NISTIR 7348

Interprocess Communication in the Process Specification Language

Conrad Bock

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

NISTIR 7348

Interprocess Communication in the Process Specification Language

Conrad Bock
Manufacturing Systems Integration Division
Manufacturing Engineering Laboratory

October 2006

U.S. Department of Commerce
Carlos M. Gutierrez, Secretary

Technology Administration
Robert Cresanti, Under Secretary of Commerce for Technology

National Institute of Standards and Technology
William Jeffrey, Director

Interprocess Communication in the Process Specification Language

Conrad Bock
October 31, 2006

Interprocess communication concerns how processes affect which entities are involved in other processes. This paper provides dimensions for characterizing interprocess communication, and places common process language capabilities within them. It formalizes loosely and tightly coupled processes as extensions of the Process Specification Language (PSL), to reduce ambiguity and increase expressiveness as compared to commonly used process languages. The paper also shows how to incrementally translate common process language elements to PSL. This generates small expressions even for large process models.

1 Introduction

Common process languages provide ways to specify which entities are involved in a process (*participation*), and to indicate that processes determine the entities involved in others (*interprocess communication*). These capabilities are available in graphical form, such as the Unified Modeling Language (UML) [1], and textual, such as programming and web service interchange languages [2][3][4]. For example, a language for manufacturing might include a way to specify that a process for drilling involves a piece of metal, a machine, and an operator, as well as a way to specify that a factory process determines the particular piece of metal, machine, and operator involved. Even a process as simple as addition involves two or more numbers, and languages supporting it provide ways for other processes, such as accounting, to specify which two particular numbers are involved.

From the point of view of a particular process, communication happens in two directions: entities come from other processes ("inbound") and go to other processes ("outbound"). Process specifications control communication along at least two dimensions:

1. Identifying other processes to communicate with. This dimension ranges from more to less restriction on the other processes (degree of *coupling*, "tight" to "loose"). For each communicated entity, a process might:

    a. specify exactly which other process to communicate with.
    b. specify only what must be achieved by the other process before it gives an inbound entity, or after it receives an outbound entity (the *effect*).
    c. not specify anything about the other processes.

    For example, a factory process might specify exactly how to drill a piece of metal (1.a, outbound), or it might only specify that the metal be drilled, but

not exactly how (1.b, outbound). It also might provide the drilled piece metal to whatever other process needs it outside the factory (1.c, outbound). A drilling process might specify exactly which process gives it metal (1.a, inbound), or might only specify that the metal must be lubricated by the process that gives it (1.b, inbound), or it might not place any restriction on which process gives it (1.c, inbound).[1]

Within options b and c above a process might:

b/c.i. specify another process (the *delegate*) to decide which other process gives or receives the communicated entity.
b/c.ii. not specify a delegate.

For example, a factory process might specify that an operator decides how to drill a piece of metal (1.b.i, outbound), or it might not restrict how it is drilled (1.b.ii, outbound). It might provide the drilled piece metal to a distributor to determine which process receives it (1.c.i, outbound), or it might not restrict which process receives it at all (1.c.ii, outbound). A drilling process might specify that the operator determine the process by which the metal is lubricated before taking the metal as input (1.b.i, inbound), or it might only require it to be lubricated somehow (1.b.ii, inbound). A drilling process might specify that the operator determine the process giving the piece of metal (1.c.i, inbound), or might not restrict which process gives it at all (1.c.ii, inbound).

2. Determining when communication happens. A process might:

   a. give entities only when it finishes, and receive entities only when it starts.
   b. give and receive entities anytime during the process ("ongoing" communication).

   This applies to processes at both ends of a communication. For example, a drilling process might receive a piece of metal only when it starts, and give the metal back only when it is done (2.a). A factory process, on the other hand, might give and receive pieces of metal as it goes, in communication with drilling and other processes outside the factory (2.b).

This paper uses the above dimensions to categorize communication elements of common process languages and guide their formalization in the Process Specification Language (PSL).[2] The paper significantly extends support for participation in PSL, which currently addresses sequences of occurrences primarily. It updates earlier PSL translations of flow

[1] Other possibilities exist between the categories above. For example, a process might specify that an inbound entity comes from a process that takes particular steps, but not restrict it from taking other steps in addition. Very few process languages have such a high level of expressiveness, the Process Specification Language (PSL) being an exception, see Section 3.

[2] This paper does not categorize how a process chooses which entities are communicated to other processes. Any entity can be communicated, see the third generalization in Section 8, except for restrictions in PSL, see footnote 22.

models, inputs, outputs, and messaging [5][6][8]. Section 2 categorizes some common software process communication terminology under the framework above. Section 3 gives a brief introduction to PSL. Section 4 discusses ways of extending languages and identifies the ones used in this paper. Section 5 gives some generically useful extensions to PSL applied in the paper. Section 6 gives translations of common process language sequencing elements needed as a basis for later sections. It includes a translation algorithm that scales to large process models. Section 7 gives PSL extensions formalizing the loose coupling end of the spectrum above (1.c). Section 8 gives extensions formalizing the medium coupling part of the spectrum (1.b). The tightest coupling end of the spectrum (1.a) is directly representable in PSL, see usage examples for functions in [8]. Section 9 shows how to use the tight coupling extensions of Section 8 to represent loose coupling, and briefly describes the corresponding techniques in software engineering. Section 10 covers future work.

2 Participation terminology

The categories of Section 1 appear in many ways and under many names in common process languages. Table 1 shows process communication terminology from software engineering for each category.[3] The table takes the viewpoint of a particular process communicating with other processes, as in Section 1. Entities come from other processes (inbound) and go to other processes (outbound). The rows cover the first dimension from tightest to loosest coupling. The columns show the second dimension, which applies to the process at the center of the viewpoint (inbound) or to other processes (outbound), as indicated in each cell of the table. Invoking a function or procedure specifies exactly which other process will receive the outbound entities (1.a). Invoking operations on objects and components delegates determination of the receiving process to an entity, only specifying a name for the effect to be achieved with the outbound entities (1.b.i).[6] Invoking a function or operation starts a new execution of another process and provides entities to it at the same time.[4] When entities are provided anytime while the function or operation is ongoing, it has streaming parameters in UML. Ongoing forms in category 1.b.i are messages to agents or active objects,[5] for outbound entities, and subscription for inbound entities.[8, 9] Invoking operations on abstract datatypes does not delegate determination of the other process, but the invoking process does not specify it either (1.b.ii).[4, 10] Preconditions restrict which other processes can provide inbound entities to those that achieve the specified effect (1.b.ii).[11] Signals are often received from other processes for the same purpose.[5, 12] An ongoing form of this category is port messages, where a process sends messages to its own ports, rather than identifying a delegate. The delegate is determined by the context in which the sending process is used, see Section 9.2. Publication and sending signals[5, 12] are ways to provide entities to other processes through a delegate without requiring an effect (1.c.i / 2.b).[9, 13] Parameters of functions, operations, messages, and signals specify the types and numbers of entities input, with no restriction on which processes give or provide them (1.c.ii).[14] The ongoing form of these are streaming parameters in UML [1][9].

[3] This section assumes familiarity with process communication techniques in software engineering.

Determination of external process	(2.a) Entities accepted only at start of process or provided only at finish	(2.b) Entities accepted or provided anytime during process
(1.a) Exact process [4]	Function / procedure invocation (outbound)	Function / procedure invocation with streaming parameters (outbound) [5]
(1.b.i) Effect,[6] **with delegate** [4]	Object / component operation invocation (outbound)	Operation invocation with streaming parameters (outbound) , [5, 7] Sending agent / active object [5] messages (outbound) , [8] Subscription (inbound) [9]
(1.b.ii) Effect,[6] **no delegate**	Abstract datatype operation invocation (outbound) , [4, 10] Preconditions (inbound) [11]	Port messages (inbound, outbound) , Receiving signals (inbound) [5, 12]
(1.c.i) Entities only, with delegate	(No common name)	Publication (outbound) , [9, 13] Sending signals (outbound) [5, 12]
(1.c.ii) Entities only, no delegate [14]	Parameters (inbound)	Streaming parameters (inbound) [5, 15]

Table 1: Participation Terminology in Software Engineering

[4] The invoked function or operation can return entities back to the invoking process (inbound to the invoking process, see footnote 14). These are always communicated to the invoking process (1.a).

[5] This is UML terminology [1].

[6] Most software languages specify the effect of processes by giving the effect a name (the *operation* in UML, or informally the "method"), rather than specifying the effect directly. The language user is expected to infer the effect from the name. For example, the effect of an operation called "balance account" is that the account will be balanced. More reliable techniques, such as preconditions and postconditions [1][10], give information about the effect that does not rely on interpreting the name. Since operations are only identified by their names, they might even be intended to imply something about how a process is carried out, see footnote 1.

[7] After a delegate has started a particular process for the operation invocation, communication through streaming parameters is between the invoking process and the one chosen by the delegate (1.a).

[8] An agent or active object is the delegate that determines the receiving process based on effect of those processes on the entities. Usually the receiving process is "inside" the agent, for example, business process in an organization.

[9] Subscription and publication usually identify a "clearing house" (the delegate) that determines the other processes with which communication happens. If not, subscription can be classified as 1.b.ii/2.b (inbound), and publication as 1.c.ii/2.b (outbound).

[10] Abstract datatypes have a single process per operation (no delegation), but that process is not specified by the invoker (1.a). Abstract datatypes do not support subclassing, and were overtaken by objects and components. See examples in [8].

[11] This considers as a single process all the processes feeding into those providing inbound entities, transitively, because preconditions on the inbound entities can be achieved by any of them.

[12] Signals can be used for asynchronous operation invocation (1.b.i/2.a with no return values, see footnote 4), as notifications of achieved effects (subscription, 1.b.i/2.b, publication, 1.c.i/2.b), or as simple provision of an object (1.c.i/2.b). The table only shows them as notification of achieved effects.

[13] Processes accepting published entities are typically those requiring notification of an achieved effect (subscription, 1.b.i). Process providing them do not require this, however, so publication falls under 1.c.i.

[14] Return parameters are a special kind of parameter for entities returned a process when it is finished, see footnote 4. The term "argument" is often used for entities outbound from the invoking process to the invoked one. Parameters might be considered a restriction on the processes giving or receiving entities because the processes must give or receive entities of a certain type and in a certain quantity.

[15] Also see flow port messages in the Systems Modeling Language (SysML) [11].

This paper clarifies the meaning of the participation patterns above with a formalization based on process execution, rather than on the terminology in which a process happens to be specified, see Section 3. This facilitates communication between subject matter experts and software practitioners, who use different terminology for specification elements that have the same process execution. Subject matter experts use the categories above, but often implicitly, or with names that vary significantly. For example, experts in manufacturing use "input" and "output" to refer to parameters of any kind (1.c.ii / 2.a-b), while some business modelers will use the term "message" for that, as well as for agent and port messages (1.b.i-ii / 2.b).

Terminology differences between the software practitioners and subject matter experts cause major inefficiencies and failures in system construction. For example, the term "message" among software practitioners often refers to object or component messages (1.b.i / 2.a), but a business expert might be referring to port messages (1.b.ii / 2.b). The software practitioner not realizing this might hear "message" from a business expert and rule out a centralized workflow system, because centralized systems usually do not result in object or component messages between the participants, even though such a system using port messages would have satisfied the business expert requirements. The inverse problem occurs in the Business Process Modeling Notation (BPMN) [12], which does not allow messages within certain boundaries. The software practitioner might assume this means a centralized workflow system must be used within the boundary, even though a distributed system would have satisfied the business process requirements just as well.[16]

Another common miscommunication between software practitioners and subject matter experts is that software practitioners draw a very strong distinction between categories 2.a and 2.b, while subject experts usually do not. Software programming languages notate elements from the two categories differently, with communication in category 2.a notated in the public declaration of process specifications, and communication in 2.b embedded in the detailed internals of the specification. Subject matter experts, on the other hand, treat these as part of a spectrum of possible process interactions [13]. For them, the public declaration can include which functions are invoked, and messages sent, if necessary.[17]

3 PSL

Capturing the meaning of participation requires a language that refers directly to processes as they actually occur. For example, in a process that drills a piece of metal, then mills the same piece, it is expected that the piece of metal will stop participating in the drilling process before it begins participating in milling. Since the processes will

[16] This miscommunication appears particularly around the terms "orchestration" and "choreography." Software practitioners often take these as referring to centrally controlled and distributed processes, respectively. The business process designer, on the other hand, usually intends an "orchestration" diagram to specify only sequencing of steps, possibly with nonstreaming parameters (1.c.ii / 2.a), and a "choreography" to include port messages (1.b.ii, 1.c.ii / 2.b). This leads to the misinterpretations above about central and distributed system control described above.

[17] This is supported in UML's Object Constraint Language [14].

happen repeatedly with different pieces of metal, this constraint must refer to actual occurrences of drilling and milling that happen at specific times, rather than those processes generally. Common flow modeling languages express this constraint, but not in a machine-verifiable way, because the occurrence-level meaning of the languages is usually specified only in natural language.

This paper uses PSL to express participant constraints, and assumes familiarity with PSL fundamentals [5].[18] PSL has language elements for referring to the separate executions of the same process (*occurrence* of an *activity* in PSL), for example to refer to the multiple milling operations that happen in a factory at different times. It defines a simple structure that covers all possible ways these occurrences can follow each other in time, called the *occurrence tree*. The process designer uses these concepts to write constraints on which sequences of occurrences are allowed for a particular process, for example to specify what must happen, may happen, or must not happen during milling.

The PSL approach supports incremental processes specification over a wide range of the process design lifecycle. Process constraints can be declared generally or specifically, as needed by the stage of design. For example, early stages usually define loose constraints, because the domain expert is just sketching out broad requirements. Constraints 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 standard 18629 at the International Organization for Standardization (ISO) [16]. It is the result of 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, written in the Common Logic Interchange Format [17], using the question mark convention for variables.

4 Extending Languages

Extending a language begins with identifying patterns of using the language without extensions ("methodology"). These patterns document how to use it to achieve the desired functionality. Methodologies use a language as it is, rather than spending time and effort to extend it. They have the disadvantages that the patterns can be complicated to write in the existing language, do not provide guidance to readers of the language as to

[18] This paper uses PSL Core and Outer Core [15].

which pattern is being used, cannot be the subject of constraints and reasoning, and do not ensure the pattern is applied in the same way wherever it is used. Language extensions address these problems, but require additional effort to define and integrate them into the existing language.

Language extensions follow usage patterns chronologically in a general cycle of moving methodological techniques into language concepts. Users apply the same techniques repeatedly, then tooling eventually adapts to support them. If the techniques are popular enough, languages evolve or are created to incorporate them as first-class concepts. For example, in arithmetic, multiplication is taught as a shorthand for repeated addition of the same number, and exponentiation is similarly a shorthand for repeated multiplication. In programming languages patterns of using the `goto` statement evolved into structured programming constructs, and code conditionalized on object type and function pointers became part of object orientation, and abstract classes evolved into interfaces. In software modeling, techniques for composable architectures were incorporated into modeling and middleware languages [1][18][19][20][21]. In ontologies, computationally efficient patterns of using first order logic underlie most of the language constructs in the Ontology Web Language [22].

There are various ways to extend a language, including mapping techniques, formal metalanguages, additional language constructs, and predefined reusable elements. Mapping techniques are for specifying translation from one language to another.[19] Formal metalanguages can be applied when the structure of sentences in the existing language are regularized enough to express the extension as "statements about statements" [25]. For example, the horn clause pattern in first order logic restricts statements that are disjunctions of literals to have at most one positive literal. Additional language constructs can be added to a language to stand in for a usage pattern. For example, typical iteration constructs in programming languages, such as the `WHILE` loop arose from common usage patterns of the `goto` construct [26]. Predefined reusable elements can be developed in the existing language, and applied repeatedly in place of a usage pattern. For example, typical programming languages have libraries of functions to perform common tasks, such as string manipulation, rather than specifying them each time they are needed [2].

This paper primarily adopts the reusable elements approach when extending PSL, by providing predefined instances of PSL concepts. This has the advantage of embedding into the existing language, unlike formal metalanguages, and does not require translation as mapping techniques do. In addition, when applied in the context of PSL, this approach enables constraints to be defined on combining the reusable elements with existing PSL relations and with each other.[20] These constraints give necessary conditions for use of

[19] One of simplest kinds are "fill in the blanks" templates, which are usually written in the syntax of the language being extended, as C++ macros [2] or PSL grammars [23], with additional syntax for specifying where the blanks are, and what should fill them. More powerful techniques include generalized mapping languages, which support machine-manipulable specification of arbitrarily complex translations [24].

[20] One of the advantages of PSL over typical process languages is that it enables formally defining the relation between new and existing language elements.

the new elements. The process designer gives sufficient conditions when using the elements. These determine whether an object is categorized by the concepts formalized in this paper, for example, whether it is an input, output, or message. Necessary conditions alone cannot specify this, only the process designer can, see Section 7.1 and Section 6 of [6]. It is an advantage of PSL that consistency of necessary and sufficient conditions can be tested automatically.[21]

5 Generic PSL Extensions

Additional relations are defined in this section for convenience, entirely in terms of existing ones in PSL. Expression 1 defines two additional participation relations based on the PSL PARTICIPATES_IN relation, which link objects with the time at which they participate in occurrences.[22] The first subexpression defines the PARTICIPANT relation as indicating that an object participates in an occurrence at some time during it. The second subexpression requires that any participant in an occurrence is also a participant in complex occurrences containing it (the PSL relation SUBACTIVITY_OCCURRENCE gives the containing complex occurrences). The third subexpression defines the ACTIVITY-PARTICIPANT relation as a generalization of PARTICIPANT for all occurrences of an activity. This includes participation in occurrences of concurrent aggregations of activities (subactivities of atomic activities in PSL), see OCCURRENCE-OFA in Expression 2.

