

The Process Specification Language (PSL) Theory and Applications

Michael Grüninger and Christopher Menzel

■ The PROCESS SPECIFICATION LANGUAGE (PSL) has been designed to facilitate correct and complete exchange of process information among manufacturing systems, such as scheduling, process modeling, process planning, production planning, simulation, project management, work flow, and business-process reengineering. We give an overview of the theories within the PSL ontology, discuss some of the design principles for the ontology, and finish with examples of process specifications that are based on the ontology.

As the use of information technology in manufacturing operations has matured, the need to integrate software applications has become increasingly important. However, interoperability among these manufacturing applications is hindered because the applications use different terminology and representations of the domain. These problems arise most acutely for systems that must manage the heterogeneity inherent in various domains and integrate models of different domains into coherent frameworks (figure 1). For example, such integration occurs in business-process reengineering, where enterprise models integrate processes, organizations, goals, and customers. Even when applications use the same terminology, they often associate different semantics with the terms. This clash over the meaning of the terms prevents the seamless exchange of information among the applications. Typically, point-to-point translation programs are written to enable communication from one specific application to another. However, as the number of applications has in-

creased and the information has become more complex, it has been more difficult for software developers to provide translators between every pair of applications that must cooperate. What is needed is some way of explicitly specifying the terminology of the applications in an unambiguous fashion.

The PROCESS SPECIFICATION LANGUAGE (PSL) (Menzel and Gruninger 2001; Schlenoff, Gruninger, and Ciocoiu 1999) has been designed to facilitate correct and complete exchange of process information among manufacturing systems. Included in these applications are scheduling, process modeling, process planning, production planning, simulation, project management, work flow, and business-process reengineering. We give an overview of the theories within the PSL ontology, discuss some of the design principles for the ontology, and finish with examples of process specifications that are based on the ontology.

Architecture of PSL

The PSL ontology is organized into PSL-CORE and a partially ordered set of extensions. All axioms are first-order sentences and are written in the KNOWLEDGE INTERCHANGE FORMAT (KIF).

There are two types of extensions within PSL: (1) core theories and (2) definitional extensions. *Core theories* introduce and axiomatize new relations and functions that are primitive. All terminology introduced in a definitional extension has conservative definitions using the terminology of the core theories. Thus, def-

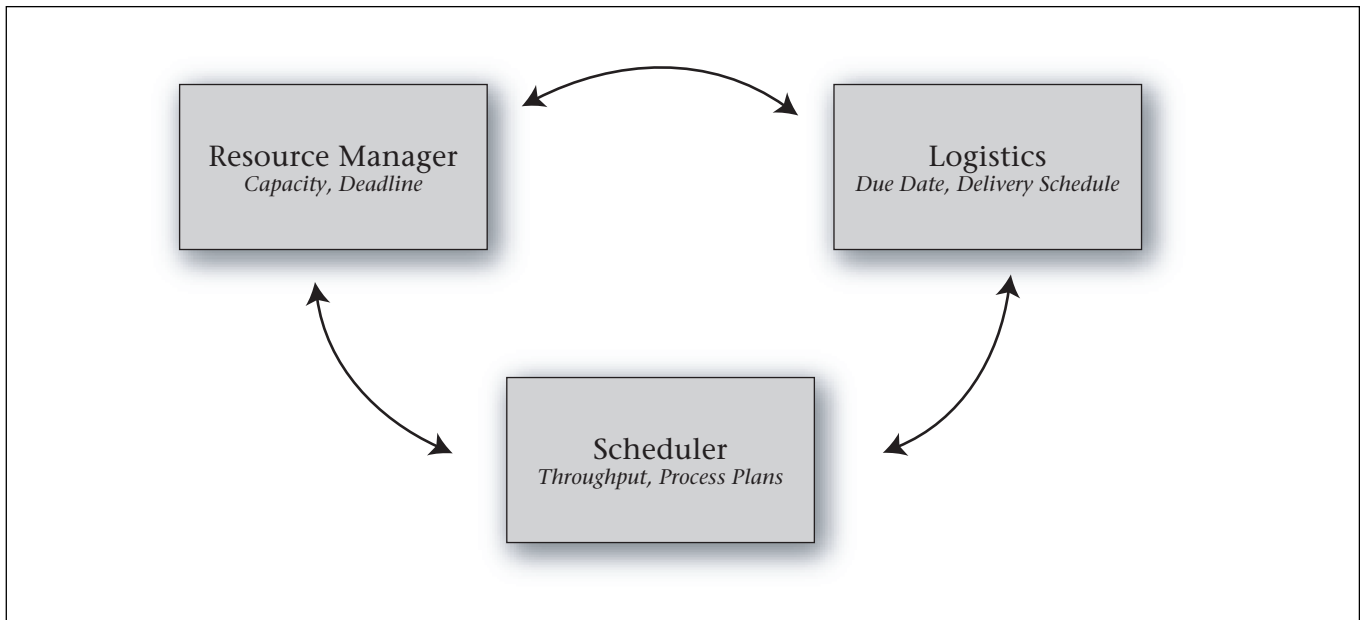


Figure 1. The Challenge of Interoperability.

Different departments within an enterprise often use their own terminology in referring to their business processes. The challenge of interoperability is to determine the relationships between the meanings of the terms used by these departments: Is the concept of “deadline” used by the resource manager semantically equivalent to the concept of “due date” used by logistics? Is the concept of “throughput” used by the scheduler semantically equivalent to the concept of “capacity” used by the resource manager?

initial extensions add no new expressive power to PSL-CORE.

