NISTIR 7152

Inputs and Outputs in the Process Specification Language

Conrad Bock
Michael Gruninger

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

NISTIR 7152

Inputs and Outputs in the Process Specification Language

Conrad Bock
Michael Gruninger

Manufacturing Systems Integration Division
Manufacturing Engineering Laboratory

August 2004

U.S. DEPARTMENT OF COMMERCE
Donald L. Evans, Secretary
TECHNOLOGY ADMINISTRATION
Phillip J. Bond, Under Secretary of Commerce for Technology
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Arden L. Bement, Jr., Director

Inputs and Outputs in the Process Specification Language

Conrad Bock
Michael Gruninger
August 9, 2004

Inputs and outputs are ubiquitous in flow modeling, including popular programming languages. This paper examines how inputs and outputs can be formalized in the Process Specification Language (PSL) to reduce ambiguity and increase expressiveness compared to conventional flow modeling representations. Inputs and outputs are shown to be early design stage notions independent of existing PSL concepts, preconditions and postconditions in particular. The paper defines axioms for input and output, and constraints on existing PSL concepts. Some of these relate early and late stage design, while others provide for multiple views of inputs and outputs. It also identifies which aspects of input are metatheoretic and consequently outside the scope of PSL.

1. Introduction

This paper assumes familiarity with PSL fundamentals [1]. PSL uses language elements referring directly to real world processes, for example milling operations as they actually occur in a factory at certain times. It defines a simple set of concepts that cover all possible ways these operations can happen, which is called the occurrence tree. The process designer uses these concepts to write constraints on which occurrences are allowed, for example to specify what is required to happen during a milling process. This approach allows process requirements to be written very generally or very specifically as needed by the stage of design. For example, early design stages define loose constraints, because the domain expert is just sketching out broad requirements. These are tightened as design moves forward, until the process is completely specified. For example, a software program is a kind of process description that places many constraints on allowable executions.

PSL is project 18629 at the International Organisation of Standardization, and part of the work is a Draft International Standard. It is based on a long period of research stemming from the situation calculus and enterprise modeling. It has been applied in scheduling, process modeling, process planning, production planning, simulation, project management, workflow, and business process reengineering. The standard is divided into core theories and extensions. The core axiomatizes a set of intuitive semantic primitives describing fundamental concepts of manufacturing processes. The core concepts include discrete states for relating processes to states of the world, as well as subactivities, atomic activities, and complex activities for composition of processes. Extensions introduce new terminology, to supplement the core concepts. They define additional relations for activities, time and state, activity ordering, duration, and resources. All axioms are first-order sentences, and are written in the Knowledge Interchange Format [2].

The paper follows the methodology of PSL by introducing axioms in increasing levels of constraint they place on processes. Section 2 gives an example highlighting the issues around inputs and outputs. Section 3 first establishes that new axioms are needed to represent inputs and outputs in PSL. Section 4 covers inputs and outputs occurrences, starting with primitive occurrences, then complex occurrences, including and multiple complex occurrences for the same primitive occurrence. Section 5 gives some convenience relations for using inputs and outputs at the level of activities. Section 6 addresses input and output issues that cannot be addressed by PSL because they involve how inputs and outputs are chosen in the first place. This leads to suggestions on future work.

2. Inputs and Outputs

Common flow modeling and process languages, whether in graphical form like the Unified Modeling Language (UML) [3], or textual form as in programming languages, usually describe inputs and outputs colloquially as entities "passed in" or "passed out" of a process. Inputs are normally required to be passed in for a process to start and outputs are required to be passed out for it to complete. The intuition is that the process has a "boundary" and the employer of the process need only provide the proper inputs to get the desired outputs.

A simple example shows the ambiguities in this common conception of input and output. A milling machine changes the shape of a piece of metal based on some instructions, and depending on the particular machine used in the process, might require oil and electricity, as shown in Figure 1. Which are the inputs and which are the outputs, and which are neither? Consider these alternative views:

- The piece of metal and instructions are input, and the piece of metal is output, whereas oil and electricity are "infrastructural" and not of concern as inputs and outputs.

- An infrastructure view might only take the oil and electricity as inputs, and oil as output, since it becomes dirty and needs to be cleaned or recycled.

- Another view might identify metal shavings that are produced by milling and call those outputs because they must be removed from the machine periodically.

- It might be decided that although sound, heat, and vibration are all external effects of the milling process, vibration is an output, because special arrangements are made to absorb it.

- Perhaps the machine is controlled under an agent architecture, which determines the shape to be made through a brokering interaction with other agents needing the milling service. In this case, the instructions are not an input because the machine chooses them for itself.

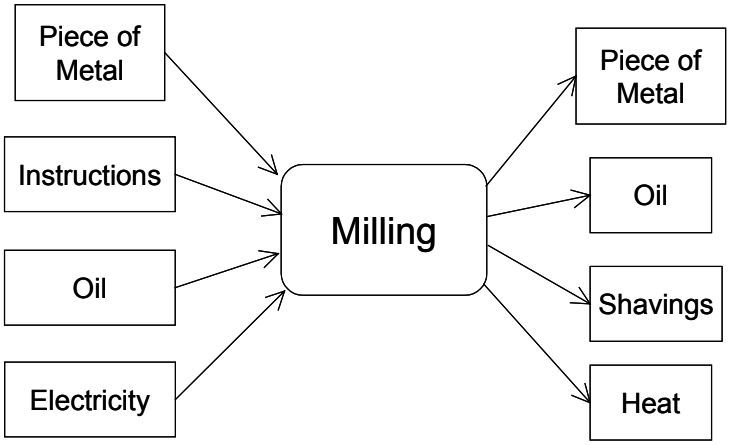

Figure 1: Inputs and Outputs

This simple example presents a challenge for both conventional flow models and PSL. Flow models, including textual programming languages, provide for only one view of a process, that is, one set of inputs and outputs. And within that single view, they provide no guidance on what to choose as an input or output. However, flow models support the temporal ordering constraints that require inputs of one process to be filled by outputs of another occurring earlier. PSL supports multiple views of a single process, because any element of a process can also be categorized under multiple other processes. However, PSL currently lacks support for representing input and output, and in particular, distinguishing inputs from outputs from other entities participating in a process. It also lacks the temporal ordering constraints between outputs and inputs. This paper addresses these issues with extensions to PSL, and identifies aspects that are not representable in PSL.

3. Independence of Inputs and Outputs from Current PSL Axioms

PSL provides a way to refer to processes as they occur in the world (hereafter called "runtime" or "execution"), and a way to state constraints on those occurrences. PSL defines the occurrence representation in first-order logic, so process designers can write constraints on it also in first-order logic, which is how process descriptions are constructed. The occurrence representation covers all possible paths that a process can take, so process descriptions can be as loose or tight as necessary to reflect the process designer's intent.

In this approach to process representation, it is not obvious that input and output are primitive concepts, rather than derivations from existing ones. A process actually occurring in the world at a certain time involves many entities, for example a piece of metal, instruction codes, oil, heat, and so on. There are various ways one might attempt to distinguish which are inputs, which are outputs, and which are neither:

- Perhaps an input is any entity participating in an occurrence of the process that also participates in some other occurrence earlier in time, and the reverse for output, using PSL's support for participation. However, such an entity is not always an input or an output. For example, suppose the oil used by a drill happened to be the same oil used by a milling machine in the same factory, but was cleaned in between uses. It would be an unusual view that took drilling as outputting oil for milling to take as input. Even this hydraulic view of the factory would interpose the cleaning process in between drilling and milling, and not be concerned with the specific machines that happened to be in the factory.

- Another approach is to use preconditions and postconditions, which PSL supports on process occurrences. Input can be represented as a precondition on an occurrence requiring a particular entity to be available to the occurrence in some specified way, and to define an output as a postcondition that a particular entity is available in some specified way from the occurrence to participate in others. For example, the precondition on the piece of metal input to a milling machine is that it is in a certain location where the operator can reach it, or the machine can detect it.

  However, it may be that this location is different for various milling machines or may change even for the same milling machine due to its configuration. PSL takes these as different occurrences, requiring different preconditions, because the milling machine is different. The flow modeler's intuition would be that milling processes in general take a piece of metal as input, regardless of how or where the machine detects its presence. The location may change and the piece of metal is still an input.[1]

[1] An example from software is that inputs and outputs to procedures in a higher level language are often compiled into a lower level language that arranges for the input and output data to be copied into memory locations relative to the location of the called procedure, or to particular kinds of memory for that purpose.

