The Knowledge Engineering Review, Vol. 13:1, 1998, 91–120
Printed in the United Kingdom

The Process Interchange Format and Framework

JINTAE LEE[1], MICHAEL GRUNINGER[2], YAN JIN[3], THOMAS MALONE[4], AUSTIN TATE[5], GREGG YOST[6] and OTHER MEMBERS OF THE PIF WORKING GROUP*

[1]*Department of Decision Sciences, University of Hawaii, 2401 Maile Way, Honolulu, HI 96822, USA*
[2]*Department of Industrial Engineering, University of Toronto,*
[3]*Department of Civil Engineering, Stanford University,*
[4]*Center for Coordination Science, MIT,*
[5]*Artificial Intelligence Applications Institute, University of Edinburgh,*
[6]*Digital Equipment Corporation*

Abstract

This document provides the specification of the Process Interchange Format (PIF) version 1.2. The goal of this work is to develop an interchange format to help automatically exchange process descriptions among a wide variety of business process modelling and support systems such as workflow software, flow charting tools, planners, process simulation systems and process repositories. Instead of having to write *ad hoc* translators for each pair of such systems each system will only need to have a single translator for converting process descriptions in that system into and out of the common PIF format. Then any system will be able to automatically exchange basic process descriptions with any other system. This document describes the PIF-CORE 1.2, i.e. the core set of object types (such as activities, agents and prerequisite relations) that can be used to describe the basic elements of any process. The document also describes a framework for extending the core set of object types to include additional information needed in specific applications. These extended descriptions are exchanged in such a way that the common elements are interpretable by any PIF translator, and the additional elements are interpretable by any translator that knows about the extensions. The PIF format was developed by a working group including representatives from several universities and companies, and has been used for experimental automatic translations among systems developed independently at three of these sites. This document is being distributed in the hopes that other groups will comment upon the interchange format proposed here, and that this format (or future versions of it) may be useful to other groups as well. The PIF Document 1.0 was released in December 1994, and the current document reports the revised PIF that incorporate the feedback received since then.

1 Introduction

More and more companies today are attempting to improve their business by engaging in some form of Business Process Redesign (BPR). BPR focuses on a "process view" of a business, and attempts to identify and describe an organization's business processes; evaluate the processes to identify problem areas; select or design new processes, possibly radically different from those currently in place; predict the effects of proposed process changes; define additional processes that will allow the organization to more readily measure its own effectiveness; and enact, manage and

*The PIF Working Group consists of people from industry and academia who are actively participating toward the development of PIF. There is also the PIF Comments Group (`pif-comments@mit.edu`), which consists of people who want to keep track of the progress on PIF without active participation. The PIF Home Page can be found at `http://soa.cba.hawaii.edu/pif/`. Your comments are always appreciated. Please address them to `pif-comments@mit.edu` or `jl@hawaii.edu`.

monitor the new processes. The goal is a leaner, more effective organization that has better insight into how it does business and how its business processes affect the organization's health. Successful BPR projects involve the cooperation of many people over extended time periods, including workplace analysts, systems engineers, and workers at all levels of the organization.

Computer applications that support one or more aspects of BPR are becoming increasingly common. Such applications include:

- Modelling tools that help a workplace analyst identify and describe an organization's processes.
- Process editors and planning aids to synthesize new processes or to modify existing processes.
- Process library browsers that help organizations find new processes that might better meet their needs.
- Process animators and simulators that help organizations visualize the effects of existing processes or potential new processes.
- Workflow management tools that help workers follow business processes.
- Outcomes analysis tools that help organizations monitor the effectiveness of their processes.

No single application supports all aspects of a BPR engagement, nor is it likely that such an application will ever exist. Furthermore, applications that do support more than one aspect rarely do them all well. For example, a workflow tool may also provide some process simulation capabilities, but those additional capabilities are unlikely to be on par with the best dedicated simulation applications. This is to be expected—building an application that supports even one of these aspects well requires a great deal of specialized knowledge and experience.

Ideally, then, a BPR team would be able to pick a set of BPR-support applications that best suits their needs: a process modeling tool from one vendor, a simulator from another, a workflow manager from another, and so forth. Unfortunately, these applications currently have no way to interoperate. Each application typically has its own process representation (often undocumented), and many applications do not provide interfaces that would allow them to be easily integrated with other tools.

Our goal with the PIF project is to support the exchange of process descriptions among different process representations. The PIF project supports sharing process descriptions through a description format called PIF (Process Interchange Format) that provides a bridge across different process representations. Tools interoperate by translating between their native format and PIF.[1]

There are several process representation languages such as LOTOS and IDEF, which could be potentially used for the purpose of sharing process descriptions. However, most of these languages are originally designed to satisfy a specific set of domain and task needs. PIF differs from them for being a translation language or an interlingua by design. As discussed in section 3, this difference yields a different set of design tradeoffs. Generality is preferred over efficiency. Extensibility is critical as any process representation language is unlikely to ever completely suit the needs of all applications that make use of business process descriptions. Therefore, in addition to the PIF format, we have defined a framework around PIF that accommodates extensions to the standard PIF description classes. The framework includes a translation scheme called Partially Shared Views that attempts to maximize information sharing among groups that have extended PIF in different ways.

