Translation Patterns to Specify Processes in the PSL Ontology

Arturo J. Sánchez-Ruíz
University of North Florida – CIS Department
4567 St Johns Bluff Rd S
Jacksonville, FL 32224-2669, U.S.A
asanchez@unf.edu

Gregory Hansen
Computer-Aided Process Improvement
830-13 A1A North, Suite 327
Ponte Vedra Beach, FL 32082, U.S.A.
http://www.capi.net/

Abstract – *PSL stands for Process Specification Language, an ontology developed by the National Institute of Standards and Technology (NIST) to formally describe concepts, along with their properties and relationships, in the context of manufacturing systems. In this paper we derive patterns that can be used to describe processes comprised of activities, which can be either complex or primitive. We discuss the cases of processes described by sequential and concurrent composition of activities.*

Keywords: Process Specification Language (PSL), domain-specific ontologies, manufacturing systems, visual modeling of manufacturing processes, interoperability of manufacturing systems.

1 Introduction

The Process Specification Language (PSL) is an ontology developed by the National Institute of Standards and Technology (NIST) with the goal of providing a domain- and tool-neutral representation of manufacturing processes. One of the main implications behind achieving this goal is that of enabling interoperability among disparate process manufacturing/modeling tools. By giving PSL the role of mediator, any two tools can interoperate provided that a mapping is built, between the underlying ontologies such tools use to describe processes, and PSL. This represents a linear approach to interoperability as opposed to the quadratic all-to-all-mapping approach.

In Section 2 of this paper we give a gentle introduction to the PSL ontology for the theories that we use in our translation patterns. At the end of this section we present an annotated example with the goal of helping readers to follow our approach.

Section 3 discusses how we derived our translation patterns for processes described by the sequential composition of complex and primitive activities. Section 4 discusses the case of concurrent composition of activities.

The paper concludes with Sections 5 and 6, which present how our approach compares to others that have appeared in the literature, and a summary with some ideas for future work, respectively.

The relevance of this paper to this workshop stems from the fact that PSL can be seen as a domain-specific model, in this case the domain is that of manufacturing processes, and also as a

common (formally specified) lingo that can be used to achieve interoperability among disparate tools. It is therefore important to derive patterns that can be used to express processes based on the concepts specified by this ontology.

2 The PSL Ontology

The PSL ontology is structured as a set of theories in first-order logic, whose axioms and definitions are presented using the Knowledge Interchange Format (KIF) [1]. The so-called "psl_core" theory (T_{psl_core}) defines fundamental concepts (e.g. "activity"). Other theories are said to extend the core in that they introduce concepts to supplement those already in T_{psl_core}. For instance $T_{subactiviy}$ provides axioms to characterize the fact that, for instance, given two activities (say x and y), x is a subactivity of y and that y might have not subactivities associated with it. In this section we give brief [1] intuitive descriptions of the theories that are used in this paper and an annotated example that will allow us to show their use when defining process relationships in PSL. A comprehensive coverage of PSL is beyond the scope of this paper. Detailed information can be found in the website maintained by NIST [2] and in the literature cited in section "Related Work" of this paper.

2.1 T_{psl_core}

In this paper we use two concepts of this theory, namely activities and their occurrences (i.e. activity occurrences). Activities can have many or no occurrences, and a specific occurrence is associated with a unique activity. Table 1 shows the KIF terms used to refer to these concepts and their intuitive meanings.

Table 1: Activities and their occurrences in PSL[2]

(activity ?a)	**?a is an activity**
(occurrence_of ?o ?a)	**?o is an occurrence of activity ?a**

2.2 $T_{subactivity}$

When activity x is a subactivity of activity y, then y is called "complex". Activities that do not have subactivities are called "primitives". Table 2 shows the corresponding KIF terms.

Table 2: Complex and primitives activities

(subactivity ?x ?y)	**?x is a subactivity of ?y**
(primitive ?x)	**?x does not have subactivities**

[1] We will not describe all axioms and definitions associated with these PSL theories but rather those (theories, axioms, and definitions) used in this paper.

[2] In KIF, the expression "?x" denotes a variable, to differentiate it from a potential value of this variable such as "x". Therefore, the expression (activity eat) indicates that "eat" is an activity.

2.3 $T_{occtree}$

In PSL, the occurrence tree characterizes all possible sequences of activity occurrences (for all activities in the world under modeling). A given process can therefore be characterized by some subtree of this tree, which is called an occurrence tree. This implies that an occurrence tree may have branches that may not make sense in the world under modeling (i.e. they are "illegal"). Table 3 shows the KIF terms that are used to reason about occurrences.

Table 3: Terms associated with occurrence trees

(successor ?a ?o)	**Denotes an occurrence of ?a that follows occurrence ?o in the occurrence tree**
(legal ?o)	**?o is a legal occurrence in the occurrence tree**

2.4 $T_{complex}$

Suppose that activity `x` is composed of subactivities `y` and `z`, and that these two are primitive activities (hence, `x` is complex). This theory introduces terms to reason about the connections between occurrences of `x` and occurrences of `y`, and `z`. An activity tree for `x` is defined as all the possible sequences of primitive activity occurrences of `y` and `z`. Equivalently, an activity tree characterizes a complex occurrence, that is to say, the occurrence of a complex activity. Notice that all activity trees are occurrence trees, but not vice-versa. The following table shows the term that expresses when a subactivity occurrence is the strict successor of another in an activity tree. Notice that this term does not rule out the possibility of having another subactivity occurrence, say `?o3`, that follows `?o1`, as long as `?o3` is not an occurrence associated with a subactivity of `?a`.