- A representation for resources is being developed for PSL that defines input and output based on effects that the activity has on the input or output. In this extension, input material is defined as any entity consumed or modified by the process, and output material refers to the participants that are created or modified. This would mean instructions to a milling are not inputs because they are not modified, and similarly for processes that use catalysts. Another aspect of the resource extension is contention, which is not a requirement for inputs and outputs. An entity can be input to two processes that operate on it at once without contention, for example, two machining operations on different parts of the same piece.

- First-order logic provides for parameterized functions that can be easily mistaken for a representation of inputs and outputs. For example, we might define the milling process as a function that specifies an activity, as the milling term does in Expression 1.

```
(forall (?a ?m ?i ?o)
     (implies (= ?a milling(?m ?i ?o))
          (and (activity ?a)
               (metal ?m)
               (instructions ?i)
               (oil ?o))))
```

Expression 1: Parameterized Term for Activities

  However, parameterized terms do not differentiate inputs from outputs, or entities that are neither. It is a useful technique to build on, however, as shown in the remainder of this article.

These examples suggest that the notions of input and output are justifiably primitive for a language that takes actual process occurrences as its base. In particular, inputs and outputs cannot be reduced to existing PSL concepts such as participation, pre/postconditions, or resources. In addition, the examples show that inputs and outputs relative to viewpoint are not currently addressed in PSL, though PSL provides a key enabler for process views that is missing from conventional flow models, as shown in section 4.3.

The inputs and outputs of the procedure do not need to change when the compiler chooses a different way to pass inputs into a procedure.

4. Input and Output Axioms at Occurrence Level

When defining axioms in PSL, one consideration is whether they should apply to occurrences of a process, or processes defined independently of when they occur, which are called *activities* in PSL. Axioms on activities usually constrain all occurrences of the activity. This is fine for many applications of input and output, however, for flexibility, such as for optional inputs and outputs, it is best to acknowledge that some occurrences of the same activity will have an input or output that others may not. In addition, constraints on which outputs are provided to which inputs is defined at the occurrence level, because activities are usually defined independently of how they are connected together by input and output in any particular usage. To support these applications, we define input and output for occurrences, and provide convenience axioms for inputs and outputs at the activity level in Section 5.

The two subsections 4.1 and 4.2 below cover input and output with and without processes composed of other processes. These are called complex occurrences and activities in PSL. Process designers will almost always use complex processes, because processes are intended to reach a desired goal, which is achieved by coordination of other processes. For example, drilling and milling a piece of metal can occur together, to attain a specific shape for the metal. They are coordinated by a complex process that has the shape as its aim. For simplicity of presentation, and application to early stage design, we begin with inputs and outputs without complex processes in Section 4.1, to introduce ordering and flows, then in Section 4.2 show how these are modified in the context of complex processes.

4.1 Without Complex Occurrences

We begin by defining relations for input and outputs of occurrences. Expression 2 constrains the application of the occurrence input and output relations to particular kinds of elements. Expression 3 and Expression 4 tie occurrence input and output to the existing PSL PARTICIPATES_IN relation, which is used to constrain which objects are involved in a particular occurrence of an activity. Expression 3 generalizes PARTICIPATES_IN so that participation at any time during the occurrence satisfies a new relation called PARTICIPATES. Expression 4 requires that inputs and outputs are always participants.

```
(forall (?x ?s)
   (implies (or (occurrence-input ?x ?s)
                (occurrence-output ?x ?s))
            (and (object ?x)
                 (not (state ?x))
                 (activity_occurrence ?s))))
```

Expression 2: Types for Occurrence Inputs and Outputs

```
(forall (?x ?s)
   (iff (participant ?x ?s)
        (exists (?t)
           (participates_in ?x ?s ?t))))
```

Expression 3: Participant Extension

```
(forall (?x ?s)
   (implies (or (occurrence-input ?x ?s)
                (occurrence-output ?x ?s))
            (participant ?x ?s)))
```

Expression 4: Participation Axiom for Inputs and Outputs

One of the basic intuitions of inputs and outputs is that inputs of an occurrence are provided by the outputs of another one happening earlier than the first. For simplicity, we assume that an occurrence cannot begin without its inputs, and cannot provide outputs before it ends.[2] This means an occurrence needing an input must begin after the occurrence providing the output has ended, as shown in Expression 5. The PSL relation EARLIER is true for an occurrence that is before another in the occurrence tree defined by the PSL relation SUCCESSOR.[3]

```
(forall (?x ?s2)
    (implies (and (occurrence-input ?x ?s2)
                  (legal ?s2))
             (exists (?s1)
                (and (occurrence-output ?x ?s1)
                     (earlier ?s2 ?s1)))))
```

Expression 5: Basic Input and Output Ordering

The definition of the milling term in Expression 1 can be used with other activities to constrain occurrence ordering. For example, if a drilling activity term were defined, Expression 6 ensures that drilling happens before milling, once at least (see Section 4.2 for stronger process constraints). The constraint in Expression 5 would not necessarily be satisfied by Expression 6, however, because Expression 6 allows a situation where there is no occurrence providing input of metal to drilling. This is how Expression 5 ensures that the inputs and outputs are well formed in constraints defined by the process designer. Expression 6 also does not address whether the other parameters of drilling and milling, such as oil, are inputs or outputs, see section 4.3.

[2] See Footnote 11.

[3] Purely physical applications might also require the object being passed to not participate in any occurrence in between the output and input. However, almost all processes have some non-physical aspects to them, due to embedded software in particular. Information objects can be referred to from multiple others and do not necessarily obey the constraint on participation above.

```
(exists (?sDrill ?sMill ?m ?i ?o)
   (and (occurrence_of ?sDrill drilling(?m ?i ?o)
        (occurrence_of ?sMill milling(?m ?i ?o)
        (occurrence-input ?m? sDrill)
        (occurrence-output ?m ?sDrill)
        (occurrence-input ?m ?sMill)
        (occurrence-output ?m ?sMill)
        (earlier ?sDrill ?sMill)
        (legal ?sMill)))))
```

Expression 6: Example Process Constraint Using Occurrence Inputs and Outputs

Expression 6 highlights that Expression 5 is weaker than normally required, because Expression 5 allows any earlier occurrence that outputs the needed input entity to satisfy the constraint. For example, the occurrence of any process outputting the needed piece of metal earlier than an occurrence of the milling process satisfies the axiom. This is fine as a general rule, but usually flow models specify a particular output that must "flow" to a particular input.

As with Expression 5, formalizing flow intuitions in PSL requires determining how they constrain which occurrences are legal. The approach taken here has two parts:

1. Link input and output to preconditions and postconditions on occurrences, which reflects the relation of early and late stage design as outlined in Section 2. For example, a milling machine might have a sensor that detects when a piece of metal has arrived to work on. The piece of metal will need to be in a certain location to be noticed. This is a precondition for the milling process, as well as a concrete realization of the abstract notion of input to the process.

   Expression 7, Expression 8, and Expression 9 represent this by introducing relations that identify which pre/postconditions are the realization of which inputs and outputs, called INPUT-STATE and OUTPUT-STATE. Expression 7 constrains the application of these relations to specific elements. Expression 8 and Expression 9 show how the relations connect occurrence inputs and outputs to PSL preconditions and postconditions, PRIOR and HOLDS. These expressions also constrain input objects to exist at the beginning of the occurrence and output objects to exist at the end.[4] The PSL relation ACHIEVED means that an occurrence caused a state to be true, that is, it was not true before the occurrence and is true afterwards.[5] Expression 10, Expression 11, and Expression 12 require that input and output states are unique to occurrences of the same activity, and different from each other on the same occurrence. Otherwise, for example, a milling process could take an input intended for a drilling process.

[4] Ideally input and output states could be constrained to be "about" the input and output objects, but this is not frst order. See Section 6.

