NISTIR 7258

Messaging in the Process Specification Language

Conrad Bock
Michael Gruninger

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

NISTIR 7258

Messaging in the Process Specification Language

Conrad Bock
Michael Gruninger

Manufacturing Systems Integration Division
Manufacturing Engineering Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8260

August 2005

U.S. DEPARTMENT OF COMMERCE
Carlos M. Gutierrez, Secretary
TECHNOLOGY ADMINISTRATION
Michelle O'Neill, Acting Under Secretary of Commerce for Technology
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
William Jeffrey, Director

Messaging in the Process Specification Language

Conrad Bock
Michael Gruninger
August 24, 2005

Messaging is ubiquitous in modern computing, from the simplest forms of object-orientation to service-oriented architectures. This paper formalizes messages by the constraints they place on participation of entities in processes. Constraints are expressed in two ways: informally as usage patterns of the Process Specification Language (PSL), and formally as extensions to it. The extensions reduce ambiguity and increase expressiveness compared to conventional process modeling languages, by defining the relation of the extensions to existing PSL execution concepts. Examples drawn from a spectrum of message types illustrate the range of variability in the reaction of the message receiver. The extensions are used to formalize message forwarding, a useful technique in specifying composable processes. These suggest simpler alternatives to formalize inputs and outputs.

1. Introduction

Common flow modeling and process languages, whether in graphical form like the Unified Modeling Language (UML) [1], or textual form as in programming and web service interchange languages [2] [3][4], usually describe 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 that is the receiver has some freedom in reacting to the message. 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. Sending a message is a way for the sending process to affect which objects are involved in the receiving process.

3. The things sent can be any kind of object, including physical objects, information, or both. The process designer can interpret these as physical transport or knowledge transmission, but a useful core formalization can be defined independently of the kind of thing sent.

The first generalization reflects the active nature of sending and receiving entities [1][5]. 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 performers, or as source and target objects of the message, obscures the critical dynamics around sending and receiving. Treating senders and receivers as communicating processes, which may 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 so on. Participation also covers information, see next generalization.

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 is a message, as is the electronic communication of the instructions. In both, the sender affects participation in the milling process by determining which instructions or oil should participate. Whether participating 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, and serves as a base for extension with specifically epistemic or physical theories [6].

Taken together, these generalizations enable messages to be formalized as constraints on participation. For example, an order sent from a customer to vendor constrains the order to participate in the vendor process only after it participates in the customer process, specifically after the customer sends it. The Process Specification Language (PSL) provides a good base for this formalization, because it supports processes, objects, and participants, states of the world that can be used for epistemic messaging, and is axiomitized in first order logic to facilitate expression of constraints on these [7][8]. Participation constraints due to messaging can even be written directly in PSL without extension, and this paper shows how to do that, but some simple extensions to PSL also proposed here make messaging more explicit and enable definition of a core set of axioms for messaging semantics.

Section 2 briefly reviews PSL. Section 3 discusses various ways to describe the usage of an existing language, like PSL, in formalizing process modeling constructs in general, and identifies the ones employed in the rest of the paper. Section 4 presents a spectrum of messaging techniques formalized in later sections. Section 5 gives patterns of using unextended PSL across the spectrum of messaging. Section 6 defines messaging extensions to PSL and applies them across the spectrum. Section 7 describes additional extensions to support process composition, and compares these to analogous extensions for inputs and outputs from earlier work [10]. Section 8 describes future work.

2. PSL

Expressing participant constraints for process models requires a language that refers directly to processes as they actually occur. For example, in a process that drills a hole in 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 piece of metal will participate in both processes at some point in time, 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 happen to express this constraint, but it is not machine-verifiable, because the occurrence-level meaning of these diagrams is usually specified only in natural language.

This paper uses PSL to express participant constraints, and assumes familiarity with PSL fundamentals [7]. PSL has language elements for referring directly to real world processes as they occur, for example milling operations as they actually happen in a factory at certain times. It defines a simple set of concepts that cover all possible ways these operations can happen, which is called the *occurrence tree*. The process designer uses these concepts to write constraints on which occurrences are allowed, for example to specify what is required to happen during a milling process. This approach allows process requirements to be written generally or specifically as needed by the stage of design. For example, early design stages define loose constraints, because the domain expert is just sketching out broad requirements. These are tightened as design moves forward, until the process is completely specified. For example, a software program is a kind of process description that places many constraints on allowable executions.

PSL is project 18629 at the International Organisation of Standardization, and part of the work is a Draft International Standard [8]. It is based on a long period of research stemming from the situation calculus and enterprise modeling. It has been applied in scheduling, process modeling, process planning, production planning, simulation, project management, workflow, and business process reengineering. The standard is divided into core theories and extensions. The core axiomatizes a set of intuitive semantic primitives describing fundamental concepts of manufacturing processes. The core concepts include discrete states for relating processes to states of the world, as well as subactivities, atomic activities, and complex activities for composition of processes. Extensions introduce new terminology, to supplement the core concepts. They define additional relations for activities, time and state, activity ordering, duration, and resources. All axioms are first-order sentences, written in the Knowledge Interchange Format [9].

Additional PSL relations are defined in this paper just for convenience, entirely in terms of existing ones, as shown in Expression 1. The PARTICIPANT relation is carried over from [10], and the ACTIVITY-PARTICIPANT generalizes it for activities. The others are shorthands for existing PSL relations.

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

(forall (?x ?a)
   (iff (activity-participant ?x ?a)
        (forall (?occ)
           (implies (occurrence_of ?occ ?a)
                    (participant ?x ?occ)))))

(forall (?a ?occ ?a1)
   (iff (subactivity-occurrence-of ?s ?occ ?a)
        (and (occurrence_of ?occ1 ?a)
             (subactivity_occurrence ?s ?occ))))

(forall (?s ?occ)
   (iff (= ?s root-occ-fn(?occ))
        (root_occ ?s ?occ)))