Table 4: Strict successor of a subactivity occurrence in an activity tree

(next_subocc ?o1 ?o2 ?a)	**Subactivity occurrence ?o2 is the successor of subactivity occurrence ?o1 in a activity tree for ?a, and there are no other subactivity occurrences of ?a in between them**

2.5 T_{actocc}

This theory provides the elements to reason about subactivity occurrences in connection with complex occurrences. The terms used in this paper are shown in Table 4. The intuition associated with the "root" and "leaf" primitive occurrences is that they mark the "beginning" and the "end" of complex occurrences, and therefore they do not admit occurrences before and after them, respectively.

Table 5: Terms associated with complex activity occurrences

(subactivity_occurrence ?o1 ?o2)	**The primitive occurrence ?o1 is part of a complex occurrence ?o2**
(leaf_occ ?o1 ?o2)	**Primitive occurrence ?o1 is the leaf of complex occurrence ?o2**
(root_occ ?o1 ?o2)	**Primitive occurrence ?o1 is the root of complex occurrence ?o2**

2.6 An annotated example

Consider a complex activity `a`, with primitive subactivities `a1` and `a2`, respectively. Assume that we want to express the process characterized by occurrences of `a1` followed by occurrences of `a2`, such that:

- There are no occurrences before `a1` and after `a2`.
- There are no occurrences of `a` between a1 and `a2`.

Such process can be specified in PSL as shown in Table 6, using the theories that were described in previous sections. Table 7 shows a way or reading this specification in "plain English".

Table 6: A simple sequential process in PSL

```
[1](activity a)
[2](activity a1)
[3](activity a2)
[4](subactivity a1 a)
[5](subactivity a2 a)
[6](primitive a1)
[7](primitive a2)
[8](forall (?occ_a)
[9] (implies
[10](and (occurrence_of ?occ_a a)
[11]      (legal ?occ_a))
[12](exists (?occ_a1 ?occ_a2)
[13] (and
[14]    (occurrence_of ?occ_a1 a1)
```

```
[15]   (legal ?occ_a1)
[16]    (occurrence_of ?occ_a2 a2)
[17]   (legal ?occ_a2)
[18]   (subactivity_occurrence ?occ_a1 ?occ_a)
[19]    (subactivity_occurrence ?occ_a2 ?occ_a)
[20]    (root_occ ?occ_a1 ?occ_a)
[21]    (next_subocc ?occ_a1 ?occ_a2 a)
[22]    (leaf_occ ?occ_a2 ?occ_a)))))
```

Table 7: The PSL specification in "plain English"

```
[1]let a be an activity
[2]let a1 be an activity
[3]let a2 be an activity
[4]let a1 be a subactivity of a
[5]let a2 be a subactivity of a
[6]let a1 be primitive
[7]let a2 be primitive
[8]for all ?occ_a:
[9] if
[10]?occ_a is an occurrence of a and
[11]  ?occ_a is legal, then
[12]there exist ?occ_a1, ?occ_a2, such that
[13]
[14]  ?occ_a1 is an occurrence of a1, and
[15]  ?occ_a1 is legal, and
[16]  ?occ_a2 is an occurrence of a2, and
[17]  ?occ_a2 is legal, and
[18]  ?occ_a1 is a
      subactivity occurrence of ?occ_a, and
[19]  ?occ_a2 is a
      subactivity occurrence of ?occ_a, and
[20]  ?occ_a1 is the
      root occurrence of ?occ_a, and
[21]  ?occ_a2 strictly follows ?occ_a1 in
      the activity tree of a, and
[22]  ?occ_a2 is the leaf occurrence of ?occ_a
```

3 Translation Patterns: Sequential Composition

The previous section concluded with what can be considered our first translation pattern, which we would like to call P1 – Sequential Composition of two Primitive Activities. In this section we will show a progression of generalizations of this pattern.

3.1 P2: Generalized Sequential Composition of Primitive Activities

Let a be a complex activity comprised of a sequence of primitive activities `a1,…,an`, where `n>1`. We will assume that:

- There are no occurrences before `a1` and after `an`
- There are no occurrences of `a` between `ai` and `ai+1`, for `i=1,…,n-1`.

In order to generalize P1, we first define in Table 8 some functions that produce translation strings from their arguments.