Core Theories

All core theories within the ontology are consistent extensions of PSL-CORE, although not all extensions need be mutually consistent. Also, the core theories need not be conservative extensions of other core theories. The relationships among the core theories in the PSL ontology are depicted in figure 2.

PSL Core The purpose of PSL-CORE is to axiomatize a set of intuitive semantic primitives that is adequate for describing the fundamental concepts of manufacturing processes. Consequently, this characterization of basic processes makes few assumptions about their nature beyond what is needed for describing these processes, and the core is, therefore, rather weak in terms of logical expressiveness. Specifically, the core ontology consists of four disjoint classes: (1) activities, (2) activity occurrences, (3) time points, and (4) objects. *Activities* can have zero or more occurrences, *activity occurrences* begin and end at time points, and *time points* constitute a linearly ordered set with end points at infinity. *Objects* are simply those elements that are not activities, occurrences, or time points.

PSL-CORE is not strong enough to provide definitions of the many auxiliary notions that be-

come necessary to describe all intuitions about manufacturing processes. To supplement the concepts of PSL-CORE, the ontology includes a set of extensions that introduce new terminology. Any PSL extension provides the logical expressiveness to axiomatize intuitions involving concepts that are not explicitly specified in PSL-CORE. All extensions within PSL are consistent extensions of PSL-CORE and can be consistent extensions of other PSL extensions. However, not all extensions within PSL need be mutually consistent. Also, the core theories need not be conservative extensions of other core theories.

A particular set of theories is grouped together to form the outer core; this distinction is pragmatic because in practice, these theories have been necessary for axiomatizing all other concepts in the PSL ontology (figure 2).

Occurrence Trees An *occurrence tree* is the set of all discrete sequences of activity occurrences. They are isomorphic to substructures of the situation tree from situation calculus (McCarthy and Hayes 1969; Pinto 1994; Reiter 1991), the primary difference being that rather than a unique initial situation, each occurrence tree has a unique initial activity occurrence. As in the situation calculus, the *poss* relation is introduced to allow the statement of constraints on activity occurrences within the occurrence tree. Because the occurrence trees include sequences that modelers of a domain