(forall (?s ?occ)
   (iff (= ?s leaf-occ-fn(?occ))
        (leaf_occ ?s ?occ)))
```

Expression 1: Convenience Relations for PSL

3. Usage Patterns

For any sufficiently powerful language, such as PSL, messaging and other common process modeling constructs can be taken as specialized ways of using of that language, or patterns of usage. Patterns have the advantage of semantic clarity due to direct application of a known language. An earlier paper described patterns of using PSL for some common flow modeling constructs [7]. For example, UML decision points can be expressed as patterns of PSL constraints that have disjunctions over MIN_PRECEDES statements.

However, if usage patterns are left as "know how" that process designers learn through textual explanations and examples, they have a number of problems. In particular, patterns:

- can be complicated to write in the existing language.
- do not provide guidance to readers of the language as to which pattern is being used.
- cannot be the subject of constraints and reasoning.
- do not ensure the pattern is applied in the same way wherever it is used.

The problems above are due to informal descriptions of the pattern. Formalizing a pattern enables it to be defined once, constrained against existing language concepts, and applied uniformly many times. There are a number of more precise ways to express language usage:

- Mapping techniques

  These techniques support specifying translation from one language to another. 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 [11], with additional syntax for specifying where the blanks are, and what should fill them.[1,2] More powerful techniques include generalized mapping languages, which support machine-manipulable specification of arbitrarily complex translations [14].

- Formal metalanguages

  If the structure of sentences in the existing language are regularized enough, then a formal metalanguage can express these "statements about statements" [15]. 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

  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 [16].

- Predefined, reusable elements

  Elements can be defined using an 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].

Formalization of usage patterns is part of a larger cycle of moving methodological techniques into language concepts. Users find themselves applying the same techniques repeatedly, then tooling eventually adapts to support them, and 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

[1] The capabilities of Common Logic to quantify over a finite domain of relations can be used as a kind of template language, with if-and-only-if statements defining the expansion of the template [12].

[2] Some template techniques perform a computation to transform the filled template into an existing language, for example, compute macros in [13]. The transformation can be affected by the specific values that fill the blanks, and the current state of the transformation environment.

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 languages [1][17][18]. In ontologies, computationally efficient patterns of using first order logic were used to define most of the language constructs in the Ontology Web Language [19].[3]

This paper primarily adopts the reusable elements approach, by providing predefined instances of PSL concepts, plus with a few additional language constructs in the form of PSL relations. An earlier paper on inputs and outputs in PSL [10] relied on defining new PSL relations. Both techniques have the advantage of being embedded in the existing language, unlike formal metalanguages, and do not require translation as mapping techniques do.[4] In addition, when applied in the context of PSL, these approaches support formal definition as constraints on how the new constructs or reusable elements can be combined with existing PSL relations and with each other. This addresses the deficiency of introducing new constructs into typical process languages where there is no semantic relation between the new and existing constructs.

4. Kinds of Messages

Message types form a spectrum based on how much freedom the receiver has in reacting to a message. This is important to process designers, who need to know what to expect when sending a message. It also helps address the ambiguity of the intuitions in Section 1 as to how much freedom is allowed in the receiver for participant transfer to qualify as a message.[5] One end of the spectrum has little or no freedom, while the other has complete freedom:

1. Functions

   Messages start processes that follow a single specification for the function. For example, a function for adding integers will follow the same instructions whenever it is invoked. The function may be considered as receiving the message, which gives values for function parameters.

[3] One of the precursors of PSL gave a form of statements that expressed state-based preconditions, which were formalized in PSL by the addition of states.

[4] Formal metalanguages are more powerful in some respects and not others. They can express constraints on what statements a process designer is allowed to write, for example, to say which inputs to a function are optional, but they can be cumbersome when used to express complicated usage patterns.

[5] Other dimensions could be used to distinguish message types, for example, on whether state is preserved across multiple message receipts. Such a dimension is important to software development in that it restricts which processes have access to which data states. It also allows the same collection of data to receive many kinds of messages. This paper focuses on the variability in reaction to messaging because it has finer graduations for distinguishing message types, rather than just whether state is preserved or not. See footnote 21.

2. Abstract Datatypes

   Messages start processes as determined by the abstract operation and the type of operation parameter values given in the message, where the process will be same for each type (there is no subtyping). For example, an abstract operation for addition will follow different instructions for integers, reals, imaginaries, but they will be the same for all integers, the same for all reals, and so on, even though integers are a subset of reals.

3. Object-orientation and Components

   Messages start processes as determined by the requested operation and the type of operation parameter values given in the message, where different processes can apply to entity subtypes than supertypes. Most object-oriented languages determine the process by the type of object operated on only [2], called *dispatching* to a *method*.[6] For example, a drawing operation for graphical entities will apply to all the shapes falling under that category, triangles, circles, squares, and so on, and each specialized shape can use the instructions inherited from a supertype, define its own, or some combination of the two.[7]

   Components and object orientation provide the similar levels of flexibility to the receiver in choosing the process to start upon receiving a message, except that objects specify the data that is internal to the object, and inherit this to subtypes, whereas components do not. Since the scope of this paper is processes that change logical entities, rather than data, the difference between object and components will be omitted.[8]

4. Communicating Processes

   Messages can start or affect processes as determined by other processes already occurring in the receiver, and by the requested operation and the type of parameter values given in the message. These are often coordinated with other messages in a protocol acceptable to the receiving entity [1][22]. For example, making a selection at a vending machine will make the desired product available, but only if the proper change is provided first.[9]

---

[6] Some languages include the types of all the operands in the determination, such as multi-methods in [20] and overloaded member functions in [2]. Some include the individual instance receiving the message in the determination [20].

[7] The inheritance of process specifications has never been well addressed in object-orientation, due to the focus on combining the specification itself, rather than the runtime semantics. Languages with execution models, such as PSL, provide much better platforms for process inheritance [7].

[8] Components separate operation interface and data implementation in classes to address concerns of modularity in software construction, such as binary compatibility and white and black box reuse [21]. This article is intended to capture runtime constraints implied by these techniques, rather than how they facilitate software development.

[9] This category could be subdivided based on whether the processes form subtytpe hierarchies, as distinguished in abstract datatypes and object-orientation.

5. Agents

   Messages can start or affect processes determined by the receiving object, but the receiving object has complete discretion as to whether to grant the request and how to fulfill it [23]. Agents can even choose dynamically whether to be bound by any particular protocol with other agents.

The major transition in the spectrum occurs between object-orientation (3) and communicating processes (4), where messages are no longer guaranteed to start processes, and might not even affect the execution of existing ones. Messages at the higher end of the spectrum are usually handled by an ongoing process in the receiving entity, which may or may not start other processes. Not coincidentally, this is also the point where the reaction to the message becomes more complicated than simply choosing a process to start based on the operation and the type of receiving entity or parameter values. For example, the receiving process might queue the incoming message because it is busy, handle it immediately by interrupting what it was doing, forward it to another entity, or reject it entirely.

Most software developers would consider communicating processes and agents as definitely giving receivers enough discretion to be called messaging, while functions and abstract datatypes definitely do not. The middle category, object-orientation, might be classified as messaging by some and not others. In object-orientation, subtypes can be introduced at any time, resulting in completely different processes started for the same operation to objects that conform to a supertype. This adds an element of variability to object-orientation more characteristic of communicating processes and agents than abstract data types.

5. PSL Usage Patterns for Messages

This section gives patterns of using unextended PSL to express the various kinds of messages. Like many process languages, PSL provides constructs that facilitate specification of functions, abstract data types, and object-orientation. These constructs are on the side of the spectrum where the message is guaranteed to start a process at the receiver, and so are less "message-like." At the other end, for communicating processes and agents, the unextended patterns are more cumbersome. They are provided for completeness and for comparison to applying messaging extensions at this end of the spectrum in Section 6.2.

1. Functions

   Functions translate to PSL as constraint patterns requiring all executions of the function to follow the same rules. For example, Expression 2 requires the function DRILLANDMILL to execute the same way each time, due to the universal quantification over occurrences of the function, and the semantics of the PSL relations ROOT_OCC, NEXT_SUBOCC, and LEAF_OCC. These do not allow any other suboccurrences under DRILLANDMILL [7].

```
(forall (?a ?m)
  (implies (or (= ?a drillAndMill(?m))
               (= ?a drill(?m))
               (= ?a mill(?m)))
           (and (activity ?a)
                (metal ?m)
                (activity-participant ?m ?a))))

(forall (?m)
  (and (subactivity drill(?m) drillAndMill(?m))
       (subactivity mill(?m) drillAndMill(?m))))

(forall (?occDrillAndMill ?m)
   (implies
      (occurrence_of ?occDrillAndMill drillAndMill(?m))
      (exists (?sDrill ?sMill)
         (and (subactivity-occurrence-of
                ?sDrill ?occDrillAndMill drill(?m))
              (subactivity-occurrence-of
                ?sMill ?occDrillAndMill mill(?m))
              (root_occ ?SDrill ?occDrillAndMill)
              (next_subocc leaf-occ-fn(?SDrill)
                           root-occ-fn(?sMill))
              (leaf_occ ?SDrill ?occDrillAndMill)))))
```

Expression 2 : Pattern Example for Functions

2. Abstract datatypes

   Operations on abstract data types translate to PSL as constraint patterns that allow execution to vary only by the type of operation parameters. For example, Expression 3 is a constraint requiring the operation SHAPEPART to execute the same way for metal parts and the same way for plastic parts, but with separate instructions for each kind of material, and no specialization between them. It uses the function DRILLANDMILL for metal and MOLDANDTRIM for plastic, which specify a single, unvarying set of instructions each. The constraint of MOLDANDTRIM is omitted for brevity, but defined similarly to DRILLANDMILL.[10] Each subactivity of SHAPEPART operates on different type of material using the corresponding function.[11]

[10] Although Expression 3 uses the type of the material to determine the procedure to apply, it does not suffer from the lack of modularity that this causes in conventional programming languages. Each material is addressed in a separate set of statements, so as new materials become available for the abstract operation SHAPEPART, new sets of statements can be written without changing existing ones.

[11] The subactivity statements are limited by the type of material so they will not conflict with the function constraints above them.

```
(forall (?x ?a)
  (implies (= ?a shapePart(?x))
           (and (activity ?a)
                (activity-participant ?x ?a))))

(forall (?p ?a)
  (implies (= ?a moldAndTrim(?p))
           (and (activity ?a)
                (plastic ?p)
                (activity-participant ?p ?a))))

(forall (?x)
  (implies (metal ?x)
           (subactivity drillAndMill(?x) shapePart(?x))))

(forall (?x)
  (implies (plastic ?x)
           (subactivity moldAndTrim(?x) shapePart(?x))))

(forall (?occShapePart ?x)
   (implies
      (and (occurrence_of ?occShapePart shapePart(?x))
           (metal ?x))
      (exists (?occDrillAndMill)
         (and (root_occ ?occDrillAndMill ?occShapePart)
              (subactivity-occurrence-of ?occDrillAndMill
                 ?occShapePart drillAndMill(?x))
              (leaf_occ ?occDrillAndMill
                        ?occShapePart)))))

(forall (?occShapePart ?x)
   (implies
      (and (occurrence_of ?occShapePart shapePart(?x))
           (plastic ?x))
      (exists (?occMoldAndTrim)
         (and (root_occ ?occMoldAndTrim ?occShapePart)
              (subactivity-occurrence-of ?occmoldAndTrim
                 ?occShapePart moldAndTrim(?x))
              (leaf_occ ?occMoldAndTrim
                        ?occShapePart)))))
```

Expression 3: Pattern Example for Abstract Datatypes

3. Object-orientation and Components

   Operations on objects or components translate to PSL with the same patterns as abstract data types, with the added flexibility of varying execution by subtyping. Expression 3 covers inheritance, since it would apply for specialized materials types, for example, as shown in Expression 4. Expression 3 can also be loosened to enable subtype variation by removing the ROOT_OCC and LEAF_OCC clauses, as in Expression

5. Then metal and plastic can add occurrences to the inherited ones [7], sometimes called "method wrapping" [20]). However, a more common application is to replace the method entirely, for example, to operate on steel in a completely different way than metal in general. To achieve this, steel needs to be explicitly excluded from the axioms involving metal, and new axioms written for it, as shown in Expression 5. This has the disadvantage of changing existing axioms when adding new ones, but is explicit enough for automated reasoning.