Table 8: Template functions used to build P2

```
Declarations(a, a1,…,an):=
 (activity a) (activity a1)…(activity an)
 (primitive a1)…(primitive an)

ExistP(a1,…,an):= (?occ_a1 … ?occ_an)

Legal_Occurrences(a1,…,an):=
 (occurrence_of ?occ_a1 a1) (legal ?occ_a1)
                …
 (occurrence_of ?occ_an an) (legal ?occ_an)


Subactivities(a,a1,…,an):=
 (subactivity_occurrence ?occ_a1 ?occ_a)
                …
 (subactivity_occurrence ?occ_an ?occ_a)

RootP(a,a1):=
 (root_occ ?occ_a1 ?occ_a)

NextP(a,a1,…,an):=
 (next_subocc ?occ_a1 ?occ_a2 a)
                …
 (next_subocc ?occ_an-1 ?occ_an a)

LeafP(a,an):=
 (leaf_occ ?occ_an ?occ_a)
```

In Table 9 we show how to use these functions to define P2 by generalizing regions in the PSL specification of Table 6.[3]

Table 9: Translation Pattern P2

```
Declarations(a, a1,…,an)
(forall (?occ_a)
  (implies
    (and (occurrence_of ?occ_a a) (legal ?occ_a))
    (exists ExistP(a1,…,an)
      (and
        Legal_Occurrences(a1,…,an)
        Subactivities(a, a1,…,an)
        RootP(a,a1)
        NextP(a,a1,…,an)
        LeafP(a,an)))))
```

[3] The suffix "P" used as part of the name of some of the functions means "Primitive". Later in the paper we will use "C" for "Complex".

3.2 Activity dependency trees and activity execution trees

Pattern P2 in Table 9 models processes as complex activities that can be expressed as the sequential composition of primitive ones. In this section we generalize this idea by allowing processes to be expressed as sequential composition of activities which can be either complex or primitive.

An activity dependency tree (ADT) models the PSL subactivity relation and has two types of nodes. Internal nodes are associated with complex activities, and leaves are associated with primitive activities. For an internal node x, the descendants of x are associated with the subactivities of x. These subactivities can be either complex or primitives. The root of this tree characterizes the whole process under modeling.

An activity execution tree (AET) is an ADT such that its depth-first traversal characterizes activity occurrence sequences that are legal in association with the process under modeling, in the sense that if we take any given sequence of legal occurrences in the process description, and change the name of each occurrence for its corresponding activity, then this sequence is a subsequence of the depth-first traversal of the AET, and vice-versa.

The following are examples of ADT (see Figure 1) and AET (see Figure 2). Circles represent internal nodes, and rectangles represent leaves. For the AET we have added arrows to emphasize the depth-first order.

Given an AET T, the PSL characterization of the process that is described by the sequence of *primitive activities* resulting from the depth-first traversal of T can be obtained by applying pattern P2 with "a" being the root of the tree and a1,…,an being the leaves of T in depth-first order. As an example, consider the subtree of AET in Figure 3 with root "Build Structure", which has as descendants only "Paint Walls", and "Build Walls". In this case, the PSL characterization of this process is obtained after applying pattern P2 with a="Build Structure", a1="Prepare Concrete", a2="Let Dry", a3="Take out Cast", a4="Paint", a5="Dry", a6="Polish".

This approach "flattens" the AET, and therefore does not express in PSL dependencies among inner nodes and leaves. In order to accomplish this we would need a pattern that does not require the a1,...,an in P2 to be primitive.

In order to model the cases for which descendants of complex activities can in turn be complex, we can introduce activity occurrences that model the root and leaf of such complex activities. So every time a complex activity is "visited" (in the depth-first traversal) we use the primitive occurrences that span from the primitive root occurrence, to the leaf root occurrence of such complex occurrence. This leads to the creation of new template functions which are shown in Table 10. We can use these functions to build P3 that we call "Generalized Sequential Composition of General Activities Pattern (GSCGAP)" (see Table 11).

3.3 Generating the PSL code associated with an AET

We are now in a position to give a general algorithm to generate the PSL code associated with an Activity Execution Tree by iteratively using P3. In our notation, `GSCGAP(x)` computes a string that contains the PSL code associated with node `x` in the AET. This string corresponds to the application of P3 to the sequential composition `a1,a2,…,ak(x)` where the `ai` are the descendants of `x` in the AET. Notice that each `ai` can be either primitive or complex. The algorithm is shown in Table 12.