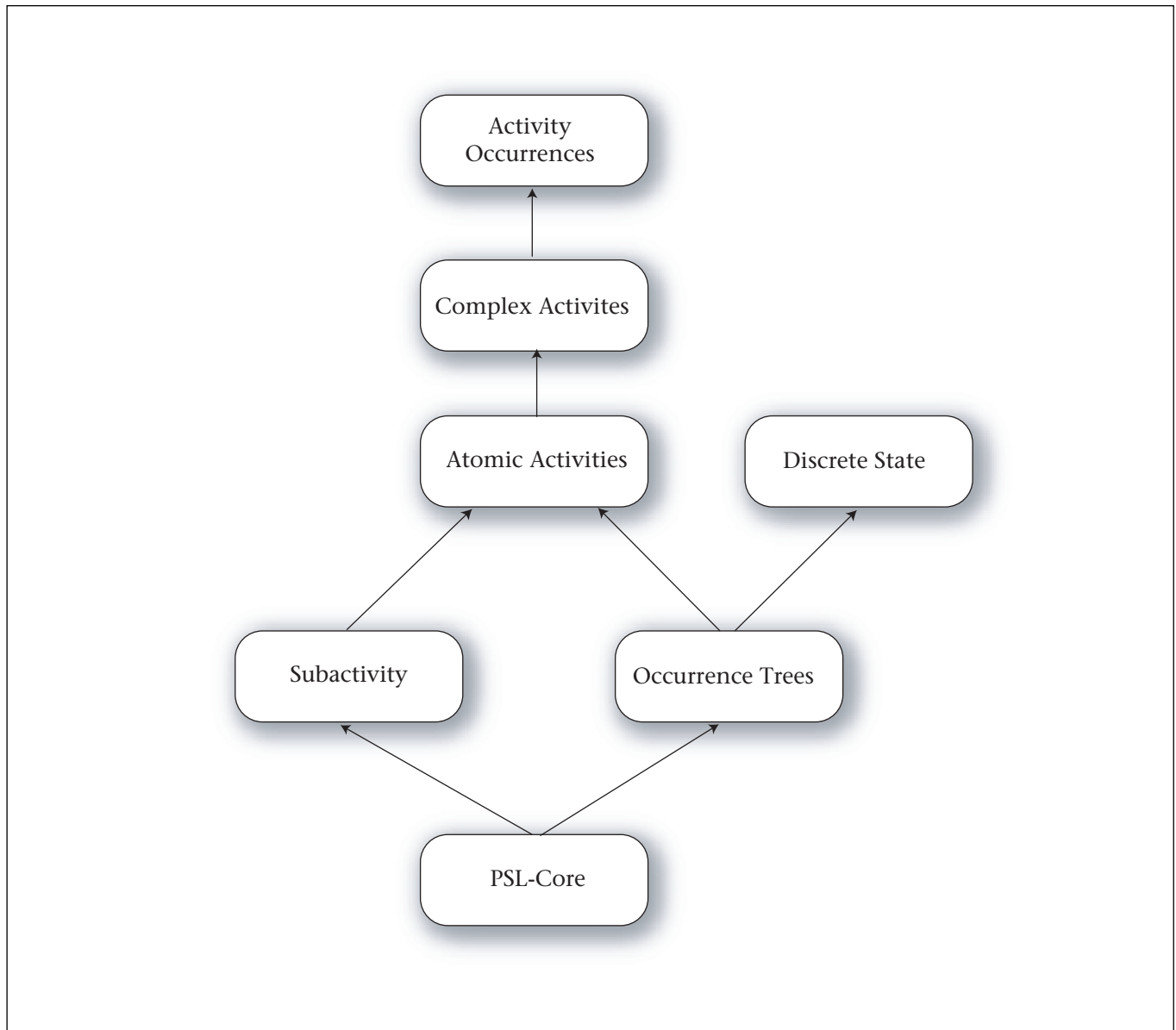


Figure 2. The Theories in the Outer Core of PSL.

will consider impossible, the poss relation “prunes” away branches from the occurrences tree that correspond to such impossible activity occurrences.

It should be noted that the occurrence tree is not the structure that represents the occurrences of subactivities of an activity. The occurrence tree does not represent a particular occurrence of an activity but, rather, all possible occurrences of all activities in the domain.

Discrete States The discrete-states core theory introduces the notion of *state* (*fluents*). Fluents are changed only by the occurrence of activities, and fluents do not change during the occurrence of primitive activities. In addition,

activities have *preconditions* (fluents that must hold before an occurrence) and *effects* (fluents that always hold after an occurrence).

Subactivities This core theory axiomatizes intuitions about *subactivities*. The only constraint imposed within this theory is that the subactivity relation is isomorphic to a discrete partial ordering. Other core theories impose additional constraints.

Atomic Activities The core theory of *atomic activities* axiomatizes intuitions about the concurrent aggregation of primitive activities. This concurrent aggregation is represented by the occurrence of concurrent activities rather than concurrent-activity occurrences.

Definitional Extensions	Core Theories	Example Concepts
Activity Extensions	Complex Activities	Deterministic/Nondeterministic Activities Concurrent Activities Partially Ordered Activities
Temporal and State Extensions	Complex Activities Discrete States	Preconditions Effects Conditional Activities Triggered Activities
Activity Ordering and Duration Extensions	Subactivity Occurrence Ordering Iterated Occurrence Ordering Duration	Complex Sequences and Branching Iterated Activities Duration-based Constraints
Resource Role Extensions	Resource Requirements	Reusable, Consumable, Renewable, Deteriorating Resources

Table 1. Definitional Extensions of PSL.

Complex Activities This core theory provides the foundation for representing and reasoning about *complex activities* and the relationship between occurrences of an activity and occurrences of its subactivities. Within models of the complex-activities theory, occurrences of complex activities correspond to subtrees of the occurrence tree. An activity can have subactivities that do not occur; the only constraint is that any subactivity occurrence must correspond to a subtree of the activity tree that characterizes the occurrence of the activity. Not every occurrence of a subactivity is a subactivity occurrence. There can be other external activities that occur during an occurrence of an activity. Different subactivities can occur on different branches of the activity tree so that different occurrences of an activity can have different subactivity occurrences.

Activity Occurrences The complex-activities theory only axiomatizes constraints on atomic subactivity occurrences. The *activity occurrences theory* generalizes these intuitions to arbitrary complex subactivities.

Additional Core Theories The remaining core theories in the PSL ontology include *subactivity occurrence ordering* (axiomatizing different partial orderings over subactivity occurrences), *iterated occurrence ordering* (axioms necessary for defining iterated activities), *duration* (augmenting PSL-CORE with a metric over the timeline), and *resource requirements* (which specify the conditions that must be satisfied by any object that is a resource for an activity).

Definitional Extensions

The *definitional extensions* are grouped into parts according to the core theories that are required for their definitions. Table 1 gives an overview of these groups together with example concepts that are defined in the extensions. The definitional extensions in a group contain definitions that are conservative with respect to the specified core theories; for example, all concepts in the temporal and state extensions have conservative definitions with respect to both the complex activities and discrete-states theories.

Design Principles

The organization of the PSL ontology and the properties of its extensions have been shaped by several design principles. In presenting these principles, we make a distinction between *hypotheses* (that constrain uses of the PSL ontology) and *criteria* (that specify properties of the PSL ontology itself).

Supporting Interoperability

Intuitively, two applications will be interoperable if they share the semantics of the terminology in their corresponding theories. Sharing semantics between applications is equivalent to sharing models of their theories; that is, the theories have isomorphic sets of models. However, applications do not explicitly share the models of their theories. Instead, they exchange sentences in such a way that the semantics of the terminology of these sentences is preserved.

We say that a theory T_A is *shareable* with a theory T_B if for any sentence Φ_A in the language of T_A , there exists an exchange that maps to a sentence Φ_B such that there is a one-to-one mapping between the set of models of T_A that satisfy Φ_A and the set of models of T_B that satisfy Φ_B . We say that a theory T_A is *interoperable* with a theory T_B if for any sentence Φ that is provable from T_A , there exists an exchange that maps Φ to a sentence that is provable from T_B . We make the following hypothesis to restrict our attention to domains in which shareability and interoperability are equivalent:

Interoperability Hypothesis

We are considering interoperability among complete first-order inference engines that exchange first-order sentences.