```
(forall (?x)
   (implies (steel ?x)
            (metal ?x)))

(forall (?x)
   (implies (high-density-plastic ?x)
            (plastic ?x)))
```

Expression 4: Pattern Example for Subtyping

```
(forall (?x)
  (implies (and (metal ?x)
                (not (steel ?x)))
           (subactivity drillAndMill(?x) shapePart(?x))))

(forall (?x)
  (implies (steel ?x)
           (subactivity drillAndMillSteel(?x)
                        shapePart(?x))))

(forall (?occShapePart ?x)
   (implies
      (and (occurrence_of ?occShapePart shapePart(?x))
           (metal ?x)
           (not (steel ?x)))
      (exists (?occDrillAndMill)
         (subactivity-occurrence-of ?occDrillAndMill
            ?occShapePart drillAndMill(?x)))))

(forall (?occShapePart ?x)
   (implies
      (and (occurrence_of ?occShapePart shapePart(?x))
           (steel ?x))
      (exists (?occDrillAndMill)
         (subactivity-occurrence-of ?occDrillAndMill
            ?occShapePart drillAndMillSteel(?x)))))
```

Expression 5: Pattern Example for Object-orientation

4. Communicating Processes

Communication between processes translates to PSL as more general patterns of constraint on participation in which the communicated things (messages) are participants. One of the most basic restrictions on messages expressed in these patterns is that messages are received after they are sent. For example, Expression 6 conforms to this constraint for the order and account messages sent between a customer and vendor, due to the PSL relation EARLIER between the sending and receiving occurrences.[12] It also shows the flexibility of communicating processes to require a specific protocol in responding to requests. In this example, the account and order must be sent before filling the order, though it does not matter which is first. The customer process does not indicate exactly when the account is sent, only that it must be. Expression 6 is a cumbersome way to write messaging constraints between communicating processes, because the processes must be specified together, impairing modularity. This is addressed by the messaging extensions in Section 6.1.[13]

```
(activity customer)
(activity vendor)

(forall (?order ?account ?occVendor)
   (implies (and (order ?order)
                 (account ?account)
                 (occurrence_of ?occVendor vendor))
            (and (subactivity createOrder(?order) customer)
                 (subactivity sendOrder(?order) customer)
                 (subactivity sendAccount(?account)
                              customer)
                 (subactivity receiveOrder(?order) vendor)
                 (subactivity receiveAccount(?account)
                              vendor)
                 (subactivity fillOrder(?order) vendor))))
```

[12] The PSL relation EARLIER is used, rather than MIN_PRECEDES or NEXT_SUBOCC, because the constraint applies to occurrences that might not be under a larger superoccurrence.

[13] The messaging constraint above happens to also be satisified by Expression 2, where a piece of metal is sent between drilling and milling processes. The occurrence of drilling ends before the occurrence of milling, and the piece of metal participates in both. This can be taken as one extreme of communicating processes, where the processes are not concurrent. In general, there is a close relationship between the notions of input and output and messaging with ports, see Section 7.2.

```
(forall (?occCustomer ?occVendor ?order ?account)
   (implies
      (and (occurrence_of ?occCustomer
                          customer(?order ?account))
           (occurrence_of ?occVendor
                          vendor(?order ?account)))
      (exists (?sCreateOrder ?sSendOrder ?sSendAccount
               ?sReceiveOrder ?sReceiveAccount ?sFillOrder
               ?sReceiveAccount)
         (and
              (subactivity-occurrence-of ?sCreateOrder
                 ?occCustomer createOrder(?order))
              (subactivity-occurrence-of ?sSendOrder
                 ?occCustomer sendOrder(?order))
              (subactivity-occurrence-of ?sSendAccount
                 ?occCustomer sendAccount(?account))
              (min_precedes ?sCreateOrder ?sSendOrder
                            customer(?order ?account))

              (subactivity-occurrence-of ?sReceiveOrder
                 ?occVendor receiveOrder(?order))
              (subactivity-occurrence-of ?sReceiveAccount
                 ?occVendor receiveAccount(?account))
              (subactivity-occurrence-of ?sFillOrder
                 ?occVendor fillOrder(?order))
              (min_precedes ?sReceiveOrder ?sFillOrder
                            vendor(?order ?account))
              (min_precedes ?sReceiveAccount ?sFillOrder
                            vendor(?order ?account))

              (earlier ?sSendOrder ?sReceiveOrder)
              (earlier ?sSendAccount ?sReceiveAccount)))))
```

Expression 6: Pattern Example for Communicating Processes

5. Agents

   Like communicating processes, messages between agents also translate to PSL as patterns of constraint on participation, where messages are the participants. The same message constraints apply. Agents employ more sophisticated techniques to choose when and if to react to incoming messages. They can choose when to commit to required protocols, and even negotiate new protocols [24]. In theory, they can react to a message in any way, but in practice, they are restricted by their overall goals and intended purpose. See Section 6.2 for example of communicating information between agents.

6. PSL Extensions for Messages

This section gives extensions to PSL for messaging, as predefined instances of existing PSL concepts, in Section 6.1. Section 6.2 applies the extensions across the spectrum of message types in Section 4.

6.1 Message Extensions

This section defines PSL activities for messaging, and constraints on their occurrences, to reflect the usage patterns of the previous section. Following the intuitions about messages and generalizations for formalizing them in Section 1, a message is taken as sending a participant from one process to another. An activity is defined for sending any object to any process, as well as activities for transmission and receipt of messages. The constraints are divided into core axioms that apply to all messaging applications, shown in the expressions, and axioms that commonly apply, but are not universal, as discussed informally in the remainder of the section.

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 occurrence, such as when and by whom. Following this approach, activities defined for transmission and receipt of the message identify the message by the occurrence of the sending activity, and the sending occurrence identifies the object sent. [14]

- Expression 7 introduces messaging activity functions for sending, transmitting, and receiving messages, and the types of their parameters.[15] Messages can be sent to any occurrence, following the first generalization in Section 1.[16,17] Sending and receiving activities are atomic, which in PSL prevents states from holding during partially sent or received messages. Transmission is not atomic, because it is not restricted in how it is done, or how long it takes to complete.

[14] 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, which particularly affects forwarding, see Section 7. It also does not capture the concept of messaging, which inherently is in the sending, not in the entity sent. Identifying messages by the sending occurrences also enables techniques for controlling unwanted concurrency, see logical time stamps for message sends in [5].
[15] Expression 7 does not require that the messaging activities are different when the parameters of the activity functions are. This means an occurrence of a messaging activity could send multiple messages, or send and receive messages at the same time. This does not seem to affect the other axioms, but constraints against it could be added to simplify inference.
[16] It would not make much sense to send to occurrences of a primitive activity or even an atomic activity, including sending and receiving activities, but these are not ruled out at this basic level of specification.
[17] Types of messages can be defined as separate axioms, for example, ones that send the same entity, or occurrences of sending that play the same role over multiple occurrences of the larger process that constrains them.