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

(forall (?x ?s ?occSuper)
   (if (and (participant ?x ?s)
            (subactivity_occurrence ?s ?occSuper))
       (participant ?x ?occSuper)))

(forall (?x ?a)
   (iff (activity-participant ?x ?a)
        (forall (?occ)
           (if (occurrence-ofA ?occ ?a)
               (participant ?x ?occ)))))
```

Expression 1: Participation Relations

Expression 2 defines a variant of the PSL OCCURRENCE_OF relation that accounts for concurrent activities. A concurrent activity in PSL is an aggregation of multiple activities into a single atomic one, so an occurrence of the activity can appear on the occurrence

[21] Necessary conditions can be used to check that the process designer uses the reusable elements properly. Detecting contradictions requires the designer to write closure expressions to eliminate unspecified usages of the elements. Otherwise their existence might be inferred by the necessary conditions.

[22] PSL does not restrict the participants of a process, except they cannot be processes or points in time.

tree (complex activities are groupings of atomic activities, but their occurrences are grouping of atomic occurrences on the occurrence tree, so are not on the occurrence tree directly). The OCCURRENCE-OF-CONC relation indicates which activities are happening in an atomic occurrence, including occurrences of concurrent aggregations. The last subexpression defines the OCCURRENCE-OFA relation to apply to both atomic and complex occurrences, using OCCURRENCE-OF-CONC for atomics.

```
(forall (?s ?a ?aConc)
   (if (occurrence-of-conc ?s ?a)
       (and (activity_occurrence ?s)
            (activity ?a)
            (atomic ?a))))

(forall (?s ?a ?aConc)
   (if (and (occurrence-of-conc ?s ?a)
            (occurrence_of ?s ?aConc))
       (atomic ?aConc)))

(forall (?s ?a ?a1 ?a2)
   (if (and (occurrence_of ?s ?a)
            (atomic ?a)
            (subactivity ?a1 ?a))
       (occurrence-of-conc ?s ?a1)))

(forall (?s ?a ?aConc)
   (if (and (occurrence-of-conc ?s ?a)
            (occurrence_of ?s ?aConc))
       (exists (?aConc)
          (subactivity ?a ?aConc))))

(forall (?s ?occ ?a)
   (iff (occurrence-ofA ?s ?a)
        (or (occurrence_of ?s ?a)
            (occurrence-of-conc ?s ?a))))
```

Expression 2: OCCURRENCE-OF-CONC and OCCURRENCE-OFA Relations

Expression 3 defines convenience relations for subactivity occurrences. The SUBACTIVITY-OCCURRENCE-NEQ relation identifies subactivity occurrences that are not the same as the superoccurrence. The SUBACTIVITY-OCCURRENCE-OF relation identifies subactivity occurrences of a particular activity, including under concurrent aggregation.

```
(forall (?s ?occ ?a)
   (iff (subactivity-occurrence-neq ?s ?occ ?a)
        (and (subactivity_occurrence ?s ?occ)
             (not (= ?s ?occSuper)))))

(forall (?s ?occ ?a)
   (iff (subactivity-occurrence-of ?s ?occ ?a)
        (and (occurrence-ofA ?s ?a)
             (subactivity_occurrence ?s ?occ))))
```

Expression 3: SUBACTIVITY-OCCURRENCE-OF Relation

Expression 4 defines relations for identifying all parts of the occurrence tree for single "execution" of a complex activity. An execution of a complex activity may have many possible variations in the steps that happen under it, especially if there are few constraints placed by the process designer. In PSL terms, a complex occurrence is a branch in a subtree of the occurrence tree (the *activity tree*, see Section 5.1 of [5]). Since even the first step may not be completely determined, the full execution is represented as sets of activity trees that "start" at the same time (a *grove* of activity trees). All the trees in the grove have roots that occur immediately after the same occurrence (the *grove root*).[23] A grove is identified by the grove root and the complex activity. The OCC-GROVE-ROOT relation is between a grove root and the complex occurrences in the activity trees of the grove. The OCC-IN-GROVE relation is similar, but requires the grove to be of a particular complex activity. The GROVE relation identifies groves by their grove root and complex activity. The SUBOCC-IN-GROVE relation is for all suboccurrences in a grove. Finally, SAME-GROVE relates occurrences in the same grove.[24]

```
(forall (?sGroveRoot ?occ ?aOccRootAct)
    (iff (and (occ-grove-root ?sGroveRoot ?occ)
              (occurrence_of (root_occ ?occ) ?aOccRootAct))
         (= (root_occ ?occ)
            (successor ?sGroveRoot ?aOccRootAct))))

(forall (?occ ?sGroveRoot ?aGroveAct)
   (iff (occ-in-grove ?occ ?sGroveRoot ?aGroveAct)
        (and (occurrence_of ?occ ?aGroveAct)
             (occ-grove-root ?sGroveRoot ?occ))))

(forall (?sGroveRoot ?aGroveAct)
   (iff (grove ?sGroveRoot ?aGroveAct)
        (exists (?occ)
          (occ-in-grove ?occ ?sGroveRoot ?aGroveAct))))
```

[23] This assumes complex occurrences do not have the initial occurrence as root. The initial occurrence has no predecessor. It would simplify representation of executions to assume a "dummy" root occurrence that has no effect and takes no time to occur. Then groves would always have a single activity tree. It would also address other issues in representing workflow engines, see Section 6.3.

[24] It is equivalent to PSL's SAME_GROVE.

```
(forall (?s ?sGroveRoot ?aGroveAct)
   (iff (subocc-in-grove ?s ?sGroveRoot ?aGroveAct)
        (exists (?occ)
           (and (occ-in-grove ?occ ?sGroveRoot ?aGroveAct)
                (subactivity-occurrence-neq ?s ?occ)))))

(forall (?s ?aOcc1Act ?aOcc2Act)
   (iff (same-grove ?occ1 ?occ2)
        (exists (?sGroveRoot ?aGroveAct)
           (and (occ-in-grove ?occ1 ?sGroveRoot ?aGroveAct)
                (occ-in-grove ?occ2 ?sGroveRoot ?aGroveAct)))))
```

Expression 4: Grove Relations

Expression 5 defines the SAME-OCC-BRANCH and SAME-ACT-BRANCH relations, which are useful in connection with groves to tell whether two occurrences are separate occurrences on the same branch of the occurrence tree or activity tree, respectively.

```
(forall (?s1 ?s2 ?a)
   (if (same-occ-branch ?s1 ?s2)
       (or (= ?s1 ?s2)
           (earlier ?s1 ?s2)
           (earlier ?s2 ?s1))))

(forall (?s1 ?s2 ?a)
   (if (same-act-branch ?s1 ?s2 ?a)
       (or (= ?s1 ?s2)
           (min_precedes ?s1 ?s2 ?a)
           (min_precedes ?s2 ?s1 ?a))))
```

Expression 5: SAME-OCC-BRANCH and SAME-ACT-BRANCH Relations

Expression 6 defines relations that generalize PSL relations on atomic occurrences to both atomic and complex occurrences.

```
(forall (?s)
  (iff (legalA ?s)
       (legal (root_occ ?s))))

(forall (?s1 ?s2)
   (iff (earlierA ?s1 ?s2)
        (exists (?s1Leaf)
          (and (leaf_occ ?s1Leaf ?s1)
               (earlier ?s1Leaf (root_occ ?s2))))))

(forall (?s ?occ)
   (iff (root-occA ?s ?occ)
        (and (= (root_occ ?s) (root_occ ?occ))
             (subactivity_occurrence ?s ?occ)))))
```

```
(forall (?s ?occ)
   (iff (leaf-occA ?s ?occ)
        (exists (?sLeaf)
          (and (leaf_occ ?sLeaf ?s)
               (leaf_occ ?sLeaf ?occ)
               (subactivity_occurrence ?s ?occ)))))

(forall (?s1 ?s2)
   (iff (min-precedesA ?s1 ?s2)
        (exists (?s1Leaf)
           (and (leaf_occ ?s1Leaf ?s1)
                (min_precedes ?s1Leaf (root_occ ?s2))))))

(forall (?s1 ?s2)
   (iff (same-occ-branchA ?s1 ?s2)
        (exists (?s1Leaf ?s2Leaf)
           (and (leaf_occ ?s1Leaf ?s1)
                (leaf_occ ?s2Leaf ?s2)
                (same-occ-branch ?s1Leaf ?s2Leaf)))))

(forall (?s1 ?s2)
   (iff (same-act-branchA ?s1 ?s2)
        (and (same-act-branch (root_occ ?s1) root_occ ?s2))
             (exists (?s1Leaf ?s2Leaf)
                (and (leaf_occ ?s1Leaf ?s1)
                     (leaf_occ ?s2Leaf ?s2)
                     (same-act-branch ?s1Leaf ?s2Leaf))))))
(forall (?f ?s)
   (iff (priorA ?f ?s)
        (prior ?f (root_occ ?s))))

(forall (?f ?s ?sLeaf)
   (iff (holdsA ?f ?s)
        (exists (?s1Leaf)
           (and (leaf_occ ?sLeaf ?s)
                (holds ?f ?sLeaf)))))

(forall (?f ?occ)
   (iff (achievedA ?f ?occ)
        (exists (?sSub)
           (and (subactivity_occurrence ?sSub ?occ)
                (achieved  ?f ?sSub)))))

(forall (?f ?occ)
   (iff (falsifiedA ?f ?occ)
        (exists (?sSub)
           (and (subactivity_occurrence ?sSub ?occ)
                (falsified  ?f ?sSub)))))
```

Expression 6: Complexity-agnostic Relations

Expression 7 defines a template for an exclusive disjunction boolean operator. Given a list of boolean subexpressions, it expands to an expression that is true if exactly one of the subexpressions is true. It is a recursive template expansion with a base case for no subexpressions, assuming OR and XOR of no subexpressions is false.

```
<xor ?p1 ?p2 ?p3 ... > expands to
  (or (and ?p1 (not (or ?p2 ?p3 ...)))
      (and (not ?p1) (xor ?p2 ?p3 ...)))

<xor> expands to false
```

Expression 7: XOR Relation

6 Context for participation constraints

Participation constraints usually are defined in the context of a larger process. For example, a production process on a factory floor will at least partly determine which piece of metal, machine, and operator are involved in drilling.[25] In PSL, this larger factory process is a *complex activity* with constraints on the occurrence of drilling under each *complex occurrence* of the factory. A typical complex activity will also have constraints other than participation, for example time sequencing of the occurrences of subactivities under it, and reaction to changes caused by other processes. All these constraints apply to occurrences of the complex activity and their suboccurrences.

This section shows how to define the complex activity context for participation constraints. Section 6.1 covers subactivities and relations between complex occurrences and suboccurrences. Section 6.2 gives a translation for typical sequencing elements in common process languages, including an incremental algorithm that scales to large process models.

6.1 Subactivities and their usages

Subactivities in PSL indicate which activities can happen under a complex one. For example, a drilling activity might have subactivities for stabilizing the piece to be drilled, tapping, drilling, reaming, and deburring. Expression 8 shows the ACTIVITY and SUBACTIVITY relations in PSL for this example.[26] The SUBACTIVITY relation does not require that the specified subactivities are all used in all occurrences of drilling. For example, some occurrences of drilling might not have the reaming step. It also does not prevent inferring additional subactivities, unless additional expressions disallow these. For example, other expressions may support inferring that preinspection of the piece is a subactivity of drilling, unless the process designer adds an expression that preinspection

[25] Even in agent-based processes, there are larger processes under which the agents assume they can interact with each other, for example, a production process based on bidding for resources.

[26] The activity expressions can be omitted, because they are deducible from the subactivity expressions, but the specification is more robust if they are included, because the subactivity expressions may change.

is not allowed as a subactivity. Finally, the subactivity relation does not imply ordering among the steps. For example, the subactivities of drilling could occur in any order, even though they are given an order when writing them out in Expression 8.[27]

```
(activity drilling)
(activity stabilizing)
(activity tapping)
(activity drilling2)
(activity reaming)
(activity deburring)

(subactivity stabilizing drilling)
(subactivity tapping drilling)
(subactivity drilling2 drilling)
(subactivity reaming drilling)
(subactivity deburring drilling)
```

Expression 8: Subactivity Examples

It significantly simplifies translation from common process languages to PSL to identify the occurrences of subactivities within a complex occurrence corresponding to elements in the process language. For example, a flow chart for the drilling process may have a node labelled "stabilizing" with an arrow pointing to another node labelled "tapping," to indicate the order of their occurrences under drilling.[28] The correspondence between occurrences of stabilizing and the nodes in the flow chart are formalized as *usage relations* between the complex occurrence and its suboccurrences. This enables translation of each step in the process description separately to its own expression, as shown in Sections 6.2 and 6.3, because multiple expressions can refer to the same suboccurrence through a usage relation. The resulting translations have much smaller expressions, which read more easily and potentially support faster automated inference. It also enables separate identification of suboccurrences when the same activities appear multiple times in the same process description. Finally, usage relations enable the same suboccurrence to be under multiple processes that are not under each other, giving further support for incremental process specification, see Expression 42.

The definitions of usage relations are generated as part of the translation to PSL, as shown by the example in Expression 9, for a portion of the drilling example above. They are specializations of the OCCURRENCE-USAGE relation defined in Expression 10. The first subexpression requires the "used" occurrence to be a suboccurrence of the complex occurrence "using" it, but not the same as the complex occurrence (occurrences are their

[27] All common process models require occurrences of complex processes to be "strongly nested," but PSL and the axioms and translations in this paper do not. In PSL terms, strong nesting means occurrences are required to be direct suboccurrences of no more than one complex occurrence (the PSL subactivity_occurrence relation forms a tree, rather than a directed acyclic graph). Applications needing this can define an additional PSL relation for direct suboccurrences (see Expressions 39 and 40 in [6]), and a constraint that limits direct superoccurrences to no more than one.

[28] Programming languages notate this textually with one line appearing under another, or separated by a statement delimiter.

own subactivity occurrences in PSL). The second subexpression requires that a suboccurrence be used only by complex occurrences in the same activity tree (by the same "execution" of the complex activity).[29] The second subexpression does not affect the minimum or maximum number of suboccurrences that will be related to a complex occurrence. This is addressed by the sequencing expressions in Section 6.2.

```
(forall (?occ ?s)
   (if (stabilizing-usage ?s ?occ)
       (occurrence-usage ?s ?occ stabilizing drilling)))

(forall (?occ ?s)
   (if (tapping-usage ?s ?occ)
       (occurrence-usage ?s ?occ tapping drilling)))
```

Expression 9: Usage Relation Examples

```
(forall (?s ?occ ?asAct ?aOccAct)
   (if (occurrence-usage ?s ?occ ?asAct ?aOccAct)
       (and (subactivity-occurrence-of ?s ?occ ?asAct)
            (not (= ?s ?occ))
            (occurrence_of ?occ ?aOccAct))))

(forall (?s ?occ1 ?asAct ?aOccAct ?occ2)
   (if (and (occurrence-usage ?s ?occ1 ?asAct ?aOccAct)
            (occurrence-usage ?s ?occ2 ?asAct ?aOccAct)
       (= (root_occ ?occ1) (root_occ ?occ2)))))
```

Expression 10: General Usage Relation

When the same process description has multiple steps for the same subactivity, then usually an additional constraint should be added to prevent the usage relations from linking the same complex occurrence to the same suboccurrence (if there is only one step for each activity, this can be inferred because an occurrence is of exactly one activity in PSL). The constraints must be stated for each process description separately to remain first-order. A usage relation might link multiple complex occurrences to the same suboccurrence, however, because each execution of a process has multiple complex occurrences, see description of Expression 4. Different usage relations in separate process descriptions might link multiple complex occurrences to the same suboccurrence, because the constraints of both process descriptions might be satisfied by the same suboccurrences, see footnote 27 and Expression 42.

6.2 Sequencing

Most common processes languages have elements to indicate that one process starts after another finishes (sometimes called "control flow"). These translate to PSL as constraints

[29] The same complex occurrence cannot appear in separate activity trees, so it is not necessary to require that the complex occurrences be in the same grove. This is implied by having the same root.

on the order of occurrences on the occurrence tree. Some languages have elements that indicate multiple sequences can happen in parallel (sometimes called “forks” or “and splits”), as well to specify alternative sequences, based on some decision about which to do (sometimes called “decisions” or “or splits”). These are usually accompanied by elements indicating that multiple sequences are to complete before starting another (sometimes called “joins” or “and joins”), as well as elements indicating multiple sequences share a common later sequence (sometimes called “merge” or “or joins”).

This section gives example translations of sequencing elements to PSL, and a translation algorithm that scales to large process models. The translation starts with the sequencing elements of typical process languages, employing the usage relations of Section 6.1 to identify the suboccurrences being sequenced. Usage relations enable each path of sequencing elements from one suboccurrence to another to be translated separately from other paths, making the expressions smaller than if the entire process were translated at once. This is facilitated by assuming the sequencing elements are intended to allow other steps to be inserted later, rather than be a complete description. Separate closure expressions can be defined to indicate the description is complete.

To simplify the translation, a number of “preprocessing” changes to a typical process description are made before applying the PSL translations:

- If a step has no incoming control flows, a special “initial node” and control flow from it to the step are introduced. This assumes that steps with no incoming control flow can start when the containing process does.

- If a step has no outgoing control flows, a special “final node” and a control flow from the step to it are introduced. This assumes that a process is finished when all its steps with control flow to final nodes are finished.

- If a step has multiple control flows coming in, then a join or merge is inserted before the step to gather the flows together, depending on the semantics of the process description.

- If a step has multiple control flows going out, then a fork or decision is inserted after the step to split the flows apart, depending on the semantics of the process description. The effect of this change and ones above is that each step has exactly one control flow coming in and one going out.

- If a decision in the process has an “else” option, this is changed to be the negation of the other options on the decision. If the decision has its own activity for deciding, this is broken out into another step.

A step in a process language can correspond to multiple suboccurrences, for example, when the step is in a loop, or is after a merge of parallel flows. The usage relation for such a step will link a single complex occurrence to multiple suboccurrences. The translation to PSL must identify which suboccurrences for one usage occur after which

suboccurrences for another usage due to the control flow between the corresponding steps. This is more specialized than the PSL MIN_PRECEDES relation, because the process may have other steps going on in parallel that overlap any particular control flow. Expression 11 defines a relation that implies MIN_PRECEDES, but not the inverse. Since the suboccurrences identified by a usage may be complex, MIN-PRECEDESA is applied instead of MIN_PRECEDES (see Expression 4). The suboccurrences related by FOLLOWS must be identified by usage relations, but this requirement must be stated for each process description separately to remain first-order, see examples below. The minimum and maximum number of suboccurrences that can follow another depends on the particular process description being translated, see below.

