NISTIR 7165

Applications of PSL to Semantic Web Services

Michael Gruninger

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

NISTIR 7165

Applications of PSL to Semantic Web Services

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

September 2004

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

Applications of PSL to Semantic Web Services

Michael Gruninger[1]

Institute for Systems Research, University of Maryland, College Park, MD 20742
gruning@cme.nist.gov

Abstract. In this paper we will show how the ontology of the Process Specification Language can be used as an upper-level process ontology that serves as the semantic foundation for the DAML-S ontology for web services.

1 Semantics for Web Services

To achieve the vision of the Semantic Web, software agents will need a computer-interpretable description of the services they offer and the information that they access. Such a description can be provided by an ontology, which explicitly represents the intended meanings of the terms being used. Within the DARPA Agent Markup Language program, an ontology of services called DAML-S has been proposed to support the discovery, invocation, and composition of the services offered by software agents on the Semantic Web.

The Process Specification Language (PSL) ([2], [4], [5]) has been designed to facilitate correct and complete exchange of process information among manufacturing systems [1]. Included in these applications are scheduling, process modeling, process planning, production planning, simulation, project management, workflow, and business process reengineering. In this paper we will show how PSL can be used as an upper-level process ontology that serves as the semantic foundation for an ontology for web services that extends DAML-S.

Any ontology that supports the representation of web services will consist of generic classes to support service specification as well as classes of constraints in service specifications, such as ordering, temporal, occurrence, and duration.

The ontology must also support reasoning problems for web service specifications such as determining the consistency of a service specification and the composability of services, particularly with incomplete service specifications.

The approach taken in this paper will be to specify a first-order semantics for DAML-S concepts through PSL translation definitions and then use the grammars associated with PSL classes as an abstract syntax for service specifications.

[1] PSL is project ISO 18629 within the International Organisation of Standardisation, and has been accepted as a Draft International Standard.

2 The Role of First-Order Logic

The PSL Ontology is a set of theories in the language of first-order logic. There are several other approaches to semantics for web services, such as BPEL [1], for which Petri nets and π-calculus have been proposed as the basis for their semantics. However, a first-order semantics has several advantages. First, we can specify and implement inference techniques that are sound and complete with respect to models of the theories. Also, a process ontology with a first-order axiomatization can be more easily integrated with other ontologies (which are almost all first-order theories themselves). Finally, a first-order semantics allows a simple characterization of incomplete service specifications.

The semantics of a first-order theory are based on the notion of an interpretation that specifies a meaning for each symbol in a sentence of the language. In practice, interpretations are typically specified by identifying each symbol in the language with an element of some algebraic or combinatorial structure, such as graphs, linear orderings, partial orderings, groups, fields, or vector spaces; the underlying theory of the structure then becomes available as a basis for reasoning about the concepts and their relationships.

First-order logic is sound and complete – a theory is consistent if and only if there exists a model that satisfies the axioms of the theory. This allows us to evaluate the adequacy of the application's ontology with respect to some class of structures that capture the intended meanings of the ontology's terms by proving that the ontology obeys the following two fundamental theorems:

- Satisfiability: every structure in the class is a model of the ontology.
- Axiomatizability: every model of the ontology is isomorphic to some structure in the class.

The purpose of the Axiomatizability Theorem is to demonstrate that there do not exist any unintended models of the theory, that is, any models that are not specified in the class of structures. In general, this would require second-order logic, but the design of PSL makes the following assumption (hereafter referred to as the Interoperability Hypothesis): *The ontology supports interoperability among first-order inference engines that exchange first-order sentences.* By this hypothesis, we do not need to restrict ourselves to elementary classes of structures when we are axiomatizing an ontology. Since 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.

Classes of structures for theories within the PSL Ontology are therefore axiomatized up to elementary equivalence – the 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 up to isomorphism.

3 PSL Ontology

Within the PSL Ontology, there is a further distinction between core theories and definitional extensions. Core theories introduce new primitive concepts, while all terms introduced in a definitional extension that are conservatively defined using the terminology of the core theories [2].