```
(forall (?a ?x ?receiver)
  (implies (= ?a send-message(?x ?receiver))
           (and (activity ?a)
                (atomic ?a)
                (activity-participant ?x ?a)
                (activity_occurrence ?receiver))))

(forall (?a ?x ?receiver ?sSend)
  (implies (= ?a transmit-message(?x ?receiver ?sSend))
           (and (activity ?a)
                (activity-participant ?x ?a)
                (activity-occurrence ?receiver)
                (occurrence_of ?sSend
                   send_message(?x ?receiver)))))

(forall (?a ?x ?sSend)
  (implies (= ?a receive-message(?x ?sSend))
           (and (activity ?a)
                (atomic ?a)
                (activity-participant ?x ?a)
                (exists (?receiver)
                   (occurrence_of ?sSend
                      send_message(?x ?receiver))))))
```

Expression 7: Types for Messaging Activity Functions

The TRANSMIT-MESSAGE and RECEIVE-MESSAGE functions have sending occurrences as parameters to uniquely identify the sending of the message. This follows the general PSL principle of defining concepts starting with their most concrete manifestation, in this case the sending of an individual message. The TRANSMIT-MESSAGE and RECEIVE-MESSAGE functions could have been parameterized only by the sending occurrence, since the activity of the occurrence provides the sent participant and receiver, but the other parameters are included for convenience.[18]

- Expression 8 ensures that messages are transmitted, and received at most once. In PSL, this means once per branch of the occurrence tree, that is, no occurrence of transmission or receipt is EARLIER than another. Message uniqueness enables constraints to be written on the sending, transmission, and receiving occurrences, as in the next bullet.

[18] Parameterizing RECEIVE-MESSAGE by the sending occurrence does not affect support for anonymous messaging. Whatever the technique for ensuring message uniqueness, the receiver could query for the sender. Anonymity of the sender can only be achieved by restricting queries, for example, denying access to the subactivity occurrences of sending processes.

```
   (forall (?s1 ?s2 ?x ?receiver ?sSend)
     (implies (and (occurrence_of ?s1
                      transmit-message(?x ?receiver ?sSend))
                   (occurrence_of ?s2
                      transmit-message(?x ?receiver ?sSend)))
              (or (= ?s1 ?s2)
                  (not (or (earlier ?s1 ?s2)
                           (earlier ?s2 ?s1))))))
(forall (?s1 ?s2 ?x ?sSend)
  (implies (and (occurrence_of ?s1 receive-message(?x ?sSend))
                (occurrence_of ?s2 receive-message(?x ?sSend)))
           (or (= ?s1 ?s2)
               (not (or (earlier ?s1 ?s2)
                        (earlier ?s2 ?s1))))))
```

Expression 8: Transmission and Receipt Uniqueness

- Expression 9 requires that messages arrive where they are sent, are transmitted after they are sent, and received after they are transmitted.

```
(forall (?x ?receiver ?sSend ?sReceive)
   (implies
      (and (occurrence_of ?sSend send-message(?x ?receiver))
           (occurrence_of ?sReceive receive-message(?x ?sSend)))
      (subactivity_occurrence ?sReceive ?receiver)))

(forall (?x ?receiver ?sSend ?sTransmit ?sReceive)
   (implies
      (and (occurrence_of ?sSend send-message(?x ?receiver))
           (occurrence_of ?sTransmit
              transmit-message(?x ?receiver ?sSend))
           (occurrence_of ?sReceive receive-message(?x ?sSend)))
      (and (earlier ?sSend root-occ-fn(?sTransmit))
           (earlier leaf-occ-fn(?sTransmit) ?sReceive))))
```

Expression 9: Basic Messaging Constraints

The core axioms above provide for sending messages to specific occurrences, following PSL's general principle of representing processes at the most concrete level. In practice, however, the receiver will be a nondeterministic process, which means it covers multiple complex occurrences spanning multiple branches of the occurrence tree. These complex occurrences form a subtree of the occurrence tree, called the *activity tree*, see Section 5 of [7]. The sender usually will not want to constrain which branch of the receiving activity tree the message is sent to, in fact, the sender will want to send to every branch, so the message can be received regardless of nondeterminism in the receiver. Although sending a single message multiple times is counterintuitive from a typical process modeling viewpoint, it captures the intended execution semantics of those models, that the message is sent to the receiving process regardless of which particular execution trace the receiver happens to follow.

Expression 10 defines an additional convenience activity SEND-MESSAGE-ALL that sends a message to all complex occurrences in an activity tree. It identifies the tree by the activity that the receivers are occurrences of, and by the occurrence that happens just before the root of all the receiving occurrences, the "preroot." This identifies a particular execution of the receiving process, and accommodates receivers that do not share the same root (the activity tree is potentially a grove of trees). The SEND-MESSAGE-ALL activity is a concurrent aggregation of SEND-MESSAGE activities, targeting all the complex occurrences in the activity tree. This appears in PSL as an atomic SEND-MESSAGE-ALL being atomic and having SEND-MESSAGE subactivities (subactivities of atomic activities are always concurrent). To simplify the presentation, the rest of the paper only uses the core SEND-MESSAGE activity, but SEND-MESSAGE-ALL could be used instead.[19]

```
(forall (?x ?a ?receiver-preroot ?receiver-act)
   (implies (= ?a send-message-all(?x ?receiver-preroot
                                   ?receiver-act))
      (and (activity ?a)
           (atomic ?a)
           (activity_occurrence ?receiver-preroot)
           (atomic ?receiver-preroot)
           (activity ?receiver-act))))

(forall (?x ?a ?receiver-preroot ?receiver-act ?receiver-root
           ?receiver-occ)
   (implies (and (= ?a send-message-all(?x ?receiver-preroot
                                        ?receiver-act))
                 (successor ?receiver-preroot ?receiver-root)
                 (root ?receiver-root ?receiver-act)
                 (root_occ ?receiver-root ?receiver-occ)
                 (occurrence_of ?receiver-occ ?receiver-act)))
            (subactivity send-message(?x ?receiver-occ) ?a))
```

Expression 10: Convenience Activity for Sending to Nondeterministic Processes

A number of useful constraints may be added to the core axioms above as needed:

- Quality of service agreements might guarantee that messages are transmitted when sent, received when transmitted, and that these happen within certain time limits. The constraints are not included in the core axioms, to support flexibility in message handling, for example, to prevent transmission and receipt of unwanted messages, and avoid restricting quality of service agreements. However, messaging implementations cannot constrain away unwanted messages, as in PSL. They must examine each message and decide if it is wanted, then forward as appropriate, see Section 7. Formalization of these implementations will require that all messages sent are also received.

[19] Another approach is to reify the notion of activity tree and send message to those. This has the advantage of providing a PSL concept corresponding to the common notion of nondeterministic process, but requires defining many more relations to constrain it against existing PLS concepts.

- Some applications may wish to prevent a sent object from participating in the receiver before it is sent, or participating in the sender after it is sent. This is true for software messages assembled from other data once per sending, by primitive functions that do not give wider access to the message object. A looser form would allow participants to be sent and received by the same process, but still prevent participation between each send and receive of the same participant. This constraint is clearly true for physical objects sent between separate processes. These constraints are not included in the core axioms to generalize from particular implementations in software or physical systems.

- Related to the previous bullet would be to ensure messages do not participate in sibling or cousin occurrences during sending, transmitting, or receiving.

- 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.

For brevity, replies are not addressed, but these would be messages sent from the function to the sender of the original message. A new relation can be defined to identify where the reply should be sent, given the sending occurrence. This would usually be a superoccurrence of the sending occurrence, which could be the origin of a series of forwarded messages, see Section 7, but also could be a completely different process.[20] The sender can either wait for the reply before proceeding or not, which is synchronous and asynchronous messaging.

6.2 Applying Message Extensions

The extensions in Section 4 facilitate the specification of messaging for communicating processes and agents. These are on the side of the spectrum where messages are handled at the receiving entity by existing processes. At the other end of the spectrum, where messages always start processes at the receiver, and the reaction of the receiver is more predictable, the message extensions are more cumbersome, since their flexibility is not needed. They are provided here for completeness and comparison. For uniformity across the spectrum, the receiving entity is taken to be the requested operation, defined as a process.[21]

[20] Fault notification is a special case of reply that occurs in exceptional situations.

[21] An alternative categorization of message types based on state preservation, as described in footnote 5, would take the entity receiving the message as handling multiple requested operations, at least for abstract datatypes and higher. In this categorization, the message would contain the operation being requested. The expressions of this section would apply to each operation the object can receive.

1. Functions

   Expression 11 shows an example of a function defined as a process receiving messages to perform the operation it represents. The receiving process must be able to handle all values for the operation parameters. In PSL, this is done with subactivities for each parameter value. The activity DRILLANDMILL is unparameterized, but has subactivities for drilling and milling activities for all possible pieces of metal. It also has subactivities for receiving messages, where the thing being received is a piece of metal, one subactivity for each piece of metal. Not all of these are necessarily used in each occurrence of DRILLANDMILL, but the last statement in Expression 11 says which ones must be. It requires that receiving a piece of metal is followed by drilling and milling it, using NEXT_SUBOCC between occurrences to ensure the same reaction to every message. If no message is sent for a particular piece of metal, then it cannot be received, according to the axioms of Section 6.1.