[1]It is important to understand that a process specification in PIF will be utilized in a context where it is passed to a person, tool or system in such a way that the task to be performed on it is understood (e.g. analyse the specifications for certain features, perform a simulation using the specification, execute a process which meets the specification, avoid executing any process which meets the specification, etc.). This imperative information about the task to be performed with a PIF process specification is not represented in the specification itself, but should be considered as the context within which the specification is used.

It is also worth noting that PIF is not intended as a solution to the problem of multiple descriptions in translation. Any language with sufficient expressiveness will permit multiple ways of describing the same thing. PIF does not claim to be a canonical language in which all the sentences with the same meaning will be expressed in exactly one way. Hence, multiple descriptions in the source language may or may not translate to the same PIF expression depending on the way that the translator maps the source language to PIF.

The PIF framework aims to support process translation such that:

- Process descriptions can be automatically translated back and forth between PIF and other process representations with as little loss of meaning as possible. If translation cannot be done fully automatically, the human efforts needed to assist the translation should be minimized.
- If a translator cannot translate part of a PIF process description to its target format, it should:
  - —Translate as much of the description as possible (and not, for example, simply issue an error message and give up).
  - —Represent any untranslatable parts as such and present them in a way that lets a person understand the problem and complete the translation manually if desired.
  - —Preserve any uninterpretable parts so that the translator can add them back to the process description when it is translated back into PIF.

These requirements on the translators are very important. We believe that a completely standardized process description format is premature and unrealistic at this point. Therefore, as mentioned earlier, we have provided ways for groups to extend PIF to better meet their individual needs. As a result, we expect that PIF translators will often encounter process descriptions written in PIF variants that they can only partially interpret. Translators must adopt conventions that ensure that items they cannot interpret are available for human inspection and are preserved for later use by other tools that are able to interpret them. Section 6 describes PIF's Partially Shared Views translation scheme, which we believe will greatly increase the degree to which PIF process descriptions can be shared.

2 History and current status

The PIF project began in October 1993 as an outgrowth of the Process Handbook project (Malone et al., 1993) at MIT and the desire to share process descriptions among a few groups at MIT, Stanford, the University of Toronto and Digital Equipment Corporation. The Process Handbook project at the MIT Center for Coordination Science aims to create an electronic handbook of process models, their relations and their tradeoffs. This handbook is designed to help process designers analyse a given process and discover innovative alternatives. The Spark project at Digital Equipment Corporation aims to create a tool for creating, browsing and searching libraries of business process models. The Virtual Design Team (VDT) project at Stanford University aims to model, simulate and evaluate process and organization alternatives. The Enterprise Modeling project at the University of Toronto aims to articulate well-defined representations for processes, time, resources, products, quality and organization. These representations support software tools for modeling various aspects of enterprises in business process reengineering and enterprise integration.

In one way or another, these groups were all concerned with process modelling and design. Furthermore, they stood to benefit from sharing process descriptions across the different representations they used. For example, the Enterprise Modelling group might model an existing enterprise, use the Process Handbook to analyse its tradeoffs and explore its alternatives, evaluate the different alternatives via VDT simulation, and then finally, translate the chosen alternative back into its own representation for implementation.

Over the past years, through a number of face-to-face, email and telephone meetings, the PIF Working Group members have:

- Articulated the requirements for PIF.
- Specified the core PIF process description classes.
- Specified the PIF syntax.
- Elaborated the Partially Shared View mechanism for supporting multiple, partially overlapping class hierarchies.

- Created and maintained a database of the issues that arose concerning PIF's design and the rationales for their resolutions.
- Implemented several translators, each of which translated example process descriptions (such as a portion of the ISPW-6 Software Change Process) between PIF and a group's own process representation.
- Used the translators to port process descriptions across heterogeneous representations (between Kappa PC representation and Lotus Notes representation of process handbook data).

Based on this work, the PIF Document 1.0 was released on December 1994. Since then, we have received a number of questions and comments on topics that range from individual PIF constructs to how certain process descriptions can be represented in PIF. We have been also assessing the adequacy of the PIF 1.0 by testing it against more complex process descriptions than before. AIAI at the University of Edinburgh also joined the PIF Working Group at this time, bringing along their interests in planning, workflow and enterprise process modelling. The Edinburgh group is also providing a valuable service as a liaison between the PIF group and the Workflow Management Coalition as well as the AI planning community (in particular the DARPA/Rome Laboratory Planning Initiative) which has been concerned with the activity representation issues for a while. The Ontology Group at the Stanford University has also joined the PIF Working Group, and is sharing the lessons from its experiences in providing the ontology library and the editor.

The revised structure of PIF reflects the lessons extracted from these external and internal input. In particular, two points emerged clearly. One is that the PIF-CORE has to be reduced to the bare minimum to enable translation among those who cannot agree on anything else. The other point is the importance of structuring PIF as a set of modules that build on one another. This way, groups with different expressive needs can share a subset of the modules, rather than the whole monolithic set of constructs. As a result, the PIF-CORE has been reduced to the minimum that is necessary to translate the simplest process descriptions and yet has built-in constructs for "hanging off" modules that extend the core in various ways.