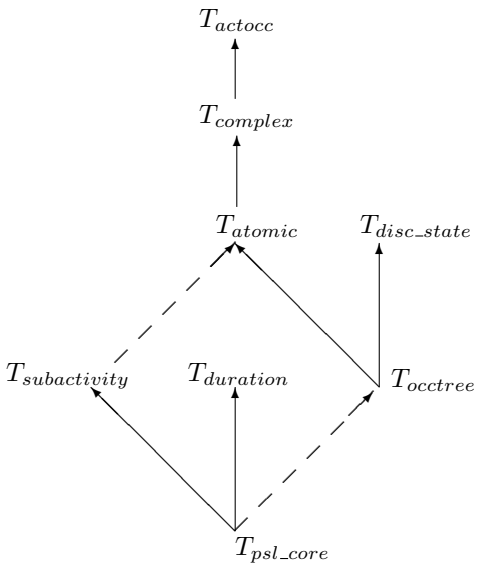

Fig. 1. The core theories of the PSL Ontology. Solid lines indicate conservative extension, while dashed lines indicate an extension that is not conservative.

3.1 Core Theories

All core theories within the ontology are consistent extensions of PSL-Core (T_{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 1.

[2] The complete set of axioms for the PSL Ontology can be found at `http://www.mel.nist.gov/psl/psl-ontology/`. Core theories are indicated by a .th suffix and definitional extensions are indicated by a .def suffix.

Occurrence Trees The occurrence trees that are axiomatized in the core theory $T_{occtree}$ are partially ordered sets of activity occurrences, such that for a given set of activities, all discrete sequences of their occurrences are branches of the tree (see Figure 2). An occurrence tree contains all occurrences of *all* activities; it is not simply the set of occurrences of a particular (possibly complex) activity. Because the tree is discrete, each activity occurrence in the tree has a unique successor occurrence of each activity.

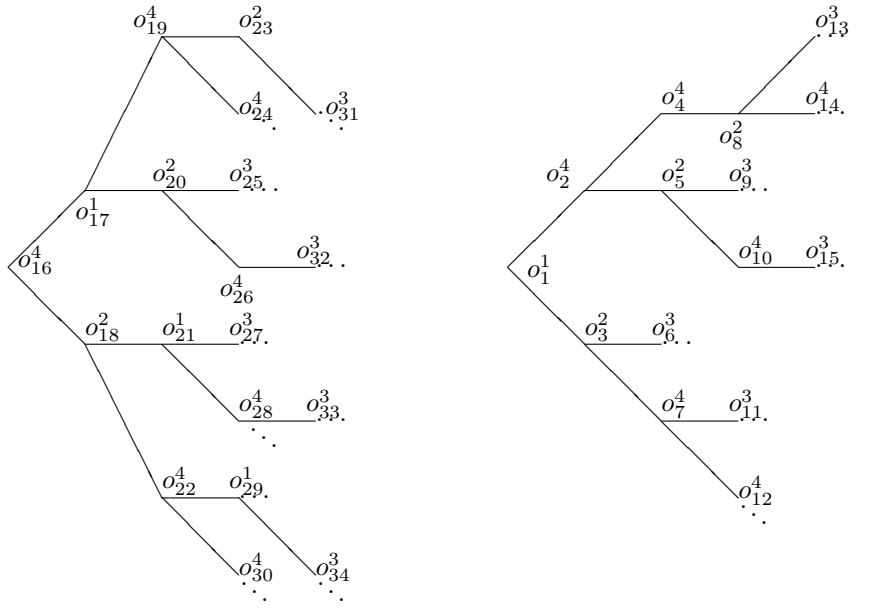

Fig. 2. Example of legal occurrence trees. The elements o^1_i denote occurrences of the activity a_1, o^2_i denote occurrences of the activity a_2, o^3_i denote occurrences of the activity a_3, and o^4_i denote occurrences of the activity a_4. The activity occurrences o^1_1 and o^4_{16} are the initial occurrences in their respective occurrence trees.

There are constraints on which activities can possibly occur in some domain. This intuition is the cornerstone for characterizing the semantics of classes of activities and process descriptions. Although occurrence trees characterize all sequences of activity occurrences, not all of these sequences will intuitively be physically possible within the domain. We will therefore want to consider the subtree of the occurrence tree that consists only of *possible* sequences of activity occurrences; this subtree is referred to as the legal occurrence tree.