```
(forall (?s1 ?s2 ?a)
   (if (follows ?s1 ?s2 ?a)
       (min-precedesA ?s1 ?s2 ?a)))
```

Expression 11: FOLLOWS Relation

The definition of FOLLOWS above allows other suboccurrences between the two that it orders. For example, a process description for drilling may say that tapping follows stabilizing a piece of metal. This does not prevent other steps from happening in between tapping and stabilizing, but does not require them either. The translations below assume that intervening steps are allowed. A version of FOLLOWS can be defined using the PSL NEXT_SUBOCC relation, which does not allow intervening steps.

Expression 12 is a typical constraint on follows that requires occurrences following later ones to be later themselves. For example, if drilling occurred repeatedly, it would require an occurrence of tapping that follows an occurrence of stabilizing to be before another occurrence of tapping that follows a later occurrence of stabilizing.[30] Additional constraints are added on each usage to prevent its occurrences from overlapping in time. Combined with Expression 12 this gives the execution semantics for queueing systems, where "tokens" are consumed in first-in, first-out buffers [1][7].[31]

```
(forall (?s1 ?s2 ?s11 ?s22 ?a)
   (if (and (follows ?s1 ?s2 ?a)
            (follows ?s11 ?s22 ?a))
       (if (min-precedesA ?s1 ?s11 ?a)
           (min-precedesA ?s2 ?s22 ?a))))
```

Expression 12: Ordered Follows

[30] This situation arises in flow models whenever execution proceeds multiple times to same sequence of steps faster than the steps can complete. For example, changes might be detected faster than they can be processed by a sequence of steps, or a loop might have a fork to a separate sequence of steps, and the loop iterates faster than the sequence of steps can complete.

[31] For systems with buffers other than first-in, first-out, Expression 12 is modified for the order in which tokens are drawn from the buffers. If the order varies with a single flow model, variations of Expression 12 and FOLLOWS are defined for each kind of buffering order.

Expression 13 is an example of translating a control flow between an initial node and a step, based on the example of Section 6.1. The first subexpression requires stabilizing a piece of metal to occur if drilling occurs, and that stabilizing does not follow any other suboccurrence due to a control flow. The subexpression does not rule out occurrences of stabilizing being after other suboccurrences of drilling, just being after any others due to control flow, which is the purpose of the FOLLOWS relation. For example, drilling may have multiple initial nodes, with no specification of which is to happen first. Other constraints can require stabilizing to be the PSL root of drilling if needed. This way the translator does not need to traverse more of the process description to generate Expression 13.

The second subexpression is needed if the initial step is inside a larger loop or after a merge of parallel flows (there is a merge intervening in the control flow between the initial node and the first step). In these cases, the STABILIZING-USAGE relation will link a single complex occurrence to multiple suboccurrences. The second subexpression requires that at most one of these occurrences follow no other occurrence. Combined with the existential in first subexpression, exactly one stabilizing occurrence will follow no other occurrence. All the other occurrences identified by STABILIZING-USAGE will follow some other occurrence.

```
(forall (?occ)
   (if (occurrence_of ?occ drilling)
       (exists (?sStabilizing)
          (and (stabilizing-usage ?sStabilizing ?occ)
               (not (exists (?sFollowed)
                       (follows ?sFollowed ?sStabilizing
                                drilling)))))))

(forall (?occ ?s1 ?s2)
   (if (and (occurrence_of ?occ drilling)
            (stabilizing-usage ?sStabilizing1 ?occ)
            (stabilizing-usage ?sStabilizing2 ?occ)
            (not (exists (?sFollowed)
                    (follows ?sFollowed ?sStabilizing1
                             drilling)))
            (not (exists (?sFollowed)
                    (follows ?sFollowed ?sStabilizing2
                             drilling))))
       (= ?sStabilizing1 ?sStabilizing2)))
```

Expression 13: Translation of Initial Control Flow Example

Expression 14 shows an example of translating control flow between steps when there are no intervening elements between them, or only intervening forks. The first subexpression requires tapping to occur after stabilizing when stabilizing occurs. The subexpression does not require stabilizing itself to occur, Expression 13 does that.

The second subexpression is needed when the control flow is inside a larger loop or after a merge of parallel flows. In these cases, the stabilizing and tapping usages will link a single complex occurrence to multiple suboccurrences for each usage. The FOLLOWS relation identifies which pairs of stabilizing and tapping occurrences happen due to a single control flow. The subexpression allows no more than one occurrence of tapping following each occurrence of stabilizing, and at most one occurrence of stabilizing to be followed by each occurrence of tapping. Without the second constraint, the first subexpression could be satisfied by an occurrence of tapping that follows all the occurrences of stabilizing. The second subexpression does not require stabilizing or tapping to exist, or that they follow each other, this is done by the first subexpression. The two subexpressions combined require exactly one tapping occurrence to follow each stabilizing occurrence, and exactly one stabilizing occurrence to be followed by each tapping occurrence. Control flows to other steps after a fork are translated in the same way as Expression 14.

```
(forall (?occ ?sStabilizing)
   (if (and (occurrence_of ?occ drilling)
            (stabilizing-usage ?sStabilizing ?occ))
       (exists (?sTapping)
          (and (tapping-usage ?sTapping ?occ)
               (follows ?sStabilizing ?sTapping drilling)))))

(forall (?occ ?sStabilizing1 ?sTapping1
              ?sStabilizing2 ?sTapping2)
   (if (and (occurrence_of ?occ drilling)
            (stabilizing-usage ?sStabilizing1 ?occ)
            (tapping-usage ?sTapping1 ?occ)
            (stabilizing-usage ?sStabilizing2 ?occ)
            (tapping-usage ?sTapping2 ?occ)
            (follows ?sStabilizing1 ?sTapping1 drilling)
            (follows ?sStabilizing2 ?sTapping2 drilling))
       (iff (= ?sStabilizing1 ?sStabilizing2)
            (= ?sTapping1 ?s1Tapping2))))
```

Expression 14: Translation of Control Flow between Subactivities Example

To simplify the presentation below, Expression 15 defines a template for generating the second subexpression in Expression 14 for other usage relations and complex activities. Given two usage relations and a complex activity, the template expands to an expression constraining each two suboccurrences identified by the usage relations to be paired by FOLLOWS as described for the second subexpression of Expression 14. This is true for usages in many process descriptions. See exceptions in the algorithm for chained sequence elements below.

```
<follows-at-most-one ?usage1 ?usage2 ?aProcess> expands to
  (forall (?occ ?s1 ?s1Follow ?s2 ?s2Follow)
     (if (and (occurrence_of ?occ ?aProcess)
              (?usage1 ?s1 ?occ)
              (?usage2 ?s1Follow ?occ)
              (?usage1 ?s2 ?occ)
              (?usage2 ?s2Follow ?occ)
              (follows ?s1 ?s1Follow ?aProcess)
              (follows ?s2 ?s2Follow ?aProcess))
         (iff (= ?s1 ?s2)
              (= ?s1Follow ?s2Follow))))
```

Expression 15: FOLLOWS-AT-MOST-ONE Relation

Expression 16 shows an example of translating control flow between steps when there is an intervening join element. The first subexpression requires inspection to occur after lubricating and deburring. It does not require lubricating and deburring to occur, other expressions for control flow into those steps do that. It also assumes the additional subactivities, usages, and other control flow expressions are defined for deburring and inspection. The second and third subexpressions are needed in the same situations as the second subexpression of Expression 14. After template expansion, and in combination with the existential of the first subexpression, they require exactly one occurrence of inspection to follow exactly one pair of lubricating and deburring occurrences.

```
(forall (?occ ?sLubricating ?sDeburring)
   (if (and (occurrence_of ?occ drilling)
            (lubricating-usage ?sLubricating ?occ)
            (deburring-usage ?sDeburring ?occ))
       (exists (?sInspection)
          (and (inspection-usage ?sInspection ?occ)
               (follows ?sLubricating ?sInspection drilling)
               (follows ?sDeburring ?sInspection drilling)))))

(follows-at-most-one lubricating-usage inspection-usage drilling)

(follows-at-most-one deburring-usage inspection-usage drilling)
```

Expression 16: Translation of Join in Control Flow Example

Expression 17 shows an example of translating control flow between steps when there is an intervening decision element. The first subexpression defines the condition tested by the decision, using the PSL STATE predicate and the HOLDSA relation, which relates occurrences to conditions that are present immediately after the occurrence completes (see Section 6.3). The second subexpression requires reaming to occur if the result of drilling2 is not within tolerance. The third subexpression requires deburring to occur if the result of drilling2 is within tolerance. These two subexpressions do not require drilling2 to occur, other expressions for control flow into the drilling2 step do that. This assumes the additional subactivities, usages, and other control flow expressions are defined for drilling2 and reaming.

The third and fourth subexpressions are needed in the same situations as the second subexpression of Expression 14. After template expansion, and in combination with the existentials of the first two subexpressions, they require exactly one occurrence of reaming or inspection, but not both, to follow exactly one occurrence of drilling2. The last subexpression is needed when the decision conditions are not provided in the process description, or to ensure they are exclusive and complete. It requires that drilling2 be followed by either deburring or reaming, but not both.

```
(state withinTolerance)

(forall (?occ ?sDrilling2)
   (if (and (occurrence_of ?occ drilling)
            (drilling2-usage ?sDrilling2 ?occ)
            (not (holdsA withinTolerance ?s1)))
       (exists (?sReaming)
          (and (reaming-usage ?sReaming ?occ)
               (follows ?sDrilling2 ?sReaming drilling)))))

(forall (?occ ?sDrilling2)
   (if (and (occurrence_of ?occ drilling)
            (drilling2-usage ?sDrilling2 ?occ)
            (holdsA withinTolerance ?s1))
       (exists (?sDeburring)
          (and (deburring-usage ?sDeburring ?occ)
               (follows ?sDrilling2 ?sDeburring drilling)))))

(follows-at-most-one drilling2-usage reaming-usage drilling)

(follows-at-most-one drilling2-usage deburring-usage drilling)

(forall (?occ ?sDrilling2)
   (if (and (occurrence_of ?occ drilling)
            (drilling2-usage ?sDrilling2 ?occ))
       (xor (exists (?sReaming)
               (and (reaming-usage ?sReaming ?occ)
                    (follows ?sDrilling2 ?sReaming drilling)))
            (exists (?sDeburring)
               (and (deburring-usage ?sDeburring ?occ)
                    (follows ?sDrilling2 ?sDeburring
                             drilling))))))
```

Expression 17: Translation of Decision in Control Flow Example

Expression 18 shows an example of translating control flow between steps when there is an intervening merge element. The first subexpression requires inspection to occur if drilling2 or reaming occur. It does not require drilling2 or reaming to occur, other expressions for control flow into those steps do that. This assumes the additional subactivities, usages, and other control flow expressions are defined for drilling2 and reaming. The second subexpression requires that an occurrence of inspection not follow

both drilling2 and reaming. Combined with the first expression, there will be at least two occurrences of inspection if drilling2 and reaming both occur.

The last two subexpressions are needed in the same situations as the second subexpression of Expression 14. After template expansion, and in combination with the existentials of the first two subexpressions, they require exactly one occurrence of inspection to follow exactly one occurrence of drilling2, and the similarly for inspection and reaming. Combined with the third expression, this means there will be exactly two occurrences of inspection if drilling2 and reaming both occur.

```
(forall (?occ ?sDorR ?sReaming)
   (if (and (occurrence_of ?occ drilling)
            (or (drilling2-usage ?sDorR ?occ)
                (reaming-usage ?sDorR ?occ)))
       (exists (?sInspection)
          (and (inspection ?occ ?sInspection)
               (follows ?sDorR ?sInspection drilling)))))

(forall (?occ ?sDrilling2 ?sReaming ?sInspection)
   (if (and (occurrence_of ?occ drilling)
            (drilling2-usage ?sDrilling2 ?occ)
            (reaming-usage ?sReaming ?occ)
            (inspection-usage ?sInspection ?occ))
       (not (and (follows ?sDrilling2 ?sInspection drilling)
                 (follows ?sReaming ?sInspection drilling)))))

(follows-at-most-one drilling2-usage inspection-usage drilling)

(follows-at-most-one reaming-usage inspection-usage drilling)
```

Expression 18: Translation of Merge in Control Flow Example

Expression 19 is an example of translating a control flow between a step and a final node. The first subexpression requires inspection to have no following occurrences due to a control flow. It does not rule out other occurrences being after inspection, just being after inspection due to control flow, which is the purpose of the FOLLOWS relation. For example, drilling may have multiple final nodes, with no specification of which is to happen last. Other constraints can require stabilizing to be the PSL leaf of drilling if needed. This way the translator does not need to traverse more of the process description to generate Expression 19.

The second subexpression is needed if the final step is inside a larger loop or after a merge of parallel flows (there is a decision intervening in the control flow between the last step and the final node). In these cases, the INSPECTION-USAGE relation will link a single complex occurrence to multiple suboccurrences. The second subexpression requires that at most one of these occurrences have no follower. Combined with the existential in first subexpression, exactly one inspection occurrence will have no

follower. All the other occurrences identified by INSPECTION-USAGE will be followed by some other occurrence.

```
(forall (?occ ?sInspection)
   (if (and (occurrence_of ?occ drilling)
            (inspection-usage ?sInspection ?occ))
       (not (exists (?sFollows)
              (follows ?sInspection ?sFollows drilling)))))

(forall (?occ ?sInspection1 ?sInspection2)
   (if (and (occurrence_of ?occ drilling)
            (inspection-usage ?sInspection1 ?occ)
            (inspection-usage ?sInspection2 ?occ)
            (not (exists (?sFollows)
                   (follows ?sInspection1 ?sFollows drilling)))
            (not (exists (?sFollows)
                   (follows ?sInspection2 ?sFollows drilling))))
       (= ?sInspection1 ?sInspection2)))
```

Expression 19: Translation of Final Control Flow Example

A translation based on the patterns described above can be applied to the incoming flow of each step separately (there is exactly one after preprocessing). Usage relations identifying the suboccurrences to be constrained are independently applicable in each constraint (compare to using logical variables, which requires combining the smaller expressions to share the variables). The translation is applicable to a process description that has only control flow between steps, and at most one other sequencing element between each two steps, such as fork or decision.

If there are multiple sequencing elements chained together between steps, the translation for each step potentially will be multiple expressions filling the templates shown in Expression 20, plus other expressions as described in the algorithm below. It traces back from the step being translated (the *target* step) along the incoming control flow, until reaching either an initial node or another step (the *source* step). The result will be a disjunction of conjunctions (disjunctive normal form) of usage relations of the sources. The result also includes the conditions on any decision elements. Each conjunction is inserted into a separate copy of the template in Expression 20, in the left side of the implication. The FOLLOWS subexpressions are generated for each template from the usage relations in the conjunction.

```
(forall (?occ < ?sSource<n> variables > )
   (if (and (occurrence_of ?occ <complex activity>)
            <result from recursion below>)
       (exists (?sTarget)
          (and (<target step usage relation> ?sTarget ?occ)
               < follows subexpressions for source usage
                 relations in recursion result and ?sTarget >)))
```

Expression 20: Template for Chain of Sequencing Elements

The algorithm below assumes that:

- flows between sequence elements do not loop between any two steps.
- decisions have conditions on all outgoing control flows.
- only merges intervene in a control flow between an initial node and a step.
- there is no intervening sequence element in a control flow between a step and a final node.

The algorithm also assumes that merges and joins do not appear on the same path between any two steps in combinations that cause the same occurrence in the target usage to follow multiple occurrences in a source usage, or multiple occurrences in the target to follow the same occurrence in a source. The FOLLOWS-AT-MOST-ONE expressions contradict these situations. Well-nested decisions and merges, and joins and forks, satisfy this assumption, for example in typical structured programming languages.

The algorithm is:

1. Before traversing backward from the target step, if the target step has a control flow to a final node, then generate a separate expression per Expression 19.

2. When traversing backward from the target step, on reaching an initial node, generate a separate expression for the target step per Expression 13. This is one of the base cases of the recursion.

3. On reaching another step, generate a separate expression with the FOLLOWS-AT-MOST-ONE template, using the usage relation of this step, the usage relation of the target step, and the name of the complex activity being translated. Return a usage relation subexpression for the step with variables `?sSource<n>` and `?occ`, where the source variable name is constructed by replacing "<n>" with the name of the usage relation of the source step. This is one of the base cases of the recursion.

4. On reaching a sequencing element, proceed as follows, depending on the kind of element:

    a. For forks, return the result of recurring on the incoming edge.

    b. For joins, return the conjunction of the results from recurring on incoming edges.

    c. For decisions, return a conjunction of the decision condition and the result of recurring on the incoming edge.

    d. For merges, return the disjunction of the results of recurring on the incoming edges.

5. When the recursion is complete, transform the result into a disjunction of conjunctions (disjunctive normal form) by repeatedly distributing conjunctions over disjunctions. This replaces subexpressions of the form `(and p1 (or p2 p3))` with `(or (and p1 p2) (and p1 p3))`.

6. For each conjunction in the disjunction in step 5, generate an expression by filling in the template in Expression 20, inserting the subexpressions into the conjunction on the left side of the implication. Insert source variables in the universal quantification that are used in the inserted subexpressions.

7. For each expression resulting from step 6, and for each usage relation subexpression on the left side of the implication, insert a FOLLOWS statement into the right side as shown in Expression 20, using the source variable name corresponding to the usage relation, the `?sTarget` variable, and the name of the complex activity being translated.

8. For each expression resulting from step 7, insert the usage relation for the target step into the right side of the implication, as shown in Expression 20.

9. If there are multiple expressions resulting from step 8, for each pair (unordered) generate an expression by filling in the template in Expression 21. Insert the usage relation subexpression for the target step into the left side of the implication. Insert all the source usage relation subexpressions in the pair of expressions into the left side also. Insert source variables in the universal quantification that are used in the inserted subexpressions. For each usage relation subexpression except for the target, insert a FOLLOWS subexpression on the right side as shown in Expression 21, with the source variable from the usage relation subexpression, the `?sTarget` variable, and the complex activity being translated.

10. If any of the steps involved above are for the same subactivity, generate a separate expression for each (unordered) pair of usage relations corresponding to the steps requiring the relations to be disjoint. This prevents them from linking the same complex occurrence to the same suboccurrence.