Recently we have been working with other groups whose aim is also to share process descriptions though in their own domains. The goal of the Process Specification Language (PSL) project at NIST is to facilitate process sharing in the domain of manufacturing. It has finished compiling the list of requirements that a process specification language should satisfy, and is evaluating the existing process representations with respect to these requirements. We are working with the PSL group in assessing these requirements and comparing the different process representations in the hope that the PSL will be compatible with PIF. The goal of the Workflow Process Description Language (WPDL) is to be an interlingua for sharing workflow descriptions. We have compared the WPDL with PIF, identified similarities and differences, and are communicating with them to make both PIF and WPDL interoperable.

3 PIF overview

The PIF ontology has grown out of the efforts of the PIF Working Group to share process descriptions among the group members' various tools. We have used the following guidelines in developing this hierarchy:

- Generality is preferred over computational efficiency when there is a tradeoff, for the reason that PIF is an interchange language, not a programming language designed for efficient execution[2]. Therefore, the organization of the entity classes is not necessarily well-suited to performing any particular task such as workflow management or process simulation. Instead, our goal has been

[2]Although PIF is not an execution language, an execution language can be PIF-compliant. That is, an execution language can be designed to include the PIF constructs as a part of it so that it does not require a translator to process a set of PIF specifications.

to define classes that can express a wide variety of processes, and that can be readily translated into other formats that may be more suitable for a particular application.

- The PIF constructs should be able to express the constructs of some existing common process representations such as IDEF (SADT) or Petri nets.
- PIF should start with the minimal set of classes and then expand only as it needs to. The minimal set was decided at the first PIF Workshop (October 1993) by examining those constructs common to some major existing process representations and to the process representations used by members of the PIF Working Group.
- Additions to the standard PIF classes could be proposed by anybody, but the proposal had to be accompanied by concrete examples illustrating the need for the additions. The Working Group decided, through discussions and votes if necessary, whether to accept the proposal. PIF allows groups to define local extensions at will (see section 6), so new classes or attributes should be added to the standard PIF classes only if they seem to be of sufficiently general usefulness.

A PIF process description consists of a set of frame definitions (cf. Appendix I and II), which are typically contained in a file. Each frame definition refers to an entity instance and is typed (e.g. ACTIVITY, OBJECT, TIMEPOINT) and they form a class hierarchy (see Figure 1). A frame definition has a particular set of attributes defined for it. Each of the attributes describes some aspect of the entity. For example, a PERFORMS definition has an Actor and an Activity attributes that specifies who is performing which activity. The instance of a frame definition has all the attributes of all of its superclasses, in addition to its own attributes. For example, all the instances of ACTIVITY have the Name attribute, since ENTITY, which is a superclass of ACTIVITY, has the Name attribute.

When an attribute of one frame has a value that refers to another frame, the attribute represents a relationship between the two instances that the two frames refer to. For example, if the Begin attribute of ACTIVITY-1 takes TIMEPOINT-32 as its value, then the Begin attribute represents a relationship between the ACTIVITY-1 and TIMEPOINT-32 instances. The value of a given attribute in a PIF file holds independent of time. Figure 2 depicts the relationships among the PIF classes. Section 5 describes all of the current PIF classes.