The soundness and completeness of first-order logic guarantees that the theorems of a deductive inference engine are exactly those sentences that are satisfied by all models and that any truth assignment given by a consistency checker is isomorphic to a model. If we move beyond the expressiveness of first-order logic, we lose completeness, so that for any deductive inference engine, there will be sentences that are entailed by a set of models but are not provable by that engine. We could therefore have two theories that are shareable but not interoperable.

Note that we are not imposing the requirement that the ontologies themselves be categorical or even complete. The two applications must simply share the same set of models (to isomorphism). Ambiguity does not arise from the existence of multiple models for an ontology; it arises because the two applications have nonisomorphic models—the ontology for application A has a model that is not isomorphic to any model for the ontology of application B .

The Ontological Stance

When building translators, we are faced with the additional challenge that almost no application has an explicitly axiomatized ontology. However, we can model a software application as if it were an inference system with an axiomatized ontology and use this ontology to predict the set of sentences that the inference system decides to be satisfiable. This is the *ontological stance*, which is analogous to the *intentional stance* (Dennett 1987), the strategy of interpreting the behavior of an entity by treating it as if it were a rational agent who performs activities in accordance with some set of intentional constraints.

In practice, the ontological stance requires the following assumption about the ontologies that are attributed to an application:

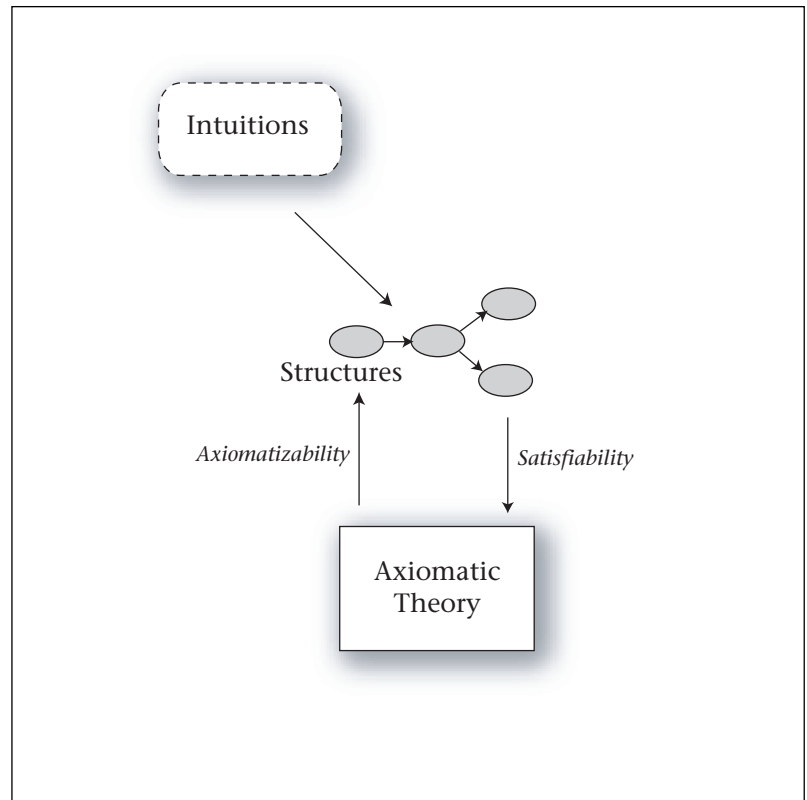


Figure 3. Methodology for the Evaluation of Axiomatic Theories.

Conformance Hypothesis

Every structure that is a model of the application ontology is isomorphic to a model of a foundational theory that is an extension of PSL-CORE.

Although this hypothesis is rather strong because it entails that all application ontologies are consistent with PSL-CORE, it also imposes conditions on the PSL ontology, which must be rich enough to axiomatize the application ontology.

Characterization of Models

Using the interoperability hypothesis, we impose the following condition on the core theories of the PSL ontology:

Definability Criterion

Classes of structures for core theories within the PSL ontology are axiomatized to elementary equivalence—the core theories are satisfied by any model in the class, and any model of the core theories is elementarily equivalent to a model in the class. Further, each class of structures is characterized to isomorphism.

The definability criterion can also be applied as a methodology for evaluating the axiomatization of an ontology (figure 3).

The first aspect of this approach is to identify the primary intuitions in some domain. Within *PSL*, for example, we have intuitions about concepts such as activity, activity occurrences, and time points. These intuitions also restrict the scope of the axiomatic theories, and they serve as informal requirements that get formally specified in the classes of structures and later axiomatized in the theory itself.

The objective of the second aspect of the methodology is to identify each concept with an element of some mathematical structure. In particular, given a nonlogical lexicon in some language, the specified structures are isomorphic to the extensions of the relations, functions, and constants denoted by the predicate symbols, function symbols, and constant symbols of the lexicon. The class of structures corresponding to the intuitions of the ontology will be defined either by specifying some class of algebraic or combinatoric structures or extending classes of structures defined for other theories within the ontology. Examples of structures include graphs, linear orderings, partial orderings, groups, fields, and vector spaces.