```
(forall (?occ < ?sSource<n> variables > ?sTarget)
   (if (and (occurrence_of ?occ <complex activity>)
            (<target step usage relation> ?sTarget ?occ)
            <usage relation expressions>)
       (not (and <follows expressions>))))
```

Expression 21: Template for FOLLOWS for Merges in Sequence Element Chain

Inference using the translations above requires closure expressions to prevent deduction of unwanted occurrences. For example, Expression 14 does not prevent tapping from from happening before stabilizing. The first subexpression only requires tapping to occur when stabilizing does, and the second only requires the occurrences to be matched one-

to-one when tapping follows stabilizing. They allow tapping to occur even if stabilizing does not. In general, the translations only infer the existence of desired occurrences, not the lack of undesired ones. Expression 22 shows closure expressions for Expression 14. The first subexpression requires every occurrence of tapping to follow an occurrence of stabilizing. The second requires tapping to follow only stabilizing, not other steps in the process. Combined with Expression 14, this requires one occurrence of tapping for each occurrence of stabilizing and no other occurrences of tapping are allowed.

```
(forall (?occ ?sTapping)
   (if (and (occurrence_of ?occ drilling)
            (tapping-usage ?sTapping ?occ))
       (exists (?sStabilizing)
          (and (stabilizing-usage ?sStabilizing ?occ)
               (follows ?sStabilizing ?sTapping drilling)))))

(forall (?occ ?sTapping ?s)
   (if (and (occurrence_of ?occ drilling)
            (tapping-usage ?sTapping ?occ)
            (follows ?s ?sTapping drilling))
        (stabilizing-usage ?s ?occ)))
```

Expression 22: Closure for Expression 14

6.3 Reacting to Changes

Many process languages have elements for reacting to changes in circumstances (sometimes called "events" or "triggers").[32] These elements do not restrict which other processes bring about the changes, which may even be the process detecting the change. Change detection is useful when a process should be specified independently of its environment, but still react to changes in it. This section gives translations of change detection patterns to PSL. It is based on the PSL STATE relation, which identifies entities representing conditions of the "world" in which a process operates. Specific states are true immediately before and after each occurrence in the occurrence tree, as defined with the PSL PRIOR and HOLDS relations (Expression 6 in Section 5 defines the extensions PRIORA and HOLDSA for complex occurrences). This section only addresses change detection in the context of sequencing constraints (Section 6.2). See Section 7.3 for change detection with participation constraints.

Expression 23 shows an example translation for starting a process whenever a certain kind of change happens. In this example a drilling process starts whenever a piece of metal becomes available. The first subexpression defines the kind of state to be detected. It defines a class of states instead of specific one, so the translation can be used when any piece of metal becomes available, rather than a particular one. The second subexpression defines a relation between occurrences and groves of drilling that start after the

[32] In software terminology, "event" usually means the receipt of notification that a change has occurred. The formalization in this paper represents changes separately from receiving notifications of them.

occurrence. The third and fourth subexpressions in Expression 23 require the BEFORE-DRILLING relation to link at most one occurrence making a piece of metal available to at most one grove of drilling on the same branch of the occurrence tree (there may be multiple groves of drilling in the activity tree as a whole). This will ensure each piece of metal is drilled at most once, assuming each METAL-AVAILABLE state is true for exactly one piece of metal.

The last subexpression requires that BEFORE-DRILLING link at least one drilling grove to each occurrence that makes a piece of metal available, and that the grove begins after the metal becomes available. It uses the PSL ACHIEVED relation to identify changes brought about by occurrences, which requires the state to be false immediately before an occurrence and true immediately after. Combined with the previous two subexpressions, BEFORE-DRILLING will link each occurrence of making a piece of metal available to exactly one grove of drilling. Expression 23 does not require the drilling grove to only occur when metal is available. This would be a constraint similar to the last subexpression, with the two sides of the implication switched to require every grove of drilling to be linked by BEFORE-DRILLING to a occurrence making a piece of metal available.

```
(forall (?f)
    (if (metal-available ?f)
        (state ?f)))

(forall (?s ?groveRoot)
   (if (before-drilling ?s ?groveRoot)
       (and (legal ?s)
            (grove ?groveRoot drilling))))

(forall (?s1 ?s2 ?groveRoot)
   (if (and (before-drilling ?s1 ?groveRoot)
            (before-drilling ?s2 ?groveRoot))
       (= ?s1 ?s2)))

(forall (?s ?groveRoot1 ?groveRoot2)
   (if (and (before-drilling ?s ?groveRoot1)
            (before-drilling ?s ?groveRoot2))
       (not (or (earlierA ?groveRoot1 ?groveRoot2))
                (earlierA ?groveRoot2 ?groveRoot1))))

(forall (?s)
   (if (and (legal ?s)
            (achieved ?f ?s)
            (metal-available ?f))
       (exists (?groveRoot)
          (and (before-drilling ?s ?groveRoot)
               (earlier ?s ?groveRoot)))))
```

Expression 23: Starting a Process after a Change

Expression 23 can be tightened to require drilling to start within a certain amount of time after metal becomes available. This is an additional clause in the existential of the last subexpression above, constraining the ending and beginning timepoints of `?s` and `?groveRoot`, respectively. An additional expression can be defined to require drilling begin on the pieces of metal as the become available, rather than drilling a later piece of metal first. This is a constraint on BEFORE-DRILLING that requires later occurrences of metal becoming available to be linked to later occurrences of drilling.

Processes might not always react when a change happens. For example, a process may reach a point where it waits for a change to happen, and continues when it does, but does not react to the change before the process begins waiting or after it continues. Expression 24 adapts the previous translation example to react to a piece of metal becoming available only after preparation is done and before drilling starts. Any other metal becoming available is ignored. It uses the state defined in the first subexpression of Expression 23. The first two subexpressions of Expression 24 define a relation to identify the first achievement of metal becoming available in a factory process after preparing is done.[33] The last subexpression is a translation of sequencing, as in Section 6.2, assuming the usage relation is defined for preparing. It ensures that drilling occurs if a piece of metal becomes available after preparing is done. It will infer only one occurrence of drilling, for the first time metal becomes available.[34]

```
(forall (?s ?occ ?sPreparing)
   (iff (provide-metal-all ?s ?sPreparing ?occ)
        (and (occurrence_of ?occ FactoryProcess)
             (preparation-usage ?sPreparing ?occ)
             (legal ?s)
             (exists (?f)
                (and (achieved ?f ?s)
                     (metal-available ?f)))
             (earlierA ?sPreparing ?s))))

(forall (?s ?occ ?sFirst)
   (iff (provide-metal-first ?sFirst ?sPreparing ?occ)
        (and (provide-metal-all ?sFirst ?occ)
             (not (exists (?s)
                     (and (provide-metal-all ?s ?occ ?sPreparing)
                          (earlier ?s ?sFirst)))))))
```

[33] See footnote 45 on page 35.

[34] Negated existentials are only inferable with closure expressions that rule out the existence of unknown occurrences. In the example above, the process designer would need to enumerate all the possible ways metal can become available, so it one of them can be proved to be the earliest after preparing.

```
(forall (?occFactory ?sPreparing ?s ?f)
   (if (and (occurrence_of ?occ FactoryProcess)
            (preparation-usage ?sPreparing ?occ)
            (provide-metal-first ?s ?occ ?sPreparing))
       (exists (?sDrilling)
          (and (drilling-usage ?sDrilling ?occ)
               (earlierA ?s ?sDrilling)
               (follows ?sPreparing ?sDrilling
                        FactoryProcess)))))
```

Expression 24: Waiting for a Change

Additional constraints can be added to Expression 24 requiring drilling to start within a certain amount of time after metal becomes available. This is an additional clause in the existential of the last subexpression above, constraining the ending and beginning timepoints of `?s` and `?sdrilling`, respectively.

Combining the above examples would be a process that waits for a change to continue, as in Expression 24, but waits at the very beginning, before any other steps in the process have occurred, as in Expression 23. Workflow engines support this capability by distinguishing processes that are "open" from those that are "running" [27][28]. This requires extending PSL, because a complex occurrence does not begin until its first suboccurrence does (the root occurrence).[35] It cannot be "waiting" for a change before the first suboccurrence.[36] A PSL extension could introduce activities that open complex occurrences before the root of those occurrences starts, and support other aspects of representing the process of "executing" a process, such as states of a process.

7 Loosely Coupled Communication

Loose coupling refers to processes that place fewer restrictions on which other processes give entities to them or receive entities from them. The loosest kind of coupling does not even specify a delegate to choose the other processes (see category 1.c.ii in Sections 1 and 2). The rest of this paper refers to this as *input* and *output*.[37] A process using only inputs and outputs will accept and provide entities regardless of which other processes it communicates with,[38] what those processes do with the entities, and exactly how the entities are given and received (see discussion of pre/postconditions in Section 7.1). A process that identifies specific other processes from which to accept entities and to which to provide entities is using function invocation (1.a). A process that identifies a delegate

[35] See footnote 23 on page 10.
[36] The occurrence bringing about the change cannot be a root, because it might never happen. It also cannot be the leaf of a "waiting" suboccurrence for the same reason. See footnote 45 on page 35.
[37] Common flow modeling and process languages, whether in graphical form like the UML [1], or textual form as in programming languages, usually define inputs and outputs colloquially as entities "passed in" or "passed out" of a process. The intuition is that the process has a "boundary" across which entities "flow" (sometimes called "data flow" or "object flow"). This definition is not sufficient to distinguish inputs and outputs from the other categories in Section 1.
[38] Except see footnote 14 on page 4.

to make that determination is using messages (1.b/c.i, and Section 8), but also see Section 9 on loosely coupled messaging.

Another important aspect of input and output is they are dependent on the view taken of the process. For example, a milling process might involve a piece of metal, oil, electricity, heat, metal shavings, and some instructions on how the metal is to be milled. Which are inputs and which are outputs, and which are neither? Consider these alternative views:

- From an operator's view, 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, while shavings and heat are just bi-products.

- From an infrastructural view, the oil and electricity are inputs, and oil is an output, since it becomes dirty and needs to be cleaned or recycled.

- Another view might assume metal shavings are output because they must be removed from the machine periodically, or are useful as scrap.

- Perhaps special arrangements are made to absorb sound, so it is an output, but heat and vibration are not, since nothing is done with these.

- The machine might be 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.

This simple example of view-dependent input and output presents a challenge for both conventional process languages models and PSL. Conventional languages, including textual programming languages, provide for only one set of inputs and outputs. And within that single view, they provide no guidance on what to choose as an input or output. They enforce the temporal ordering constraints that require inputs of one process to be filled by outputs of another occurring earlier, but these must be specified under a single larger process containing the communicating processes. PSL supports multiple views of a single process, because any element of a process can be contained under multiple other processes, each of which can establish its own constraints on the elements. However, PSL currently lacks support for representing input and output, in particular, distinguishing inputs from outputs from other entities participating in a process, as discussed in Section 7.1. It also lacks temporal ordering constraints between inputs and outputs.

This section defines an extension of PSL for inputs and outputs. Section 7.1 shows inputs and outputs to be early design stage notions that are not equivalent to existing PSL concepts, preconditions and postconditions in particular. Section 7.2 defines activities for input and output that include support for multiple views, along with constraints on their

usage. Section 7.3 gives translations for typical input and output elements in common process languages. Section 6.3 gives translations for reacting to changes.

7.1 Inputs and Outputs in Current PSL Relations

Before extending PSL for inputs and outputs, it is important to show they are require an extension to define accurately. Here are some ways one might attempt in current PSL to distinguish which objects involved in an activity occurrence are inputs, which are outputs, and which are neither:

- Perhaps an input is any object 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 object is not always an input or an output. For example, processes for repairing and painting beams in a factory might happen to choose to operate on the same beam at different times, but not because one process outputs beams to the other. The beam is a participant in both, but would not be considered an output of one process to the other, even if this accidentally happened all the time.

- Another approach is to use preconditions and postconditions, which PSL supports on process occurrences. For example, the precondition on the piece of metal input to a milling machine might be it is in a certain location where the operator or machine notices it. However, sometimes inputs are received by a process after it starts, and provided by a process before it finishes. For example, a milling process might receive oil after it starts, and produce shavings before it finishes. This means conditions for input and output do not necessarily apply at the start and finish of a process, as required by preconditions and postconditions.

  Even for inputs received at the beginning of a process and outputs provided at the end, some preconditions and postconditions are not about inputs or outputs. For example, perhaps a milling machine must be inspected before the milling process starts, but this does not imply the inspector is an input to milling. Finally, the preconditions and postconditions for input and output are too specific in most cases. For example, the specification of a milling process with a piece of metal as input probably does not include how or where to detect the presence of the metal. The exact location for input might be different for various kinds of milling processes or may even change for the same one to other factors.

- 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 object 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 object 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 represents a PSL activity, as the milling term does in Expression 25. However, parameterized terms do not differentiate inputs from outputs, or objects that are neither.[39]

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

Expression 25: Parameterized Term for Activities

These examples suggest the notions of input and output are not equivalent to existing PSL concepts, such as participation, preconditions, postconditions, or resources. They also suggest it is not possible to define generally sufficient conditions to identify an object as an input and output. The PSL extension in this paper defines only necessary conditions on input and object objects, leaving sufficient conditions to be defined by the designer of each process (see discussion at the end of Section 4).

7.2 Axioms for Input and Output

This section extends PSL for inputs and outputs rather than apply usage patterns, for the reasons given in Section 4. In particular, extensions enable the process designer to define inputs and outputs of separate processes independently of each other. For example, a drilling process providing a piece of metal to a milling process is only concerned with providing the output, and the milling process only with taking the input. This is more modular than writing one constraint for both as would be required when applying usage patterns.[40]

The extension consists of activities for accepting inputs and posting outputs (giving and receiving entities), plus some functions for identifying entities that are the subject of these activities. Occurrences of accepting and posting activities enable inputs and outputs to be received and given at any time during a process. They are also the basis for defining the relations that identify entities given and received at particular times, for example, at the start and finish of a process.[41]

---

[39] Parameterized functions support multiple views by mapping different functions to the same activity.

[40] See Section 5 of [8] for example usage patterns for input and output.

[41] An earlier paper relied mainly on defining new PSL relations for input and output [6], which has the advantage of not requiring anything to actually occur when inputting or outputting an object. However, new relations cannot represent exactly when inputs and outputs are communicated during a process without specifying states as well, which require activities to bring them about. The predefined activities in this

Expression 26 defines activity functions for accepting and posting entities to and from a process. These are relations between an activity, an object that is accepted or posted, and a grove to or from which the object is accepted or posted (see Expression 4 for description of groves). They are defined as functions because they identify exactly one activity for each pair of object and grove.[42] The grove argument is needed because groves are not necessarily strongly nested, as in typical process languages (see footnote 27 and Expression 42). The grove argument identifies the "boundary" between the process and other processes giving or receiving the inputs and outputs.[43]

Expression 26 allows the same object to be accepted or posted multiple times under the same superoccurrence, to support multiple inputs or outputs of the same type of thing, see Expression 45. It does not prevent the same activity from accepting or posting multiple objects for multiple superoccurrences, or multiple objects for the same superoccurrence.[44] An accept or post activity can also have user-defined capabilities, for example a post activity for a piece of metal can also move it. Accept and post activities are not required to be atomic in the extension, but the process designer can constrain theirs to be atomic as needed.

```
(forall (?a ?x ?sGroveRoot ?aGroveAct)
   (if (or (= ?a (accept-input ?x ?sGroveRoot ?aGroveAct))
           (= ?a (post-output ?x ?sGroveRoot ?aGroveAct)))
           (and (activity ?a)
                (activity-participant ?x ?a)
                (grove ?sGroveRoot ?aGroveAct)
                (subactivity ?a ?aGroveAct))))

(forall (?a ?x ?sGroveRoot ?aGroveAct ?s)
   (if (and (or (= ?a (accept-input ?x ?sGroveRoot ?aGroveAct))
                (= ?a (post-output ?x ?sGroveRoot ?aGroveAct)))
            (occurrence-ofA ?s ?a))
       (subocc-in-grove ?s ?sGroveRoot ?aGroveAct)))
```

Expression 26: Accepting Inputs and Posting Outputs

Expression 27 defines relations that facilitate application of the activity functions above. They relate inputs and outputs to occurrences in which they are accepted or posted. A particular accept or post will usually not be in all the occurrences of the grove to which

---

paper abstract from the state-oriented approach of [6] for more expressiveness and simplicity, while preserving the transition to states (see discussion of preconditions and postconditions in Section 7.1, and Expression 47 and Expression 48). The suggestion to use activities appeared in [8].

[42] A function is a relation with one of the arguments identified as always the same for a particular combination of the other arguments.

[43] This replaces the "relative" input and output relations in [6].

[44] The same function can be applied to different arguments and still map to the same individual. The "combined" activities defined this way cannot be uncombined. For example, if an activity accepting or posting multiple objects occurs multiple times, all the objects are accepted or posted multiple times, once at each occurrence.

they refer. Even for a process that always accepts the same input or posts the same outout, the occurrences of the accept and post activities will usually not be the same. They are spread across branches of the activity trees due to occurrences external to the process.