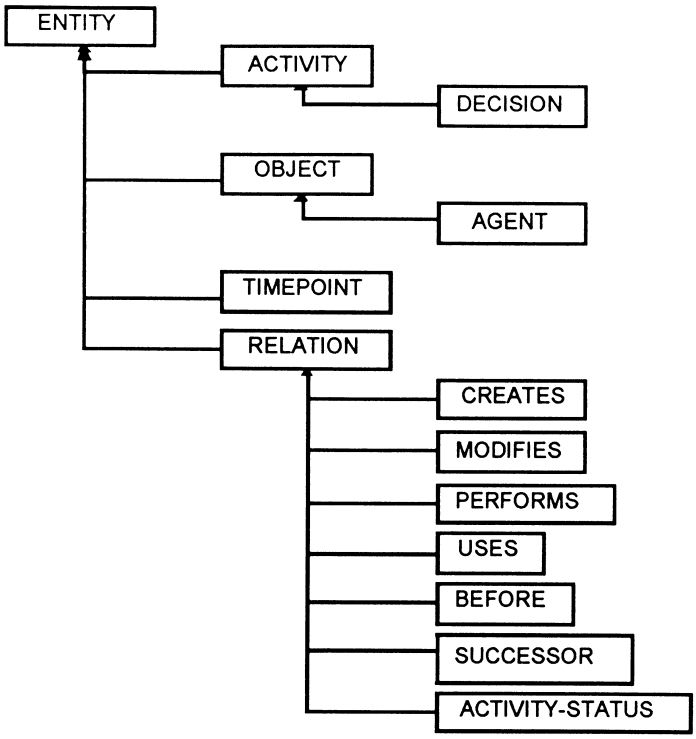

Figure 1 The PIF class hierarchy

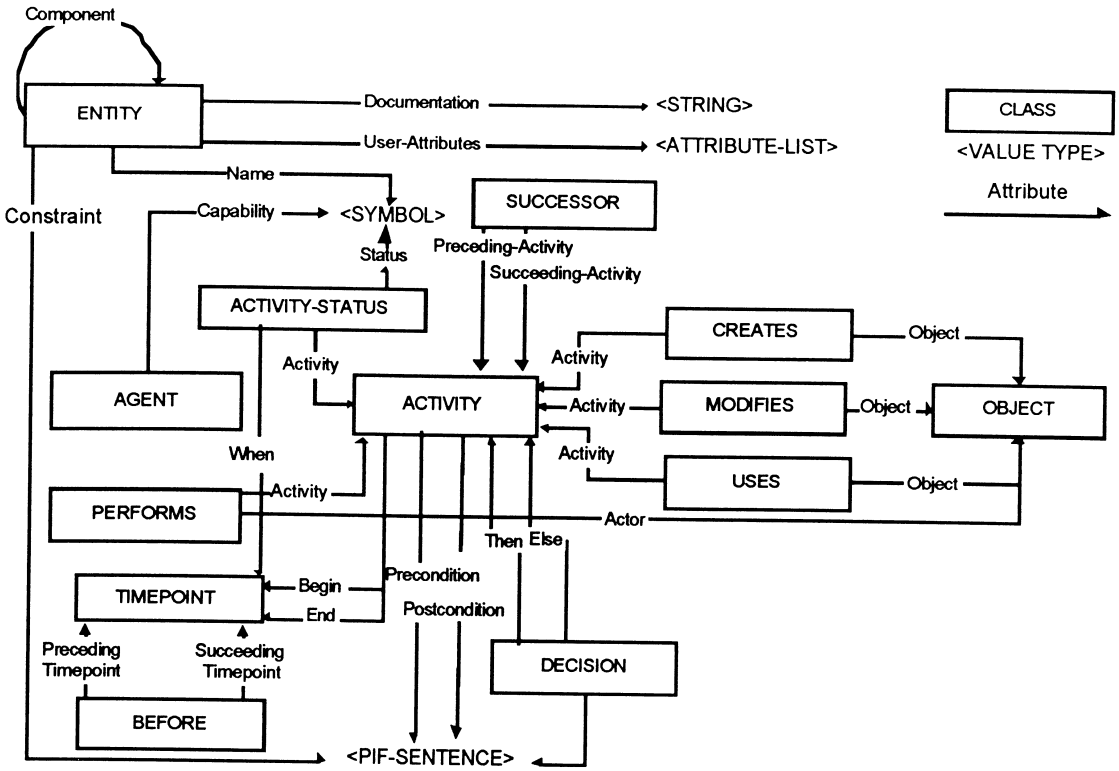

Figure 2 Relationships among PIF classes

An attribute in a PIF entity can be filled with the following and only the following PIF expressions: a literal value of a PIF primitive value type or an expression of a composite value type. The PIF primitive value types consist of: NUMBER, STRING, and SYMBOL:

- NUMBER: A numeric value. The NUMBER type is subdivided into INTEGER and FLOAT types.
- STRING: A sequence of characters.
- SYMBOL: Symbols are denoted by character sequences, but have somewhat different properties than strings. PIF symbols are a much-simplified version of symbols in the Lisp programming language (Steele, 1990). In PIF, the main difference between strings and symbols is that symbols have their references (e.g. variables and constants) and are not case-sensitive unless specially quoted, whereas strings are always case-sensitive.

The PIF composite value types consist of: LIST and PIF-SENTENCE.

- LIST: A list.
- PIF-SENTENCE: A logical expression that evaluates to TRUE or FALSE.

An object variable is of the form, object-name[.slot-name]*, which refers to either the object named or the object which is the value of the named slot (or, if there are more than one slot-names specified, the object which is the value of the named slot of the object which is the value of the next named slot, and so on.) Appendix I describes PIF's syntax, including the syntax of the primitive literals as well as the composite value types.

4 Rationales

The goal of PIF is to support maximal sharing of process descriptions across heterogeneous process representations. To better serve this goal, PIF consists of not a monolithic set of constructs, but a partially ordered set of modules. A module can build on other modules in that the constructs in a

module are specializations of the constructs in the other modules. One can adopt some modules but not others, depending on one's expressive needs. Hence, a module typically contains a set of constructs that are useful for a particular domain or a type of task. More details of this module structure are discussed in section 6.