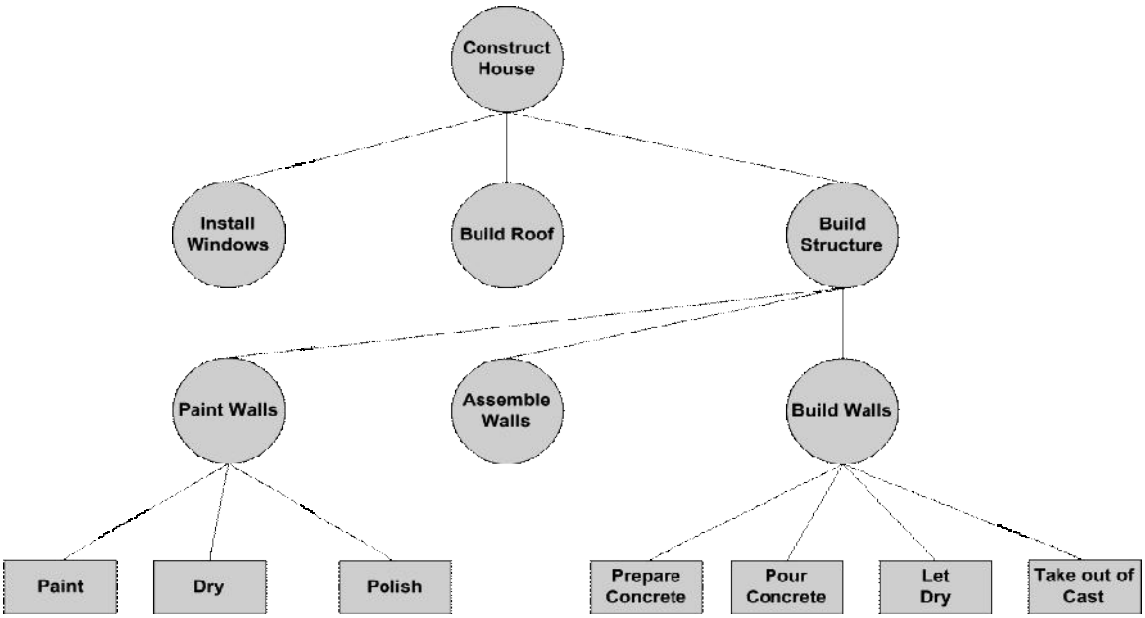

Figure 1: Example of ADT

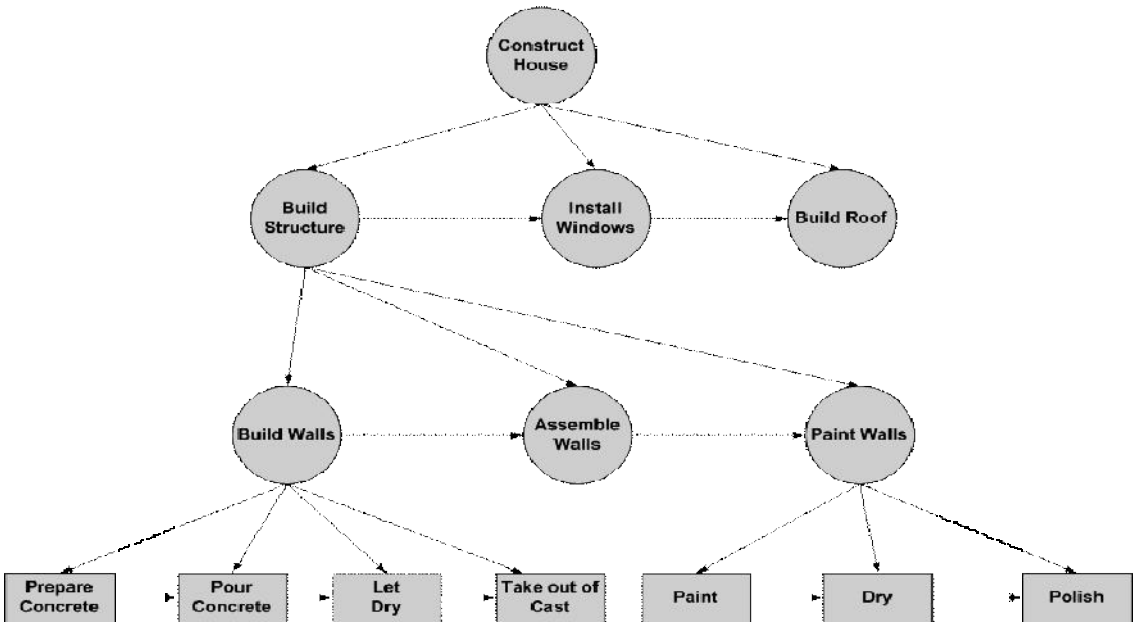

Figure 2: Example of AET

Table 10: New template functions

```
ExistC(a1,…,an):= (?occ_a1 … ?occ_an
                   ?root_a1 ?leaf_a1 …
                   ?root_an ?leaf_an)

RootLeafOcc(a1,…,an):=
 (root_occ ?root_a1 ?occ_a1)
 (leaf_occ ?leaf_a1 ?occ_a1)
          …
 (root_occ ?root_a1 ?occ_a1)
 (leaf_occ ?leaf_a1 ?occ_a1)

RootC(a,a1):= (root_occ ?root_a1 ?occ_a)

NextC(a,a1,…,an):=
 (next_subocc ?leaf_a1 ?root_a2 a)
                …
 (next_subocc ?leaf_an-1 ?root_an a)

LeafC(a,an):= (leaf_occ ?leaf_an ?occ_a)
```

Table 11: Translation Pattern P3

```
Declarations(a, a1,…,an)
(forall (?occ_a)
  (implies
    (and (occurrence_of ?occ_a a) (legal ?occ_a))
    (exists ExistC(a1,…,an)
      (and
        Legal_Occurrences(a1,…,an)
        Subactivities(a, a1,…,an)
        RootLeafOcc(a,a1,…,an)
        RootC(a,a1)
        NextC(a,a1,…,an)
        LeafC(a,an)))))
```

