing is done. This indirection enables a task to be carried out in various ways that may change over time, while preserving the interface of the task to the rest of the system. It also means

```
        (implies (holds TooHot ?occServe)
           (exists (?occLetCool)
              (and (occurrence_of ?occLetCool LetCool)
                   (subactivity_occurrence ?occLetCool
                                           ?occFoodService)
                   (min_precedes ?occServe ?occLetCool FoodService)
                   (min_precedes ?occLetCool ?occEat
                                 FoodService))))
(activity LetCool)
(subactivity LetCool FoodService)
```

Expression 25: Flow constraints for UML Extension Points

the semantics of any specification is inherently ambiguous, since the executed behavior is not determined until runtime.[15] Systems that are truly dynamic in behavior selection, such as the Common Lisp Object System [17], are at least amenable to constraints on their effects as described in this section, even if the ultimate runtime behavior cannot be determined or even guessed at specification time.

The techniques in this section are also helpful in formalizing UML use cases, which have behaviors and form classification hierarchies. Extension points in use cases identify where other behaviors can be inserted as necessary. This is an answer to question 3 in Sect. 1. Conditions can be placed on when the inserted behavior is executed. If the use case behavior is a UML activity, the control links where new behaviors can be inserted have the semantics of MIN_PRECEDES, and control links that are not extendable have the semantics of NEXT_SUBOCC. Behaviors inserted conditionally are expressed in PSL similar to optional ones, except using an implication to enforce the existence of the optional step under certain conditions. For example, an additional element of the top-level conjunction in Expression 19 might be the one shown in Expression 25.

Expression 25 uses the PSL HOLDS relation to specify a state of the world that is true after an activity has executed (see Sect. 4.3). It determines whether the occurrence of FoodService should include a step for letting the food cool between serving and eating.

PSL can also be used for "client-side" constraints on operations. For example, UML's OCL 2 supports postconditions on operations that constrain the messages that the method of the operation may send. The techniques in this section can provide a formalization for OCL 2 used this way, and increase the breadth of statements that can be made about messages sent by an operation.

[15] Most systems do not offer completely unrestricted runtime behavior selection. Popular object-oriented languages, for example, do not permit runtime reclassification of instances, or support runtime modification of methods in an instance. This means the behavior resulting from a message to an object of a certain type will be one of a small set of methods on subtypes of the type of the object, and often only one. Component-based approaches are more flexible because a component can be replaced without recompilation. However, once a system is executing and the component is selected, it usually cannot be changed. For these applications, the system configurations can be enumerated at specification time, and tools can support the modeler in traversing through the formalizations of possible combinations of behaviors.

7 Closure

An issue with the PSL behavior specifications so far is whether they should rule out executions that have additional elements than the ones explicitly given in the specification. For example, Expression 1 says that all legal occurrences of Paint must have a legal occurrence of Dry as a successor. However, the expression does not prevent legal occurrences of other activities to be successors of Paint also, because the occurrence tree provides for multiple possible successors. This means the expression allows execution traces where Paint is not followed immediately by Dry, contrary to the original intention. To be accurate, it requires a kind of "closed world" constraint that rules out possibilities not included in the specification, as shown in Expression 26. Even if particular inference engines make the closed world assumption implicitly, it is beneficial to include closure explicitly in behavior specifications to ensure they are independent of the engine used.

```
(forall (?occPaint)
  (implies
    (and (occurrence_of ?occPaint Paint)
         (legal ?occPaint))
    (and (legal (successor Dry ?occPaint))
         (forall (?otherSuccessor)
           (implies
             (not (equal ?otherSuccessor
                    (successor Dry ?occPaint)))
             (not (legal ?otherSuccessor)))))))
```

Expression 26: Closed version of Expression 1

Similarly, the subactivity statements in Expression 2 do not prevent additional subactivities from being added to the specification later. In particular, a complete inference engine might introduce new subactivities without contradicting the specification. This prevents the engine from deducing, for example, the amount of time Change-