The PIF-CORE, on the other hand, consists of the minimal set of constructs necessary to translate simple but non-trivial process descriptions. There is the usual trade-off between simplicity and expressiveness. The PIF-CORE could have been chosen to contain only the constructs necessary for describing the simplest process descriptions such as a precedence network. Such a PIF-CORE then would not be able to translate many process descriptions. On the other hand, the PIF-CORE could have contained constructs sufficient for expressing the information contained in process descriptions of richer complexity. Such a PIF-CORE then would contain many constructs that may not be needed for many simpler descriptions. The PIF-CORE strikes a balance in this tradeoff by first collecting process descriptions, starting from the simplest and continuing with more complex until we have reasonably many of them, and then by looking for a set of constructs that can translate the process descriptions in this collection. The following describes the rationales for each of the constructs in the PIF-CORE. The attributes of each of these constructs are described in section 5. Appendix II provides the complete specification of the PIF-CORE 1.2.

In PIF, everything is an ENTITY; that is, every PIF construct is a specialization of ENTITY. There are four types of ENTITY: ACTIVITY, OBJECT, TIMEPOINT, and RELATION. These four types are derived from the definition of process in PIF: a process is a set of ACTIVITIES that stand in certain RELATIONS to one another and to OBJECTS over TIMEPOINTS.

The following provides intuitive rationales for each of these four constructs. Their precise semantics, however, are defined by the relations they have with other constructs (cf. section 5).

ACTIVITY represents anything that happens over time. DECISION, which represent conditional activities, is the only special type of ACTIVITY that the PIF-CORE recognizes. In particular, the PIF-CORE does not make any distinction among process, procedure, or event. A TIMEPOINT represents a particular point in time, for example "Oct. 2, 2.32 p.m. 1995" or "the time at which the notice is received." An OBJECT is intended to represent all the types of entities involved in a process description beyond the other three primitive ones of ACTIVITY, TIMEPOINT and RELATION. AGENT is a special type of OBJECT.

RELATION represents relations among the other constructs. The PIF-CORE offers the following relations: BEFORE, SUCCESSOR, CREATES, USES, MODIFIES and PERFORMS.

BEFORE represents a temporal relation between TIMEPOINTS. SUCCESSOR (Activity-1, Activity-2) is defined to be the relation between ACTIVITIES where BEFORE (Activity-1.End, Activity-2.Begin) holds. It is provided as a shorthand for simple activity precedence relations.

CREATES, USES and MODIFIES represent relations between ACTIVITY and OBJECT. In these relations, the object is assumed to be created, used, modified at some non-determinate timepoint(s) in the duration of the activity (i.e. between its Begin and its End timepoint inclusively). Hence, the object would have been created, used or modified by the End timepoint, but no commitment is made as to when the object is actually created, used or modified. PERFORMS represents a relation between OBJECT (normally an AGENT specialization) and ACTIVITY. In the PERFORMS relation, the actor is assumed to perform the activity at some non-determinant timepoint(s) in the duration of the activity (possibly for the whole duration, but not necessarily). We understand that there are other possible interpretations of these relations. For example, we might want to specify that a given actor is the only one who performs the activity during the whole activity interval. Such a specification, however, will require a PSV extension of the PIF-CORE (for example, by introducing a relation such as PERFORMS-EXCLUSIVELY; cf. section 6).

SUCCESSOR in PIF may not correspond exactly to the notions of successor as used in some workflow or enactment systems because it is common in these systems to bundle into a single relationship a mixture of temporal, causal and decomposition relationships among activities. PIF provides precise, separate relationships for all three of these activities-to-activity specifications. For example, the temporal relationship is specified with the BEFORE relation, the causal relation with

the Precondition and Postcondition attributes of ACTIVITY, and the decomposition relation with the Component attribute. Its intention is to allow the exact meaning to be communicated. Hence, one might have to combine some of these constructs to capture exactly the meaning of SUCCESSOR as used in one's own system.

The attribute value of a PIF-CORE object holds independent of time (i.e. no temporal scope is associated with an attribute value in the PIF-CORE). Any property of an object which can change over time, should be represented by a RELATION that links the property to a timepoint. An example of one such RELATION in the PIF-CORE is ACTIVITY-STATUS which is used to represent the status (e.g. DELAYED, PENDING) of an ACTIVITY at different times. The ACTIVITY-STATUS is provided in the PIF-CORE because it is the one example of a dynamic property of those objects commonly used in process modeling and workflow systems and modelled in the PIF-CORE. Other properties of those objects included in the PIF-CORE are, for the most part, true for all time. As mentioned before, it is possible to extend the PIF-CORE to express additional temporally scoped properties by introducing additional RELATIONS. It is also possible to add temporally scoped version of the static attributes already in the PIF-CORE. In this case, any such static attributes actually specified in a PIF file holds true for all time.

The attribute value of a PIF object can be one of the PIF value types specified above. The PIF primitive value types consist of NUMBER, STRING and SYMBOL. The PIF composite value types are LIST and PIF-SENTENCE. LIST is used for conveying structured information that is not to be evaluated by a PIF interpreter, but simply passed along (e.g. as in the User-Attribute attribute of ENTITY). PIF-SENTENCE is used to specify a condition that is either true or false, as required, for example, for the Precondition and the Postcondition attributes of ACTIVITY.