```
(activity drillAndMill)

(forall (?m)
   (implies (metal ?m)
            (and (subactivity drill(?m) drillAndMill)
                 (subactivity mill(?m) drillAndMill))))

(forall (?x ?ra ?sSend)
   (implies (and (metal ?x)
                 (?ra = receive-message(?x ?sSend)))
            (subactivity ?ra drillAndMill)))

(forall (?occDrillAndMill ?x ?sReceive ?sSend)
   (implies (and (occurrence_of ?occDrillAndMill
                                drillAndMill)
                 (metal ?x)
                 (subactivity-occurrence-of
                    ?sReceive ?occDrillAndMill
                    receive-message(?x ?sSend)))
            (exists (?sDrill ?sMill)
               (and (subactivity-occurrence-of ?sDrill
                       ?occDrillAndMill drill(?x))
                    (subactivity-occurrence-of ?sMill
                       ?occDrillAndMill mill(?x))
                    (next_subocc ?sReceive ?sDrill)
                    (next_subocc ?sDrill ?sMill
                                 drillAndMill)))))
```

Expression 11: Function Message Example

2. Abstract datatypes

Abstract data types receive messages in a similar way to functions, but the reactions are based on the type of thing operated on, as shown in Expression 12. The SHAPEPART activity has subactivities for receiving pieces of metal and plastic, and for sending drill and mill messages, or mold and trim messages, in response.[22] Then the last two statements of Expression 12 require receiving a shape part message to be followed immediately by sending a drilling and milling message, or molding and trimming messages, depending on whether the part is metal or plastic.

```
(activity shapePart)

(forall (?x ?occDrillAndMill)
   (implies (and (metal ?x)
                 (occurrence_of ?occDrillAndMill
                                drillAndMill))
            (subactivity send-message(?x ?occDrillAndMill)
                         shapePart)))

(forall (?x ?occMoldAndTrim)
   (implies (and (plastic ?x)
                 (occurrence_of ?occMoldAndTrim moldAndTrim))
            (subactivity send-message(?x ?occMoldAndTrim)
                         shapePart)))

(forall (?x ?ra ?sSend)
   (implies (and (metal ?x)
                 (= ?ra receive-message(?x ?sSend)))
            (subactivity ?ra shapePart)))

(forall (?x ?ra ?sSend)
   (implies (and (plastic ?x)
                 (= ?ra receive-message(?x ?sSend)))
            (subactivity ?ra shapePart)))

(forall (?occShapePart ?sReceive ?x ?sSend)
   (implies (and (occurrence_of ?occShapePart shapePart)
                 (subactivity-occurrence-of
                    ?sReceive ?occShapePart
                    receive-message(?x ?sSend))
                 (metal ?x))
            (exists (?occDrillAndMill ?sSendDrillAndMill)
               (and (occurrence_of ?occDrillAndMill
                                   drillAndMill)
                    (subactivity-occurrence-of
                       ?sSendDrillAndMill ?occShapePart
                       send-message(?x ?occDrillAndMill))
                    (next_subocc ?sReceive
                                 ?sSendDrillAndMill)))))
```

[22] See footnote 10.

```
(forall (?occShapePart ?sReceive ?x ?sSend)
   (implies (and (occurrence_of ?occShapePart shapePart)
                 (subactivity-occurrence-of
                    ?sReceive ?occShapePart
                    receive-message(?x ?sSend))
                 (plastic ?x))
            (exists (?occMoldAndTrim ?sSendMoldAndTrim)
               (and (occurrence_of ?occMoldAndTrim moldAndTrim)
                    (subactivity-occurrence-of
                       ?sSendMoldAndTrim ?occShapePart
                       send-message(?x ?occMoldAndTrim))
                    (next_subocc ?sReceive ?sSendMoldAndTrim)))))
```

Expression 12: Abstract Datatype Message Example

3. Object-orientation and Components

   Objects and components react to messages in a similar way to abstract datatypes, with the added flexibility of varying execution by subtyping. Expression 12 covers inheritance, since it would apply for specialized materials types, for example, as shown in Expression 4 in Section 5. Expression 12 can also be loosened to enable subtype variation by using MIN_PRECEDES instead of NEXT_SUBOCC, as in Expression 13. Then steel and high-density plastic can add occurrences to the inherited ones [7], sometimes called "method wrapping" [20]). However, a more common application is to replace the method entirely, for example, to operate on steel in a completely different way than metal in general. To achieve this, steel needs to be explicitly excluded from the axioms involving metal, and new axioms written for it, as shown in Expression 13. This has the disadvantage of changing existing axioms when adding new ones, but is explicit enough for automated reasoning.[23]

```
(forall (?occShapePart ?sReceive ?x ?sSend)
   (implies (and (occurrence_of ?occShapePart shapePart)
                 (subactivity-occurrence-of
                    ?sReceive ?occShapePart
                    receive-message(?x ?sSend))
                 (metal ?x)
                 (not (steel ?x)))
            (exists (?occDrillAndMill ?sSendDrillAndMill)
               (and (occurrence_of ?occDrillAndMill)
                    (subactivity-occurrence-of
                       ?sSendDrillAndMill ?occShapePart
                       send-message(?x ?occDrillAndMill))
                    (min_precedes ?sReceive
                       ?sSendDrillAndMill shapePart)))))
```

[23] Expression 13 will not operate on the same piece of metal multiple times, even if it is sent multiple times, due to the use of MIN_PRECEDES instead of NEXT_SUBOCC. If it is supposed to, the statement could be constrained to match each message send with a single receipt, using a relation between message receipts and sending occurrences, or more strongly, to require sending to occur before receiving another.

```
(forall (?occShapePart ?sReceive ?x ?sSend)
   (implies (and (occurrence_of ?occShapePart shapePart)
                 (subactivity-occurrence-of
                    ?sReceive ?occShapePart
                    receive-message(?x ?sSend))
                 (steel ?x))
            (exists (?occDrillAndMillSteel
                     ?sSendDrillAndMillSteel)
               (and (occurrence_of ?occDrillAndMillSteel
                                   drillAndMillSteel)
                    (subactivity-occurrence-of
                       ?sSendDrillAndMillSteel ?occShapePart
                       send-message(?x ?occDrillAndMillSteel))
                    (min_precedes ?sReceive
                                  ?sSendDrillAndMillSteel
                                  shapePart)))))
```

Expression 13: Object-oriented Message Example

4. Communicating Processes

   The messaging relations of Section 6.1 facilitate specification of communicating processes, by separating the constraints on each process, as shown in Expression 14. The customer process 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 in Expression 6. In addition, the messaging constraints of Section 6.1 are formally enforceable, rather than just usage patterns. For brevity, the expression omits correlation information the receiver uses to link orders to the accounts.

```
(activity customer)
(activity vendor)

(forall (?order ?account ?occVendor)
   (implies (and (order ?order)
                 (account ?account)
                 (occurrence_of ?occVendor vendor))
            (and (subactivity createOrder(?order) customer)
                 (subactivity
                    send-message(?order ?occVendor)
                    customer)
                 (subactivity
                    send-message(?account ?occVendor)
                    customer))))
```

```
(forall (?occCustomer ?order ?account)
  (implies
     (occurrence_of ?occCustomer customer)
     (exists (?sCreateOrder ?sSendOrder ?sSendAccount
              ?occVendor)
        (and (subactivity-occurrence-of ?sCreateOrder
                ?occCustomer createOrder(?order))
             (subactivity-occurrence-of
                ?sSendOrder ?occCustomer
                send-message(?order ?occVendor))
             (subactivity-occurrence-of
                ?sSendAccount ?occCustomer
                send-message(?account ?occVendor))
             (min_precedes ?sCreateOrder ?sSendOrder
                           customer)))))

(forall (?order ?account ?sSend)
   (implies (and (order ?order)
                 (account ?account))
            (and (subactivity
                    receive-message(?order ?sSend) vendor)
                 (subactivity
                    receive-message(?account ?sSend)
                    vendor))))

(forall (?occVendor ?order ?account ?sReceiveOrder
         ?sReceiveAccount ?sSend)
  (implies
     (and (occurrence_of ?occVendor
                         vendor(?order ?account))
          (subactivity-occurrence-of ?sReceiveOrder
             ?occVendor receive-message(?order ?sSend))
          (subactivity-occurrence-of ?sReceiveAccount
             ?occVendor receive-message(?account ?sSend)))
     (exists (?sFillOrder)
        (and (subactivity-occurrence-of ?sFillOrder
                ?occVendor fillOrder(?order))
             (min_precedes ?sReceiveOrder ?sFillOrder
                           vendor)
             (min_precedes ?sReceiveAccount ?sFillOrder
                           vendor)))))
```

Expression 14: Communicating Process Message Example

5. Agents

   The messaging relations of Section 6.1 have the same benefits for agents as for communicating processes, to modularize the specifications of the agents. For agents, there are no restrictions on how they react to messages, except ones the agents commit to. Sophisticated forms of agents may have "beliefs" and "knowledge," they use to constrain their processes. Expression 14 is an example of this, where the

vendor must be notified of which account to bill before processing the order, which in a sense means the vendor is "told" which account to use. A more explicit approach to knowledge-constrained processes is to employ PSL states, which are objects representing states of the world. Since states are objects, they can be participants, and participation of a state in a process can be interpreted as the process "knowing" or at least "believing" that the state actually holds in the world. For example, Expression 15 shows a process for a bank guard that can open a safe only after receiving a message giving the combination of the safe. The fact that a particular safe has a certain combination is a state of the world. The last constraint in Expression 15 requires that the guard receive the proper state before opening a safe.