Color will take to execute, because it could infer that other subactivities exist. The closure constraint to prevent other subactivities from being added can be concisely written using the fact that PSL activities are subactivities of themselves, as shown in Expression 27.

```
(forall (?ccSubactivity)
  (iff (subactivity ?ccSubactivity ChangeColor)
       (or (subactivity ?ccSubactivity Paint)
           (subactivity ?ccSubactivity Dry))))
```

Expression 27: Closed version of Expression 2

If flow constraints are complete enough, then closure is not needed. For example, using roots and leaves with a series of NEXT_SUBOCC relations between them, as in Expressions 7 and 13, gives a complete constraint in which no other occurrences can be inserted. On the other hand, partial constraints such as those using MIN_PRECEDES in Expressions 5 and 6 intentionally allow extensions to the specification. If some extensions are not allowed, these must be explicitly ruled out. For example, in Expression 19 we might want to prevent the subactivities of FoodService from happening more than once under the same complex occurrence. For each subactivity, the clause in Expression 28 can be added to the outer conjunct of Expression 19 to eliminate duplicate subactivity occurrences.

```
(forall (?occ1 ?occ2 ?activity)
 (implies (and (subactivity_occurrence ?occ1
                                        ?occFS)
               (subactivity_occurrence ?occ2
                                        ?occFS)
               (occurrence_of ?occ1 ?activity)
               (occurrence_of ?occ2 ?activity))
          (= ?occ1 ?occ2)))
```

Expression 28: A closure addition to Expression 19

Closure constraints can be too tight in some cases, eliminating execution traces that might actually occur. For example, Expression 18 can be changed to allow only those concurrent executions of activities that do not interfere with painting. The expression could be tightened to the one in Expression 29 using PSL's resource extension, which represents whether two activities are contending for the same resource. With this closure, an inference engine would no longer be able to detect when a set of processes contend for a resource, because the possibility is prevented by the specification. However, the processes actually implemented might contend, because the inference engine did not alert implementers to that possibility, and no arrangement was made to prevent it.

```
(or (occurrence_of ?occPaint Paint)
    (exists (?activity)
       (and (occurrence_of ?occPaint (conc
                                  ?activity Paint))
            (not (interfering
                              ?activity Paint)))))
```

Expression 29: Closed version of Expression 18

Closure is also cumbersome to introduce during the development of a behavior specification, because more changes need to be made to add elements. For example, to add another subactivity to ChangeColor with closure requires editing the closure constraints as well as the subactivity statements.

The issues with closure can be addressed by adding closure constraints only when needed for inference. Then they do not burden development and can be adjusted to account for the information required from the inference engine. For example, the subactivities of ChangeColor can be left open until the specification is ready for testing and inference. If it is required that the engine show where the contending processes are, then closure constraints for contention can be omitted.

8 Other approaches

PSL has a simple semantics based on first-order logic. Intuitions about executing behaviors are mapped to elements of well-understood mathematical structures, and we can prove that these structures are isomorphic to the logical axioms of the PSL. Development within this methodology has distinct advantages. The use of mathematical structures validates that we formalized our intuitions on a commonly agreed upon basis. The application of first-order logic means that behavior descriptions based on PSL concepts can support automated reasoning with a wide array of theorem provers [15] and constraint satisfaction techniques. First-order logic also provides a framework for specifying semantic mappings between different software applications [3, 24].

This development methodology can be compared to other approaches, for example:

– Petri nets is perhaps the most powerful and widely-known alternative for process modeling [16, 18]. It is designed to model the synchronization of concurrent processes.
– Within the artificial intelligence community, the Planning Domain Definition Language (PDDL) is used extensively in the Artificial Intelligence Planning Systems competition [7]. It covers the domain of planning, including a specification of states, the set of possible activities, the structure of complex activities, and the effects of activities.
– The Cognitive Robotics Group at the University of Toronto proposed the language GOLog as a high-level robotics programming language [19]. GOLog provides mechanisms for specifying complex activities as programs in a second-order language that extends the axiomatization of situation calculus [28].
– The Workflow Management Coalition developed a standard terminology which can serve as a common framework for different workflow management system vendors. The ontology for this effort is the Workflow Process Definition Language (WPDL) [32].