PIF-SENTENCE is a logical expression that may include variables, quantifiers, and the Boolean operators for expressing conditions or constraints. A PIF-SENTENCE is used in the Constraint slot of ENTITY, the Precondition and the Postcondition slots of ACTIVITY, and the If slot of DECISION. A variable in a PIF-SENTENCE takes the following positions in the three dimensions that define the possible usage.

1 The scope of the variable is the frame. That is, variables of the same name within a frame definition are bound to the same object, whereas they are not necessarily so if they occur in different frames.
2 A variable is assumed to be bound by an implicit existential quantifier.
3 The constraints on variables in a frame definition are expressed in the Constraints slot of that frame. These constraints are local to the frame.

These positions are expected to be extended by some PSV Modules. Some PSV modules will extend the scope of a variable beyond a single object. Some will introduce explicit existential and universal quantifiers; yet others will allow global constraints to be stated, possibly by providing an object where such global constraints that hold across all the objects in a PIF file (e.g. All purchase order must be approved by the finance supervisor before sent out.).

Notable absence

We have decided not to include ROLE because a role may be defined wherever an attribute is defined. For example, the concept of RESOURCE is a role defined by the Resource attribute of the USE relation. Any object, we view, is a resource if it can be USEd by an ACTIVITY. As a consequence, we have decided not to include ROLE or any construct that represents a role, such as RESOURCE. ACTOR is not included in PIF because it is another role-concept, one defined by the Actor attribute of the PERFORMS relation. Any object, as long as it can fill the Actor attribute, can be viewed as an ACTOR. Hence, we resolved that explicit introduction of the constructs such as ACTOR or RESOURCE is redundant and may lead to potential confusions. We should note, however, that the PIF-CORE provides the construct AGENT, which is not defined by a role an entity plays but by its inherent characteristic, namely its capability (for example, of making intelligent decisions in various domains).

5 Alphabetic class reference

ACTIVITY

Parent classes: ENTITY

Attribute	*Value Type*	*Multiple Values Allowed*
Component	ACTIVITY	Yes
Precondition	PIF-SENTENCE	No
Postcondition	PIF-SENTENCE	No
Begin	TIMEPOINT	No
End	TIMEPOINT	No

Attribute descriptions

- **Component**: the subactivities of the activity. For example, if the activity is ''Develop Software'', its Component may include: ''Design Software'', ''Write Code'', ''Debug Software'', and so on. The field is inherited from ENTITY, but here it is restricted so that its values must all be ACTIVITY entities.
- **Precondition**: the conditions that have to be satisfied at the Begin timepoint of the activity before it can get executed. For example, a precondition of the activity ''Run Software'' might state that the executable code must be available. Such conditions are expressed using PIF-SENTENCES, as described in Appendix I.
- **Postcondition**: the conditions that are true at the End timepoint of the activity. For example, a postcondition of the activity ''Run Software'' might be that a log file has been updated. Such conditions are expressed using PIF-SENTENCES, as described in Appendix I.
- **Begin**: the TIMEPOINT at which the activity begins.
- **End**: the TIMEPOINT at which the activity ends.

In the PIF-CORE, the condition in the Precondition is to be true before the Begin timepoint of the ACTIVITY. Similarly, the condition in the Postcondition is to be true after the End timepoint of the ACTIVITY. This requirement may be relaxed later in PSV modules (cf. section 6) to allow the precondition and the postcondition to be stated relative to other time points.

Many preconditions and postconditions can be expressed in PIF without using the Precondition and Postcondition attributes of ACTIVITY. For example, the USE relation between an activity A and an object O implies that one of A's preconditions is that R is available. In general, the Precondition and Postcondition attributes of ACTIVITY should only be used to express conditions that cannot be expressed any other way in PIF. Doing so will maximize the degree to which a process description can be shared with others.

ACTIVITY-STATUS

Parent classes: RELATION

Attribute	*Value Type*	*Multiple Values Allowed*
Activity	ACTIVITY	Yes
Status	SYMBOL	Yes
When	TIMEPOINT	No

Attribute descriptions

- **Activity**: the activity whose status is being specified.
- **Status**: the status being specified such as DELAYED and PENDING.
- **When**: the timepoint at which the status of the activity is being specified.

AGENT

Parent classes: OBJECT -> ENTITY

Attribute	*Value Type*	*Multiple Value Allowed*
Capability	SYMBOL	Yes
Component	AGENT	Yes

Attribute descriptions

- **Capability**: its possible values are SYMBOLS that represent the kinds of skills the agent is capable of providing. The symbols are supplied by the source language and simply preserved across translations by PIF. A PSV Module may introduce a restricted set of symbol values.

An AGENT represents a person, group, or other entity (such as a computer program) that participates in a process. An AGENT is distinguished from other ENTITIES by what it is capable of doing or its skills.

BEFORE

Parent classes: RELATION -> ENTITY

Attribute	*Value Type*	*Multiple Values Allowed*
Preceding-Timepoint	TIMEPOINT	No
Succeeding-Timepoint	TIMEPOINT	No

Attribute descriptions

- **Preceding timepoint**: the time point that is before the Succeeding Timepoint
- **Succeeding timepoint**: the time point that is after the Preceding Timepoint.