[5] An alternative to introducing new relations would be to extend OCCURRENCE-INPUT and OCCURRENCE-OUTPUT with a state. This would reduce the number of relations, but make specification of the abstract level of input and output more cumbersome by requiring existentials for the state.

```
(forall (?x ?s ?f)
   (implies (or (input-state ?x ?s ?f)
                (output-state ?x ?s ?f))
            (and (object ?x)
                 (not (state ?x))
                 (activity_occurrence ?s)
                 (state ?f))))
```

Expression 7: Types for Input and Output States

```
(forall (?x ?s ?f)
       (implies (input-state ?x ?s ?f)
                (and (occurrence-input ?x ?s)
                     (prior ?f ?s)
                     (exists_at ?x (begin_of ?s)))))
```

Expression 8: Preconditions for Occurrence Inputs

```
(forall (?x ?s ?f)
   (implies (output-state ?x ?s ?f)
            (and (occurrence-output ?x ?s)
                 (achieved ?f ?s)
                 (exists_at ?x (end_of ?s)))))
```

Expression 9: Postconditions for Occurrence Outputs

```
(forall (?x1 ?s1 ?a1 x2 ?s2 ?a2 ?f)
   (implies (and (input-state ?x1 ?s1 ?f)
                 (input-state ?x2 ?s2 ?f)
                 (activity_occurrence ?s1 ?a1)
                 (activity_occurrence ?s2 ?a2))
            (and (= ?x1 ?x2)
                 (= ?a1 a2))))
```

Expression 10: Input State Uniqueness

```
(forall (?x1 ?s1 ?x2 ?s2 ?f)
   (implies (and (output-state ?x1 ?s1 ?f)
                 (output-state ?x2 ?s2 ?f)
                 (activity_occurrence ?s1 ?a1)
                 (activity_occurrence ?s2 ?a2))
            (and (= ?x1 ?x2)
                 (= ?a1 a2))))
```

Expression 11: Output State Uniqueness

```
(forall (?x1 ?x2 ?s ?f)
    (not (and (input-state ?x1 ?s ?f)
              (output-state ?x2 ?s ?f))))
```

Expression 12: Input and Output State Uniqueness

2. Connect occurrence inputs and outputs with flows that constrain preconditions and postconditions. For example, a drilling process that comes before a milling process must ensure that the piece of metal is placed in the proper position to start milling.[6] It is required that no other process puts the piece of metal on the milling machine or takes it away before milling starts. This prevents any process preceding drilling from passing the piece in too early, and any process occurring after milling from flowing it out too early.

   Expression 13, Expression 14, Expression 15, and Expression 16 represent this by introducing a relation that identifies which entities are flowing between the outputs and inputs of which occurrences, called OCCURRENCE-FLOW. Expression 13 constrains which elements this relation applies to. Expression 14 constrains flows to go between earlier and later occurrences, to match output to input states, and to have no occurrences in between that alter these states.[7] Finally, some approaches might require outputs to have inbound flows, and inputs to have outbound flows, as shown in Expression 15 and Expression 16.[8]

```
(forall (?x ?s1 ?s2)
   (implies (occurrence-flow ?x ?s1 ?s2)
            (and (object ?x)
                 (not (state ?x))
                 (activity_occurrence ?s1)
                 (activity_occurrence ?s2))))
```

Expression 13: Types for Flows

[6] Usually there would be a separate transport process in between drilling and milling, such as a conveyor belt. This is omitted from the example for simplicity of presentation.

[7] The relation CHANGED is an addition to PSL. It is defined to require that an occurrence either change a state from true to false or false to true:

```
(forall (?f ?s)
   (iff (changed ?f ?s)
        (or (achieved ?f ?s)
            (falsified ?f ?s))))
```

[8] Purely physical applications could require only one outflow per output of an occurrence, because physical objects can only flow to one place at a time. However, this would eliminate the application of multiple processes to the same object at the same time, for example oiling and cutting a piece of metal. Some applications might constrain the number of input and output objects of the same type. These constraints are cumbersome to write in first-order logic, because it does not have operators for referring to the number of elements in a set, but see extensions in [4].

```
(forall (?x ?s1 ?s2)
   (implies (and (occurrence-flow ?x ?s1 ?s2)
                 (legal ?s2))
            (and (occurrence-output ?x ?s1)
                 (occurrence-input  ?x ?s2)
                 (forall (?f)
                    (iff (ouput-state ?x ?s1 ?f)
                         (input-state ?x ?s2 ?f)))
                 (earlier ?s1 ?s2)
                 (not (exists (?s3)
                         (and (changed ?f ?s3)
                              (earlier ?s1 ?s3)
                              (earlier ?s3 ?s2)))))))
```

Expression 14: Basic Flow Constraint

```
(forall (?x ?s2)
   (implies (occurrence-input ?x ?s2)
            (exists (?s1)
               (occurrence-flow ?x ?s1 ?s2))))
```

Expression 15: Inbound Flow Constraint

```
(forall (?x ?s1)
   (implies (occurrence-output ?x ?s1)
            (exists (?s2)
               (occurrence-flow ?x ?s1 ?s2))))
```

Expression 16: Outbound Flow Constraint

Applying these relations to Expression 6, we get Expression 17, which identifies the piece of metal that flows between the occurrences of drilling and milling. The flow axioms above can infer the output of drilling and the input of milling, as well as the equivalence of the output state of drilling and the input state of milling. They also restrict other occurrences from being introduced that alter that state between the drilling and milling. Expression 17 does not satisfy Expression 5, Expression 15, or Expression 16, because Expression 17 allows a situation where there is no output providing the metal input to drilling.

```
(exists (?sDrill ?sMill ?m ?i ?o
         ?fDrillMetalOutState ?fMillMetalInState)
   (and (occurrence_of ?sDrill drilling(?m ?i ?o))
        (occurrence_of ?sMill milling(?m ?i ?o))
        (occurrence-input ?m ?sDrill)
        (occurrence-output ?m ?sMill)
        (earlier ?sDrill ?sMill)
        (legal ?sMill)
        (input-state ?m ?sMill ?fMillMetalInState)
        (occurrence-flow ?m ?sDrill ?sMill)))
```

Expression 17: Example Process Constraint Using Flows

4.2 With Complex Activity Occurrences

The axioms and examples of the previous sections allow processes to happen on their own, uncoordinated by any larger process. These axioms are too weak, because normally processes are part of larger ones that also have inputs and outputs, and coordinate subprocesses to achieve particular goals. For example, Expression 5 and Expression 14 only say that an input must be provided sometime earlier than the corresponding output, without regard to an overall coordinating process aimed at making a piece of metal of a certain shape. Expressions 6 and Expression 17 are similar in saying that there exists a sequence of drilling and milling sometime, without any other requirement. Processes that coordinate others are called *complex activities* in PSL and their occurrences are *complex occurrences*. Processes that do not are called *primitive activities* and *primitive occurrences*.

In this section, we revise the axioms of the last section to support complex processes, and add new ones to relate inputs and outputs of complex occurrences to inputs and outputs of contained primitive occurrences. Providing for the specification of inputs and outputs on complex occurrences separately from primitive occurrences means that complex inputs and outputs can be specified even when primitive occurrences have not, as usually happens at an early stage of design. This requires more relations and consistency rules than simply deriving complex inputs and outputs from primitive ones, but supports incremental construction of process models, resulting in designs that are more robust.

First, Expression 5 is updated to apply to complex occurrences as well as primitive, because the PSL relation EARLIER applies only to primitive occurrences. Expression 18 uses the PSL relations ROOT_OCC and LEAF_OCC to identify beginning and ending primitive suboccurrences of complex ones. The updated axiom also applies to primitive occurrences, because primitives are roots and leaves of themselves, see Expression 7 of [1]. Expression 14 has the same problem, and requires updates for complex occurrences, see Expression 31 and following.

```
(forall (?x ?s2 ?root2)
   (implies (and (occurrence-input ?x ?s2)
                 (root_occ ?root2 ?s2)
                 (legal ?root2))
            (exists (?s1 ?leaf1)
               (and (occurrence-output ?x ?s1)
                    (leaf_occ ?leaf1 ?s1)
                    (earlier ?leaf1 ?root2)))))
```

Expression 18: Revised Expression 5 for Primitive and Complex Occurrences

A common characteristic of complex flow models, including procedures in programming languages, is that inputs and outputs must either pass between suboccurrences of the same complex occurrence, or through inputs and outputs of their complex occurrence, that is, across the complex occurrence "boundary," as shown in Expression 19 through Expression 22. Expression 19 constrains suboccurrence inputs to come from earlier suboccurrences under the complex one, or from the complex occurrence itself. It uses the PSL relation MIN_PRECEDES, which is a partial ordering of legal occurrences that happen as part of a complex occurrence, and SUBACTIVITY_OCCURRENCE, which relates complex occurrences to the suboccurrences in them.[9,10] Expression 20 is the corresponding constraint for the outputs of suboccurrences. Expression 21 is the constraint for inputs to complex occurrences, that they must be matched to the output of some suboccurrence. The constraint does not allow a complex output to be provided directly from a complex input, because some occurrence must change the input state to the output state, due to PSL's "inertia" principle, and Expression 9. Expression 22 is the corresponding constraint for outputs of complex occurrences. Many flow models consider Expression 22 optional, because a process may be predefined and provide an output that happens not to be needed in any particular usage. Some models also consider Expression 21 optional, but a process with unused inputs is usually not properly defined.