Table 12: Algorithm to generate the PSL code associated with an AET

```
Algorithm: Generates the PSL code associated with an AET
Input:
  T: AET
Output:
  psl: String that contains the PSL code associated with T
Method:
```

```
String psl = empty_string;
for each (node x in T in depth-first order)
   if (x is complex)
     psl = psl + GSCGAP(x)
```

4 Translation Patterns: Concurrent Composition

When modeling "concurrency" it is important to first define the interpretation the modeler is giving to the term. Two acceptations of the term that are often found are:

a) A single processor must execute various processes. Each process is conceived of as a sequence of tasks (each of them usually considered to be *atomic* in the sense that their execution is not interrupted). The processor executes tasks from each process in an *interleaved* manner. A real-life example that conforms to this model is the way some people play chess with various opponents. The person in question would be the processor. The processes would be the games associated with each opponent. At any moment the player is playing with exactly one opponent. However, if a time line were plotted showing the progress of each process it would depict each game evolving in an interleaved way without overlapping.[4]

b) There are specialized processors for various classes of tasks, and they must execute various processes. Since it is usually the case that they must cooperate, there can be some dependencies among them. One way of exposing such dependencies is by building a "dependency graph". In the graph, there is a directed edge from u to v if and only if task u must finish before task v starts (i.e. v depends on u). A topological sort of this graph reveals independent threads of execution [3]. Each thread is a sequence of tasks and threads can be executed in *parallel* among themselves. A time line showing the progress of computation shows overlapping of execution among the threads. A real-life example that conforms to this model is a jazz ensemble. The processors would be the players and the processes would be the music they play.

When modeling processes using "fork/join" nodes in activity diagrams found in UML2 [4], the second interpretation seems to be the one that better suits their intended semantics. An example from the wine-making application domain can be seen in Figure 3. We are adopting this graphical notation to represent concurrent processes.

From the perspective of the PSL ontology, theory $\boldsymbol{T_{atomic}}$ introduces the concept of atomic activities as a specialization of the concept "Activity" and a generalization of the concept of "Primitive Activity". Additionally, this theory introduces the concept of concurrent composition of atomic activities "`conc (a1, a2)`" which defines a new atomic activity as the composition of a1 and a2. This new activity can be referred to as a whole in the description of a process.

The following section describes an approach to generate the PSL code associated with a process specified using fork/join nodes.

[4] Sometimes this is called simultaneous chess playing, even though none of the games progress simultaneously.

4.1 Specifying implicit parallelism in PSL

This approach is equivalent to expressing in PSL a dependency graph associated with activities. This implies that if two given activities are dependent among each other, then the use of "`min_precedes`" (of theory $\boldsymbol{T}_{complex}$) expresses the enforcement of the required execution sequence.

Activities that are not dependent among each other are considered to proceed concurrently (or, more precisely, in parallel). To fix ideas, let us consider the diagram in Figure 4, which corresponds to a certain activity "`a`" whose subactivities are "`x`", "`y`", and "`z`". The PSL code associated with this diagram is presented in Table 13.

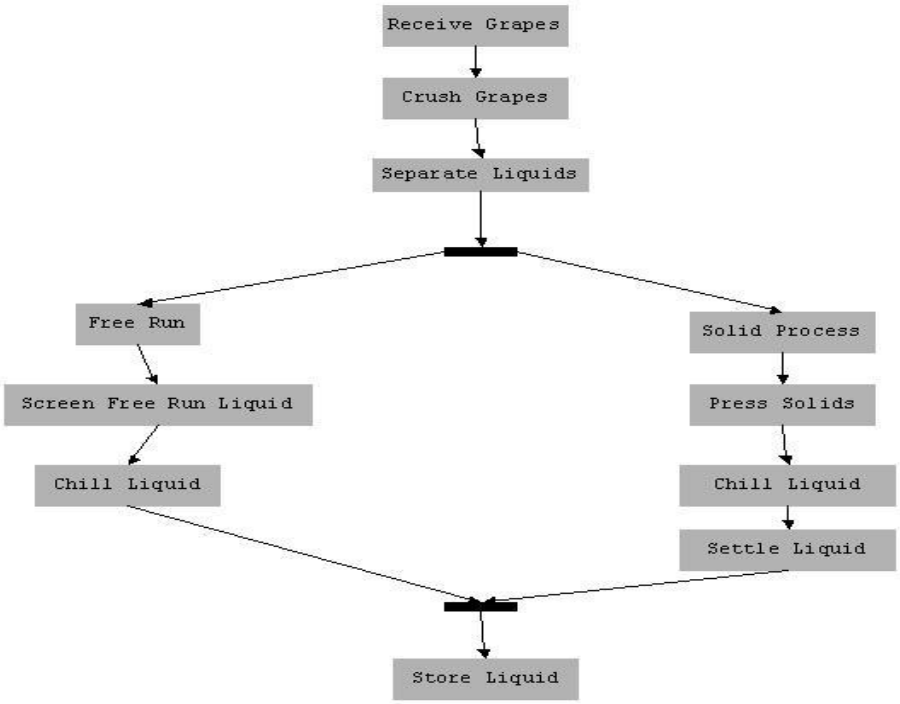

Figure 3: Process with fork/join nodes

For the sake of brevity, we have omitted the predicate "`(legal ?occ…)`" associated with occurrences "`?occ…`".