BEFORE is a relation between TIMEPOINTS not between ACTIVITIES. A shorthand for a common example of the BEFORE relation is available via the SUCCESSOR relation.

CREATES

Parent classes: RELATION -> ENTITY

Attribute	*Value Type*	*Multiple Values Allowed*
Activity	ACTIVITY	No
Object	OBJECT	Yes

Attribute descriptions

- **Activity**: the activity that creates the object. The object is assumed to be created at some non-determinate timepoint(s) between its Begin and its End timepoint inclusive.
- **Object**: the object that the activity creates.

DECISION

Parent classes: ACTIVITY -> ENTITY

Attribute	*Value Type*	*Multiple Values Allowed*
If	PIF-SENTENCE	No
Then	ACTIVITY	Yes
Else	ACTIVITY	Yes

Attribute descriptions

- **If**: the condition being tested to decide which successor relations to follow. Such conditions are expressed using PIF-SENTENCES, as described in Appendix I
- **Then**: the activity to follow if the condition in the If field holds (that is, if the PIF-SENTENCE in the If field evaluates TRUE).
- **Else**: the activity to follow if the condition in the If field does not hold (that is, if the PIF-SENTENCE in the If field evaluates to FALSE).

A DECISION is a special kind of activity that represents conditional branching. If the PIF Sentence in its If attribute is TRUE, the activity specified in its Then attribute follows. If not, the activity in its Else attribute follows. If the Else attribute is empty, it means no activity follows the DECISION activity in the case where the decision condition is false. If more than one activity in a process is dependent on a decision, then they may be included in the multiple value "then" or "else" attributes. To ease description of a complex sub-process which is dependent on the decision, it is possible to describe a set of sub-activities (and any ordering or other constraints on them) in a separate process and to include that process itself within the "then" or "else" attributes.

ENTITY

Parent classes: None. ENTITY is the root of the PIF class hierarchy.

Attribute	*Value Type*	*Multiple Values Allowed*
Name	STRING	No
Documentation	STRING	No
Component	ENTITY	Yes
Constraint	PIF-SENTENCE	No
User-Attribute	LIST	No

Attribute descriptions

- **Name**: the entity's name.
- **Documentation**: a description of the entity.
- **Component**: this attribute is used to specify an homogeneous aggregate of the type itself. For example, in an AGENT object, this attribute can be used to specify that the agent is in fact a group of sub-agents. In an ACTIVITY object, this attribute is used to specify its subactivities that make up the activity. If one needs to specify a group of objects of different types, then one can do so by going up to an object of their common ancestor type and specify them in the Component attribute of this object. When interpreted as a relation, this relation holds between the entity and each value, not between the entity and the set of all the values.
- **Constraint**: this attribute is used to specify any constraint that should be true of the other attribute values in the current entity, e.g. constraints on the variables.
- **User-attribute**: this attribute is used to store additional *ad hoc* attributes of an entity that are not part of its class definition. For example, a process modelling application might allow users to specify additional attributes for AGENT entities that are not included in AGENT's PIF definition—the user might want to add an attribute recording the AGENT's age, for example. Such additional attributes can be stored in the User-Attribute attribute, which all PIF entities inherit from ENTITY. Another common use is in the Partially Shared Views translation scheme that we propose for interchanging PIF files (see section 6). Each value of User-Attribute is a list containing an attribute name and its value(s). For example, an OBJECT entity might have (User-Attribute (Color RED GREEN) (Weight 120))

MODIFIES

Parent classes: RELATION -> ENTITY

Attribute	*Value Type*	*Multiple Values Allowed*
Activity	ACTIVITY	No
Object	OBJECT	Yes

Attribute descriptions

- **Activity**: the activity that modifies the object. The object is assumed to be modified at some non-determinate timepoint(s) between its Begin and its End timepoint inclusive.
- **Object**: the object that the activity modifies.

OBJECT

Parent classes: ENTITY
Attribute descriptions: No attribute.

An OBJECT is an entity that can be used, created, modified or used in other relationships to an activity. This includes people (represented by the AGENT subclass in PIF), physical materials, time, and so forth. The PIF Working Group has discussed adding OBJECT attributes such as Consumable, Sharable, and so forth, but so far no decision has been made on what attributes are appropriate.

PERFORMS

Parent classes: RELATION -> ENTITY

Attribute	*Value Type*	*Multiple Values Allowed*
Actor	OBJECT	Yes
Activity	ACTIVITY	Yes

Attribute descriptions

- **Actor**: the object that performs the activity.
- **Activity**: the activity that is performed. The actor is assumed to perform the activity at some non-determinate timepoint(s) between its Begin and its End timepoint inclusive.

RELATION

Parent classes: ENTITY
Attribute descriptions: no attribute.

RELATION entities have no attributes of their own. PIF uses it as an abstract parent class for more specific relation classes such as USES and PERFORMS.

SUCCESSOR

Parent classes: RELATION -> ENTITY

Attribute	*Value Type*	*Multiple Values Allowed*
Preceding-Activity	ACTIVITY	No
Succeeding-Activity	ACTIVITY	Yes

Attribute descriptions

- **Preceding-Activity**: the preceding activity.
- **Succeeding-Activity**: the succeeding activity.