```
(activity guard)

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

(forall (?f ?x ?y)
   (implies (= ?f safe-has-combination(?x ?y))
            (and (state ?f)
                 (safe ?x)
                 (combination ?y))))
(forall (?occGuard ?sOpen ?x ?y)
   (implies (and (occurrence_of ?occGuard guard)
                 (subactivity-occurrence-of ?sOpen ?occGuard
                    openSafe(?x ?y)))
            (exists (?sReceive ?f ?sSend)
               (and (= ?sReceive
                       receive-message(?f ?sSend))
                    (= ?f safe-has-combination(?x ?y))
                    (min_precedes ?sReceive ?sOpen
                                  guard)))))
```

Expression 15: Agent Message Example

7. Composable Process Specifications with Messages

The messaging approaches defined so far require the sender of a message to identify a process to receive it. This is a severe limitation on the composability of process specifications, since they include direct reference to specific other processes, rather than any process that might be able to receive the message and react appropriately. For example, a customer is required to identify a particular vendor to send an order to, as in Expression 14, when it may be more effective for the customer to send the message through a broker or other arrangement that determines the vendor.

Modern approaches to component-based and service-oriented architecture address this by introducing the concept of ports on components through which messages are sent and forwarded [1][3][25]. In these techniques, a component sends messages to its own ports, where they are forwarded to other components. Channels between ports and components determine where the message is sent. This way components can send messages to other components without being constrained ahead of time to exactly which ones. For example, a customer sends a message to one of its own ports for purchasing a computer, which may be forwarded to a vendor for that type of computer. The determination of where to forward a message can be, for example:

- a constant, as in some of the patterns so far.
- a channel with a receiver on the other end that can vary from time to time.
- a publish and subscribe mechanism where receivers sign up for receiving messages.
- an arbitrarily complex process for determining the receiver.

These techniques significantly improve composability compared to messages sent directly between components. Even the simplest ports with constant receivers localize the determination of the receiver to a particular subprocess, rather than having it potentially anywhere in the component specification. When used with channels, ports leave the determination to how components are "wired" together into larger components, rather than in the component itself.[24]

Section 7.1 gives a formalization of ports and channels. Section 7.2 compares this to formalization of inputs and outputs from earlier work [10] and proposes improvements.

7.1 Ports

Following the presentation so far, this section formalizes components as processes and ports as a special kind of subprocess that forwards messages. A component process sends messages to its port subprocesses, which forward them to other component processes in whatever way they are specified to. As in the earlier formalizations, this can be done as a pattern of using existing PSL, a pattern of using the messaging axioms of Section 6.1, or new relations defined to reflect these patterns. This section will use the last two of these approaches.

Expression 16 and Expression 17 show how the customer axioms in Expression 14 would change with the introduction of ports. Expression 16 establishes subactivities for CUSTOMER and PURCHASEPORT, in particular, that customers send messages to purchase ports, purchase ports receive them, and send messages to vendors.[25] Expression 17

[24] For additional flexibility, the style of determiniation can change over the life of the component or port.
[25] These can be derived from the constraints on SUBACTIVITY_OCCURRENCE in Expression 17, but are stated for clarity.

constrains exactly which customers send to which purchase ports, and which purchase ports forward to which vendors. It localizes identification of the receiving vendor to occurrences of PURCHASEPORT, which can be defined independently of the rest of the customer specification, rather than allowing the vendor to be identified anywhere in the customer process. The relation RECEIVINGVENDOR stands in for whatever technique the port uses to determine the vendor. As in Expression 14, correlation of order and account at the receiver is omitted for brevity.

```
(activity customer)
(activity purchasePort)
(subactivity purchasePort customer)

(forall (?order ?account ?occPurchasePort)
   (implies (and (order ?order)
                 (account ?account)
                 (occurrence_of ?occPurchasePort
                                purchasePort)))
            (and (subactivity createOrder(?order) customer)
                 (subactivity
                    send-message(?order ?occPurchasePort)
                    customer)
                 (subactivity
                    send-message(?account ?occPurchasePort)
                    customer)))

(forall (?order ?account ?occCustomer ?occVendor
         ?occPurchasePort)
   (implies (and (order ?order)
                 (account ?account)
                 (occurrence_of ?occVendor vendor)
                 (occurrence_of ?occCustomer customer)
                 (occurrence_of ?occPurchasePort purchasePort)
                 (subactivity-occurrence-of ?sSend
                                            ?occCustomer
                    send-message(?x ?occPurchasePort)))
            (and (subactivity receive-message(?order ?sSend)
                              purchasePort)
                 (subactivity receive-message(?account ?sSend)
                              purchasePort)
                 (subactivity send-message(?order ?occVendor)
                              purchasePort)
                 (subactivity
                    send-message(?account ?occVendor)
                    purchasePort)))))
```

Expression 16: Pattern Example of Port Subactivities

```
(forall (?occCustomer ?occPurchasePort ?x ?Send ?sReceive
         ?occVendor)
   (implies
      (and (occurrence_of ?occCustomer customer)
           (subactivity-occurrence-of ?occPurchasePort
              ?occCustomer purchasePort)
           (or (order ?x)
               (account ?x))
           (subactivity-occurrence-of ?sSend ?occCustomer
              send-message(?x ?occPurchasePort))
           (subactivity-occurrence-of
              ?sReceive
              ?occPurchasePort
              receive-message(?x ?sSend))
           (receivingVendor ?x ?occVendor))
      (exists (?sForwardMessage)
         (and (subactivity-occurrence-of ?sForwardMessage
                 ?occPurchasePort
                 send-message(?x ?occVendor))
              (min_precedes ?sReceive ?sForwardMessage)))))

(forall (?occCustomer ?order ?account)
  (implies
     (occurrence_of ?occCustomer customer)
     (exists (?occPurchasePort ?sCreateOrder ?sSendOrder
              ?sSendAccount ?occVendor)
        (and (subactivity-occurrence-of ?occPurchasePort
                ?occCustomer purchasePort)
             (subactivity-occurrence-of ?sCreateOrder
                ?occCustomer createOrder(?order))
             (subactivity-occurrence-of ?sSendOrder
                                        ?occCustomer
                send-message(?order ?occPurchasePort))
             (subactivity-occurrence-of
                ?sSendAccount ?occCustomer
                send-message(?account ?occPurchasePort))
             (min_precedes ?sCreateOrder ?sSendOrder)))))
```

Expression 17: Pattern Example of Message Forwarding with Ports

The usage pattern in Expression 16 and Expression 17 can be formalized as a PSL activity, as shown in Expression 18 through Expression 20. These define a port process that forwards messages to other processes based on predefined channels.[26] Expression 18 defines channel ports as processes able to send and receive any message. Expression 19 defines relations PORT-CHANNEL-1 and PORT-CHANNEL-2 between port processes and other processes that send and receive messages through the port. They require that messages sent and received by the port go to and come from processes on the other end of channels. The channel relations divide channels into two "sides." Expression 20

[26] This is the semantics of ports in [1], where the channels are called *connectors*.

requires messages arriving on one side to be forwarded along the other.[27] Channel ports could be further constrained to match each forward with a single receipt, using a relation between message receipts at the port and the forwarding occurrences that requires exactly one message receipt per forwarding occurrence, or more strongly, to finish all forwarding for each receipt before receiving another message.[28] These additional constraints would not apply, for example, to ports that handle bids coming in along multiple channels. The port would choose which bid to forward based on lowest price or other criteria.