Discrete States The core theory T_{disc_state} 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 The PSL Ontology uses the *subactivity* relation to capture the basic intuitions for the composition of activities. This relation is a discrete partial ordering, in which primitive activities are the minimal elements.

Atomic Activities The core theory T_{atomic} 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.

Complex Activities The core theory $T_{complex}$ characterizes the relationship between the occurrence of a complex activity and occurrences of its subactivities. Occurrences of complex activities correspond to sets of occurrences of subactivities; in particular, these sets are subtrees of the occurrence tree. An activity tree consists of all possible sequences of atomic subactivity occurrences beginning from a root subactivity occurrence. In a sense, activity trees are a microcosm of the occurrence tree, in which we consider all of the ways in which the world unfolds *in the context of an occurrence of the complex activity.*

Different subactivities may occur on different branches of the activity tree i.e., different occurrences of an activity may have different subactivity occurrences or different orderings on the same subactivity occurrences. In this sense, branches of the activity tree characterize the nondeterminism that arises from different ordering constraints or iteration.

An activity will in general have multiple activity trees within an occurrence tree, and not all activity trees for an activity need be isomorphic. Different activity trees for the same activity can have different subactivity occurrences. Following this intuition, the core theory $T_{complex}$ does not constrain which subactivities occur. For example, conditional activities are characterized by cases in which the state that holds prior to the activity occurrence determines which subactivities occur. In fact, an activity may 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.

3.2 Definitional Extensions

Many ontologies are specified as taxonomies or class hierarchies, yet few ever give any justification for the classification. If we consider ontologies of mathematical structures, we see that logicians classify models by using properties of models, known as invariants, that are preserved by isomorphism. For some

classes of structures, such as vector spaces, invariants can be used to classify the structures up to isomorphism; for example, vector spaces can be classified up to isomorphism by their dimension. For other classes of structures, such as graphs, it is not possible to formulate a complete set of invariants. However, even without a complete set, invariants can still be used to provide a classification of the models of a theory.

Following this methodology, the set of models for the core theories of PSL are partitioned into equivalence classes defined with respect to the set of invariants of the models. Each equivalence class in the classification of PSL models is axiomatized using 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. In this way, the terminology of the PSL Ontology arises from the classification of the models of the core theories with respect to sets of invariants. The terminology within the definitional extensions intuitively corresponds to classes of activities and objects.

4 Translation Definitions

Translation definitions specify the mappings between PSL and application ontologies. Such definitions have a special syntactic form – they are biconditionals in which the antecedent is a class in the application ontology and the consequent is a formula that uses only the lexicon of the PSL Ontology.

Translation definitions are generated using the organization of the definitional extensions. Each invariant from the classification of models corresponds to a different definitional extension. Any particular activity, activity occurrence, or fluent will have a unique value for the invariant. Each class of activity, activity occurrence, or fluent corresponds to a different value for the invariant. The consequence of a translation definition is equivalent to the list of invariant values for members of the application ontology class.

4.1 DAML-S Translation Definitions

In this section we will present the translation definitions [3] for concepts in the DAML-S Process Ontology. Such translation definitions provide a first-order axiomatization of the intended semantics for the DAML-S constructs. Moreover, this axiomatization inherits the proofs of the Axiomatizability and Satisfiability Theorems from the underlying PSL Ontology.

Atomic Activities The `composedOf` property in DAML-S is equivalent to the `subactivity` relation in PSL:

[3] The translation definitions in this paper are written in the Knowledge Interchange Format. For more information on this language, see `http:cl.tamu.edu`.

```
(forall (?a1 ?a2)
        (iff    (composedOf ?a1 ?a2)
                (subactivity ?a2 ?a1)))
```

Within DAML-S, an `AtomicProcess` has no subprocesses; consequently, this corresponds to a primitive activity within PSL.

```
(forall (?a)
        (iff    (AtomicProcess ?a)
                (and    (primitive ?a)
                        (markov_precond ?a)
                        (or  (markov_effects ?a)
                             (context_free ?a)))))
```

The most common cconstraint on the legal occurrences of an activity specify the activity's preconditions. Activities whose preconditions depend only on the state prior to the occurrences. The class of activities with markov preconditions is defined in the PSL definitional extension $state\_precond.def$.