This relationship between the intuitions and the structures is, of course, informal, but we can consider the domain intuitions as providing a physical interpretation of the structures. In this sense, we can adopt an experimental or empirical approach to the evaluation of the class of intended structures in which we attempt to falsify these structures. If we can find some objects or behavior within the domain that do not correspond to an intended structure, then we have provided a counterexample to the class of structures. In response, we can either redefine the scope of the class of structures (that is, we do not include the behavior within the characterization of the structures), or we can modify the definition of the class of structures so that they capture the new behavior.

For example, physicists use various classes of differential equations to model different phenomena. However, they do not use ordinary linear differential equations to model heat diffusion, and they do not use second-order partial differential equations to model the kinematics of springs. If we want to model some phenomena using a class of differential equations, we can use the equations to predict behavior of the physical system; if the predictions are falsified by observations, then we have an incorrect set of equations. Similarly, in our case, we can use some class of structures to predict behavior or characterize states of affairs; if there is no physical scenario in the domain that corresponds to these behaviors or

states of affairs, then we intuitively have an incorrect set of structures.

Once we specify the class of structures, we can formally evaluate an axiomatic theory with respect to this specification. In particular, we want to prove two fundamental properties: (1) *satisfiability*, where every structure in the class is a model of the axiomatic theory, and (2) *axiomatizability*, where every model of the axiomatic theory is isomorphic to some structure in the class.

Strictly speaking, we only need to show that a model exists to demonstrate that a theory is satisfiable. However, in the axiomatization of domain theories, we need a complete characterization of the possible models. For example, because we are considering the domain of activities, occurrences, and time points to show that a theory is satisfiable, we need only specify an occurrence of an activity that, together with the axioms, is satisfied by some structure. The problem with this approach is that we run the risk of having demonstrated satisfiability only for some restricted class of activities. For example, a theory of activities that supports scheduling can be shown to be consistent by constructing a satisfying interpretation, but the interpretation might require that resources cannot be shared by multiple activities, or it might require all activities to be deterministic. Although such a model might be adequate for such activities, it would in no way be general enough for our purposes. We want to propose a comprehensive theory of activities, so we need to explicitly characterize the classes of activities, time points, objects, and other assumptions that are guaranteed to be satisfied by the specified structures.

The purpose of the axiomatizability theorem is to demonstrate that any unintended models of the theory do not work, that is, any models that are not specified in the class of structures. With the interoperability hypothesis, we do not need to restrict ourselves to elementary classes of structures when we are axiomatizing an ontology. Because the applications are equivalent to first-order inference engines, they cannot distinguish between structures that are elementarily equivalent. Thus, the unintended models are only those that are not elementarily equivalent to any model in the class of structures.

The Role of Definitional Extensions

The terminology within the definitional extensions intuitively corresponds to classes of activities and objects. Within the *PSL* ontology, the terminology arises from the classification of the models of the core theories with respect to

sets of invariants. *Invariants* are properties of models that are preserved by isomorphism. A set of invariants is complete for a class of structures if and only if it can be used to classify the structures to isomorphism. For example, a finite abelian group can be classified to isomorphism by the subgroups whose orders are factors of the group's order. In general, it is not possible to formulate a complete set of invariants; for example, there is no known set of invariants that can be used to classify graphs to isomorphism. However, even without a complete set, invariants can still be used to provide a classification of the models of a core theory in PSL, which leads to the classification criterion and the definitional extension criterion:

Classification Criterion

The set of models for the core theories of PSL is partitioned into equivalence classes defined with respect to the set of invariants of the models.

Definitional Extension Criterion

Each equivalence class in the classification of PSL models can be axiomatized within a definitional extension of PSL.

In particular, each definitional extension in the PSL ontology is associated with a unique invariant; the different classes of activities or objects that are defined in an extension correspond to different properties of the invariant.

PSL in Action: A Foundation for Process Modeling

In this section, we present simple examples to illustrate one of the uses that PSL can be put to, namely, as a foundation for the semantics and, hence, integration, of business-process models.

Processes are patterns of activities. *Process modeling* is the linguistic, diagrammatic, or numeric representation of such patterns. Process models, in these various forms, are ubiquitous in industry: There is a plethora of business and engineering applications—work flow, scheduling, discrete-event simulation, process planning, business-process modeling, and others—that are designed explicitly for the construction of process models of various sorts. The vision of enterprise integration, therefore, will be realized only if it is possible to integrate business-process models. It is somewhat scandalous that so little work on this rather practical issue has been done.

As is widely recognized, process-model integration will be possible only if there is a common semantics of process information to draw on. Among potential semantic frameworks, the theory of Petri nets is perhaps the most power-

ful. However, we find Petri nets undesirable as a formal foundation for process modeling for at least two reasons: First, there is still no standard, widely agreed-on semantics for Petri nets, and those semantic systems that exist are highly complex and require a sophisticated knowledge of certain areas of mathematics. (The most common approach to providing a semantics for Petri nets is to map the apparatus into linear logic and then exploit one of several semantic theories for linear logic; see, for example, Marti-Oliet and Meseguer [1991]). Second, Petri nets do not provide any sort of axiomatic theory of processes. It is therefore rather difficult to gain any insight, from Petri nets alone, into the nature of the things that process models are about and, hence, difficult to see how it can serve as a basis for process-model sharing and integration.

By contrast, PSL scores well on both these counts. The language of PSL has a rigorous semantics that draws on familiar model-theoretic and algebraic structures. This semantics, in turn, is fully captured in a complete set of axioms. We illustrate how it can be used as a foundation for process modeling with a simple example.