We define a "fork/join-to-fork/join path" (or fj-path) to be a sequence of activities that emanate from a fork-join node and culminate in a fork-join node. Figure 5 shows the fj-paths associated with the activity diagram in Figure 4. If we were able to decompose a given activity diagram into its fj-paths, we could simply apply pattern P3 (see Table 11) to generate the code associated with such paths, and then use the technique shown in Table 13 to enforce dependencies at the fork/join nodes. The algorithm in Table 14 summarizes the idea.

If we apply the algorithm to the diagram in Figure 3, we have:

§ The decomposition into fj-paths would yield {Fj-p1, Fj-p2, Fj-p3, Fj-p4}.
§ Fork/Join nodes are n1 and n2.
§ I(n1) = {Separate Liquids}
§ I(n2) = {Chill Liquid, Settle Liquid}
§ O(n1) = {Free Run, Solid Process}
§ O(n2) = {Store Liquid}

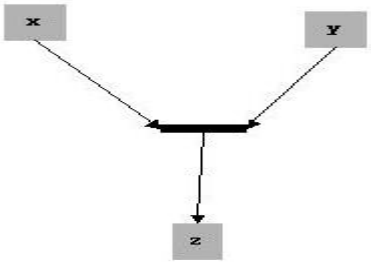

Figure 4: A simple process with concurrent activities

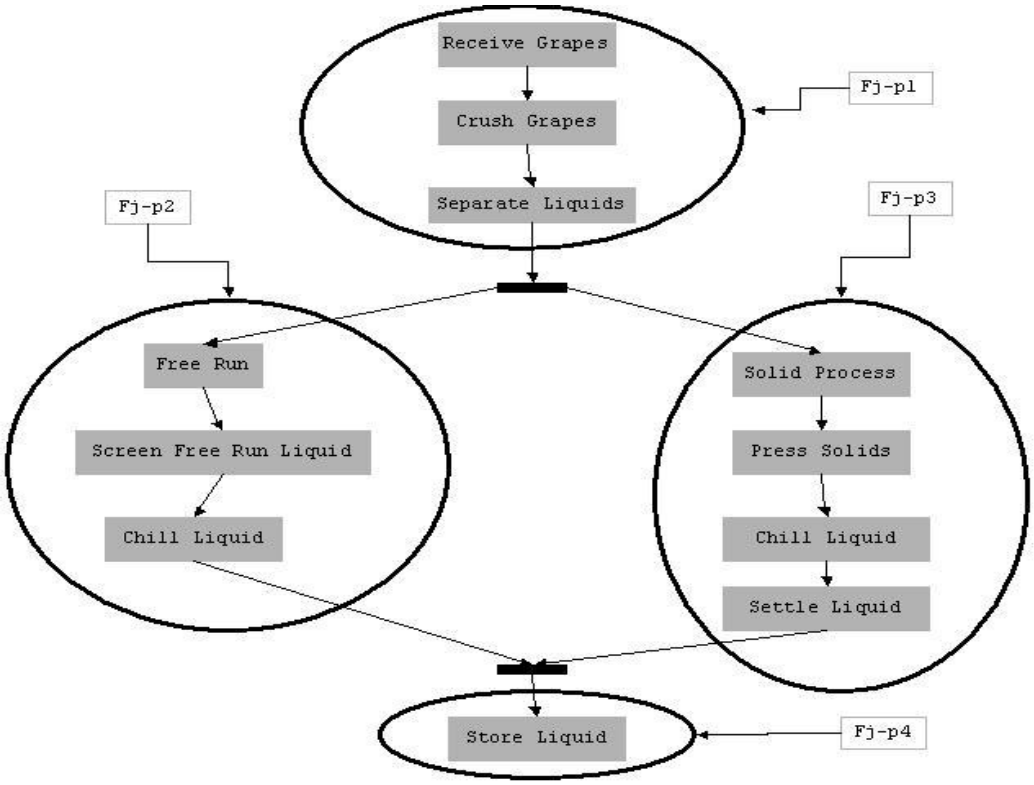

Figure 5: fj-paths associated with the diagram in Figure 3

Table 13: The PSL code associated with the diagram in Figure 4

```
(activity a)
(activity x)
(activity y)
```

```
(activity z)

(subactivity x a)
(subactivity y a)
(subactivity z a)

(forall (?occ_a)
 (implies (occurrence_of ?occ_a a)

    (exists (?occ_x ?occ_y ?occ_z)
     (and
      (occurrence_of ?occ_x x)
      (occurrence_of ?occ_y y)
      (occurrence_of ?occ_z z)
      (subactivity_occurrence ?occ_x ?occ_a)
      (subactivity_occurrence ?occ_y ?occ_a)
      (subactivity_occurrence ?occ_z ?occ_a)

      (min_precedes (leaf_occ ?occ_x)
                    (root_occ ?occ_z) a)
      (min_precedes (leaf_occ ?occ_y)
                    (root_occ ?occ_z) a))))
```

Table 14: Algorithm to generate the PSL code associated with a fork/join diagram

Algorithm: Generating the PSL code associated with a fork/join diagram that describes a process.
Input: fork/join diagram.
Output: string representing the PSL code.