– In the context of the Semantic Web, much work has been done using the DARPA Agent Markup Language (DAML) [14]. In particular, the DAML-S ontology provides a set of process classes that can be specialized to describe a variety of Web services [22].
– Work on semantic domains for UML has been ongoing at the Object Management Group (OMG) [1, 30]. These are based on how objects change over time, using a series of object snapshots to model entities changing over time. Behavior execution is modeled as constraints on object snapshots before and after execution of the behavior.

These approaches to process semantics lack one or more formal properties of PSL. Petri nets has no standard, widely agreed upon semantics, and those that exist are highly complex, and require a sophisticated knowledge of advanced areas of mathematics.[16] They also have not been axiomitized in first-order logic. PDDL maps to commonly understood mathematical structures, but does not provide the equivalent first-order logic expressions. GOLog describes its mathematical structures informally and does not prove equivalence to its axiomization. Languages such as DAML-S and WPDL do not provide a mapping to mathematical structures to validate their concepts, and WPDL does not even provide a semantic domain. The OMG proposals give semantic domains, but do not validate them with mathematical structures, or axiomitize them completely in first-order logic.[17]

9 Summary

This article describes an approach to flow model semantics based on constraining sequences of runtime execution: the Process Specification Language (PSL). Since modelers already simulate runtime traces in their minds to understand the semantics of behavior specifications, PSL concepts are an intuitive basis for making those intentions precise and machine-interpretable, and disambiguating modeling shorthands. This improves communication between modelers and ensures fidelity of implementations. Also, more precise semantics reveals distinctions that add expressiveness and power to flow modeling, for example the capability to incrementally combine partial behavior specifications. Behavior specifications in PSL are expressed in first order logic, making properties about their runtime execution provable with standard inference engines.

[16] The most common approach to semantics for Petri nets is to map them into linear logic and then exploit one of the semantics for that [20].

[17] A number of narrower issues in comparison can be addressed in a separate paper. For example, the GOLog and PDDL semantic domains do not include reusable composed processes, so cannot support reasoning about complex activity occurrences. The sequences in the OMG semantic domains do not branch, so they cannot require indeterminacy, only allow it, and cannot model interrupted or hypothetical processes.