In general, business and engineering processes are described at the type level: A *process model* characterizes a certain general, repeatable process structure. This structure, in turn, might admit to many instances that, depending on how constrained the structure is, might differ considerably from one another. A robust foundation for process modeling, therefore, should be able to characterize both the general process structure described by a model as well as the class of possible instances of that structure. Moreover, such a foundation must clearly be able to represent the constraints that a process model places on something's counting as an instance of the process, the constraints, as we might say, on process realization.

A typical process is best thought of informally as a structured collection of activities that are related to one another in a manner that reflects the process flow and temporal relations that can appear in any given occurrence of the process. For example, consider the painting process depicted in figure 4 (we use the graphic notation of the IDEF3 process-description capture method to illustrate the intuitive process). This diagram depicts a general process that must begin with an occurrence of Paint Widget (represented by the Paint Widget box with no predecessor), followed by an occurrence of Test Coverage. At this point, depending on the outcome of the test, an occurrence of the process

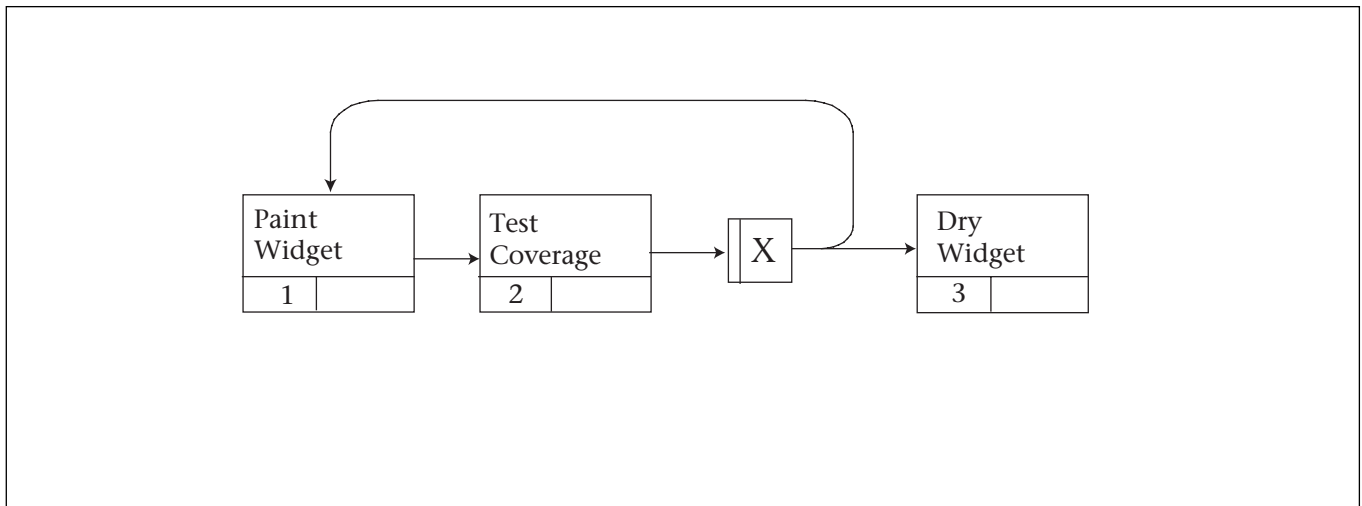


Figure 4. Paint, Test, and Dry Process.

can either loop back to another occurrence of Paint Widget (wherein our current widget is repainted) or continue to have the widget dried. Thus, there are, in principle, infinitely many possible ways this single process can be instantiated by a particular series of activity occurrences, depending on how many times such a series loops back to produce another occurrence of the Paint Widget activity.

The PSL ontology axiomatizes the classes of activities and resources that are used when defining a process. However, when using PSL, software applications are not exchanging definitions of classes of activities; rather, they are exchanging sentences that are satisfied by activities that belong to these classes. Such sentences are known as process specifications, and they include preconditions and effects, temporal constraints on occurrences, and ordering constraints on subactivity occurrences. In the remainder of this article, we present several examples of processes and a simplified syntax that can be used for process specifications.

The first step is the notion of an activity role declaration (ARD), characterized generally as follows:

```

(define-activity-role
  :id <number>
  :name <string>
  :successors <number>*
  :preconditions <PSL sentence>*
  :postconditions <PSL sentence>*)
  
```

The value of the `:id` field in an ARD *D* is known as its *role identifier*, and the value of the `:name` field of *D* is its *activity name*. The values of the `:subactivity-successor` field are known as *D's successor identifiers*, and the values of the `:preconditions` and `:postconditions` fields are

known as *D's preconditions* and *postconditions*, respectively.

ARDs correspond roughly to IDEF3 boxes, as seen in figures 4 and 5. Thus, in the context of a process model, an ARD represents the structural role that the indicated activity plays in the process represented by the model. An ARD has both name (`:name`) and identifier (`:id`) fields because the same activity can play different roles in the same process. In such cases, we typically have two or more distinct declarations with the same activity name but with distinct activity identifiers because it is the identifiers that indicate the distinct structural roles being played by the activity in the overarching process. The successor field will contain the identifiers of other ARDs (or possibly the same ARD) in the model and the precondition and postcondition PSL sentences that express conditions that must hold before and after an occurrence of the given activity—in the indicated role—in a realization of the model.