[9] The SUBACTIVITY_OCCURRENCE relation should apply between complex occurrences as well as primitive and complex occurrences, but one of the PSL axioms inadvertantly prevents this. Axiom3 of the Activity Occurrence theory should be restricted to atomic occurrences:

```
(forall (?a ?occ ?s1 ?s2)
     (implies  (and  (occurrence_of ?occ ?a)
                     (subactivity_occurrence ?s1 ?occ)
                     (subactivity_occurrence ?s2 ?occ)
                     (atomic ?s1)
                     (atomic ?s2)
               (or   (min_precedes ?s1 ?s2 ?a)
                     (min_precedes ?s2 ?s1 ?a)
                     (= ?s1 ?s2))))
```

[10] Expression 19 uses the same technique as Expression 18 and Expression 31 to apply to both primitive and complex activities. The inequality is needed to prevent an occurrence input providing input to itself (PSL occurrences are subactivity occurrences of themselves).

```
(forall (?x ?s2 ?root2)
   (implies (and (occurrence-input ?x ?s2)
                 (root_occ ?root2 ?s2)
                 (legal ?s2))
            (or (exists (?s1 ?leaf1 ?a)
                   (and (occurrence-output ?x ?s1)
                        (leaf_occ ?leaf1 ?s1)
                        (min_precedes ?leaf1 ?root2 ?a)))
                (exists (?occ)
                   (and (occurrence-input ?x ?occ)
                        (subactivity_occurrence ?s2 ?occ)
                        (not (= ?occ ?s2)))))))
```

Expression 19: Suboccurrence Inputs

```
(forall (?x ?s1 ?leaf1)
   (implies (and (occurrence-output ?x ?s1)
                 (leaf_occ ?leaf1 ?s1)
                 (legal ?s1))
            (or (exists (?s2 ?root2 ?a)
                   (and (occurrence-input ?x ?s2)
                        (root_occ ?root2 ?s2)
                        (min_precedes ?leaf1 ?root2 ?a)))
                (exists (?occ)
                   (and (occurrence-output ?x ?occ)
                        (subactivity_occurrence ?s1 ?occ)
                        (not (= ?occ ?s1)))))))
```

Expression 20: Suboccurrence Outputs

```
(forall (?x ?occ ?a)
   (implies (and (occurrence-input ?x ?occ)
                 (occurrence_of ?occ ?a)
                 (not (primitive ?a)))
            (exists (?s)
               (and (occurrence-input ?x ?s)
                    (subactivity_occurrence ?s ?occ)
                    (not (= ?s ?occ))))))
```

Expression 21: Complex Inputs

```
(forall (?x ?occ ?a)
   (implies (and (occurrence-output ?x ?occ)
                 (occurrence_of ?occ ?a)
                 (not (primitive ?a)))
            (exists (?s)
               (and (occurrence-output ?x ?s)
                    (subactivity_occurrence ?s ?occ)
                    (not (= ?s ?occ))))))
```

Expression 22: Complex Outputs

Expression 24 is the complex version of Expression 6. It constrains the suboccurrences of all occurrences of a complex activity drillAndMill, shown in Expression 23, to happen in a certain order and have certain inputs and outputs. It satisfies the input and output axioms of complex activities above. In particular, the input of drilling is matched to the input of the complex occurrence containing it. It still does not address the question of input and output from other parameters beside metal, see Section 4.3.

```
(forall (?a ?m)
   (implies (= ?a drillAndMill(?m ?i1 ?i2))
            (and (activity ?a)
                 (metal ?m)
                 (instructions ?i1)
                 (instructions ?i2)
                 (exists (?o)
                    (subactivity drilling(?m ?i1 ?o) ?a))
                 (exists (?o)
                    (subactivity milling(?m ?i2 ?o) ?a)))))
```

Expression 23: Example Parameterized Term for a Complex Activity

```
(forall (?occDrillAndMill ?m ?i2 ?i2)
   (implies
      (occurrence_of ?occDrillAndMill drillAndMill(?m ?i2 i2))
      (exists (?sDrill ?sMill ?m ?o ?root ?leaf)
         (and (occurrence_of ?sDrill drilling(?m i1 ?o))
              (occurrence_of ?sMill milling(?m i2 ?o))
              (subactivity_occurrence ?sDrill ?occDrillAndMill)
              (subactivity_occurrence ?sMill ?occDrillAndMill)
              (occurrence-input ?m ?occDrillAndMill)
              (occurrence-input ?m ?sDrill)
              (occurrence-output ?m ?sDrill)
              (occurrence-input ?m ?sMill)
              (occurrence-output ?m ?sMill)
              (occurrence-output ?m ?occDrillAndMill)
              (root_occ ?root ?sDrill)
              (min_precedes ?root ?sDrill drillAndMill)
              (min_precedes ?sDrill ?sMill drillAndMill)
              (min_precedes ?sMill ?leaf drillAndMill)
              (leaf_occ ?leaf ?sMill)))))
```

Expression 24: Example Process Constraint for a Complex Occurrence

The boundary intuitions also require loosening Expression 10 and Expression 11 to allow input and output states to be in common between complex occurrences and their suboccurrences, as shown in Expression 25 and Expression 26. An occurrence is its own subactivity_occurrence, so these expressions do not weaken Expression 10 and Expression 11 for primitive occurrences.

```
(forall (?x1 ?s2 ?x2 ?s2 ?f)
   (implies (and (input-state ?x1 ?s1 ?f)
                 (input-state ?x2 ?s2 ?f))
            (and (= ?x1 ?x2)
                 (subactivity_occurrence ?s1 ?s2))))
```

Expression 25: Revised Expression 10 for Complex Activities

```
(forall (?x1 ?s2 ?x2 ?s2 ?f)
   (implies (and (output-state ?x1 ?s1 ?f)
                 (output-state ?x1 ?s2 ?f))
            (and (= ?x1 ?x2)
                 (subactivity_occurrence ?s1 ?s2))))
```

Expression 26: Revised Expression 11 for Complex Activities

Applying boundary intuitions to flows between outputs and inputs of complex processes requires that preconditions, postconditions, input states, output states, and flows between complex occurrences must be consistent with the corresponding aspects of the occurrences they contain. The most basic constraint is that preconditions and postconditions for roots and leaves respectively are the same as their containing complex occurrences, as shown in Expression 27 and Expression 28. These are general axioms, independent of inputs and outputs, but apply to input and output states also. Expression 27 combined with Expression 8 provides the typical constraint on complex input states that they are established before the complex occurrence starts. This still allows a suboccurrence that is not a root to take the complex input as its own input.

```
(forall (?occ ?s ?f)
    (iff (and (prior ?f ?s)
              (root_occ ?s ?occ))
         (prior ?f ?occ)))
```

Expression 27: Complex Preconditions

```
  forall (?occ ?s ?f)
    (iff (and (holds ?f ?s)
              (leaf_occ ?s ?occ))
         (holds ?f ?occ)))
```

Expression 28: Complex Postconditions

The analogous constraint on complex output states is stronger, because these are usually only established by the leaf occurrences. This corresponds to the intuition that outputs are available from a process all at once at the end, which is necessary to prevent later processes from being triggered by an output state of the complex occurrence before it is complete. This is shown in Expression 29, which requires the complex output state to be achieved by the leaf only. The expression still allows a suboccurrence that is not a leaf to provide the complex output object as output, and even achieve a postcondition that is required of the output object by the goals of the complex occurrence, see Section 6, as

long it does establish the complex output state. For example, a step in the milling process may achieve the required shape for the metal, but it will not place the piece of metal in the location defined as the output of the milling process until the process is completely done.[11]

```
(forall (?occ ?s ?f ?x ?s2)
    (implies (and (output-state ?x ?f ?occ)
                  (leaf_occ ?s ?occ))
             (and (achieved ?f ?s)
                  (implies (holds ?f ?s2)
                           (= ?s ?s2)))))
```