Method:

```
Decompose the activity diagram into its fj-paths;

for each (fj-path f) do

  Generate the PSL code for f using translation pattern P3;

for each (fork/join node n in the diagram) do

  Let I(n) be the set of all activities ai such that ai is the last one
  in a sequence associated with a fj-path which is incident upon n;

  Let O(n) be the set of all activities bi such that bi is the first one
  in a sequence associated with a fj-path which emanates from n;

  for each (pair (ai, bj) such that ai is in I(n) and bj is in O(n)) do

     Generate the expression "min_precedes(leaf_occ(?occ_ai),
     root_occ(?occ_bj))" as part of the translation in the appropriate
     place;
```

Figure 6 illustrates the situation. The algorithm generates the code for the threads of execution that correspond to each fj-path. The application of the innermost loop implies generating PSL code that expresses the following facts:

- Separate Liquids come before Free Run and Solid Process.
- Chill Liquid and Settle Liquid come before Store Liquid.

5 Related Work

The PSL ontology is under frequent revisions and improvements. The latest version, along with supplemental material, can be found in the website maintained by NIST [2]. One of the main goals of PSL is to become a standard similar to STEP, "a comprehensive ISO standard (ISO 10303) that describes how to represent and exchange digital product information" [5]. In fact, "PSL is being standardized within Joint Working Group 8 of Sub-committee 4 (Industrial Data) and Sub-committee 4 (Manufacturing integration) of Technical committee ISO TC 184 (Industrial Automation Systems and Integration)." [2], [6].

PSL is intended to be a very general ontology for manufacturing processes and the typical approach to specify processes consists of imposing restrictions on configurations that describe valid scenarios associated with the world under modeling. Examples of this approach, including insights that reveal the intuition behind several PSL theories can be found in a paper by Gruninger [7], who leads the group working on this standardization effort. This paper is part of a recent compilation of ontologies [8].

In [9], Bock and Gruninger present UML conceptual models that describe the relationships among concepts of the PSL theories used in our paper. They also show examples that illustrate the use of PSL to associate semantics with simple UML activity diagrams. The patterns presented in this paper can be seen as extensions and generalizations of some of the examples presented in their paper.

The work reported in [9] also raises the issue of closure, namely whether additional restrictions should be added to process specifications to avoid sequences not explicitly allowed by the modeler. For instance, we mentioned that the expression `(next_subocc ?o1 ?o2 ?a)` (see Table 4) does allow activity occurrences between `?o1` and `?o2` as long as they are not a subactivity of `?a`. It is therefore possible to add terms to P3 (see Table 11) to explicitly disallow occurrences of any other activity but the intended ones. This can be accomplished by inserting the term shown in Table 15 right after `NextP(…)` in P3 (in Table 11).

Our definition of fj-paths is akin to that of "decision-to-decision" paths which has been amply used in the context of the automatic generation of test cases from graphs that model the flow of control of programs [10], [11].

We have applied the translation patterns discussed here to the implementation of a visual modeler that allows users to graphically represent processes and which automatically generates the PSL code associated with such representation.

6 Summary and Future Work

In this paper we have derived patterns that can be used to translate graphical representations of processes based on the arbitrary application of the subactivity relationship among activities, and which can be composed sequentially and concurrently. Such translations are expressed in PSL, an ontology that provides process modelers with concepts formalized by a lattice of first-order theories.The use of PSL enables the interoperability among disparate manufacturing process tools. Using PSL is not an easy task even for modelers who are well seasoned in specific application domains. Our approach completely liberates modelers from having to pursue such a task, for it bridges the gap between graphical representation of concepts, which are generally intuitively closer to the application domain, and the PSL ontology.

We are currently working on extending our visual modeler to include concepts such as resources, time constraints, inputs/outputs, and queues. We are also investigating on generating output that can be used to simulate manufacturing processes [12].