References

1. Action Semantics Submission Team (2000) Action Semantics for the UML. `http://www.omg.org/cgi-bin/doc?ad/2000-08-02`
2. Bock C (2003) UML 2 Activity and Action Models. Journal of Object Technology 2:4, July–August. `http://www.jot.fm/issues/issue_2003_07/column3`
3. Ciocoiu M, Gruninger M, Nau D (2001) Ontologies for Integrating Engineering Applications. Journal of Computing and Information Science and Engineering 1(1):12–22
4. Comon Logic Working Group (2003) Common Logic Standard. `http://cl.tamu.edu`, `http://cl.tamu.edu`
5. Fox MS (1992) The TOVE Project: A Common-sense Model of the Enterprise, Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. In: Belli F, Radermacher FJ (eds) Lecture Notes in Artificial Intelligence # 604, Springer-Verlag, pp 25–34
6. Fox MS, Gruninger M (1998) Enterprise Modelling. AI Magazine, AAAI Press, pp 109–121, Fall
7. Ghallab M, Howe A, Knoblock C, McDermott D, Ram A, Veloso M, Daniel W, Wilkins D (1998) PDDL: The Planning Domain Definition Language v.2. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control
8. Genesereth MR, Fikes R (1992) Knowledge Interchange Format 3.0. Technical Report KSL-92-01, Knowledge Systems Laboratory, Stanford University
9. Gruninger M (2003) Guide to the Ontology of the Process Specification Language. In: Staab S (ed) Handbook of Ontologies in Information Systems, Springer-Verlag
10. Gruninger M (2003) PSL 2.0 Ontology – Current Theories and Extensions. `http://www.nist.gov/psl/psl-ontology/`
11. Gruninger M (2004) Model Theory of PSL-Core. To appear in Technical Report of the Institute for Systems Research at the University of Maryland, College Park
12. Gruninger M, Menzel C (2003) Process Specification Language: Principles and Applications. AI Magazine 24(3): 63–74
13. Hayes P, Menzel C (2001) A Semantics for the Knowledge Interchange Format. Workshop on the IEEE Standard Upper Ontology, IJCAI, Seattle
14. Hendler J, McGuinness DL (2001) DARPA Agent Markup Language. IEEE Intelligent Systems. 15:72–73
15. Kalman J (2001) Automated reasoning with Otter. Rinton Press, Princeton
16. Karp R, Miller R (1966) Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. SIAM Journal of Applied Mathematics 14(6):1390–1411, November
17. Kiczales G, des Rivieres J, Bobrow D (1991) The Art of the Metaobject Protocol. MIT Press
18. Peterson J (1981) Petri Net Theory and the Modelling of Systems. Prentice-Hall
19. Levesque H, Reiter R, Lesperance Y, Lin F, Scherl R (1997) GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming 31:92–128
20. Marti-Oliet N, Meseguer J (1991) From Petri Nets to Linear Logic. Mathematical Structures in Computer Science 1(1): 69–101
21. McCarthy J, Hayes P (1969) Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer B, Michie D (eds) Machine Intelligence 4, Edinburgh University Press, Edinburgh, pp 463–502
22. McIlraith S, Son TC, Zeng H (2001) Semantic Web Services. IEEE Intelligent Systems, Special Issue on the Semantic Web 16:46–53, March/April
23. Menzel C, Gruninger M (2001) A formal foundation for process modeling. In: Welty C, Smith B (eds) Formal Ontology in Information Systems, ACM Press
24. Nau D (2003) Mapping and merging ontologies. In: Staab S (ed) Handbook of Ontologies in Information Systems, Springer-Verlag
25. Object Management Group (2003) OCL 2.0 Specification. `http://www.omg.org/cgi-bin/doc?ptc/03-10-14`, March
26. Object Management Group (2003) OMG Unified Modeling Language Specification, version 1.5, Part 6. `http://www.omg.org/cgi-bin/doc?formal/03-03-01`, March

27. Object Management Group (2004) UML 2.0 Superstructure Specificatoin. `http://www.omg.org/cgi-bin/doc?ptc/03-08-02`, March
28. Reiter R (2001) Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press
29. Schlenoff C, Knutilla A, Ray S (1997) Requirements for Modeling Manufacturing Process: A New Perspective. In: Proceedings of Design Engineering Conferences, Sacremento, September
30. Unambiguous UML Submission Team (2003) Unambiguous UML (2U) 3rd Revised Submission to UML 2 Infrastructure RFP. `http://www.omg.org/cgi-bin/doc?ad/2003-01-07`
31. Workflow Management Coalition (1999) Workflow Standard – Interoperability Abstract Specification. `http://www.wfmc.org/standards/docs/TC-1012_Nov_99.pdf`, November
32. Workflow Management Coalition (1999) Interface 1: Process Definition Interchange Process Model. Technical Report WfMC-TC-1016-P. `http://www.wfmc.org/standards/docs.htm`
33. Wyner GM, Lee J (2003) Defining Specialization for Process Models. In: Malone TW, Crowston K, Herman GA (eds) Organizing Business Knowledge: The MIT Process Handbook, MIT Press, pp 131–174

Conrad Bock is the workgroup lead for UML 2 activity and action modeling, and participated in UML process model development in earlier versions of UML. He has extensive experience in process modeling over a range of methodologies, including those at SAP and Microsoft. He is currently a Computer Scientist at the U.S. National Institute of Standards and Technology (NIST) in the Process Specification Language project.

Michael Gruninger is the project leader for the PSL project at NIST, and for International Standards Organization (ISO) 18629 standardizing PSL. His previous work on process ontologies included TOVE, a commonsense model for the enterprise. Michael is currently a Research Scientist in the Institute for Systems Research at the University of Maryland College Park and Guest Researcher at NIST.