Expression 29: Complex Output States

The most basic consistency rule about flows between complex occurrences is that a flow exists between their suboccurrences, and vice-versa, as shown in Expression 30. This follows PSL's general principle of representing processes at the most concrete level, which means that flows between complex occurrences are just views onto the flows between primitive occurrences.

```
(forall (?occ1 ?occ2 ?x ?a1 ?a2)
    (iff (and (occurrence-flow ?x ?occ1 ?occ2)
              (occurrence_of ?occ1 ?a1)
              (occurrence_of ?occ2 ?a2)
              (not (primitive ?a1))
              (not (primitive ?a2)))
         (exists (?s1 ?s2)
            (and (occurrence-flow ?x ?s1 ?s2)
                 (subactivity_occurrence ?s1 ?occ1)
                 (subactivity_occurrence ?s2 ?occ2)
                 (not (= ?s1 ?occ1))
                 (not (= ?s2 ?occ2)))))))
```

Expression 30: Flow Between Complex Activities

Expression 30 does not restrict which suboccurrences realize the flows between complex occurrences. To specify this requires loosening Expression 14 to allow flows between complex inputs and those of suboccurrences, and likewise for complex outputs, as well as updating it to apply between complex occurrences, as Expression 18 does for occurrence inputs and outputs. These can be achieved in a single axiom, as shown in Expression 31. It uses the root and leaf technique of Expression 18, but narrows from Expression 14 to only the occurrences that are not subactivities of one another.

[11] Expression 8, Expression 9, and Expression 29 are usually true, but too restrictive for some applications. A process might take input or provide output while it is executing, rather than just at the beginning and end. These are called streaming parameters in UML 2. For example, a milling machine will take in and put out oil as it is running. Additional axioms can be defined to distinguish these kinds of inputs and outputs, and temporal constraints loosened for these kinds.

```
(forall (?x ?s1 ?s2 ?leaf1 ?root2)
   (implies (and (occurrence-flow ?x ?s1 ?s2)
                 (not (subactivity_occurrence ?s1 ?s2))
                 (not (subactivity_occurrence ?s2 ?s1))
                 (root_occ ?root2 ?s2)
                 (legal ?root2))
            (and (occurrence-output ?x ?s1)
                 (occurrence-input  ?x ?s2)
                 (forall (?f)
                    (iff (ouput-state ?x ?s1 ?f)
                         (input-state ?x ?s2 ?f)))
                 (leaf_occ ?leaf1 ?s1)
                 (earlier ?leaf1 ?root2)
                 (not (exists (?s3 ?root3 ?leaf3)
                         (and (changed ?f ?s3)
                              (root_occ ?root3 ?s3)
                              (leaf_occ ?leaf3 ?s3)
                              (earlier ?leaf1 ?root3)
                              (earlier ?leaf3 ?root2)))))))
```

Expression 31: Revised Expression 14 for Primitive and Complex Occurrences

This opens the way to constraints on complex inputs and outputs and those of suboccurrences, as shown in Expression 32 and Expression 33. Expression 32 requires that the input state of the complex occurrence only be modified by the suboccurrence to which the input flows. For example, if a piece of metal must appear at a certain location to be input to a milling process, only the specific suboccurrence to which the part is flowing can move the piece of metal once it is taken as input. Expression 33 is similar, but places weaker constraints on the effects of occurrences between the suboccurrence at the source of the flow and the leaf of the complex occurrence. This is to adhere to the requirement that only leaves achieve output states, in Expression 29.

```
(forall (?x ?s1 ?s2 ?root1 ?root2 ?a)
   (implies (and (occurrence-flow ?x ?s1 ?s2)
                 (subactivity_occurrence ?s2 ?s1)
                 (root_occ ?root2 ?s2)
                 (legal ?root2))
            (and (not (= ?s1 s2))
                 (occurrence-input ?x ?s1)
                 (occurrence-input ?x ?s2)
                 (forall (?f)
                    (iff (input-state ?x ?s1 ?f)
                         (input-state ?x ?s2 ?f)))
                 (root_occ ?root1 ?s1)
                 (occurrence_of ?s1 ?a)
                 (not (exists (?s3 ?root3 ?leaf3)
                         (and (changed ?f ?s3)
                              (root_occ ?root3 ?s3)
                              (leaf_occ ?leaf3 ?s3)
                              (earlier ?root1 ?root3 ?a)
                              (earlier ?leaf3 ?root2 ?a)))))))
```

Expression 32: Flows from Complex Inputs to Suboccurrences

```
(forall (?x ?s1 ?s2 ?leaf1 leaf2 ?a)
   (implies (and (occurrence-flow ?x ?s1 ?s2)
                 (subactivity_occurrence ?s1 ?s2)
                 (leaf_occ ?leaf2 ?s2)
                 (legal ?leaf2))
            (and (not (= ?s1 s2))
                 (occurrence-output ?x ?s1)
                 (occurrence-output ?x ?s2)
                 (leaf_occ ?leaf1 ?s1)
                 (occurrence_of ?s1 ?a)
                 (not (exists (?s3 ?root3 ?leaf3)
                         (and (participant ?x ?s3)
                              (root_occ ?root3 ?s3)
                              (leaf_occ ?leaf3 ?s3)
                              (earlier ?leaf1  ?root3 ?a)
                              (earlier ?leaf3 ?leaf2 ?a)))))))
```

Expression 33: Flows from Complex Suboccurrences to Outputs

The loosened Expression 31 also lets us relate suboccurrence inputs and outputs to flows by updating Expression 19 through Expression 22 to require flows connecting inputs and outputs within complex activities, as shown in Expression 34 through Expression 37. Expression 38 shows the complex version of Expression 17 conforming to the updated flow axioms.

```
(forall (?x ?s2 ?root2)
   (implies (and (occurrence-input ?x ?s2)
                 (root_occ ?root2 ?s2)
                 (legal ?s2))
            (or (exists (?s1 ?leaf1 ?a)
                   (and (occurrence-flow ?x ?s1 ?s2)
                        (leaf_occ ?leaf1 ?s1)
                        (min_precedes ?leaf1 ?root2 ?a)))
                (exists (?occ)
                   (and (subactivity_occurrence ?s2 ?occ)
                        (not (= ?occ ?s2))
                        (occurrence-flow ?x ?occ ?s2))))))
```

Expression 34: Suboccurrence Input Flows

```
(forall (?x ?s1 ?leaf1)
   (implies (and (occurrence-output ?x ?s1)
                 (leaf_occ ?leaf1 ?s1)
                 (legal ?s1))
            (or (exists (?s2 ?root2 ?a)
                   (and (occurrence-input ?x ?s2)
                        (root_occ ?root2 ?s2)
                        (min_precedes ?leaf1 ?root2 ?a)))
                (exists (?occ)
                   (and (subactivity_occurrence ?s1 ?occ)
                        (occurrence-output ?x ?occ)
                        (not (= ?occ ?s1))
                        (occurrence-flow ?x ?s1 ?occ))))))
```

Expression 35: Suboccurrence Output Flows

```
(forall (?x ?occ ?a)
   (implies (and (occurrence-input ?x ?occ)
                 (occurrence_of ?occ ?a)
                 (not (primitive ?a)))
            (exists (?s)
               (and (occurrence-input ?x ?s)
                    (subactivity_occurrence ?s ?occ)
                    (not (= ?s ?occ))
                    (occurrence-flow ?x ?occ ?s)))))
```

Expression 36: Complex Input Flows

```
(forall (?x ?occ ?a)
   (implies (and (occurrence-output ?x ?occ)
                 (occurrence_of ?occ ?a)
                 (not (primitive ?a)))
            (exists (?s)
               (and (occurrence-output ?x ?s)
                    (subactivity_occurrence ?s ?occ)
                    (not (= ?s ?occ))
                    (occurrence-flow ?x ?s ?occ)))))
```

Expression 37: Complex Output Flows

```
(forall (?occDrillAndMill ?m ?i2 ?i2)
   (implies
      (occurrence_of ?occDrillAndMill drillAndMill(?m ?i2 i2))
      (exists (?sDrill ?sMill ?m ?o ?root ?leaf)
         (and (occurrence_of ?sDrill drilling(?m i1 ?o))
              (occurrence_of ?sMill milling(?m i2 ?o))
              (subactivity_occurrence ?sDrill ?occDrillAndMill)
              (subactivity_occurrence ?sMill ?occDrillAndMill)
              (root_occ ?root ?sDrill)
              (min_precedes ?root ?sDrill drillAndMill)
              (min_precedes ?sDrill ?sMill drillAndMill)
              (min_precedes ?sMill ?leaf drillAndMill)
              (leaf_occ ?leaf ?sMill)
              (occurrence-flow ?m ?occDrillAndMill ?sDrill)
              (occurrence-flow ?m ?sDrill ?sMill)
              (occurrence-flow ?m ?sMill ?occDrillAndMill)))))
```

Expression 38: Example Complex Process Using Flows

All common process models require occurrences of complex processes to be "strongly nested." In PSL terminology, this means:

- Occurrences of a process cannot be directly under more than one complex occurrence. For example, the occurrences for drilling and milling within a complex occurrence for making a metal part cannot be directly contained in any other complex occurrence. This means SUBACTIVITY_OCCURRENCE forms a tree, rather than a directed acyclic graph.