```
(forall (?x ?occ ?sAccept)
   (iff (accept-input-in-occ ?x ?occ ?sAccept)
        (exists (?sGroveRoot ?aGroveAct)
           (and (occ-grove-root ?sGroveRoot ?occ)
                (occurrence_of ?occ ?aGroveAct)
                (occurrence-ofA ?sAccept
                   (accept-input ?x ?sGroveRoot ?sGroveAct))
                (subactivity_occurrence-neq ?sAccept ?occ)))))

(forall (?x ?occ ?sPost)
   (iff (post-output-in-occ ?x ?occ ?sPost)
        (exists (?sGroveRoot ?aGroveAct)
           (and (occ-grove-root ?sGroveRoot ?occ)
                (occurrence_of ?occ ?aGroveAct)
                (occurrence-ofA ?sPost
                   (post-output ?x ?sGroveRoot ?sGroveAct))
                (subactivity_occurrence-neq ?sPost ?occ)))))
```

Expression 27: Inputs and Outputs in Occurrences

Inputs can be taken from changes brought about by other processes, without requiring them to be provided as outputs. For example, a process for driving will take traffic light changes as input, but might be specified without requiring the process for changing traffic lights to provide colors as output. This enables the specification of driving to be independent of the specification for traffic light changes. Changes in PSL are represented using states, which represent any condition of the world (see Section 6.3). PSL does not refine state any further, in particular, it does not give a relation to link a state to the objects involved in the conditions represented by a state.

Expression 28 defines a relation linking states to the objects they are about. For example, the state of a traffic light being green will be about a particular traffic light and the color green. The first subexpression defines the types of things being related, states and objects. The second requires occurrences changing states to have the objects the states are about as participants. It uses the ACHIEVEDA and FALSIFIEDA relations, which link occurrences to states that become true and false during the occurrences, respectively.

```
(forall (?x ?f)
   (if (about ?x ?f)
       (and (state ?f)
            (object ?x))))

(forall (?x ?f ?s)
   (if (and (about ?x ?f)
            (or (achievedA ?f ?s)
                (falsifiedA ?f ?s)))
       (participant ?x ?s)))
```

Expression 28: The ABOUT Relation

Expression 29, Expression 30, and Expression 31 constrain when accepting and posting activities can occur. Expression 29 is the most general constraint on occurrences of accepting and posting activities. It requires every legal occurrence of accepting to be after a posting of the same object, or after achievement of a state about that object.[45] This follows the common intuition that receipt of an object must happen after it is becomes available. It uses the EARLIERA relation, which only requires the occurrences be somewhere on the occurrence tree, rather than within an activity tree. It does not require posted objects to be accepted.

```
(forall (?sAcceptInput ?x ?sGroveRoot1 ?aGroveAct1)
   (if (and (occurrence-ofA ?sAcceptInput
              (accept-input ?x ?sGroveRoot1 ?aGroveAct1))
            (legalA ?sAcceptInput))
       (or (exists (?sPostOutput ?sGroveRoot2 ?aGroveAct2)
             (and (occurrence-ofA ?sPostOutput
                     (post-output ?x ?sGroveRoot2 ?aGroveAct2))
              (earlierA ?sPostOutput ?sAcceptInput)))
           (exists (?sAchieves ?f)
             (and (achieved ?f ?sAchieves)
                  (about ?f ?x)
                  (earlierA ?sAchieves ?sAcceptInput))))))
```

Expression 29: Ordering Constraint on Accept and Post

[45] Many process languages represent change detection as a step in a process ("listening"). This has the advantage of representing the changed object as output of a step, with the same semantics as output from any other step. In particular, the PSL axioms for input and output are simpler. It also enables representation of change detection in a similar way to messaging (Section 8.1), as the final stage of a process that begins with the occurrence bringing the change about, and ends with the change being noticed by the listener. Representing change detection as a step has the disadvantage that it might be unintuitive to consider "listening" as something which occurs. For example, a listening occurrence would have no suboccurrences (other than itself) if the change to be detected never happens. In particular, listening activities require an extension to PSL for representing processes that are started but not doing anything, one of the problems listening activities are intended to avoid, see discussion of workflow engines in Section 6.3. Listening activities also require an extension for aborting processes, needed to prevent them from waiting forever for a change that never occurs.

Expression 30 and Expression 31 are stronger constraints following the common style of process languages where all "flows" of inputs and outputs are either within a "boundary" of a process, through inputs and outputs of the process itself, or from detecting the achievement of a state about the input object sometime during the process. In Expression 30 the accept (`?sAcceptInput`) is for suboccurrences in complex occurrences (`?occSuper`) under which flow happens. The premise of the implication identifies the complex occurrences in the accepting grove in which the accept actually occurs, using ACCEPT-INPUT-IN-OCC. The accepted object must come from either a preceding post for a suboccurrence in the same complex occurrence (first disjunct in consequent), from a preceding accept for the same complex occurrence (second disjunct), or from an earlier achievement of a state about that object during the same complex occurrence (third disjunct).[46] There is no similar constraint on posting, because outputs are not required to be input to other processes.

```
(forall (?x ?occWithAccept ?sAcceptInput)
   (if (accept-input-in-occ ?x ?occWithAccept ?sAcceptInput)
       (exists (?occSuper)
          (and (subactivity-occurrence-neq  ?occWithAccept
                                            ?occSuper)
               (or (exists (?occEarlier ?sPostOutput)
                      (and (post-output-in-occ ?x ?occEarlier
                                               ?sPostOutput)
                           (subactivity-occurrence-neq
                              ?occEarlier ?occSuper)
                           (occurrence_of ?occSuper ?aSuperAct)
                           (min-precedesA ?sPostOutput
                                          ?sAcceptInput
                                          ?aSuperAct)))
                   (exists (?aSuperAct ?sAcceptForSuper)
                      (and (accept-input-in-occ ?x ?occSuper
                                                ?sAcceptForSuper)
                           (occurrence_of ?occSuper ?aSuperAct)
                           (min-precedesA ?sAcceptForSuper
                                          ?sAcceptInput
                                          ?aSuperAct)))
                   (exists (?f ?sEarlier)
                      (and (achieved ?f ?sEarlier)
                           (about ?f ?x)
                           (or (= (root_occ ?occSuper) ?sEarlier)
                               (earlier (root_occ ?occSuper)
                                        ?sEarlier))
                           (earlierA ?sEarlier
                                     ?sAcceptInput)))))))))
```

Expression 30: Ordering Constraint on Accept by Suboccurrence

[46] All common process models require occurrences of complex processes to be "strongly nested" (see footnote 27 and Expression 42), though PSL and the axioms and translations in this paper do not. In particular, Expression 30 and Expression 31 could be tightened to require accepting an input only from a direct superoccurrence, and posting an output only to a direct superoccurrence (see Expressions 39 and 40 in [6] for direct suboccurrence axioms).

Expression 31 applies to the posting of outputs to a complex occurrence. The posted object must come from either a preceding post under a suboccurrence of the same complex occurrence (first disjunct), from an accept for the same complex occurrence (second disjunct), or from an earlier achievement of a state about that object during the same complex occurrence (third disjunct). It does not require posted objects to be accepted. Accepting objects in complex occurrences is constrained by Expression 30.

```
(forall (?x ?occSuper ?sPostOutputForSuper)
   (if (post-output-in-occ ?x ?occSuper ?sPostOutputForSuper)
       (or (exists (?occEarlier ?sPostOutputEarlier)
              (and (post-output-in-occ ?x ?occEarlier
                                       ?sPostOutputEarlier)
                   (subactivity-occurrence-neq ?occEarlier
                                               ?occSuper)
                   (occurrence_of ?occSuper ?aSuperAct)
                   (min-precedesA ?sPostOutputEarlier
                                  ?sPostOutputForSuper
                                  ?aSuperAct)))
           (exists (?sAcceptForSuper)
              (and (accept-input-in-occ ?x ?occSuper
                                        ?sAcceptForSuper)
                   (occurrence_of ?occSuper ?aSuperAct)
                   (min-precedesA ?sAcceptInputForSuper
                                  ?sPostOutputForSuper
                                  ?aSuperAct)))
           (exists (?f ?occEarlier)
              (and (achieved ?f ?occEarlier)
                   (about ?f ?x)
                   (or (= (root_occ ?occSuper) ?occEarlier)
                       (earlier (root_occ ?occSuper)
                                ?occEarlier))
                   (earlierA ?occEarlier
                             ?sPostOutputForSuper))))))
```

Expression 31: Ordering Constraint on Post by Suboccurrence

Applications involving physical objects might want to require that inputs taken by a subprocess are output again before being input to a later subprocess, and before being output from a containing process. For example, a piece of metal accepted into a drilling process might be required to be posted out again before being accepted by a later milling process, or before being posted out of an overall factory process. Expression 30 and Expression 31 can be tightened to require this.

Expression 32 and Expression 33 define kinds of input and output based on whether they are accepted and posted only at the beginning and end of a process, or possibly anytime during it (see category 2 in Sections 1 and 2). The relations OCCURRENCE-INPUT-BEGIN and OCCURRENCE-OUTPUT-END require occurrences of ACCEPT-INPUT and POST-OUTPUT at the beginning and end (root and leaf) respectively, whereas the relations OCCURRENCE-

INPUT and OCCURRENCE-OUTPUT do not have that restriction.[47] The "begin" and "end" relations imply the others, but not the inverse.[48] The "begin" and "end" relations refer to specific objects, rather than object identified by relations to the superoccurrence (see Expression 45). In some applications, the same object is identified by multiple relations to the superoccurrence, and is input or output only at the beginning or end for some of those relations, and input or output anytime for others. In this case, variations of Expression 33 can be defined that use a one-direction implication instead of the inner bidirectional one. This would require the object to be accepted or posted at the beginning or end while still allowing it to be accepted or posted during the superoccurence. The process designer applies these to objects identified by relations to the superoccurrence.

```
(forall (?x ?occ ?sAccept)
   (iff (occurrence-input-begin ?x ?occ)
        (iff (accept-input-in-occ ?x ?occ ?sAccept)
             (root-occA ?sAccept ?occ))))

(forall (?x ?occ ?sPost ?sPostLeaf)
   (iff (occurrence-output-end ?x ?occ)
        (iff (post-output-in-occ ?x ?occ ?sPost)
             (leaf-occA ?sPost ?occ))))
```

Expression 32: Inputs and Outputs at Beginning and End of Complex Occurrence

```
(forall (?x ?occ)
   (iff (occurrence-input ?x ?occ)
        (exists (?sAccept)
           (accept-input-in-occ ?x ?occ ?sAccept))))

(forall (?x ?occ)
   (iff (occurrence-output ?x ?occ)
        (exists (?sPost)
           (post-output-in-occ ?x ?occ ?sPost))))
```

Expression 33: Inputs and Outputs in Complex Occurrence

7.3 Translations of Input and Output

This section gives example translations of input and output "flow" to PSL ("data flow"),[49] using the predefined activities and new relations defined in Section 7.2 for inputs and outputs. The translation follows the definition of input and output in the introduction to Section 7, in particular, interpreting data flow as constraints on

[47] These relations apply to occurrences, but similar ones can apply to groves, constraining all the occurrences in a grove, or to activities, constraining all occurrences of an activity.

[48] Many process languages support both kinds of input and output, but they appear as separate language elements. For example, programming language parameters would be formalized with OCCURRENCE-INPUT-BEGIN and OCCURRENCE-OUTPUT-END, but these same programs might operate in "publish" and "subscribe" systems, which have the semantics of OCCURRENCE-INPUT and OCCURRENCE-OUTPUT.

[49] See footnote 37 on page 29.

interprocess participation that are independent of the processes being coordinated. The translation covers data flow in typical process languages, and employs the usage relations of Section 6.1 to identify the suboccurrences between which entities are flowing. This enables each data flow to be translated separately from others, and separately from sequencing ("control flow," see Section 6.2), making the expressions smaller than if the entire process were translated at once. This is also facilitated by assuming data flow is intended to allow other steps to be inserted later, rather than be a complete description. Separate closure expressions can be defined to indicate the description is complete.

Expression 34 through Expression 39 assume data flow is independent of control flow (as translated in Section 6.2).[50] This simplifies translation of some kinds of processes that use both, and supports process languages that do not assume an implicit control flow with every data flow [12][29]. If a language has implied control flow with data flow [1][9], then control flow can be added to the process description for every data flow before translation. This approach is not suitable for processes where data flow starts a subprocess without the completion of earlier subprocesses (category 1.c.ii/2.a in Sections 1 and 2), for example, if a milling process starts when metal is input, even if the subprocess providing metal is not complete. In these applications, data flow must be folded into expressions similar to control flow, to require the existence of the subprocess accepting input.[51]

Expression 34 gives the translation of a data flow from a factory process to a drilling process occurring under it. It requires drilling to accept input only accepted by the factory, and that the factory process must accept the input before drilling accepts it. This ensures conformance to the second disjunct in Expression 30, one of the alternative requirements for legal occurrences of ACCEPT-INPUT. Expression 34 does not require that the factory process to accept input before drilling starts. Expression 34 makes the simplifying assumption that everything input to factory is input to drilling. See Expression 44 and Expression 45 for how to refine this for more realistic processes, by identifying inputs by their type or relation to the superoccurrence, enabling different constraints to apply to each. It also assumes additional activities, subactivities, and usages are defined for the factory process and drilling.

```
(forall (?occ ?sDrill ?x ?sAcceptD)
   (if (and (occurrence_of ?occ factoryProcess)
            (drilling-usage ?sDrill ?occ)
            (accept-input-in-occ ?x ?sAcceptD ?sDrill))
       (exists (?sAcceptFP)
          (and (accept-input-in-occ ?x ?occ ?sAcceptFP)
               (min-precedesA ?sAcceptFP ?sAcceptD
                              factoryProcess)))))
```

Expression 34: Translation of Data Flow from Superoccurence to Suboccurrence

[50] The expressions do not require the existence of the output, they only require input is taken when output is provided.

[51] The translations distinguish intentional participant sharing from accidental sharing (see Section 7.1) by having the complex occurrence in the left side of the implication (necessary conditions), rather than the right side (sufficient conditions). This satisfies the motivation of the first bullet in Section 7.1.

Expression 35 gives the translation of data flow from an external change during a factory process to a drilling process occurring under it. The first subexpression defines the state for an operator arriving, which is the change brought about by an unidentified process. The second two subexpressions of Expression 35 define a relation to identify the first achievement of the state in a factory process. The last subexpression requires drilling to accept the same operator the change is about. It also requires the change to happen before drilling accepts the operator. This ensures conformance to the third disjunct in Expression 30. Expression 35 will infer only one accept, for the first time an operator arrives during the factory process.[52] It does not require that the change occur before drilling starts.

```
(forall (?f ?x)
  (if (operator-arrived ?f ?x)
      (and (state ?f)
           (operator ?x)
           (about ?x ?f))))

(forall (?s ?occ ?operator ?sLeaf)
   (if (and (occurrence_of ?occ FactoryProcess)
            (legal ?s)
            (exists (?f)
               (and (achieved ?f ?s)
                    (operator-arrived ?f ?operator)))
            (earlier (root_occ ?occ) ?s)
            (leaf_occ ?sLeaf ?occ)
            (earlier ?s ?sLeaf))
       (provide-operator-all ?s ?occ ?operator)))

(forall (?sFirst ?occ ?operator)
   (if (and (provide-operator-all ?sFirst ?occ ?operator)
            (not (exists (?s)
                    (and (provide-operator-all ?s ?occ ?operator)
                         (earlier ?s ?sFirst)))))
       (provide-operator-first ?sFirst ?occ ?operator)))

(forall (?occ ?sDrill)
   (if (and (occurrence_of ?occ factoryProcess)
            (drilling-usage ?sDrill ?occ))
       (exists (?s ?operator ?sAccept)
          (and (provide-operator-first ?s ?occ ?operator)
               (accept-input-in-occ ?operator ?sDrill ?sAccept)
               (earlierA ?s ?sAccept)))))
```

Expression 35: Translation of Data Flow from Change to Suboccurrence Example

Expression 36 gives the translation of a data flow from drilling to milling under a factory process. It requires milling to accept input only from drilling, and that drilling must post

[52] See footnote 34 on page 28.

the output before milling accepts it as input. This ensures conformance to the first disjunct in Expression 30. Expression 36 does not require that drilling finish before milling starts. Expression 36 makes the simplifying assumption that everything input by milling is output from drilling. See Expression 44 and Expression 45 for how to refine this for more realistic processes, by identifying inputs by their type or relation to the superoccurrence. It also assumes additional activities, subactivities, and usages are defined for the factory process, drilling, and milling.

```
(forall (?occ ?sMill ?x ?sAccept)
   (if (and (occurrence_of ?occ factoryProcess)
            (milling-usage ?sMill ?occ)
            (accept-input-in-occ ?x ?sMill ?sAccept))
       (exists (?sDrill ?sPost)
          (and (drilling-usage ?sDrill ?occ)
               (post-output-in-occ ?x ?sDrill ?sPost)
               (min-precedesA ?sPost ?sAccept factoryProcess)))))
```

Expression 36: Translation of Data Flow between Suboccurrences Example

Expression 37 gives the translation of a data flow from milling to a factory process under which it occurs. It requires the factory process to post the outputs that milling does, and to post them after milling does. This ensures conformance to the first disjunct in Expression 31, one of the alternative requirements for legal occurrences of POST-OUTPUT. Expression 37 does not require that milling finish before the factory posts its output. Expression 37 makes the simplifying assumption that everything output from milling is output from the factory. See Expression 44 and Expression 45 for how to refine this for more realistic processes, by identifying inputs by their type or relation to the superoccurrence.