In addition to activity constraints, one also has to be able to express information about the objects that participate in the activity occurrences that jointly realize the model. Such information is often relegated to text in a graphic model but is just as critical as the process-structure information represented explicitly by the boxes and arrows of IDEF3. Hence, we introduce a similar mechanism—*object declarations*—for introducing the participating objects into process models:

```

(define-object
  :name <KIF constant>
  :constraints <PSL sentence>*)
  
```

The `:name` field of an object declaration, of course, specifies the constant to be introduced,

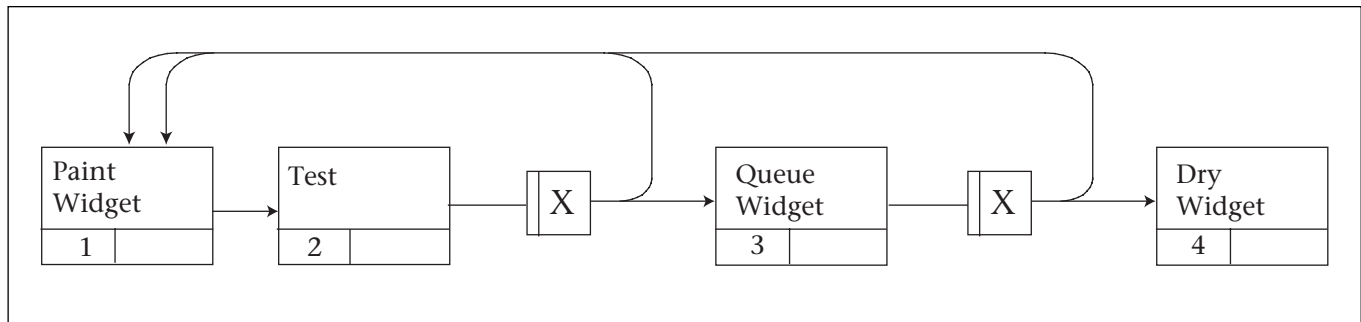


Figure 5. Paint, Test, Queue, and Dry Process.

and the :axioms field consists of PSL sentences that characterize the indicated object. Identifiers are unnecessary because the same object does not play different structural roles in a process the way that activities do.

Given this apparatus then, we can capture both the structural information indicated by the earlier IDEF3 diagram as well as implicit content about participating objects. Note that a general background ontology characterizing the relevant properties and relations in this model (Widget, In, Paint_Coverage) is being assumed (figure 6).

Note that the preconditions and postconditions all contain a free activity occurrence variable ?occ. Say that an activity occurrence e satisfies an ARD if e is an occurrence of the activity named in the ARD, and the preconditions and postconditions are true, relative to some variable assignment that assigns e as the value of the occurrence variable (?occ is the only variable in the previous ARDs). Because of the presence of the occurrence variable, many different occurrences of the activity can satisfy the same ARD, which is critical because looping can lead to a situation where the same ARD is satisfied by many different occurrences—as happens in the case of our example if the same widget is repainted because of inadequate coverage.

A collection of activity occurrences can be said to realize a process model M if (1) the temporal-ordering over occurrences in the collection can be mapped into the ordering determined by the successor fields of the ARDs in M in a structure-preserving way (that is, *homomorphically*) and (2) each activity occurrence in the collection satisfies the ARD to which it is mapped. Given this definition, we see that any series of activity occurrences in which a widget is painted and then, if necessary, repeatedly repainted until its coverage is adequate and then dried will realize the process model.

This example, of course, is rather simplistic. In particular, most iterative processes involve not simply the same object undergoing a pro-

cedure numerous times, as with the widget in the previous process, but many objects of the same sort undergoing the same procedure, as in a typical manufacturing process.

For example, the IDEF3 diagram in figure 5 contains both sorts of looping. In this process, a widget is painted until coverage is adequate and then queued, at which point either another widget is painted or, if the queue is full, all the queued widgets are dried en masse. Thus, the first loop, as in the first example, “carries” a single widget back to undergo an earlier activity, whereas the second indicates the beginning of a new paint job with a new widget; there is, so to speak, no “object flow” in the second loop.

The flexibility of variables in the PSL language enables us to capture this semantics simply and easily. Their meanings, unlike ordinary names, can shift, and we can use this fact to allow them to refer to different widgets in different events in a process realization. Like ordinary names, however, we can put constraints on the values of these variables. Following a similar notion in situation theory (Barwise and Etchemendy 1987; Devlin 1991), we refer to such constrained variables (and occasionally, ambiguously, their values) as *parameters*, and we introduce a concomitant declaration template:

```
(define-parameter
  :variable <KIF variable>
  :constraints <PSL sentence>*)
```

We now say that an occurrence e satisfies an ARD D if it satisfies it in the previous sense; in addition, for any parameter V occurring in the preconditions or postconditions of D , there is an object a participating in e such that the parameter’s constraints are true when V is assigned a as its value.