- The boundary for inputs, outputs, and flows is the immediately containing complex process in the subactivity occurrence tree. For example, the complex occurrence immediately containing the occurrences of drilling and milling is the one through which entities are passed from outside processes to drilling and milling, and vice versa. This affects in Expression 19 through Expression 22 and Expression 34 through Expression 37.[12]

PSL does not impose the above constraints, but they can be added for applications that need them. The first one is established by Expression 39 and Expression 40 defining DIRECT_SUBACTIVITY_OCCURRENCE, a special kind of SUBACTIVITY_OCCURRENCE that requires no intervening complex occurrences, then using it to require at most one directly containing subactivity occurrence. The second constraint above can be written by using DIRECT_SUBACTIVITY_OCCURRENCE to revise Expression 19, as shown in Expression 41 (the equality tests are not needed with DIRECT_SUBACTIVITY_OCCURRENCE. Expression 20, Expression 21, and Expression 22 can be similarly tightened.[13]

[12] Some software languages provide for lexical and "spaghetti stack" scoping that gives deeply nested processes access to the inputs and outputs of more than the immediately containing process. PSL constraints could also be written for these applications.

[13] All common process models also provide named inputs and outputs, so the same object can be input or output in different ways to the same occurrence. For example, a process that finds home and work phone

```
(forall (?s ?occ1)
   (iff (direct_subactivity_occurrence ?s ?occ1)
        (and (subactivity_occurrence ?s ?occ1)
             (not (= ?s ?occ1))
             (not (exists (?occ2)
                     (and (subactivity_occurrence ?s ?occ2)
                          (subactivity_occurrence ?occ2 ?occ1)
                          (not (= ?occ2 ?s))
                          (not (= ?occ2 ?occ1)))))))))
```

Expression 39: Direct Subactivity Occurrences

```
(forall (?s ?occ1 ?occ2)
   (if (and (direct_subactivity_occurrence ?s ?occ1)
            (direct_subactivity_occurrence ?s ?occ2))
       (= ?occ1 ?occ2)))
```

Expression 40: Strong Subactivity Occurrence Nesting

```
(forall (?x ?s2 ?root2)
   (implies (and (occurrence-input ?x ?s2)
                 (root_occ ?root2 ?s2)
                 (legal ?s2))
            (exists (?occ)
               (and (direct_subactivity_occurrence ?s2 ?occ)
                    (or (exists (?s1 ?leaf1 ?occ ?a)
                           (and (occurrence_of ?occ ?a)
                                (occurrence-output ?x ?s1)
                                (leaf_occ ?leaf1 ?s1)
                                (min_precedes ?leaf1 ?root2 ?a)))
                        (occurrence-input ?x ?occ)))))))
```

Expression 41: Revised Expression 19 for Strong Nesting

4.3 With Multiple Activity Occurrences

In PSL, occurrences can be contained directly under multiple complex ones, as long as the strong nesting constraints of Expression 40 and Expression 41 are not applied, providing a representation for multiple views of the same occurrences. For example, the milling process partially overlaps other processes, such as lubricating the milling machine, supplying power to it, and so on. In a typical flow model, the only way to bring these partially overlapping processes together is to make one large process. In PSL, each suboccurrence can be shared under multiple separate complex occurrences, each providing its own constraint on the occurrence, and supporting partially overlapping

---

numbers for a person may return the same number for both. If these two outputs are to be treated differently, then they need to be uniquely identified for flows to unambiguously refer to them. This can be axiomitized by introducing names as a kind of PSL object and extending the input, output, and flow relations with names.

complex occurrences. This makes the construction of the process description incremental and flexible.

We can formalize this by defining input and output relations that are relative to a complex occurrence, along with constraints limiting the earlier input and output constraints to hold only on relative inputs and outputs. Expression 42 extends the OCCURRENCE-INPUT and OCCURRENCE-OUTPUT relations from Expression 2 to include an additional complex occurrence that the inputs and outputs are relative to. This means inputs and outputs of the same occurrence can vary depending on which of the many complex occurrences it might be under. This should not apply to multiple containment due to strong nesting, as required by Expression 43 and Expression 44. For example, the inputs and outputs of the steps involved in milling a piece of metal should be the same when viewed from the overall factory process that contains it. Since occurrences are subactivity occurrences of themselves, these expressions also say that an occurrence can act as a view of its own inputs and outputs.

```
(forall (?x ?s)
   (implies (or (occurrence-input-rel ?x ?s ?occ)
                (occurrence-output-rel ?x ?s ?occ))
            (and (object ?x)
                 (not (state ?x))
                 (activity_occurrence ?s)
                 (activity_occurrence ?occ)
                 (subactivity_occurrence ?s ?occ)
                 (not (= ?s ?occ)))))
```

Expression 42: Relative Inputs and Outputs

```
(forall (?x ?s ?occ1 ?occ2)
   (implies (and (occurrence-input-rel ?x ?s ?occ1)
                 (subactivity_occurrence ?occ1 ?occ2))
            (occurrence-input-rel ?x ?s ?occ2)))
```

Expression 43: Relative Inputs Consistency

```
(forall (?x ?s ?occ1 ?occ2)
   (implies (and (occurrence-output-rel ?x ?s ?occ1)
                 (subactivity_occurrence ?occ1 ?occ2))
            (occurrence-output-rel ?x ?s ?occ2)))
```

Expression 44: Relative Outputs Consistency

With these relations, Expression 19 through Expression 22 can be updated to use relative inputs and outputs. Expression 45 shows this for Expression 19 and the others can be similarly amended, along with the flow axioms.

```
(forall (?x ?s2 ?viewocc ?root2)
   (implies (and (occurrence-input-rel ?x ?s2 ?viewocc)
                 (root_occ ?root2 ?s2)
                 (legal ?s2))
            (or (exists (?s1 ?leaf1 ?a)
                   (and (occurrence-output-rel ?x ?s1 ?viewocc)
                        (leaf_occ ?leaf1 ?s1)
                        (min_precedes ?leaf1 ?root2 ?a)))
                (exists (?s1 ?occ)
                   (and (occurrence-input-rel ?x ?occ viewocc)
                        (subactivity_occurrence ?s ?occ)
                        (not (= ?occ ?s1)))))))
```

Expression 45: Relative Suboccurrence Inputs

Applying relative inputs and outputs to the milling and drilling example of Expression 24, the drillAndMill complex activity is broken into aspects related to movement of metal from drilling to milling and aspects related to moving oil from a reservoir, as shown in Expression 46, Expression 47, and Expression 48. Inputs and outputs to drilling and milling are specified under two complex activities, as shown in Expression 49 and Expression 50. The lubricating activity involves pumping oil from a reservoir, since milling and drilling would not normally use the same oil. The outputs of oil from drilling and milling back to the reservoir or a cleaning activity are omitted, for simplicity. Finally, Expression 51 ensures the same occurrences of milling and drilling are under both of the complex occurrences, while allowing the occurrences of pumping to be only under one.[14] The expressions for drillAndMillShape and drillAndMillLubricate do not require them to be used together, even though they are in the example. This means alternative lubrication processes can be combined with drilling and milling, or vice versa.

```
(forall (?a ?m)
   (implies (= ?a drillAndMill(?m ?i1 ?i2 ?r))
            (and (activity ?a)
                 (metal ?m)
                 (instructions ?i1)
                 (instructions ?i2)
                 (reservoir ?r)
                 (subactivity drillAndMillShape(?m ?i1 ?i2) ?a)
                 (subactivity drillAndMillLubricate(?m ?r) ?a))))
```