```
(forall (?occ ?sMill ?x ?sPostM)
   (if (and (occurrence_of ?occ factoryProcess)
            (milling-usage ?sMill ?occ)
            (post-output-in-occ ?x ?sMill ?sPostM))
       (exists (?sPostFP)
          (and (post-output-in-occ ?x ?occ ?sPostFP)
               (min-precedesA ?sPostM ?sPostFP
                              factoryProcess)))))
```

Expression 37: Translation of Data Flow from Suboccurrence to Superoccurence Example

Expression 38 gives the translation of data flow from an input to output of the same factory process. It restricts the process to post only objects it also accepts, and that the posts must be after the corresponding accepts. This ensures conformance to the second disjunct in Expression 31. Expression 38 does not require the factory process to finish when it posts an output. It makes the simplifying assumption that everything output from the factory is also input. See Expression 44 and Expression 45 for how to refine this for more realistic processes.

```
(forall (?occ ?x ?sPost)
   (if (and (occurrence_of ?occ factoryProcess)
            (post-output-in-occ ?x ?occ ?sPost))
       (exists (?sAccept)
          (and (accept-input-in-occ ?x ?occ ?sAccept)
               (min-precedesA ?sAccept ?sPost factoryProcess)))))
```

Expression 38: Translation of Data Flow from Superoccurrence to Superoccurence Example

Expression 39 gives the translation of data flow from an external change during a factory process to output of the factory process. The changes is the same as in Expression Expression 35, an operator arriving, brought about by an unidentified process. Expression 39 requires any operator posted by the factory process to be the same one the change is about, and that the change happens before the post. This ensures conformance to the third disjunct in Expression 31. Expression 39 allows at most one posted operator, which must be the first operator to arrive during the factory process (see definition of PROVIDE-OPERATOR-FIRST in Expression 35). It does not require the factory process to finish when it posts an output.

```
(forall (?occ ?output ?sPost)
   (if (and (occurrence_of ?occ factoryProcess)
            (post-output-in-occ ?output ?occ ?sPost)
            (operator ?output))
       (exists (?s)
          (and (provide-operator-first ?s ?occ ?output)
               (earlierA ?s ?sPost)))))
```

Expression 39: Translation of Data Flow from Change to Superoccurence Example

Occurrences in PSL can be contained directly under multiple complex ones, providing a representation for multiple views of the same occurrences, including views of their inputs and outputs.[53] For example, drilling might partially overlap other processes, such as lubricating, supplying power, and so on, with each process specifying its own inputs and outputs for drilling. These views can be specified separately, even by separate authors, and tested for consistency. This makes the construction of the process description incremental and flexible. In a typical flow model, the only way to bring these partially overlapping processes together is in one large specification containing all of them, because the models require occurrences of complex processes to be strongly nested.

Expression 40, Expression 41, and Expression 42 show an example of suboccurrences under more than one superoccurrence, with each superoccurrence specifying different inputs and outputs for each suboccurrence. A factory process is broken into two subprocesses, one shown in Expression 40 for passing of metal from drilling to milling, and another shown in Expression 41 for provision of lubricant to drilling and milling

[53] Compare to "strong nesting," see footnote 27 on page 14.

(these assume the usage relations in shaping and lubricating can apply to occurrences of both drilling and milling). Different inputs and outputs to drilling and milling are specified under each subprocess. Expression 42 combines the two subprocesses, with the constraint that the same occurrences of milling and drilling happen under both. This ensures the input and output constraints of both subprocesses apply to the occurrences of milling and drilling. The two subprocesses do not need to be used together, even though they are in Expression 42. Alternative lubrication processes can be combined with shaping, or vice versa.

```
(forall (?occDrillAndMill)
   (if (occurrence_of ?occDrillAndMill drillingAndMilling)
       (exists (?m ?sAcceptDM ?sDrill ?sAcceptD ?sPostD
                ?sMill ?sAcceptM ?sPostM ?sPostDM)
          (and (accept-input-in-occ ?m ?occDrillAndMill
                                    ?sAcceptDM)
               (drilling-usage ?sDrill ?occDrillAndMill)
               (accept-input-in-occ ?m ?sDrill ?sAcceptD)
               (min-precedesA ?sAcceptDM ?sAcceptD)
               (post-output-in-occ ?m ?sDrill ?sPostD)

               (milling-usage ?sMill ?occDrillAndMill)
               (accept-input-in-occ ?m ?sMill ?sAcceptM)
               (min-precedesA ?sPostD ?sAcceptM)
               (post-output-in-occ ?m ?sMill ?sPostM)

               (post-output-in-occ ?m ?occDrillAndMill ?sPostDM)
               (min-precedesA ?sPostM ?sPostDM
                              drillingAndMilling)))))
```

Expression 40: Drilling and Milling Occurrence Example

```
(forall (?occLubricating)
   (if (occurrence_of ?occLubricating lubricating)
       (exists (?sPumping ?o ?sPostP ?sShaping ?sAcceptS)
          (and (pumping-usage ?sPumping ?occLubricating)
               (post-output-in-occ ?o ?sPumping ?sPostP)
               (shaping-usage ?sShaping ?occLubricating)
               (accept-input-in-occ ?o ?sAcceptS ?sShaping)
               (min-precedesA ?sPostP ?sAcceptS lubricating)))))
```

Expression 41: Lubricating Occurrence Example

```
(forall (?occFactoryProcess)
   (if (occurrence_of ?occFactoryProcess FactoryProcess)
      (exists (?occDrillAndMill ?occDrillLubricating
               ?occMillLubricating ?sDrill ?sMill)
         (and (subactivity-occurrence-of ?occDrillAndMill
                 ?occFactoryProcess drillingAndMilling)
              (subactivity-occurrence-of ?occDrillLubricating
                 ?occFactoryProcess lubricating)
              (subactivity-occurrence-of ?occMillLubricating
                 ?occFactoryProcess lubricating)
              (drilling-usage ?sDrill ?occDrillAndMill)
              (milling-usage ?sMill ?occDrillAndMill)
              (shaping-usage ?sDrill ?occDrillLubricating)
              (shaping-usage ?sMill ?occMillLubricating)))))
```

Expression 42: Suboccurrence with More Than One Superoccurrence Example

Expression 43, Expression 44, and Expression 45 show how to extend the example translations above to specify the kind of input, and to distinguish multiple inputs of the same kind. The corresponding expressions for output are similar, as are the expressions for input and output at the beginning and end of an occurrence, using the relations in Expression 32. Expression 43 is for input of a single object of a certain kind to a drilling activity. The first subexpression requires the input to be metal, the second requires at least one input, and the third limits the input to at most one object. The combination of these subexpressions means drilling takes exactly one input and that it must be a piece of metal. Expression 44 is for multiple inputs of different kinds of things. The first subexpression requires metal and operators to be different kinds of things. The second two require at least one object of each kind to be input. The last two limit the inputs to at most one of each kind. The combination of these subexpressions means drilling takes at least two inputs, that one must be a piece of metal and the other an operator, and none of the other inputs are metal or operators. Another subexpression, not shown, can prevent any other inputs by requiring that all inputs must be either metal or operators.

```
(forall (?occ ?x)
   (if (and (occurrence_of ?occ drilling)
            (occurrence-input ?x ?occ))
       (metal ?x)))

(forall (?occ ?x)
   (if (occurrence_of ?occ drilling)
       (exists (?x)
          (occurrence-input ?x ?occ))))

(forall (?occ ?x1 ?x2)
   (if (and (occurrence_of ?occ drilling)
            (occurrence-input ?x1 ?occ)
            (occurrence-input ?x2 ?occ))
       (= ?x1 ?x2)))
```

Expression 43: Translation of One Input Example

```
(forall (?x)
   (not (and (metal ?x) (operator ?x))))

(forall (?occ ?x)
   (if (occurrence_of ?occ drilling)
       (exists (?x)
          (and (occurrence-input ?x ?occ)
               (metal ?x)))))

(forall (?occ ?x)
   (if (occurrence_of ?occ drilling)
       (exists (?x)
          (and (occurrence-input ?x ?occ)
               (operator ?x)))))

(forall (?occ ?x1 ?x2)
   (if (and (occurrence_of ?occ drilling)
            (occurrence-input ?x1 ?occ)
            (occurrence-input ?x2 ?occ)
            (metal ?x1)
            (metal ?x2))
       (= ?x1 ?x2)))

(forall (?occ ?x1 ?x2)
   (if (and (occurrence_of ?occ drilling)
            (occurrence-input ?x1 ?occ)
            (occurrence-input ?x2 ?occ)
            (operator ?x1)
            (operator ?x2))
       (= ?x1 ?x2)))
```

Expression 44: Translation of Multiple Inputs of Separate Kinds Example

Expression 45 is for multiple inputs where the inputs cannot be distinguished by the kind of thing they are. It defines relations between process occurrences and the participants to identify which input is which (commonly called “parameters” in programming languages). The stamping process has two distinct operators involved, which are linked to occurrences of stamping by the OP1 and OP2 relations respectively. The first subexpression requires these relations to identify operators that are inputs to stamping. The second requires at least one operator for each relation per stamping occurrence, and that they are different ones. The last two limit the relations to one operator each per occurrence. The combination of subexpressions means each stamping occurrence takes at least two inputs, and these must be two different operators. Another subexpression, not shown, can prevent any other inputs by requiring that all inputs must be related to the stamping occurrence by either OP1 or OP2. Relations from occurrence to input and output entities can also be used in Expression 43 and Expression 44. This make the expressions more robust, because additional inputs and outputs will not change existing expressions.

```
(forall (?x ?occ)
   (if (or (op1 ?x ?occ)
           (op2 ?x ?occ))
       (and (operator ?x)
            (occurrence_of ?occ stamping)
            (occurrence-input ?x ?occ))))

(forall (?occ)
   (if (occurrence_of ?occ stamping)
       (exists (?x1 ?x2)
          (and (op1 ?x1 ?occ)
               (op2 ?x2 ?occ)
               (not (= ?x1 ?x2))))))

(forall (?occ ?x1 ?x2)
   (if (and (op1 ?x1 ?occ)
            (op1 ?x2 ?occ))
       (= ?x1 ?x2)))

(forall (?occ ?x1 ?x2)
   (if (and (op2 ?x1 ?occ)
            (op2 ?x2 ?occ))
       (= ?x1 ?x2)))
```

Expression 45: Translation of Multiple Compatible Inputs Example

Processes can have a variable number of inputs or outputs of the same kind and in the same relation to occurrences. These constraints are cumbersome to write in first-order logic, because it does not have constructs for referring to the number of elements in a set, even when the members of the set are those satisfying a first-order constraint.[54] For example, Expression 46 shows a process (AT-LEAST-THREE-INPUT-PROCESS) with at least three distinct inputs of the same kind and in the same relation to each occurrence (OP-AT-LEAST-THREE). There are as many negated equality conjuncts in the second subexpression as there are pairs (unordered) of input objects. Another conjunct, not shown, can place a maximum bound on the number of inputs by requiring all other input objects to be one of the others identified by the existential variables.

[54] See extensions to first order notation in [30] that simplify these expressions.

```
(forall (?occ ?x)
   (if (op-at-least-three ?x ?occ)
       (and (operator ?x)
            (occurrence_of ?occ at-least-three-input-process)
            (occurrence-input ?x ?occ))))

(forall (?occ)
   (if (occurrence_of ?occ at-least-three-input-process)
       (exists (?x ?y ?z)
          (and (op-at-least-three ?x ?occ)
               (op-at-least-three ?y ?occ)
               (op-at-least-three ?z ?occ)
               (not (= ?x ?y))
               (not (= ?x ?z))
               (not (= ?y ?z))))))
```

Expression 46: Translation of Variable Number of Compatible Inputs Example

Processes can have optional inputs or outputs, which might be present or not and the process still functions as intended (the inputs of Expression 43, Expression 44, Expression 45, and Expression 46 are all mandatory).[55] Optional inputs and outputs are not directly expressible in PSL because they are about how a process designer writes constraints on process occurrences, rather than about occurrences themselves.[56] In PSL, a process has optional inputs and outputs if its specification is consistent with occurrences that accept an input or post an output, and with occurrences that do not. However, it may so happen that all legal occurrences of the process accept an input or post an output, due to other factors in the overall system design, making the input or output appear mandatory when it might not be. This means no constraint on legal occurrences can express that some inputs and outputs are optional.[57] As a workaround, a process specification can be tested separately from the rest of the system specification to check if it is consistent with an expression that declares the existence of an occurrence not accepting the input or providing the output. If it is, then the input or output is optional, otherwise it is mandatory.

A more concrete specification of inputs and outputs requires tying them to changes in the world. In the example in Section 7.1, a piece of metal is detected as input to a milling machine by being put in a certain location where the operator or machine notices it. Expression 47 shows an example of this, which relates occurrences to conditions that are present immediately after the occurrence completes, using the PSL STATE predicate and the PRIORA relation, which relates occurrences to conditions that are present immediately before the occurrence completes (see Section 6.3 and Expression Expression 6). Expression 47 requires the piece of metal input to milling to be accepted only when it is

[55] A process specification can allow alternative steps when an input is not available, and allow alternative steps to posting output. In PSL, process constraints can require alternative legal suboccurrences whenever an accept or post is not legal.

[56] They are statements about PSL statements ("metatheoretic") capturing the specifications described in footnote 55. In general these might involve quantification over relations (second-order logic).

[57] Only constraints on legal occurrences are allowed in PSL, not all occurrences.

in a certain location. The states should not be shared across inputs.[58] Expression 47 can be combined with the techniques of Expression 44 and Expression 45 to specify different states for multiple inputs. An example of the corresponding expressions for outputs is shown in Expression 48. It uses the HOLDSA relation, which relates occurrences to conditions that are present immediately after the occurrence completes.

```
(forall (?f ?x)
  (if (= ?f (milling-input-location ?x))
      (and (state ?f)
           (metal ?x))))

(forall (?occ ?x ?sAcceptInput)
   (if (and (occurrence_of ?occ milling)
            (accept-input-in-occ ?x ?occ ?sAcceptInput)
            (metal ?x))
       (priorA (milling-input-location ?x) ?sAcceptInput)))
```

Expression 47: Input State Example

```
(forall (?f ?x)
  (if (= ?f (milling-output-location ?x))
      (and (state ?f)
           (metal ?x))))

(forall (?occ ?x ?sPostOutput)
   (if (and (occurrence_of ?occ milling)
            (post-output-in-occ ?x ?occ ?sPostOutput)
            (metal ?x))
       (holdsA (milling-output-location ?x) ?sPostOutput)))
```

Expression 48: Output State Example

8 Tightly Coupled Communication

Tight coupling refers to processes that restrict which others give entities to them or receive entities from them. The tightest kind of coupling specifies exactly which processes give and receive the communicated entities (see 1.a in Sections 1 and 2). This is directly expressible in unextended PSL, see usage examples for functions in [8]. The medium tight categories assign a delegate to choose the other processes (1.b/c.i). Within these, some require the other processes to give and receive entities only when they start and finish (2.a). This is representable in unextended PSL, see usage examples for object orientation in [8].[59] Other processes in this category have ongoing communications during their execution (1.b.i / 2.b). The rest of this paper refers to these as *messages*. A

[58] An earlier paper defined relations to link states, occurrences, and input objects [6], but makes the severe assumption that each object is only input once per occurrence.

[59] Applying these examples to effectless communication with delegates (1.c.i) requires a convention for operations that require no particular effect.

process that does not restrict which other process will give or receive entities is using input and output (1.c) rather than messages, see Section 7. However, some ways of using the extensions in this section, and some modern software techniques, are nearly equivalent to input and output, see Section 9.

Common process languages usually define messages colloquially as things "sent" from one entity to another. The thing sent may be, for example, a command to perform a certain task, a notification of an event having occurred, or an uninterpreted piece of information. Another important aspect is that the receiver has some freedom in reacting to the message (as a delegate). Even for a command to perform a task, the receiver determines how the task is carried out.

This paper generalizes the above intuitions in three ways, to simplify and broaden the applicability of the formalization:

1. Entities that send and receive messages are processes. Sending and receiving are specific kinds of subprocesses under the control of these entities.

2. The thing sent is a participant in the sending and receiving processes (see Section 1). Sending a message is a way for the sending process to affect which objects are involved in the receiving process.

3. Any kind of thing can be sent, including physical objects, data, and communications. The process designer can interpret these as physical transport or knowledge transmission, requests, commands or other modalities ("performatives" in [31]). Useful core formalizations can be defined independently of the kind of thing sent.

The first generalization reflects the active nature of sending and receiving entities [1][32]. They have potentially complex processes for determining when to send what to whom, and how to react to receipt. Focusing too much on these entities as objects, or as sources and targets of messages, obscures the critical dynamics around sending and receiving. Treating senders and receivers as communicating processes that might hold state unifies these viewpoints and leads to a more complete semantics.

The second generalization takes messages as a way processes affect each other by determining the objects involved in them. For example, when a customer sends an order document to a vendor, the document becomes part of the fulfillment process in the seller, guiding that process as the customer desires. The notion of participant refers to an entity involved in a process in any way. For example, in a fabrication process, the machines used to operate on a piece of metal are participants, as are the human operators, the oil that lubricates the machines, the electricity that powers them, and the instructions that guide them.

The third generalization treats information and physical things uniformly as objects, and like all objects, as potential participants in processes. For example, the coded instructions

directing an automated milling machine are participants in the milling process, as are oil and metal. The transport of oil from a pump to the milling machine transfers participants between processes, as does the electronic communication of instructions. The sender affects participation in the milling process by determining which instructions or oil participates in the receiver. Whether information is interpreted as knowledge, belief, request, command, and so on, is up to the receiver, just as its use of a physical object is. This abstraction significantly simplifies the core theory, widens its applicability (including category 1.c.i in Sections 1 and 2), and serves as a base for extension with specifically epistemic or physical theories [33].[60]

Taken together, these generalizations enable messages to be formalized as constraints on participation. For example, an order sent from a customer to a vendor is constrained to participate in the vendor process only after it participates in the customer process, in particular, after the customer sends it. Section 8.1 gives a formalization of messaging in PSL in terms of predefined activities, and constraints on how they are used. Section 8.2 applies them to some examples.

8.1 Axioms for Messaging

This paper extends PSL for messaging rather than applying usage patterns, for the reasons given in Section 4. In particular, extensions enable the process designer to define messages between processes in a way that processes can be defined independently. For example, a customer process sending an order to a vendor is only concerned with sending messages, and the vendor process only with reacting appropriately when receiving them. This is more modular than writing one constraint for both as would be required when applying usage patterns.[61]

This section gives some predefined activities and some new relations for messaging, the same approach as for inputs and outputs. It defines an activity for sending any object to any process, as well as activities for transmitting and receiving messages. Constraints on these are divided into core axioms that apply to all messaging applications and other axioms that are useful, but not universal.

An accurate formalization of messages must separate the thing sent from the sending of that thing, a distinction obscured by the informal term “message.” The distinction is necessary because not all participants are messages, just the ones that are sent. And the same participant can be sent multiple times, by different occurrences of sending. The distinction corresponds to the typical separation in implementations between the “payload,” the thing being sent, and “headers,” which contain information about the sending itself, such as when it happened and who was responsible. Following this approach, activities defined for transmission and receipt of the message identify the

[60] PSL has some restrictions on participants, see footnote 22.
[61] See Section 5 of [8] for example usage patterns for messaging.

message by the occurrence of the sending activity, and the sending occurrence identifies the object sent.[62]

Expression 49 defines activities for sending, transmitting, and receiving messages between processes. Sending activities are identified in relation to an object that is sent, a grove that sends it, and a grove that receives it (see Expression 4 for description of groves as process executions). Messages are sent between groves to relieve the sender of specifying exactly which of the receiver's complex occurrences are "happening" when the message is sent. The relations are defined as functions because they identify exactly one activity for each set of object, sender grove, and receiver grove. The activities are not required to be atomic in the extension, but the process designer can constrain theirs to be atomic as needed. The transmitting and receiving activities are functions of the object sent and the sending occurrence, in case the sending occurrence sends multiple objects.[63,64]