```
(activity channelPort)

(forall (?x ?a ?receiver ?sSend)
   (implies (or (= ?a send-message(?x ?receiver))
                (= ?a receive-message(?x ?Send)))
            (subactivity ?a channelPort)))
```

Expression 18: Channel Port Activity

```
(forall (?cport ?p)
   (implies (or (port-channel-1 ?cp ?p)
                (port-channel-2 ?cp ?p))
            (and (occurrence_of ?cp channelPort)
                 (occurrence ?p))))

(forall (?occChannelPort ?x ?receiver)
   (implies (and (occurrence_of ?occChannelPort channelPort)
                 (subactivity-occurrence-of
                    ?sSend ?occChannelPort
                    send-message(?x ?receiver)))
            (or (port-channel-1 ?occChannelPort ?receiver)
                (port-channel-2 ?occChannelPort ?receiver))))

(forall (?occChannelPort ?sReceive ?x ?sSend)
   (implies (and (occurrence_of ?occChannelPort channelPort)
                 (subactivity-occurrence-of
                    ?sReceive ?occChannelPort
                    receive-message(?x ?sSend)))
            (or (exists (?sender)
                        (and (port-channel-1 ?occChannelPort
                                             ?sender)
                             (subactivity_occurrence
                                ?sSend ?sender)))
                (exists (?sender)
                        (and (port-channel-2 ?occChannelPort
                                             ?sender)
                             (subactivity_occurrence
                                ?sSend ?sender))))))
```

Expression 19: Channel Port Relations

[27] The axioms allow for ports on both ends of the channel, as is typically the case in UML, but do not require it.

[28] Ports can also be constrained to not have the same process on both sides, so messages cannot "bounce" back to the sender. Expression 20 will forward regardless of the subactivity occurrence relations between the processes at the other end of the channels.

```
(forall (?occChannelPort ?sReceive ?x ?sSend ?sender
         ?forwardTo)
   (implies (and (occurrence_of ?occChannelPort channelPort)
                 (subactivity-occurrence-of
                    ?sReceive ?occChannelPort
                    receive-message(?x ?sSend))
                 (subactivity_occurrence ?sSend ?sender)
                 (or (and (port-channel-1 ?occChannelPort
                                          ?sender)
                          (port-channel-2 ?occChannelPort
                                          ?forwardTo))
                     (and (port-channel-2 ?occChannelPort
                                          ?sender)
                          (port-channel-1 ?occChannelPort
                                          ?forwardTo))))
            (exists (?sForward)
                 (and (subactivity-occurrence-of ?sForward
                         ?occChannelPort
                         send-message(?x ?forwardTo))
                      (min_precedes ?sReceive ?sForward
                                    channelPort)))))
```

Expression 20: Channel Port Forwarding

Expression 21 replaces the port axioms in Expression 17 with the channel port relations. It establishes a channel port on a customer with a vendor on the other end of the channel. Additional constraints on the port could be defined to restrict the messages that go through the port. For example, the port could be limited to accepting order and account messages from the customer. If Expression 9 in Section 6.1 was extended to guarantee message delivery, a theorem prover could detect if a process specification requires the wrong messages to be sent to the port.

```
(forall (?occCustomer)
   (implies (occurrence_oc ?occCustomer customer)
            (exists (?occPurchasePort ?occVendor)
               (and (subactivity-occurrence-of
                       ?occPurchasePort?occCustomer
                       channelPort)
                    (port-channel-1 ?occChannelPort
                                    ?occCustomer)
                    (port-channel-2 ?occChannelPort
                                    ?occVendor)))))
```

Expression 21: Channel Port Example

Some applications require channels to be created and destroyed dynamically. For example, channels from a customer to bidding vendors could be added or removed over

time. The channel relations could be extended with a time parameter to support this, representing when a channel exists. Dynamic channels are useful in publish and subscribe architectures. A "publishing" process accepts messages and forwards them to whichever processes happen to "subscribe" to the publisher. The interested processes subscribe by establishing channels to the publisher, possibly on ports of the publisher for the various kinds of messages. Channels can be removed to cancel subscriptions.

7.2 Inputs and Outputs

Ports and channels achieve for messaging what inputs and outputs do for process specifications in common modeling and programming languages. Ports enable process specifications to include messages with limited constraint on which processes will communicate, as shown in Section 7.1. Inputs and outputs are analogous to ports in being an "interface" between a process specification and other process specifications that use it. Neither side of the interface needs to be constrained by the other, only by the interface between them.

The affinity between ports and inputs and outputs leads us to compare the formalization in this paper to the one previously given for inputs and outputs [10]. In particular:

- Any object can be sent in a message, taken as input, or provided as ouput, and the object is a participant because of this, in Expressions 2 and 4 of [10].

- Sending and receiving messages are formalized as occurrences of predefined sending and receiving activities, whereas taking inputs or providing of outputs are formalized by input and output states holding on occurrences, in Expressions 7 through 12 of [10].

- Ports are formalized as subprocesses that forward messages, whereas inputs and outputs are formalized as new relations between participants and occurrences in Expressions 2 through 4 of [10].

- Channels are analogous to flow relations, Expression 13 through 16 of [10]. They both constrain how participants are transferred between processes without including those constraints in the specification of the processes. Both are formalized as channels and flow relations. However, channels constrain message sending and receipt occurrences, whereas flow relations constrain the new relations for inputs and outputs and their corresponding states.

The comparison above suggests the approach of this paper in formalizing messages, ports, and channels is more modular and introduces fewer new relations than the formalization of inputs and outputs in [10]. This paper uses predefined activities that address only messaging, whereas the input and output formalization introduces new relations that require the involvement of states. The connection of inputs and outputs to states is important, but can be handled separately from inputs and outputs if the approach

of the message formalization is used. Following Expression 7, Expression 22 introduces activities for taking input and posting output. Then the OCCURRENCE-INPUT and OCCURRENCE-OUTPUT relations in Expressions 2 and 4 of [10] can be redefined as shown in Expression 23.

```
(forall (?a ?x ?occ)
   (implies (or (= ?a (accept-input ?x ?occ))
                (= ?a (post-output ?x ?occ)))
            (and (activity ?a)
                 (atomic ?a)
                 (activity-participant ?x ?a)
                 (activity_occurrence ?occ))))
```

Expression 22: Types for Input and Output Activities

```
(forall (?x ?s)
   (iff (occurrence-input ?x ?s)
        (exists (?subocc)
           (and (subactivity_occurrence ?subocc ?s)
                (occurrence_of ?subocc (accept-input ?x))))))

(forall (?x ?s)
   (iff (occurrence-output ?x ?s)
        (exists (?subocc)
           (and (subactivity_occurrence ?subocc ?s)
                (occurrence_of ?subocc (post-output ?x))))))
```

Expression 23: Occurrence Input and Outputs using Input and Output Activities

Process designers can use the redefined OCCURRENCE-INPUT and OCCURRENCE-OUTPUT relations if they do not want to commit to exactly when inputs are accepted and outputs posted, or they can use occurrences of the ACCEPT-INPUT and POST-OUTPUT relations to make that explicit. For example, Expression 24 is a version of Expression 2 using the predefined input and output activities. The first two statements give inputs for occurrences of the DRILL and MILL activities, without specifying exactly when during those processes the input is actually accepted. The third statement uses occurrences of ACCEPT-INPUT and POST-OUTPUT to indicate when input is accepted and output posted during occurrences of DRILLANDMILL. Combined with Expression 23 this also gives the more general OCCURRENCE-INPUT and OCCURRENCE-OUTPUT constraints.