Expression 46: Example Parameterized Term for Complex Activity

[14] Additional constraints are needed to prevent inference engines from introducing other occurrences of drilling and milling other than the ones required to exist in Expression 49, Expression 50, and Expression 51. These are the closure constraints as described in Section 7 of [1].

```
(forall (?a ?m)
   (implies (= ?a drillAndMillShape(?m ?i1 ?i2))
            (and (activity ?a)
                 (metal ?m)
                 (instructions ?i1)
                 (instructions ?i2)
                 (exists (?o)
                    (subactivity drilling(?m ?i1 ?o) ?a)))
                 (exists (?o)
                    (subactivity milling(?m ?i2 ?o) ?a)))))
```

Expression 47: Example Parameterized Term for Shaping

```
(forall (?a ?m)
   (implies (= ?a drillAndMillLubricate(?m ?r))
            (and (activity ?a)
                 (metal ?m)
                 (reservoir ?r)
                 (exists (?o)
                    (subactivity drilling(?m ?i1 ?o) ?a)))
                 (exists (?o)
                    (subactivity milling(?m ?i2 ?o) ?a)))))
```

Expression 48: Example Parameterized Term for Lubricating

```
(forall (?occDrillAndMillShape ?m ?i1 ?i2 ?r)
   (implies
      (occurrence_of ?occDrillAndMillShape
                     drillAndMillShape(?m ?i1 ?i2))
      (exists (?sDrill ?sMill ?m ?o1 ?o2 ?root ?leaf)
         (and (occurrence_of ?sDrill drilling(?m ?i1 ?o1))
              (occurrence_of ?sMill milling(?m ?i2 ?o2))
              (subactivity_occurrence ?sDrill
                                      ?occDrillAndMillShape)
              (subactivity_occurrence ?sMill
                                      ?occDrillAndMillShape)
              (occurrence-input ?m ?occDrillAndMillShape)
              (occurrence-input-rel ?m ?sDrill
                                    ?occDrillAndMillShape)
              (occurrence-output-rel ?m ?sDrill
                                     ?occDrillAndMillShape)
              (occurrence-input-rel ?m ?sMill
                                    ?occDrillAndMillShape)
              (occurrence-output-rel ?m ?sMill
                                     ?occDrillAndMillShape)
              (occurrence-output ?m ?occDrillAndMillShape)
              (root_occ ?root ?sDrill)
              (min_precedes ?root ?sDrill drillAndMillShape)
              (min_precedes ?sDrill ?sMill drillAndMillShape)
              (min_precedes ?sMill ?leaf drillAndMillShape)
              (leaf_occ ?leaf ?sMill)))))
```

Expression 49: Example Complex Occurrences for Shaping

```
(forall (?occDrillAndMillLubricate ?m ?r)
   (implies
      (occurrence_of ?occDrillAndMillLubricate
                     drillAndMillLubricate(?m ?r))
      (exists (?sDrill ?sMill ?m ?i1 ?i2 ?o1 ?o2 ?root ?leaf)
         (and (occurrence_of ?sDrill drilling(?m ?i1 ?o1))
              (occurrence_of ?sMill milling(?m ?i2 ?o2))
              (occurrence_of ?sPumping1 pumping(?r ?o1))
              (occurrence_of ?sPumping2 pumping(?r ?o2))
              (subactivity_occurrence ?sDrill
                                      ?occDrillAndMillLubricate)
              (subactivity_occurrence ?sMill
                                      ?occDrillAndMillLubricate)
              (subactivity_occurrence ?sPumping1
                                      ?occDrillAndMillLubricate)
              (subactivity_occurrence ?sPumping2
                                      ?occDrillAndMillLubricate)
              (occurrence-output-rel ?o1 ?occPumping1
                                     ?occDrillAndMillLubricate)
              (occurrence-input-rel ?o1 ?sDrill)
              (occurrence-output-rel ?o2 ?occPumping2)
              (occurrence-input-rel ?o2 ?sMill)
              (root_occ ?root ?sDrill)
              (min_precedes ?root ?sPumping1
                            drillAndMillLubricate)
              (min_precedes ?sPumping1 ?sDrill
                            drillAndMillLubricate)
              (min_precedes ?sDrill ?sPumping2
                            drillAndMillLubricate)
              (min_precedes ?sPumping2 ?sMill
                            drillAndMillLubricate)
              (min_precedes ?sMill ?leaf drillAndMillLubricate)
              (leaf_occ ?leaf ?sMill)))))
```

Expression 50: Complex Occurrences for Lubricating

```
(forall (?occDrillAndMill ?m ?i1 ?i2 ?r)
   (implies
     (occurrence_of ?occDrillAndMill drillandMill(?m ?i1 ?i2 ?r))
     (exists (?occDrillAndMillShape ?occDrillAndMillLubricate
              ?sDrill ?sMill)
        (and (occurrence_of ?occDrillAndMillShape
                            drillAndMillShape(?m ?i1 ?i2))
             (occurrence_of ?occDrillAndMillLubricate
                            drillAndMillLubricate(?m ?r))
             (occurrence_of ?sDrill drilling(?m ?i1 ?r))
             (occurrence_of ?sMill milling(?m ?i1 ?r))
             (subactivity_occurrence ?occDrillAndMillShape
                                     ?occDrillAndMill)
             (subactivity_occurrence ?occDrillAndMillLubricate
                                     ?occDrillAndMill)
             (subactivity_occurrence ?sDrill
                                     ?occDrillAndMillShape)
             (subactivity_occurrence ?sDrill
                                     ?occDrillAndMillLubricate)
             (subactivity_occurrence ?sMill
                                     ?occDrillAndMillShape)
             (subactivity_occurrence ?sMill
                                   ?occDrillAndMillLubricate)))))
```

Expression 51: Occurrences Under More Than One Superoccurrence

5. Input and Output Axioms at the Activity Level

In typical flow modeling and programming languages, inputs and outputs are defined for activities, rather than occurrences. This reflects the common pattern that inputs and outputs of multiple occurrences of the same activity are usually the same. In addition, it is useful to delay the introduction of relative inputs and outputs to later stages of design. Convenience relations can be defined for early stage representation of inputs and outputs, as shown in Expression 52, Expression 53, and Expression 54. These can be used to extend activity terms, such as milling in Expression 1, to include inputs and outputs, as shown in Expression 55. Similar axioms can be defined for input and output states, and used with activity terms.

```
(forall (?x ?a)
   (implies (or (activity-input ?x ?a)
                (activity-output ?x ?a))
            (and (object ?x)
                 (not (state ?x))
                 (activity ?a))))
```

Expression 52: Activity Input and Output Types

```
(forall (?x ?a ?s)
  (implies (and (activity-input ?x ?a)
                (occurrence_of ?s ?a))
           (exists (?occ)
              (occurrence-input-rel ?x ?s ?occ))))
```

Expression 53: Activity Input Constraint

```
(forall (?x ?a ?s)
  (implies (and (activity-output ?x ?a)
                (occurrence_of ?s ?a))
           (exists (?occ)
              (occurrence-output-rel ?x ?s ?occ))))
```

Expression 54: Activity Output Constraint

```
(forall (?a ?m ?i ?o)
   (implies (= ?a milling(?m ?i ?o))
            (and (activity ?a)
                 (metal ?m)
                 (instructions ?i)
                 (oil ?o)
                 (activity-input ?m ?a)
                 (activity-input ?i ?a)
                 (activity-input ?o ?a)
                 (activity-output ?m ?a)
                 (activity-output ?o ?a))))
```

Expression 55: Example Activity Input and Output

Concurrency in PSL is represented by combining multiple activities into one, with the restriction that only occurrences of the combined activity appear in the occurrence tree. This is because the synergistic effects of the activities happening together often are different from the activities happening alone.[15] PSL provides the CONC function to combine activities concurrently into one. Whatever the synergistic effects of concurrency, it is still expected that inputs and outputs will combine without interference, as shown in Expression 56 and Expression 57. PSL happens to consider the occurrence of a combined activity as not having subactivity occurrences, but if this were to be allowed, then these axioms could be written on occurrences and generalized for activities as done above for activity inputs and outputs.

```
(forall (?a1 ?a2)
   (implies (activity-input ?x ?a1)
            (activity-input ?x (conc ?a1 ?a2))
```

Expression 56: Concurrent Activity Inputs

```
(forall (?a1 ?a2)
   (implies (activity-output ?x ?a1)
            (activity-output ?x (conc ?a1 ?a2))
```

Expression 57: Concurrent Activity Outputs

[15] See section 5.2 of [1].