```
(forall (?a ?x ?sReceiverGroveRoot ?aReceiverGroveAct
               ?sSenderGroveRoot ?aSenderGroveAct)
   (if (= ?a (send-message ?x
                ?sReceiverGroveRoot ?aReceiverGroveAct
                ?sSenderGroveRoot ?aSenderGroveAct))
       (and (activity ?a)
            (activity-participant ?x ?a)
            (grove ?sReceiverGroveRoot ?aReceiverGroveAct)
            (grove ?sSenderGroveRoot ?aSenderGroveAct))))

(forall (?a ?x ?sSend)
   (if (= ?a (transmit-message ?x ?sSend))
       (and (activity ?a)
            (activity-participant ?x ?a)
            (exists (?sReceiverGroveRoot ?aReceiverGroveAct
                     ?sSenderGroveRoot ?aSenderGroveAct)
               (occurrence_ofA ?sSend
                  (send_message ?x
                    ?sReceiverGroveRoot ?aReceiverGroveAct
                    ?sSenderGroveRoot ?aSenderGroveAct))))))
```

[62] An alternative approach is to require the thing sent to be unique per occurrence of message sending, where the same participant could be sent multiple times by using different "wrappers." However, wrapping is more cumbersome, since a different wrapper is needed to send the same thing multiple times. It also does not capture the most basic aspect of messaging, the act of sending. Identifying messages by the sending occurrences also enables techniques for controlling unwanted concurrency, see logical time stamps for message sends in [32].

[63] Expression 49 does not require messaging activities to be different when the parameters of the activity functions are, see footnote 44 on page 33. This means an occurrence of a messaging activity could send multiple messages, or send and receive messages at the same time.

[64] Parameterizing RECEIVE-MESSAGE by the sending occurrence does not affect support for anonymous messaging, which are constraints against queries to the particular messaging implementation about the sender of a message.

```
(forall (?a ?x ?sSend)
   (if (= ?a (receive-message ?x ?sSend))
       (and (activity ?a)
            (activity-participant ?x ?a)
            (exists (?sSenderGroveRoot ?aSenderGroveAct
                     ?sReceiverGroveRoot ?aReceiverGroveAct)
               (occurrence_ofA ?sSend
                  (send_message ?x
                    ?sReceiverGroveRoot ? aReceiverGroveAct
                    ?sSenderGroveRoot ?aSenderGroveAct))))))
```

Expression 49: Sending, Transmitting, and Receiving Messages

Expression 50, Expression 51, and Expression 52 give core constraints on the messaging activities above. Expression 50 requires sending occurrences to be in the groves they specify. Expression 51 requires transmission to start with sending. It uses PSL's support for complex occurrences that have some suboccurrences in common and not others.[65] This enables the sending occurrence to be in its grove as well as in the transmission process. Expression 52 requires at least one transmission per receipt, with the receipt at the end of the transmission. The combination of Expression 51 and Expression 52 require every receipt to be after a matching send under some transmission.[66]

```
(forall (?sSend ?x ?sReceiverGroveRoot ?aReceiverGroveAct
                  ?sSenderGroveRoot ?aSenderGroveAct)
   (if (occurrence-ofA ?sSend
          (send_message ?x ?sReceiverGroveRoot ?aReceiverGroveAct
                           ?sSenderGroveRoot ?aSenderGroveAct))
       (subocc-in-grove ?sSend
                        ?sSenderGroveRoot ?aSenderGroveAct)))
```

Expression 50: Send in Sending Grove

```
(forall (?sTransmit ?x ?sSend)
   (if (occurrence-ofA ?sTransmit (transmit-message ?x ?sSend))
       (root-occA ?sSend ?sTransmit)))
```

Expression 51: Transmission Starts with Sending

[65] Most process languages require complex occurrences to have either none of their suboccurrences under another complex occurrence, unless it is a superoccurrence (strong nesting, see footnote 27 and Expression 42).

[66] The expressions allow receipt to overlap send under the transmission, and be the same as its send if the result of the send and receive functions is the same activity, see footnote 63.

```
(forall (?sReceive ?x ?sSend)
   (if (occurrence_ofA ?sReceive (receive-message ?x ?sSend))
       (exists (?sTransmit)
          (and (occurrence_ofA ?sTransmit
                  (transmit-message ?x ?sSend)))
               (leaf-occA ?sReceive ?sTransmit))))
```

Expression 52: Receipt Always Due to Transmission

The core axioms above are very loose about the effectiveness of sending a message. For example, they do not require a message to be transmitted when sent, or received when transmitted. They allow a message to be delivered more than once, and to incorrect recipients. These conditions are true in many applications, due to accidents and unforseen circumstances. However, even in these applications it is useful to assume that messaging occurs as desired, to enable proof of properties about communicating processes.

Expression 53, Expression 55, and Expression 56 give constraints to represent reliable messaging. Expression 53 requires the receipt of a message be in the grove to which it was sent. Expression 54 requires transmission to end with receipt. Expression 55 requires a transmission for every send. Expression 56 requires that messages are transmitted and received at most once. In PSL, this means once per branch of the occurrence tree, that is, no more than one transmission for same send ending on the same branch (see SAME-OCC-BRANCHA in Expression 6). The combination of Expression 54, Expression 55, and Expression 56 require every send to have exactly one transmission and receipt.

```
(forall (?sTransmit ?x ?sSend)
   (if (occurrence-ofA ?sTransmit (transmit-message ?x ?sSend))
       (root-occA ?sSend ?sTransmit)))
```

Expression 53: Receipt in Recipient Grove

```
(forall (?sTransmit ?x ?sSend)
   (if (occurrence-ofA ?sTransmit (transmit-message ?x ?sSend))
       (exists (?sReceive)
          (and (occurrence-ofA ?sReceive
                               (receive-message ?x ?sSend))
               (leaf-occA ?sReceive ?sTransmit)))))
```

Expression 54: Transmission Ends with Receipt

```
(forall (?sSend ?x ?sReceiverGroveRoot ?aReceiverGroveAct
                  ?sSenderGroveRoot ?aSenderGroveAct)
   (if (occurrence_ofA ?sSend
          (send_message ?x ?sReceiverGroveRoot ?aReceiverGroveAct
                        ?sSenderGroveRoot ?aSenderGroveAct))
       (exists (?sTransmit)
          (occurrence-ofA ?sTransmit
             (transmit-message ?x ?sSend)))))
```

Expression 55: Sending Always Results in Transmission

```
(forall (?s1 ?s2 ?x ?sSend)
  (if (and (occurrence-ofA ?s1 (transmit-message ?x ?sSend))
           (occurrence-ofA ?s2 (transmit-message ?x ?sSend))
           (not (= ?s1 ?s2)))
      (not (same-occ-branchA ?s1 ?s2))))

(forall (?s1 ?s2 ?x ?sSend)
  (if (and (occurrence-ofA ?s1 (receive-message ?x ?sSend))
           (occurrence-ofA ?s2 (receive-message ?x ?sSend))
           (not (= ?s1 ?s2)))
      (not (same-occ-branchA ?s1Leaf ?s2Leaf))))
```

Expression 56: No More Than One Transmission and Receipt for the Same Send

Other optional constraints can be defined on messaging that are useful in some applications. For example, "quality of service" constraints can require that transmission results in a receipt within a certain time, or that transmission notify the sender if it fails to complete. A less common but useful constraint is that messages are received in the same order in which they are sent. This can only be ensured in centralized messaging systems, but facilitates interactions based on strict protocols.

8.2 Applying Messaging Extensions

This section applies the messaging extensions of 8.1 to translate some common process modeling techniques to PSL. Expression 57 shows an example of a customer process sending an order to a vendor process (delegation to a subprocess of the vendor is omitted for brevity). The first subexpression requires the customer to create an order before sending it, and to send account information for billing. The second subexpression requires the vendor to fill the order after both the order and the account have arrived. For brevity, the expression omits other relations, such as activity and subactivity relations, usages, and correlation in the receiver to link orders and accounts.

```
(forall (?occCustomer ?sCustomerGroveRoot ?order ?sSendOrder
         ?sVendorGroveRoot)
   (if (and (occurrence_of ?occCustomer customer)
            (occ-grove-root ?sCustomerGroveRoot ?occCustomer)
            (order ?order)
            (subactivity-occurrence-of ?sSendOrder ?occCustomer
               (send-message ?order ?sVendorGroveRoot vendor
                             ?sCustomerGroveRoot customer)))
       (exists (?sCreateOrder ?account ?sSendAccount)
          (and (subactivity-occurrence-of ?sCreateOrder
                  ?occCustomer (createOrder ?order))
               (min-precedesA ?sCreateOrder ?sSendOrder customer)

               (account ?account)
               (subactivity-occurrence-of ?sSendAccount
                  ?occCustomer
                  (send-message ?account
                     ?sVendorGroveRoot vendor
                     ?sCustomerGroveRoot customer))))))

(forall (?occVendor ?sVendorGroveRoot ?order ?sReceiveOrder
         ?sSendOrder ?account ?sReceiveAccount ?sSendAccount)
   (if (and (occurrence_of ?occVendor vendor)
            (occ-grove-root ?sVendorGroveRoot ?occVendor)
            (order ?order)
            (subactivity-occurrence-of ?sReceiveOrder
               ?occVendor (receive-message ?order ?sSendOrder))
            (account ?account)
            (subactivity-occurrence-of ?sReceiveAccount
               ?occVendor (receive-message ?account
                                           ?sSendAccount)))
       (exists (?sFillOrder)
          (and (subactivity-occurrence-of ?sFillOrder ?occVendor
                                          (fillOrder ?order))
               (min-precedesA ?sReceiveOrder ?sFillOrder
                              vendor)
               (min-precedesA ?sReceiveAccount ?sFillOrder
                              vendor)))))
```

Expression 57: Communicating Process Message Example

Expression 58 is an example of agents communicating "knowledge" that constrains their processes. It uses PSL states, which are objects representing conditions of the world (see Section 6.3). Since states are objects, they can be participants, and participation of a state in a process is interpreted in this example as the process "knowing" or "believing" the state actually holds in the world.[67] Expression 58 shows a process for a bank guard that can open a safe only after receiving a message giving the combination of the safe (delegation to a subprocess of the guard is omitted for brevity). The fact that a particular

[67] Participation of states can be specialized for various modalities.

safe has a certain combination is a state of the world. The last subexpression requires that the guard receive the proper state before opening the safe.

```
(forall (?a ?x ?y)
   (if (= ?a (openSafe ?x ?y))
       (and (activity ?a)
            (safe ?x)
            (combination ?y))))

(forall (?f ?x ?y)
   (if (= ?f (safe-has-combination ?x ?y))
       (and (state ?f)
            (safe ?x)
            (combination ?y))))

(forall (?occGuard ?sOpen ?safe ?combination)
   (if (and (occurrence_of ?occGuard guard)
            (subactivity-occurrence-of ?sOpen ?occGuard
               (openSafe ?safe ?combination)))
       (exists (?sReceiveCombination ?safeHasCombination ?sSend)
          (and (subactivity-occurrence-of ?sReceiveCombination
                  ?occGuard (receive-message ?safeHasCombination
                                             ?sSend))
               (= ?safeHasCombination
                  (safe-has-combination ?safe ?combination))
               (min-precedesA ?sReceiveCombination ?sOpen
                              guard)))))
```

Expression 58: Agent Message Example

9 Unifying Loosely and Tightly Coupled Communication

Input, output, and messaging as defined in Sections 7.2 and 8.1 are ways of specifying how one process determines the entities involved in another (participation, see Section 1). They are distinguished by the degree of restriction that can be placed on the other processes giving inbound entities and receiving outbound entities (coupling):

- A process specified with the accept and post activities of Section 7.2 (inputs and outputs) cannot express restrictions on which other processes receive and give the posted and accepted objects, and for posted objects cannot even require that any other process will accept them (loose coupling). For example, Expression 34 cannot restrict where the piece of metal comes from in the first accept, and Expression 37 cannot restrict where the piece of metal goes after the second post, or even require that the metal goes anywhere at all.

- A process specified with the send and receive activities of Section 8.1 (messages) can place more or less restriction on the sender and receiver, but requires a

receiver to exist, unlike outputs (nearly loose to tight coupling). For example, the first subexpression of Expression 57 requires messages to be received by a vendor process, but does not specify exactly which execution of a vendor process (which particular vendor, that is, which grove root). It could have placed no restriction on the receiver except existence, see Section 9.1. The second subexpression does not restrict the processes sending messages to the vendor at all.

The flexibility of messaging is both a strength and a weakness. Its capacity to provide any form of coupling from nearly the loosest to the tightest has led to its dominance of interprocess communication in modern software engineering. Only the very lowest level processes use only input and output anymore, mostly datatype manipulation, as for strings and numbers. The other processes in a typical modern software application all communicate through messages. However, their capacity to place strong restrictions on message senders and receivers allows applications to impose significant limitations on composability of process specifications. For example, Expression 57 requires a customer to identify a particular kind of process (vendor) to receive an order, when it may be more effective for a broker or other entity to determine the kind of receiver.[68] Messaging has more than enough flexibility left over from building good applications to build poor ones also.

The prevalence of messaging in modern software and the desirability of loose coupling suggest it is important to understand how to use messaging in a loosely coupled way. Section 9.1 describes a technique for constraining messaging between loosely coupled applications. It also shows how to constrain an input/output application to loosely coupled messaging. Section 9.2 summarizes new techniques in software messaging that achieve loose coupling.

9.1 Protocols

Protocols are a kind of process specification governing communication between other processes. When used with messages, they preserve composability by placing constraints on senders and receivers in processes without modifying the specifications of those processes. For example, Expression 59 defines a protocol by generalizing the communication between vendor and customer in Expression 57 to be applicable to any two groves, and to constrain only the communication between them, rather than "internal" suboccurrences (it also adds a message back to the customer with the ordered product). The first subexpression defines an activity for communication between two groves (PURCHASING). The second requires that the suboccurrences of these activities

[68] Messaging is also more cumbersome for specifying loosely coupled interprocess communication than inputs and outputs. For example, processes often share participants with subprocesses. Using messages, the superprocess sends the participant to the subprocess. Using inputs and outputs, the subprocess can accept any participant accepted by the superprocess without additional posts, as in Expression 34. The subprocess can also accept objects posted by other subprocesses, as in Expression 36, or even from a change, as in Expression 35. Using messages for these examples requires additional sends and receives to have the same effect, and detecting changes requires an external process to send notification of the change.

include a series of message tranmissions between the two groves.[69, 70] It uses the SAME-OCC-BRANCH and EARLIERA relations to ensure the same occurrences in each grove are communicating under each occurrence of the protocol.[71] The separate specifications for each grove must be loose enough to allow the protcol to constrain the message sends and receives as needed. The specification for the initiating grove must establish the existence of an occurrence of the protocol, which identifies the other groves involved in the protocol.[72] Theorem provers can check the consistency of grove specifications with the protocol. This assumes closure expressions to rule out other transmissions in the protocol, and the full set of messaging axioms in 8.1.