Effects characterize the ways in which activity occurrences change the state of the world. Such effects may be context-free, so that all occurrences of the activity change the same states, or they may be constrained by other conditions. The most common constraints are state-based effects that depend on the context; the class of activity associated with such constraints are defined as $\texttt{markov}_{\texttt{e}}\texttt{ffect}$ activities in the PSL extension $state\_effects.def$.

A `CompositeProcess` in DAML-S is decomposable into other processes. Within PSL, the corresponding activity cannot be primitive; it will either be atomic (in which case it is a concurrent activity) or complex:

```
(forall (?a)
        (iff    (CompositeProcess ?a)
                (and    (activity ?a)
                        (not (primitive ?a)))))
```

Ordered Activities The classification of models within the the PSL Ontology leads to classes of activities, activity occurrences, and fluents. Classes of activity occurrences correspond to invariants for activity trees. The translation definitions for remaining DAML-S concepts are all related to invariants for activity trees.

Within DAML-S, a `Sequence` is a list of processes to be done in order (see Figure 3) [4]. The translation definition for `Sequence` has two parts; one says that there exists an activity tree for the activity which is ordered and which is simple

[4] All of the examples in this section refer to the activities whose process descriptions are found in the Appendix.

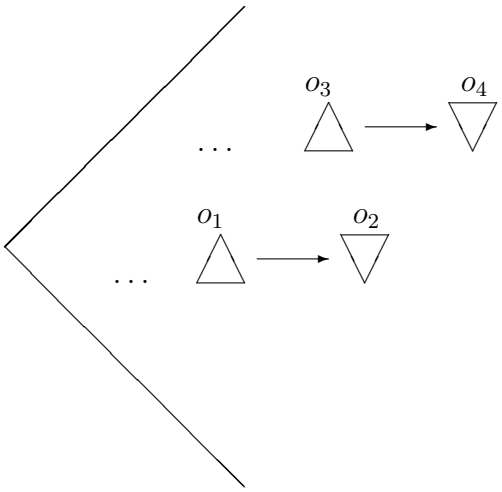

Fig. 3. Example of activity trees for $transfer$, which is a Sequence DAML-S activity. o_1 and o_3 are occurrences of the subactivity $withdraw$, while o_2 and o_4 are occurrences of the subactivity *deposit*. Note that the diagram depicts two separate activity trees within a stylized legal occurrence tree.

and rigid (that is, there are no nontrivial permutations of subactivity occurrences). The second part says that the activity is uniform, that is, all activity trees for the activity are isomorphic: [5]

```
(forall (?a)
        (iff    (Sequence ?a)
                (and    (uniform ?a)
                        (exists (?occ)
                                (and  (occurrence_of ?occ ?a)
                                      (simple ?occ)
                                      (rigid ?occ)
                                      (ordered ?occ)
                                      (strong_poset ?occ))))))
```

In a DAML-S `Split` activity, sets of subactivities are performed in parallel (see Figure 4). `Split` activities differ from `Sequence` activities in that there

[5] Two branches of an activity tree are isomorphic if there is a one-to-one mapping of subactivity occurrences that preserves the activities, e.g. occurrences of activity a_1 are mapped to occurrences of a_1. Two activity trees are isomorphic if all of their branches are isomorphic. In the visual convention adopted in this paper, occurrences of different activities are depicted by different shapes; thus, a mapping that preserves activities will map a square to a square, a circle to a circle, and so on.

exist nontrivial permutations of subactivity occurrences among the branches of the activity trees, so that the translation definition becomes:

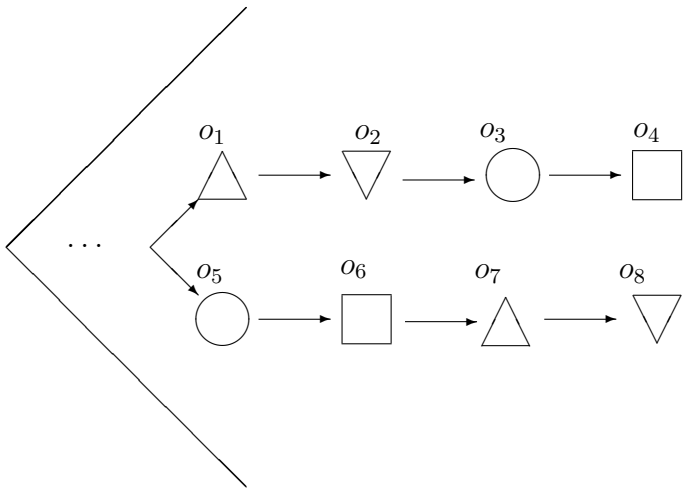

Fig. 4. Example of an activity tree for *buy_product*, which is a Split DAML-S activity. For this purposes of this example, consider $transfer$ to be a complex activity, with *deposit* and *withdraw* as subactivities.

```
(forall (?a)
        (iff    (Split ?a)
                (and    (uniform ?a)
                        (exists (?occ)
                                (and  (occurrence_of ?occ ?a)
                                      (not (simple ?occ))
                                      (ordered ?occ)
                                      (strong_poset ?occ))))))
```

For example, in Figure 4, the two branches of the activity tree consist of isomorphic subactivity occurrences that occur in different orderings on each branch.

According to [3], the `Unordered` construct allows process components to be executed in some unspecified order, although all components must be executed. This is equivalent to the class of `bag` activity trees within the PSL Ontology:

```
(forall (?a)
        (iff    (Unordered ?a)
```

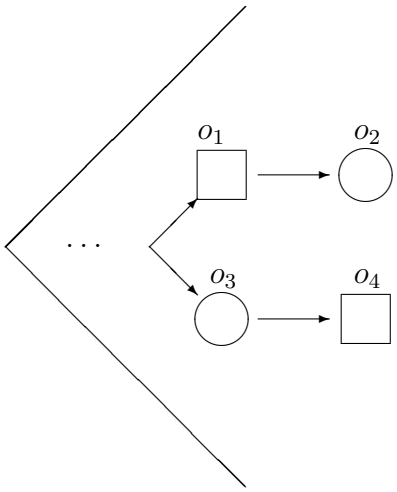

Fig. 5. Example of an activity tree for *buy_product*, which is an Unordered DAML-S activity. For the purpose of this example, consider $transfer$ to be a primitive activity; o_1 and o_4 are occurrences of the subactivity `(transfer ?Fee ?Buyer ?Broker)`, o_2 and o_4 are occurrences of the subactivity `(transfer ?Cost ?Buyer ?Seller)`.

```
(and    (uniform ?a)
        (exists (?occ)
                (and  (occurrence_of ?occ ?a)
                      (bag ?occ))))))
```

In Figure 5, we see an example of an activity tree that is the unordered activity with two subactivities.

Nondeterminism The simplest form of nondeterminism is captured by the class of activities in which some subactivity occurs (see Figure 6). Given this intended semantics, the translation definition to PSL would be:

```
(forall (?a)
        (iff    (Choice ?a)
                (and    (uniform ?a)
                        (exists (?occ)
                                (and  (occurrence_of ?occ ?a)
                                      (simple ?occ)
                                      (rigid ?occ)
                                      (unordered ?occ)
                                      (choice_poset ?occ))))))
```

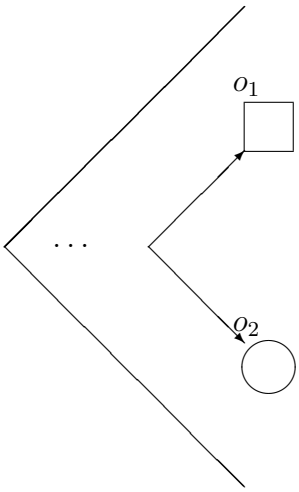

Fig. 6. Example of an activity tree for a Choice DAML-S activity that is equivalent to a *choice_poset* in PSL. In this example, o_1 is an occurrence of a withdrawal from Account1 and o_2 is an occurrence of a withdrawal from Account3.

There are some indications in [3] that the intended semantics for `Choice` activities is more general than this translation definition. For example, some possible applications of this construct may be intended to capture intuitions such as "choose subactivities and perform them in sequence" or "choose subactivities and perform them in parallel." In such cases, the corresponding PSL class would be based on the notion of weak posets (see Figure 7), so that the translation definition would be:

```
(forall (?a)
        (iff    (Choice ?a)
                (and    (uniform ?a)
                        (exists (?occ)
                                (and  (occurrence_of ?occ ?a)
                                      (simple ?occ)
                                      (weak_poset ?occ))))))
```

In addition, there are suggestions in [3] for extensions that construct new subclasses such as "choose exactly n subactivities from m." Such extensions do not correspond to any classes within Version 2.0 of the PSL Ontology.