6. Metatheoretic Issues

Some aspects of the concepts of input and output are not representable in PSL, because they concern how process specifications are written, rather than how a process executes at runtime. For example, some flow modeling languages support optional inputs and outputs. These objects are not required to participate in the occurrences of an activity, but can if they are available, which is a statement that the allowable process specifications include those that do not provide a certain input or output. It is not a statement about process execution, because it may be that the optional inputs and outputs of an activity happen to always participate in the occurrences, but rather it is a statement about what inputs and outputs are allowed to be specified. This means if all legal occurrences of the activity happen to use the optional inputs and outputs, there is no first-order constraint on the occurrences that can say it could be otherwise. It is a statement about PSL statements ("metatheoretic") or involves the quantification over relations ("second-order").[16]

Another metatheoretic issue is to determine whether a participant is an input, output, or neither. The axioms of the previous sections specify the consequences of identifying a participant as input or output, but not the requirements for making the identification in the first place. For example, the milling machine is a participant in the milling process, but would not normally be considered an input, even though it is affected by the process as much as any other participant, due to wear and tear. A view could be constructed in which the machine was an input to the factory process as a whole and output much later as a worn out machine. We might take the position that all participants are either inputs or outputs, as shown in Expression 58, which narrows the problem to distinguishing input from output.

```
(forall (?x ?s)
   (implies (participant ?x ?s)
            (exists (?occ)
               (or (occurrence-input-rel ?x ?s ?occ)
                   (occurrence-output-rel ?x ?s ?occ)))))
```

Expression 58: No Side Effects

An important metatheoretic issue is to categorize views of inputs and outputs, as enabled by the relations defined in Section 4.3. For example, Section 2 lists multiple possibilities for inputs and outputs of the milling activity. These views are intimately connected with the purposes of the process designer. For example, if the goal of the milling process were to produce the part that is most needed at any particular moment, then the shape of the part would be determined dynamically by the milling process itself, perhaps though a brokering architecture, rather than given as input.

[16] It is possible to prevent an activity from having optional inputs, however, but requiring all occurrences of an activity to have the same inputs and outputs.

Automating the categorization of views requires two steps:

1. Formalizing goals as kinds of preconditions and postconditions

   Not all preconditions and postconditions are goals, because some of the requirements and effects of an activity are not the reasons for defining the activity. For example, a milling process might make oil dirty, which is a state that holds after milling, but it is not the reason for milling. In particular, output states are not usually goals. The fact that a milling machine places a finished piece of metal at a certain location to pass it out to the next step in a factory process is not the goal of the milling process, which is just to produce a certain change in shape of the metal. Goals may vary by stakeholders of the process.

2. Using goals to determine inputs and outputs

   This hinges on the observation that process goals usually involve only some of the participants in the process [5]. For example, oil might not be referred to in the goals for milling, because a technique for milling might exist that does not need oil. Non-goal participants are only required for particular ways of reaching a goal, not for reaching the goal in general. They might still be inputs and outputs, but would be under a separate view from those related to the goal. The relation of participants and goals is complicated by the fact that process designers often assume certain technologies are available or not for implementation of their goals [6].

Extensions to PSL can be easily added for the first step above, as shown in the axioms of Expression 59, but there is little benefit without the second, which requires metatheoretic statements about the objects involved in the preconditions and postconditions. This aspect is difficult in PSL, because it represents preconditions and postconditions as primitive elements, rather than statements that might be examined in a metatheoretic way. Even if relations are defined to expose the objects referred to by preconditions and postconditions, as shown in Expression 60, there is nothing to require the process designer to expose all the objects, defeating formalization of the metatheoretic constraints between goals and inputs and outputs.

```
(forall (?f ?stakeholder ?s)
   (implies (goal ?f ?stakeholder ?s)
            (and (state ?f)
                 (object ?stakeholder)
                 (not (state ?stakeholder))
                 (activity_occurrence ?s))))

(forall (?f ?stakeholder ?s)
   (implies (goal ?f ?stakeholder ?s)
            (holds ?f ?s)))

(forall (?f ?stakeholder ?s >x)
   (implies (and (goal ?f ?stakeholder ?s)
                 (about ?x ?f))
            (participant ?x ?s)))
```

Expression 59: Goals

```
(forall (?x ?f)
   (implies (about ?x ?f)
            (and (state ?f)
                 (object ?x)
                 (not (state ?x)))))

(forall (?f ?s ?x)
   (implies (and (about ?x ?f)
                 (changes ?f ?s))
            (participant ?x ?s)))
```

Expression 60: Relation of States to Participants

However, the metatheoretic approach above can provide a more rigorous understanding of categories of inputs and outputs, for example, as provided in IDEF0 [7], a popular high-level input/output dependency graph. IDEF0 distinguishes three kinds of inputs, as shown in Figure 2: regular inputs, mechanisms, and controls. Regular inputs can be understood as those directly mentioned by the goal of a process, such as a piece of metal in a milling process. Mechanisms are those inputs that are not directly mentioned by the goal, but involved in reaching it and consumed or depleted at least partially in the process. Controls are inputs that are not consumed, because they represent information used to direct the process,[17] and can be required by goals or not. This approach suggests that IDEF0 could be extended with mechanism outputs, which are the outputs not involved in the goals of the process, for example dirty oil output by milling.

[17] A PSL theory of consumable resources is under development that is applicable to formalizing control inputs.

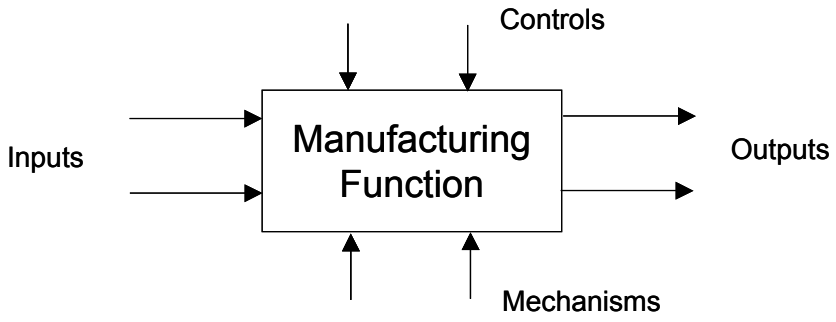

Figure 2: IDEF0

Once inputs and outputs are categorized into goal- and mechanism-related, we can define processes purely with activities that use goal-driven inputs and outputs, without committing to which mechanism implements them. We would like goal-based processes to be reused for many mechanisms that might achieve the goals. This requires a way to use the activities with mechanism inputs and outputs in the goal-driven processes without replacing all the goal-based activities with mechanism-based ones. In effect, the mechanism-based activities are special cases of the goal-driven activities. The relation of inputs and outputs to activity specialization [1] will be addressed in a later article.

7. Conclusion

This paper defines a formalization of inputs and outputs for PSL at increasing levels of detail, for application in early and late stages of design. Inputs and outputs are shown to be early stage notions independent of existing PSL concepts. They constrain application of some PSL concepts in later stages, preconditions and postconditions in particular. The paper defines these constraints, providing a concrete semantics for input and output. The constraints are defined both inside and outside the context of composed processes, and in multiple composed processes to support multiple views of inputs and outputs. Goals are identified as an important metatheoretic issue that is outside the scope of PSL, but useful in providing rigor and extensions to IDEF0. This points the way to further ontology development in the areas of process specialization and goal implementation.

8. References

[1] Bock, C., Gruninger, M., "PSL: A Semantic Domain for Flow Models," to appear in Journal of Software and Systems Modeling, 2004.

[2] Hayes, P., Menzel, C., "A Semantics for the Knowledge Interchange Format," Workshop on the IEEE Standard Upper Ontology, IJCAI, Seattle, 2001.

[3] Object Management Group, "UML 2.0 Superstructure Specification," http://www.omg.org/cgi-bin/doc?ptc/03-08-02, August 2003.

[4] Barwise, J., Etchemendy, J., "The Language of First-Order Logic," The University of Chicago Press, 1993.

[5] Bock, C., "Goal-driven Modeling," Journal Of Object-Oriented Programming, Vol. 13, No. 5, September 2000

[6] Swartout, W., Balzer, R., "On the Inevitable Intertwining of Specification and Implementation," Communications of the ACM, Vol. 25, No. 7, July 1982.

[7] National Institute of Standards and Technology, "Integration Definition For Function Modeling (IDEF0)," http://www.idef.com/idef0.html, December, 1993.