```
(forall (?occDrill ?m)
   (implies
      (occurrence_of ?occDrill drill(?m))
      (occurrence-input ?m ?occDrill)))

(forall (?occMill ?m)
   (implies
      (occurrence_of ?occMill mill(?m))
      (occurrence-input ?m ?occMill)))

(forall (?occDrillAndMill ?m)
   (implies
      (occurrence_of ?occDrillAndMill drillAndMill(?m))
      (exists (?sAccept ?sDrill ?sMill ?sPost)
         (and (subactivity-occurrence-of ?sAccept
                ?occDrillAndMill accept-input(?m))
              (subactivity-occurrence-of
                ?sDrill ?occDrillAndMill drill(?m))
              (subactivity-occurrence-of
                ?sMill ?occDrillAndMill mill(?m))
              (subactivity-occurrence-of
                ?sPost ?occDrillAndMill post-input(?m))
              (min_precedes ?sAccept root-occ-fn(?sDrill)
                            drillAndMill)
              (min_precedes leaf-occ-fn(?SDrill)
                            root-occ-fn(?sMill) drillAndMill)
              (min_precedes leaf-occ-fn(?sMill) ?sPost
                            drillAndMill)))))
```

Expression 24: Input and Output Activity Example

Typical process models accept input when a process starts and post outputs when it finishes, but many applications require more flexibility. For example, a milling machine will take in and put out oil as it is running. Expression 25 loosens the constraints in Expressions 8 and 9 of [10] by relating input and output states to the taking of inputs and posting of outputs, rather than the larger occurrence in which these happen.[29] Activities that only take inputs when they start and post outputs only when they finish can constrain the suboccurrences of TAKE-INPUT and POST-OUTPUT to reflect this.

[29] Some applications may wish to prevent an input from being accepted multiple times under the same occurrence, or at least not without an intervening output, and similarly for outputs. This is required by Expressions 10 and 11 of [10].

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

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

Expression 25: Input and Output States using Input and Output Activities

With the OCCURRENCE-INPUT and OCCURRENCE-OUTPUT relations defined independently of input and output states, the flow constraints in Expressions 14 of [10] and its refinement in Expression 31 can be simplified to the ones in Expression 26 below by removing the clauses referring to state. The flow constraints between occurrences and suboccurrences in Expressions 32 and 33 of [10] are simplified in the same way in Expression 26. The constraints on input and output states can be defined separately using the removed clauses. This modularizes the axioms so input and output can be used with states or not, as needed.[30]

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

(forall (?x ?s1 ?s2 ?leaf1 ?root2)
   (implies (and (occurrence-flow ?x ?s1 ?s2)
                 (not (subactivity_occurrence ?s1 ?s2))
                 (not (subactivity_occurrence ?s2 ?s1))
                 (root_occ ?root2 ?s2)
                 (legal ?root2))
            (and (occurrence-output ?x ?s1)
                 (occurrence-input ?x ?s2)))))
```

Expression 26: Simplified Flow Constraints

[30] Another point of comparison between the formalization of ports in this paper and inputs and outputs in [10] is that the same participant can be sent out from a process through different ports, or come into it through different ports, whereas the same participant can only be output to one flow, or input from one flow. Formally, this is a relation between the occurrence and the participant being input or output. However, first order logic does not quantify over these, which is an advantage of the messaging approach. An alternative is to reify those relations as objects, and extending the input, output activities and relations, and flow relations, with these objects.

Another advantage of formalizing input and output with predefined activities as above is that it is closer to common intuitions. Inputs to a process are not determined by the process itself, they require suboccurrences that explicitly accept them from an external source. For example, repair and painting processes on beams in a factory might happen to choose to operate on the same beam at different times, but not because one 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. However, if one process identified the beam for the other to operate on, it would be considered a flow of output to input. One might try to formalize this by adding the constraint that participation must be sequential for the beam to be an input, that is, the beam does not participate in any other process in between repair and painting. However, the painting process may happen to independently identify the same beam as an immediately preceding repair, and we would not say the repair process accepted input from painting. The sequential participation might even accidentally happen all the time and deterministically, and there is still no output to input flow between repair and painting.[31]

The above argument reinforces the conclusion that formalization of common process modeling concepts on execution-centered languages such as PSL must be based on usage patterns. Notions such as "sending a message" or "providing inputs" are about the way execution constraints are written, rather than specific execution constraints that can be written once and for all. When a usage pattern is formalized, it can itself be used in larger patterns of patterns, as shown in ports and channels, Section 7. Usage patterns are metatheoretic from the point of view of a first-order execution language, like PSL, and out of their scope. The benefit of execution-centered languages such as PSL is that patterns can be formalized as predefined, reusable elements, or new relations, and constrained against unambiguous and well-grounded concepts in the execution language.

8. Future work

Many process modeling techniques depend on messaging and can be formalized on the basis provided in this paper. 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][22]. The entity accepting the message changes state according to the messages it receives, sending out other messages as it does. Following the approach of this paper, state machines could be translated as a pattern of using the messaging extensions, or as predefined relations on entities that define their state structure. This will be addressed in future work.

Constraints on participation in general are useful for other process modeling techniques. For example, applications of Petri nets often interpret places as processes, and tokens as participants in those processes [26]. Under this interpretation, transitions and arcs can be formalized as constraints on participation, and many Petri net properties formalized as constraints on participation. For example, safety in Petri nets refers to whether places can

[31] This is a refinement to the argument in the first bullet of Section 3 of [10].

have more than one token, and reachability refers to whether they can have any at all. Conflict refers to whether there is nondeterminism in transition firing, which can be formalized in PSL as branching in the activity tree. This will also be addressed in future work.

The semantic similarity between messaging and inputs and outputs can be drawn out further, towards a possible unification. One of the issues to address is the intuitions of input and output do not usually include subprocesses, channels, and forwarding. This is especially true in the common case of processes that only accept inputs when they start and post outputs when they finish. The occurrence of these kind of processes do not exist before inputs arrive, and or after outputs leave. The formalization of channel would need to be generalized so they can exist independently of the connected occurrences [27]. The unification could also generalize input and output axioms to cover nondeterministic processes, by supporting flows between activity trees as well as individual occurrences.

9. Conclusion

This paper provides a formalization of messaging in PSL. It identifies the common intuitions of messaging to support the interpretation of messages as patterns of constraints on process participation. Approaches to formalizing usage patterns of PSL are described, and some are used in the rest of the paper. In particular, reusable activities are defined for messaging, such as sending and receiving messages. These extensions support definition of constraints between occurrences of messaging activities, for example, that messages are only received after they are sent. This paper gives usage patterns and formalizations for a spectrum of message types, illustrating a range of variability in the reaction of the message receiver. The extensions are used to formalize message forwarding through ports and channels, a technique for defining composable processes. These suggest simpler alternatives to formalize inputs and outputs from earlier work, and reinforce the conclusion that the formalization of common process modeling concepts on execution-centered languages like PSL must be based on patterns of constraints.

10. References

[1] Object Management Group, "UML 2.0 Superstructure Specification," http://www.omg.org/cgi-bin/doc?ptc/04-10-02, October 2004.

[2] Kalev, D., ANSI/ISO C++ Professional Programmer's Handbook, Que, 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] Lamport, L., "Time, Clocks and the Ordering of Events in a Distributed System," Communications of the ACM, Vol. 21, No. 7, pp. 558-565, July 1978.

[6] Scherl, R., Levesque, H., "Knowledge, Action, and the Frame Problem," Artificial Intelligence, Vol. 144, No. 1-2, pp. 1-39, March 2003.

[7] Bock, C., Gruninger, M., PSL: A Semantic Domain for Flow Models," Software and Systems Modeling Journal, Vol. 4, No. 2, pp. 209 – 231, May 2005.

[8] ISO 18269, *Process Specification Language*, 2005

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

[10] Bock, C., Gruninger, M., "Inputs and Outputs in PSL," NISTIR 7152, August 2004.

[11] Gruninger, M., "Guide to the Ontology of the Process Specification Language," in Staab, S. (ed.) Handbook of Ontologies in Information Systems, Springer-Verlag, 2003.

[12] International Standards Organization, "Common Logic (CL) – A Framework for a Family of Logic-Based Languages," WG2, SC32, ISO/IEC JTC1, http://philebus.tamu.edu/cl, June 2005.

[13] Graham, P., ANSI Common LISP, Prentice Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1995.

[14] QVT Submission Team, "MOF 2.0 Query/View/Transformation Submission," doc.omg.org/ad/05-03-02, March 2005.

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

[16] Dijkstra, E.W., "Go To Statement Considered Harmful," Communications of the ACM, Vol. 11, No. 3, pp. 147-148, March 1968.

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

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

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

[20] Kiczales G., des Rivieres J., Bobrow D., The Art of the Metaobject Protocol, MIT Press, 1991.

[21] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns, Addison-Wesley, 1995.

[22] Sangiorgi, D., Walker, D., The Pi Calculus, Cambridge University Press, 2001.

[23] Odell, J., "Objects and Agents Compared," in Journal of Object Technology, Vol. 1, No. 1, pp. 41-53, May-June 2002.

[24] Van Dyke Parunak, H., Breuckner, S., Sauter, J. "A Preliminary Taxonomy of Multi-Agent Interaction," Agent-Oriented Software Engineering IV, Giorgini, P., Müller, J., Odell, J (eds.), Lecture Notes on Computer Science, Vol. 2935, Springer, Berlin, 2004.

[25] Object Management Group, "CORBA Components," http://doc.omg.org/formal/02-06-65, 2002.

[26] 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, Vol. 7, No. 4, pp. 515-527, August 1991.

[27] Bock, C., "UML 2 Composition Model," Journal of Object Technology, Vol. 3, No. 10, pp. 47-73, November/December 2004.