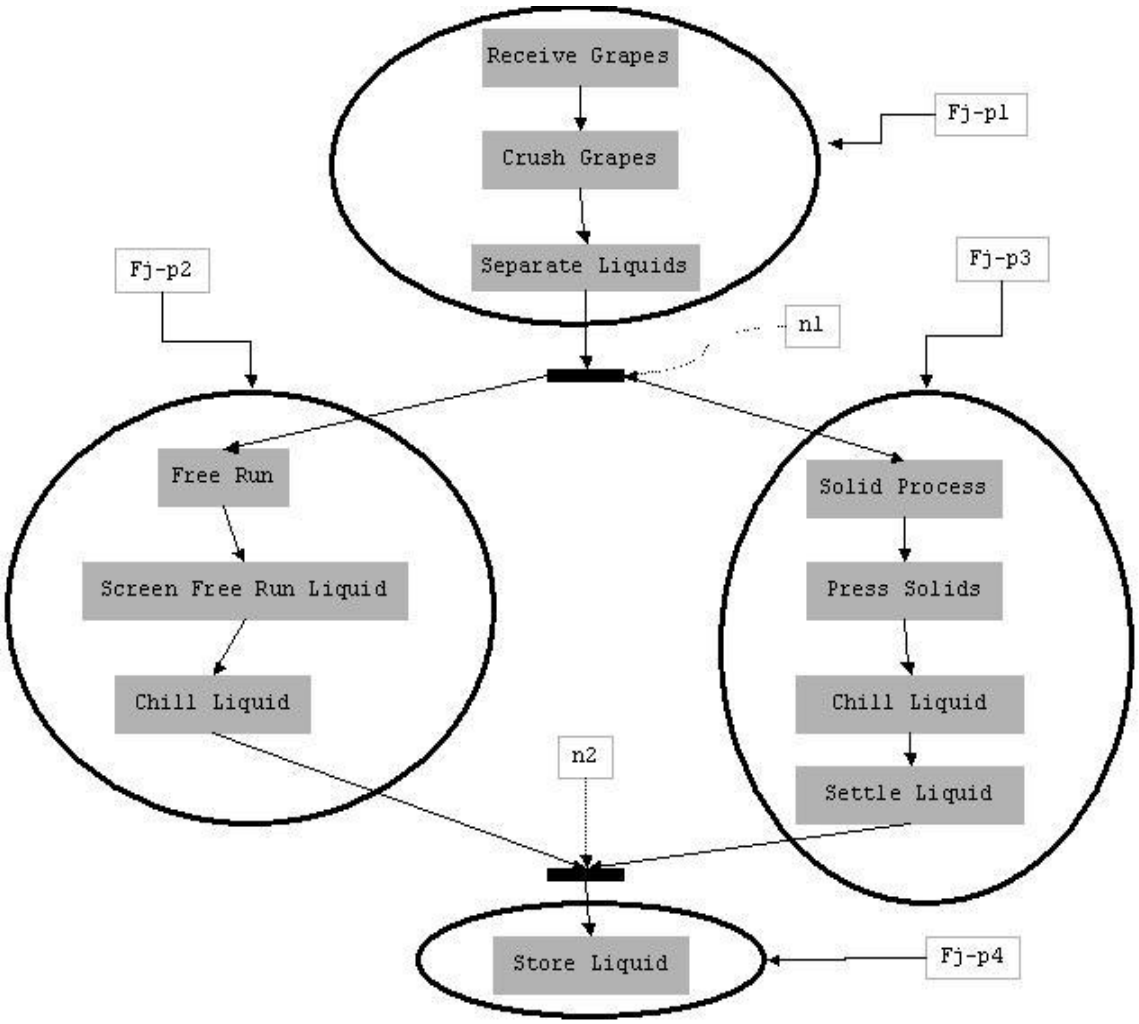

Figure 6: Applying the algorithm in Table 14 to the diagram in Figure 3

Table 15: Closure term

```
Closure(a1,…an):=
 (and
   (forall (?x)
     (implies
     (not (equal ?x (successor ?a2 ?occ_a1)))
     (not (legal ?x))))
          …
   (forall (?x)
     (implies
     (not (equal ?x (successor ?ai+1 ?occ_ai)))
     (not (legal ?x))))
          …
```

```
   (forall (?x)
     (implies
     (not (equal ?x (successor ?an-1 ?occ_an)))
     (not (legal ?x))))
 )
```

Acknowledgements

The authors would like to thank Michael Gruninger and Conrad Bock, from NIST, for their constructive feedback. The authors also thank the anonymous reviewers for their valuable comments. This work was partially funded by the National Institute of Standards and Technology as part of contract SB1341-03-W-0828.

References

[1] See http://www-ksl.stanford.edu/knowledge-sharing/kif/

[2] See http://www.mel.nist.gov/psl/

[3] T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, and C. Stein: "Introduction to Algorithms" (Second Edition). MIT Press. 2001.

[4] C. Bock: "UML2 Activity and Action Models. Part 3: Control Nodes", in Journal of Object Technology, vol. 2, no. 6, pp. 7-23. 2003.

[5] STEP Tools, Inc. See http://www.steptools.com

[6] ISO – TC184/SC4. See http://www.tc184-sc4.org/

[7] M. Gruninger: "Ontology of the Process Specification Language". In [8], pages 575-592.

[8] S. Staab, R. Studer (Editors): "Handbook on Ontologies". Springer. 2004.

[9] C. Bock, M. Gruninger: "PSL: A Semantic Domain for Flow Models". Software, Systems, and Modeling (On-Line First). November 2004. SpringerLink. See http://www.springerlink.com/

[10] J. C. Huang: "An Approach to Program Testing". ACM Computing Surveys, vol. 7, issue 3, pp. 113-128. 1975.

[11] W. R. Adrion, *et alia*: "Validation, Verification, and Testing of Computer Software".ACM Computing Surveys, vol. 14, issue 2, pp. 159-192. 1982.

[12] G. A. Hansen: "Automating Business Process Re-Engineering: Using the Power of Visual Simulation Strategies to Improve Performance and Profit" (Second Edition). Prentice-Hall. 1977.