```
(forall (?a ?order ?sCustomerGroveRoot ?aCustomerGroveAct
         ?sVendorGroveRoot ?aVendorGroveAct)
   (if (= ?a (purchasing ?order ?sCustomerGroveRoot ?aCustomerAct
                        ?sVendorGroveRoot ?sVendorAct)
       (and (activity ?a)
            (order ?order)
            (grove ?sCustomerGroveRoot ?aCustomerAct)
            (grove ?sVendorGroveRoot ?sVendorAct)))))

(forall (?aProtocol ?order ?sCustomerGroveRoot ?aCustomerGroveAct
         ?sVendorGroveRoot ?aVendorGroveAct ?occProtocol)
   (if (and (= ?aProtocol (purchasing ?order
                             ?sCustomerGroveRoot ?aCustomerAct
                             ?sVendorGroveRoot ?sVendorAct))
            (occurrence_of ?occProtocol ?aProtocol))
       (exists (?sSendOrder ?sTransmitOrder
                ?account ?sSendAccount ?sTransmitAccount
                ?product ?sSendProduct ?sTransmitProduct
                ?sReceiveProduct)
          (and (occurrence_of ?sSendOrder
                  (send-message ?order
                     ?sVendorGroveRoot ?sVendorAct
                     ?sCustomerGroveRoot ?aCustomerAct))
               (subactivity-occurrence-of
                  ?sTransmitOrder ?occProtocol
                  (transmit-message ?order ?sSendOrder))
```

[69] This uses PSL's support for complex occurrences that have some suboccurrences in common and not others, see discussion of transmission in description of Expression 51.

[70] Usage relations linking occurrences of protocols to transmissions are omitted for brevity (see Expression 9 and Expression 10), but are needed to prevent the same transmissions from appearing in more than one protocol.

[71] This has an analogous effect to the convenience relations for input and output (Expression 27) used in the translation examples in Section 7.3. In Expression 59 and Expression 60, the EARLIERA subexpression for the groves is partially redundant with the protocol because it constrains occurrence ordering the same way. SAME-OCC-BRANCH could be used instead for clarity, but it has more paths to search during proof.

[72] Identifying the groves involved in a protocol might require negotiations to reach agreement to follow the protocol. These negotiations are also protocols, which bottom out with protocols that do not require responses, such as contract nets [35].

```
(account ?account)
(occurrence_of ?sSendAccount
   (send-message ?account
      ?sVendorGroveRoot ?aVendorGroveAct
      ?sCustomerGroveRoot ?aCustomerAct))
(subactivity-occurrence-of
   ?sTransmitAccount ?occProtocol
   (transmit-message ?account ?sSendAccount))

(same-occ-branchA ?sSendOrder ?sSendAccount)

(product-matching-order ?product ?order)
(occurrence_of ?sSendProduct
   (send-message ?product
      ?sCustomerGroveRoot ?aCustomerGroveAct
      ?sVendorGroveRoot ?aVendorGroveAct))
(subactivity-occurrence-of
   ?sTransmitProduct ?occProtocol
   (transmit-message ?product ?sSendProduct))
(leaf_occ ?sReceiveProduct ?sTransmitProduct)
(earlierA ?sSendOrder ?sReceiveProduct)

(min-precedesA ?sTransmitOrder ?sTransmitProduct
               ?aProtocol)
(min-precedesA ?sTransmitAccount ?sTransmitProduct
               ?aProtocol)))))
```

Expression 59: Protocol Example

Protocols can have subactivities, like any other activity, and these can also be protocols. Expression 60 shows a protocol that uses the one above, adapted from [34]. The customer sends a catalog request to the vendor, who replies with the catalog. Then the purchasing protocol in Expression 59 occurs. Expression 60 happens to have the product vendor as the catalog provider, but it could be written to enable a third party to provide the catalog, assuming a relation between catalog providers and product vendors is defined. It uses the EARLIERA relation in the same way as Expression 59.

```
(forall (?a ?sCustomerGroveRoot ?aCustomerGroveAct
         ?sVendorGroveRoot ?aVendorGroveAct)
   (if (= ?a (purchasing-from-catalog
               ?sCustomerGroveRoot ?aCustomerAct
               ?sVendorGroveRoot ?sVendorAct)
       (and (activity ?a)
            (grove ?sCustomerGroveRoot ?aCustomerAct)
            (grove ?sVendorGroveRoot ?sVendorAct)))))
```

```
(forall (?aProtocol ?sCustomerGroveRoot ?aCustomerGroveAct
         ?sVendorGroveRoot ?aVendorGroveAct ?occProtocol)
   (if (and (= ?aProtocol (purchasing-from-catalog
                           ?sCustomerGroveRoot ?aCustomerAct
                           sVendorGroveRoot ?sVendorAct))
            (occurrence_of ?occProtocol ?aProtocol))
       (exists (?catalog-request ?sSendRequest ?sTransmitRequest
                ?catalog ?sSendCatalog ?sTransmitCatalog
                ?sReceiveCatalog ?order ?sPurchasing)
          (and (catalog-request ?catalog-request)
               (occurrence_of ?sSendRequest
                  (send-message ?catalog-request
                     ?sVendorGroveRoot ?sVendorAct
                     ?sCustomerGroveRoot ?aCustomerAct))
               (subactivity-occurrence-of
                  ?sTransmitRequest ?occProtocol
                  (transmit-message ?catalog-request
                                    ?sSendRequest))
               (catalog ?catalog)
               (occurrence_of ?sSendCatalog
                  (send-message ?catalog
                     ?sCustomerGroveRoot ?aCustomerAct
                     ?sVendorGroveRoot ?sVendorAct))
               (subactivity-occurrence-of
                  ?sTransmitCatalog ?occProtocol
                  (transmit-message ?catalog ?sSendCatalog))
               (leaf_occ ?sReceiveCatalog ?sTransmitCatalog)
               (earlierA ?sSendRequest ?sReceiveCatalog)

               (min-precedesA ?sTransmitRequest ?sTransmitCatalog
                              ?aProtocol)

               (order-from-catalog ?order ?catalog)
               (subactivity-occurrence-of
                  ?sPurchasing ?occProtocol
                  (purchasing ?order
                     ?sCustomerGroveRoot ?aCustomerAct
                     ?sVendorGroveRoot ?sVendorAct))
              (earlierA ?sTransmitCatalog ?sPurchasing)

              (min-precedesA ?sTransmitCatalog ?sPurchasing
                             ?aProtocol)))))
```

Expression 60: Subprotocol Example

Protocols can also apply to input and output, by constraining occurrences of accepting and posting in multiple groves. The expressions are similar to above, except with accepts and posts in each grove as suboccurrences, instead of sends and receives. The expressions require inputs accepted by one grove to be the same as outputs provided by another, similar to what Expression 40 does for drilling and milling in Section 7.3. Protocols have the same effect on participants in each grove regardless of whether the

groves use input/output or messaging. In both cases, an object participates in one grove, is output or sent, then participates in another after input or receipt. For messaging activities, the protocol determines the sender and receiver, assuming the separate grove specifications are loose enough to allow it. For accept/post activities, the protocol ensures outputs are the same as the inputs of specific other groves, not just any other grove as allowed by post and accept.

Applications using input and output can be constrained to use messages when posts and accepts are complex occurrences that place no constraints on their suboccurrence, and can be matched one to one, as in Expression 36. Expression 61 defines relations requiring posts and accepts to have sends and receives as suboccurrences, respectively. Each message sent or received must be for the same participant as the post or accept under which it is a suboccurrence. Each send must originate from the same grove as the one to which the participant is posted. Each receive must be for a send to the same grove as the one from which the participant is accepted. Additional constraints can require that suboccurrences of the posts and accepts are the same as suboccurrences of the sends and receives.

```
(forall (?sPost ?sSend)
   (iff (post-refinement ?sPost ?sSend)
        (exists (?x ?sSourceGroveRoot ?aSourceGroveAct
                 ?sReceiverGroveRoot ?aReceiverGroveAct)
           (and (occurrence-of ?sPost
                   (post-output ?x ?sSourceGroveRoot
                                ?aSourceGroveAct))
                (occurrence-of ?sSend
                   (send-message ?x
                      ?sReceiverGroveRoot ?aReceiverGroveAct
                      ?sSourceGroveRoot ?aSourceGroveAct))
                (subactivity-occurrence ?sSend ?sPost)))))

(forall (?sAccept ?sReceive)
  (iff (accept-refinement ?sAccept ?sReceive)
       (exists (?x ?sTargetGroveRoot ?aTargetGroveAct
                ?sSend ?sSenderGroveRoot ?aSenderGroveAct)
         (and (occurrence-of ?sAccept
                 (accept-input ?x ?sTargetGroveRoot
                               ?aTargetGroveAct))
              (occurrence-of ?sReceive
                             (receive-message ?x ?sSend))
              (occurrence-of ?sSend
                 (send-message ?x
                    ?sTargetGroveRoot ?aTargetGroveAct
                    ?sSenderGroveRoot ?aSenderGroveAct)))
              (subactivity-occurrence ?sReceive ?sAccept))))
```

Expression 61: Constraining Input and Output to Messaging

9.2 Ports

Modern software messaging achieves loose coupling by enabling them to be sent to *ports* on the sender, leaving the determination of the final receiver to the context in which the sender is operating [1][3][4] [18][19][20][21]. This significantly improves composability of processes, compared to sending messages directly to a specific process. For example, the determination of receivers can be:

- Channels between ports on the sender and ports on the receiver.
- Publish and subscribe mechanisms where other processes sign up for receiving messages.
- Arbitrarily complex processes for determining the receivers.

The above techniques give messaging the capability of input and output, including not requiring existence of the receiver. They can completely disconnect process specifications from others with which they can share participants (see Section 7). There are at least two kinds of ports [18]:

- "Lightweight" ports cannot receive messages directly. Message are "sent" to the sending process itself, with the port as additional information provided for determining the receiver.
- "Heavyweight" ports can receive messages directly. They are entities in (subprocesses of) the sender.

Sending messages to lightweight ports has the loose coupling of input and output as defined in Section 7 (1.c.ii), although it often restricts the effect of the receiving process (1.b.ii). This is the dominant interpretation in some application areas, for example in web services [3]. The formalization of lightweight ports is the same as input and output (accept and post activities defined in Section 7.1), except using the input or output object to indicate effect when needed, and applied to strongly nested processes as in typical software languages.[73] The "receiver" is determined by other constraints on the equality of outputs and inputs, defining how ports on the sender are "wired" to ports on the receiver. These constraints can range from fixed channels to arbitrarily complex determinations, such as filtering out some inputs, modifying them, or routing them dynamically.

Sending messages to heavyweight ports is a messaging application that has the effect of lightweight ports, but has intermediate stages of message delivery. The transmitted object is sent first to a port on the sending process, then from there to a port on an external process, which can forward it again to an internal subprocess as the final receiver. Formalizing heavyweight ports uses the sending, transmitting, and receiving activities of Section 8.1. See Section 7.1 of [8] for additional PSL extensions for heavyweight ports and channels.

[73] See footnote 27 and Expression 42.

10 Future Work

Other features of common process languages include aborting and suspending processes. Aborting refers to halting processes permanently before they complete. Suspending refers to halting processes temporarily and resuming them later. These require extensions to identify areas of the occurrence tree where a process is "open," but may not conform to the process specification, as discussed in Section 6.2. An aborted process is a portion of the tree that conforms to a process specification only up to some intermediate point.[74] A suspended process is a portion that conforms to a process specification except during some intermediate period. These are aspects of representing processes that manage the execution of other processes.

Constraints on participation are useful in formalizing other process languages. For example, industrial applications of state machines interpret them as specifying how an entity will react to incoming messages, in particular, what messages it will send out in response [1][36]. The entity accepting messages changes state according to the messages it receives, sending out other messages as it does. Another example is applications of Petri nets, which often interpret places as subprocesses, and tokens as participants in those processes [37]. Under this interpretation, transitions, arcs, and many Petri net properties can be formalized as constraints on participation. For example, a safe Petri net cannot have more than one token in each place at one time (suboccurrences identified by each usage relation must happen at different times). A place is reachable if it can have at least one token (its usage relation is required to identify at least one suboccurrence). Conflict refers to whether there is nondeterminism in transition firing (activity tree branches have different processes executing). A PSL representation of Petri nets can express fully concurrent transition firing, in addition to synchronous firing as represented in matrix formalizations.

11 Conclusion

This paper identifies dimensions of interprocess communication characterizing how processes effect the entities involved in other processes. The primary dimensions are identification of other processes to communicate with (varying by the degree of restriction on those processes, loose to tight), and specification of when communication happens (either anytime during a processes or only at the beginning and end). Common process language features are placed along these dimensions, such as functions, object-oriented and agent messages, publication, subscription, and parameters. The paper extends PSL with predefined activities for loosely and tightly coupled processes, along with axioms expressing constraints on their usage. These are applied to translating common process language patterns to PSL, including data flow and protocols. A technique is also presented for incrementally translating common process sequence elements to separate PSL expressions, improving readability and facilitating automated inference.

[74] See application of aborting to change detection, footnote 45 on page 35.

Acknowledgements

Thanks to Peter Denno and Jeff Smith for reviewing drafts of this paper, and to Michael Gruninger for his assistance with PSL. A number of topics in this paper were motivated by discussions with the Business Process Definition Metamodel and Business Process Modeling Notation teams at the Object Management Group.

Commercial equipment and materials might be identified to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the U.S. National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

[1] Object Management Group, "UML 2.1 Superstructure Specification," http://www.omg.org/cgi-bin/doc?ptc/06-04-02, April 2006.

[2] Kalev, D., ANSI/ISO C++ Professional Programmer's Handbook, Que, June 1999.

[3] W3C, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," http://www.w3.org/TR/wsdl20, May 2005.

[4] W3C, "Web Services Choreography Description Language Version 1.0," http://www.w3.org/TR/ws-cdl-10, December 2004.

[5] Bock, C., Gruninger, M., PSL: A Semantic Domain for Flow Models," Software and Systems Modeling Journal, 4:2, pp. 209–231, May 2005.

[6] Bock, C., Gruninger, M., "Inputs and Outputs in PSL," NIST Internal Report 7152, August 2004.

[7] Bock, C., "UML 2 Activity and Action Models Part 4: Object Nodes," Journal of Object Technology, 3:1, pp. 27-41, January 2004.

[8] Bock, C., Gruninger, M., "Messaging in the Process Specification Language," NIST Internal Report 7258, August 2005.

[9] Bock, C., "UML 2 Activity and Action Models Part 2: Actions," Journal of Object Technology, 2:5, pp. 41-56, September 2003.

[10] Meyer, B., Eiffel : The Language, October 1991.

[11] Object Management Group, "OMG SysML Specification," http://doc.omg.org/ptc/06-05-04, May 2006

[12] Object Management Group, "Business Process Modeling Notation Specification," http://www.omg.org/cgi-bin/doc?dtc/06-02-01, February 2006.

[13] Bock, C., "SysML and UML 2.0 Support for Activity Modeling," Journal of International Council of Systems Engineering, 9:2, pp. 160-186, Summer 2006.

[14] Object Management Group, "Object Constraint Language," http://doc.omg.org/formal/06-05-01, May 2005.

[15] Gruninger, M., "PSL Ontologies – Current Theories and Extensions," http://www.mel.nist.gov/psl/ontology.html, 2006.

[16] International Organization for Standardization, "Process Specification Language (ISO 18629)," ISO TC184 SC4 http://www.tc184-sc4.org/SC4_Open/SC4%20Legacy%20Products%20(2001-08)/PSL_(18629)/, June 2006.

[17] International Organization for Standardization, "Common Logic (CL) - A Framework for a Family of Logic-Based Languages," ISO/IEC JTC1 SC32 WG2, http://philebus.tamu.edu/cl, June 2006.

[18] Bock, C., "UML 2 Composition Model," Journal of Object Technology, 3:10, pp. 47-73, November 2004.

[19] Selic, B., Gullekson, G., Ward, P., Real-Time Object-Oriented Modeling, Wiley, April 1994.

[20] Ellsberger, Jan., Hogrefe, D., Sarma, A., SDL: Formal Object-Oriented Language for Communicating Systems, 2nd Edition, Prentice Hall, 1997.

[21] Object Management Group, "CORBA Component Model Specification," http://doc.omg.org/formal/06-04-01, April 2006.

[22] W3C, "OWL Web Ontology Language Semantics and Abstract Syntax," http://www.w3.org/TR/owl-semantics, February 2004.

[23] Gruninger, M., "Guide to the Ontology of the Process Specification Language," in Staab, S. (ed.) Handbook of Ontologies in Information Systems, Springer-Verlag, January 2004.

[24] Object Management Group, "MOF QVT," http://doc.omg.org/ptc/05-11-01, November 2005.

[25] Gerbaux, F., Gruber, T., "Theory KIF-META," http://www-ksl.stanford.edu/knowledge-sharing/ontologies/html/kif-meta, July 1994.

[26] Dijkstra, E.W., “Go To Statement Considered Harmful,” Communications of the ACM, 11:3, pp. 147-148, March 1968.

[27] Object Management Group, “Workflow Management Facility Specification,” http://www.omg.org/cgi-bin/doc?formal/00-05-02, May 2000.

[28] Workflow Management Coalition, “Workflow Standard - Interoperability Abstract Specification,” http://www.wfmc.org/standards/docs/TC-1012_Nov_99.pdf, November 1999.

[29] D. Oliver, T. Kelliher, and J. Keegan, Jr., Engineering complex systems with models and objects, McGraw-Hill, New York, January 1997.

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

[31] Foundation for Intelligent Physical Agents, “FIPA ACL Message Structure Specification,” http://www.fipa.org/specs/fipa00061/SC00061G.pdf, December 2002.

[32] Lamport, L., “Time, Clocks and the Ordering of Events in a Distributed System,” Communications of the ACM, 21:7, pp. 558-565, July 1978.

[33] Scherl, R., Levesque, H., “Knowledge, Action, and the Frame Problem,” Artificial Intelligence, 144:1-2, pp. 1-39, March 2003.

[34] Business Process Metamodel Submission Team, “Business Process Definition MetaModel,” http://doc.omg.org/bmi/2006-09-07, September 2006.

[35] Smith, R., “The contract net protocol: : High-Level Communication and Control in a Distributed Problem Solver,” IEEE Transactions on Computers, C-29:12, pp. 1104-1113, December 1980.

[36] Sangiorgi, D., Walker, D., The Pi Calculus: A Theory of Mobile Processes, Cambridge University Press, October 2003.

[37] Zhou, M., DiCesare F., “Parallel and Sequential Mutual Exclusions for Petri Net Modeling of Manufacturing Systems with Shared Resources,” IEEE Transactions on Robotics and Automation, 7:4, pp. 515-527, August 1991.