Conditional Activities The class of `IfThenElse` activities within DAML-S are equivalent to conditional activities in PSL:

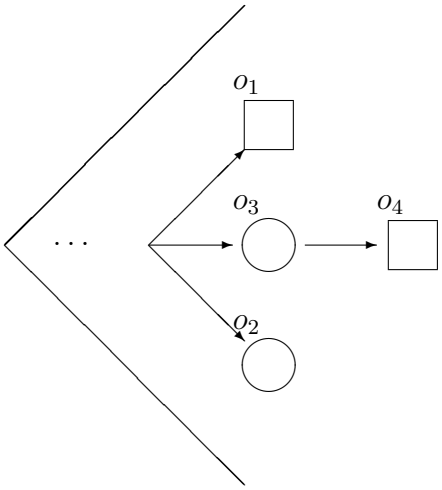

Fig. 7. Example of an activity tree for a Choice DAML-S activity that is equivalent to a *weak_poset* in PSL.

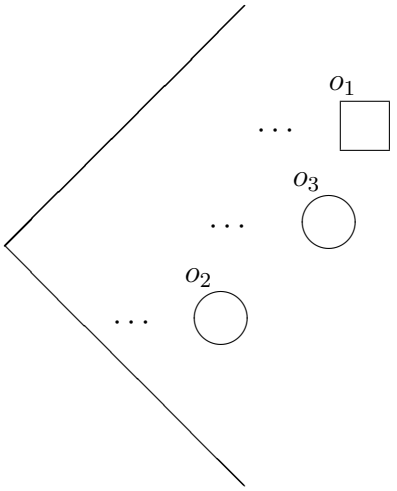

Fig. 8. Example of activity trees for *withdraw*, which is an IfThenElse DAML-S activity. o_2 and o_2 are occurrences of the subactivity *change_balance*, and o_1 is an occurrence of the subactivity $notify$.

```
(forall (?a)
        (iff    (IfThenElse ?a)
                (conditional ?a)))
```

Conditional activities are not uniform; however, if the same fluents hold prior to two occurrences of a conditional activity, then the activity trees for the activity are isomorphic. Figure 8 depicts three different activity trees, two of which are isomorphic.

Iterated Activities The intended semantics of the `Iterate` process in DAML-S makes no assumption about how many iterations are made, or when to terminate. Within PSL, this corresponds to an activity in which there exist multiple isomorphic subtrees; for example, the activity tree in Figure 9 contains three subtrees that are isomorphic to the activity tree in Figure 6. Since different activity trees may have different numbers of iterations of the subactivities, the activity is not uniform. These considerations lead to the following translation definition:

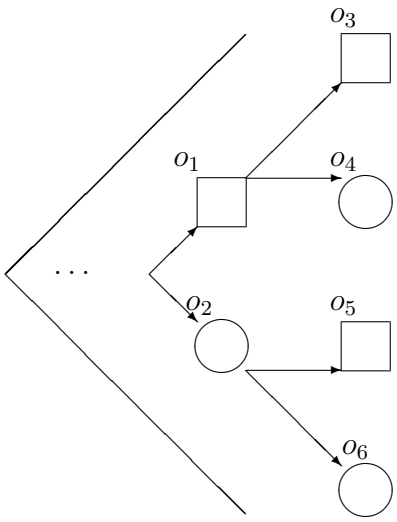

Fig. 9. Example of an activity tree for an Iterate DAML-S activity.

```
(forall (?a)
   (iff    (Iterate ?a)
           (forall (?occ)
```

```
                (implies  (occurrence_of ?occ ?a)
                       (and   (repetitive ?occ)
                              (multiple_outcome ?occ)))))
```

A `RepeatUntil` process in DAML-S executes until some state condition becomes true (see Figure 10). Because of this dependence on state, a `RepeatUntil` process is equivalent to an `Iterate` process which is conditional:

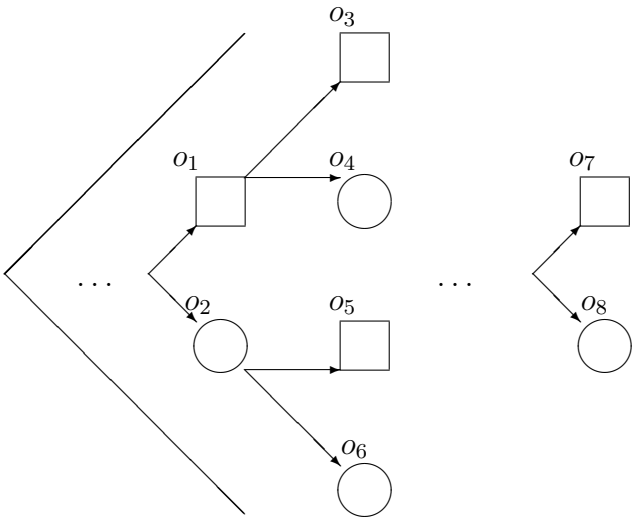

Fig. 10. Example of activity trees for a RepeatUntil DAML-S activity.

```
(forall (?a)
   (iff   (RepeatUntil ?a)
          (and   (conditional ?a)
                 (forall (?occ)
                      (implies  (occurrence_of ?occ ?a)
                                (and   (repetitive ?occ)
                                       (multiple_outcome ?occ)))))))
```

Thus, there will exist multiple nonisomorphic activity trees (corresponding to occurrences of the activity with different iterations), and activity trees that agree on state will be isomorphic.

5 Grammars for Process Descriptions

PSL makes a distinction between the ontology (which is the lexicon together with an axiomatization of their intended meaning) and the process descriptions that are exchanged between software applications. For each class in the ontology, PSL specifies a grammar that is satisfied by process descriptions of the activities or activity occurrences in that class.

For example, if two software applications both used an ontology for algebraic fields, they would not exchange new definitions, but rather they would exchange sentences that expressed properties of elements in their models. For algebraic fields, such sentences are equivalent to polynomials. Similarly, the software applications that use PSL do not exchange arbitrary sentences, such as new axioms or even conservative definitions, in the language of their ontology. Instead, they exchange process descriptions, which are sentences that are satisfied by particular activities, occurrences, states, or other objects.

DAML-S specifications are in fact grammars for service specifications. Using the translation definitions proposed in the previous section, we can use the grammars associated with the classes in the PSL Ontology to characterize the correctness and completeness of the DAML-S specification for the corresponding DAML-S constructs.

There are several classes within the DAML-S Ontology that are classes of sentences rather than classes of activities, activity occurrences, or fluents. In particular, DAML-S has two classes of conditions, ConditionalEffects and UnconditionalEffects. Within the PSL Ontology, this distinction is captured by the classes of $context\_free$ and $markov\_effects$ activities. If one considers the PSL process description grammars for a $context\_free$ activity, conditions appear as a class of formulae, but they are not a class in the ontology. Similarly comments apply to conditional activities. For example, in the process description for *withdraw* in the Appendix, the condition is the formula

```
(and  (prior (balance ?account ?Balance) (root_occ ?occ))
      (greaterEq ?Balance ?amount))
```

6 Summary

Within the increasingly complex environments of enterprise integration, electronic commerce, and the Semantic Web, 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.

DAML-S is an attempt to support semantic web services within the framework of the DARPA Agent MArkup Language. However, the intended semantics of the concepts in DAML-S cannot be axiomatized within the Ontology Web Language, and the DAML-S ontology itself combines object level classes of concepts together with metalevel classes of sentences.

The PSL Ontology draws upon 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. The PSL Ontology also provides a justification of the taxonomy of activities by classifying the models with respect to invariants. Finally, process descriptions are formally characterized as syntactic classes of sentences that are satisfied elements of the models.

The translation definitions presented in this paper are the first step towards laying firm logical foundations for semantic web services specified in DAML-S. Through these definitions, DAML-S can be given a sound and complete axiomatization and ontological distinctions can be clarified.