Armed with this construct, we can capture the detailed semantics of the process indicated in figure 5 by adding the declarations in figure 7.

```

(define-object
  :name widget
  :constraints (Widget widget))

(define-object
  :name painter
  :constraints (Paint_Sprayer painter))

(define-object
  :name oven
  :constraints (Oven oven))

(define-activity-role
  :id Act-1
  :name Paint_Widget
  :successors 2
  :preconditions
    (or (not (Painted widget (beginof ?occ)))
        (not (Adequate (Paint_Coverage widget (beginof ?occ))))))
  :postconditions
    (Painted widget (endof ?occ)))

(define-activity-role
  :id Act-2
  :name Test_Coverage
  :successors 1 3
  :preconditions (Painted widget (beginof ?occ))
  :postconditions (Adequate (Paint_Coverage widget) (endof ?occ)))

(define-activity-role
  :id Act-3
  :name Dry_Widget
  :successors
  :preconditions (Adequate (Paint_Coverage widget) (beginof ?occ))
  :postconditions (Dry widget (endof ?occ)))

```

Figure 6. Process Declarations for the Point, Test, and Dry Process.

Because variables can be assigned different values relative to an interpretation of the names and predicates of a language, we are able to capture the intended semantics of the complex looping of figure 5.

Summary

Within the increasingly complex manufacturing environment where process models are

maintained in different software applications, standards for the exchange of this information must address not only the syntax but also the semantics of process concepts. PSL draws on well-known mathematical tools and techniques to provide a robust semantic foundation for the representation of process information. This foundation includes first-order theories for concepts together with complete characterizations of the satisfiability and axiomatizability of the models of these theories.

```

(define-parameter
  :variable ?w
  :constraints (Widget ?w))

(define-activity-role
  :id Act-1
  :name Paint_Widget
  :successors 2
  :preconditions
    (or (not (Painted ?w (beginof ?occ)))
        (not (Adequate (Paint_Coverage ?w (beginof ?occ))))))
  :postconditions
    (Painted widget (endof ?occ)))

(define-activity-role
  :id Act-2
  :name Test_Coverage
  :successors 1 3
  :preconditions (Painted ?w (beginof ?occ))
  :postconditions (Adequate (Paint_Coverage widget) (endof ?occ)))

(define-activity-role
  :id Act-3
  :name Queue_Widget
  :successors 1 4
  :preconditions
    (Adequate (Paint_Coverage ?w (beginof ?occ)))
  :postconditions
    (Painted widget (endof ?occ)))

(define-activity-role
  :id Act-4
  :name Dry_Widget
  :successors
  :preconditions (Adequate (Paint_Coverage ?w) (beginof ?occ))
  :postconditions (Dry ?w (endof ?occ)))

```

Figure 7. Process Declarations for the Point, Test, Queue, and Dry Process.

Moreover, the modular organization of PSL enables the flexible support of interoperability, even when the applications involved do not have explicit ontologies.

References

- Barwise, J., and Etchemendy, J. 1987. *The Liar: An Essay on Truth and Circularity*. New York: Oxford University Press.
- Dennet, D. C. 1987. *The Intentional Stance*. Cambridge, Mass.: MIT Press.
- Devlin, K. 1991. *Logic and Information*. Cambridge, U.K.: Cambridge University Press.
- McCarthy, J., and Hayes, P. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, eds. B. Meltzer and D. Michie, 463–502. Edinburgh, U.K.: Edinburgh University Press.



The New International AI Web Site Could Use Your Help!

AAAI, in cooperation with IJCAI and other international AI organizations, has launched a new international AI web site. Launched initially as a finding aid to other AI sites around the world, this new site contains links to national AI societies, nonuniversity laboratories and research sites, and university AI departments and laboratories.

Please visit us at www.aiinternational.org and if you find that your university or company is not listed (and you think it should be!), please use the contact form to submit your suggested link.

By putting all these links together in one maintained site, we hope this site will provide useful information to you and the entire international AI community.

Marti-Oliet, N., and Meseguer, J. 1991. From Petri Nets to Linear Logic. *Mathematical Structures in Computer Science* 1(1): 69–101.

Menzel, C., and Gruninger, M. 2001. A Formal Foundation for Process Modeling. In *Formal Ontology in Information Systems*, eds. C. Welty and B. Smith, 256–269. New York: Association of Computing Machinery.

Pinto, J. 1994. Temporal Reasoning in the Situation Calculus, Technical Report, KRR-TR-94-1, Department of Computer Science, University of Toronto.

Reiter, R. 1991. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, ed. Vladimir Lifschitz, 418–440. San Diego, Calif.: Academic.

Schlenoff, C.; Gruninger, M.; and Ciociu, M.. 1999. The Essence of the Process-Specification Language. *Transactions of the Society for Computer Simulation* 16(4): 204–216.

Michael Gruninger is currently a research scientist at the Institute for Systems Research at the University of Maryland at College Park and also a guest researcher at the National Institute for Standards and Technology (NIST). He received his Ph.D. and M.Sc. in computer science at the University of Toronto and his B.Sc. in computer science at the University of Alberta. His current research focuses on the design and formal characterization of ontologies and their application to problems in manufacturing and enterprise engineering. Gruninger is also the project leader for ISO 18629 (PROCESS SPECIFICATION LANGUAGE) within the International Standards Organization.

Chris Menzel is an associate professor in the Philosophy Department at Texas A&M University. His research interests include metaphysics and philosophical logic, but he has been known to get his hands dirty studying the application of these ideas to problems of representing and managing information in engineering and manufacturing systems. He holds a Ph.D. in philosophy from the University of Notre Dame.

Reminder:

**AAAI-04 / IAAI-04
Submissions are Due
January 20, 2004. For
details, follow the link
at AAAI's home page.**