References

1. Business Process Execution Language for Web Services, Version 1.0 `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`
2. Gruninger, M. (2003) A Guide to the Ontology of the Process Specification Language", in *Handbook on Ontologies in Information Systems*, R. Studer and S. Staab (eds.). Springer-Verlag.
3. McIlraith, S., Son, T.C. and Zeng, H. (2001) Semantic Web Services, *IEEE Intelligent Systems*, Special Issue on the Semantic Web. 16:46–53, March/April, 2001.
4. Menzel, C. and Gruninger, M. (2001) A formal foundation for process modeling, *Second International Conference on Formal Ontologies in Information Systems*, Welty and Smith (eds), 256-269.
5. Schlenoff, C., Gruninger, M., Ciocoiu, M., (1999) The Essence of the Process Specification Language, *Transactions of the Society for Computer Simulation* vol.16 no.4 (December 1999) pages 204-216.

Appendix: Examples of Process Descriptions

To buy a product, pay a fee to the broker and the cost of the product to the seller, performing these steps in parallel.

The PSL process description for *buy_product* is:

```
(forall (?x ?y ?z) (subactivity (transfer ?x ?y ?z) (buy_product ?y)))
(forall (?x ?y ?z) (subactivity (withdraw ?x ?y) (transfer ?x ?y ?z)))
(forall (?x ?y ?z) (subactivity (deposit ?x ?z) (transfer ?x ?y ?z)))

(forall (?occ ?Buyer)
        (implies  (occurrence_of ?occ (buy_product ?Buyer))
                  (exists (?occ1 ?occ2 ?Fee ?Cost ?broker ?Seller)
                        (and    (occurrence_of (transfer ?Fee ?Buyer ?Broker))
                                (occurrence_of (transfer ?Cost ?Buyer ?Seller))
                                (subactivity_occurrence ?occ1 ?occ)
                                (subactivity_occurrence ?occ2 ?occ)))))
```

To transfer money from Account1 to Account2, withdraw some amount from Account1 and deposit the amount in Account2.

The PSL process description for $transfer$ is:

```
(forall (?occ)
        (implies  (occurrence_of ?occ (transfer ?Amount ?Account1 ?Account2))
                  (exists (?occ1 ?occ2 ?occ3)
                        (and    (occurrence_of ?occ1 (withdraw ?Amount ?Account1))
                                (occurrence_of ?occ2 (deposit ?Amount ?Account2))
                                (subactivity_occurrence ?occ1 ?occ)
                                (subactivity_occurrence ?occ2 ?occ)
                                (leaf_occ ?occ3 ?occ1)
                                (min_precedes ?occ3 (root_occ ?occ2))))))
```

To withdraw money from an account, if the amount is greater than the balance, then change the account balance, otherwise notify the account that there are insufficient funds available.

Suppose

```
(forall (?x ?y ?z) (activity (change_balance ?x ?y ?z)))

(subactivity (change_balance ?Account ?Balance1 ?Balance2)
             (deposit ?Amount ?Account))

(subactivity (change_balance ?Account ?Balance1 ?Balance2)
             (withdraw ?Amount ?Account))

(subactivity (notify ?Account)
             (withdraw ?Amount ?Account))
```

In this case, *deposit* and *withdraw* are conditional activities, with the following PSL process descriptions:

```
(forall (?occ)
(and (implies  (and     (occurrence_of ?occ (withdraw ?Amount ?Account))
                        (prior (balance ?account ?Balance) (root_occ ?occ))
                        (greaterEq ?Balance ?amount))
               (exists (?occ1)
                  (and    (occurrence_of ?occ1 (change_balance ?account ?Balance
                                                (plus ?Balance ?Amount)))
                        (subactivity_occurrence ?occ1 ?occ))))
     (implies  (and     (occurrence_of ?occ (withdraw ?Amount ?Account))
                        (prior (balance ?account ?Balance) (root_occ ?occ))
                        (lesser ?Balance ?amount))
               (exists (?occ2)
                  (and    (occurrence_of ?occ2 (notify ?Account))
                        (subactivity_occurrence ?occ2 ?occ)))))
```

The effects of *change_balance* are:

```
(forall (?occ)
   (implies  (and   (occurrence_of ?occ (change_balance ?Account ?Amount1 ?Amount2))
                    (leaf_occ ?occ1 ?occ))
             (and   (holds (balance ?Account ?Amount2))
                    (not (holds (balance ?Account ?Amount1))))))
```