SUCCESSOR with the Preceding-Activity ACTIVITY-1 and the Succeeding-Activity ACTIVITY-2 is exactly the same as BEFORE with Preceding-Timepoint TP-1 and Succeeding-Timepoint TP-2, where TP-1 is the Begin timepoint of ACTIVITY-2 and TP-2 is the End timepoint of ACTIVITY-1. That is, the SUCCESSOR relation is true if the ACTIVITY-1 ends before the ACTIVITY-2 begins.

TIMEPOINT

Parent classes: ENTITY
Attribute descriptions: No attribute.

TIMEPOINT represents a point in time. In PIF-CORE, it is used, for example, to specify the Begin and End times of an Activity or the Preceding and Succeeding time points of the BEFORE relation.

USES

Parent classes: RELATION -> ENTITY

Attribute	*Value Type*	*Multiple Values Allowed*
Activity	ACTIVITY	No
Object	OBJECT	Yes

Attribute descriptions

- **Activity**: the activity that uses the object from its Begin timepoint to its End timepoint. The USES relation is true from the Begin to the End timepoint of the activity. The object is assumed to be used at some non-determinate timepoint(s) between its Begin and its End timepoint inclusive.
- **Object**: the object that the activity uses.

6 Extending PIF

PIF provides a common language through which different process representations can be translated. Because there will always be representational needs local to individual groups, however, there must also be a way to allow local extensions to the description classes while supporting as much sharing as possible among local extensions. The Partially Shared Views (PSV) scheme has been developed for the purpose (Lee & Malone, 1990). PSV integrates different ways of translating between groups using different class hierarchies (e.g. pairwise mapping, translation via external common language, translation via internal common language) so as to exploit the benefits of each when most appropriate.

A PSV Module is a declaration of PIF entities which specialize other entities in the PIF-CORE or other PSV modules on which it builds. The class definitions in a PSV Module cannot delete or alter the existing definitions but can only add to them. Examples of PSV Modules are given at the end of this section. A group of users may adopt one or more PSV Modules as necessary for its task.

A group using a PSV module translates a PIF object X into their native format as follows:

1. If X's class (call it C) is known to the group and the group has developed a method that translates objects of class C into their native format, then apply that translation method. C is known to the group if either C is defined in one of the PSV Modules that the group has adopted or the group has set up beforehand a translation rule between C and a type defined in one of the PSV Modules adopted.
2. Otherwise, translate X as if it were an object of the nearest parent class of C for which (1) applies (its parent class in the most specific PSV Module that the group and the sender group both share, i.e. have adopted).

This translation scheme allows groups to share information to some degree even if they do not support identical class hierarchies. For examples, suppose that Group A supports only the standard PIF AGENT class, and that Group B in addition supports an EMPLOYEE subclass. When Group

A receives a process description in Group B's variation on PIF, they can still translate any EMPLOYEE objects in the description as if they were AGENT objects. What happens to any information that is in an EMPLOYEE object that is not in a generic AGENT object? That will vary according to the sophistication of the translator and the expressive power of the target process representation. However, the translator will preserve the additional information so that it can be viewed by users and reproduced if it is later translated back into PIF.

For example, suppose EMPLOYEE has a "Medical-plan" attribute, which is not part of the AGENT object in the PIF-CORE. Then Group A's translator would

- Translate any Medical-plan attributes into a form that the user could view in the target system (even if it only as a textual comment)[3] AND
- When the information is re-translated into PIF in the future (from Group A's native format), it is emitted as an EMPLOYEE object with the same value for the Medical-plan attribute (and not simply as an AGENT object with no Medical-plan attribute). MIT researchers are currently investigating this general problem of preserving as much information as possible through "round trips" from one representation to another and back (Chan, 1995).

Translators that can follow these conventions will minimize information loss when processes are translated back and forth between different tools. The details of PSV can be found in Lee & Malone (1990). In the current version of PIF, each PIF file begins with a declaration of the class hierarchy for the objects described in the file. PSV uses this class hierarchy to translate objects of types that are unknown to a translator. To eliminate the need for PIF translators to do any other inheritance operations, however, all PIF objects should contain all of their attributes and values. For instance, even if the value of a given attribute is inherited without change from a parent, the attribute and value are repeated in the child.

As the number of PSV modules grows large, we need a mechanism for registering and coordinating them so as to prevent any potential conflict such as naming conflict. Although the exact mechanism is yet to be worked out, we are envisioning a scenario like the following. The user who needs to use PIF would first consult the indexed library of PSV modules, which documents briefly the contents of each of the modules and the information about the other modules it presupposes. If an existing set of modules does not serve the user's purpose in hand and a new PSV module has to be created, then the information about the new module and its relation to other modules is sent to a PSV registration server, which then assigns to it a globally unique identifier and updates the indexed library. We foresee many other issues to arise such as whether any proposed PSV module should be accepted, if not who decides, whether to distinguish an *ad hoc* module designed for temporary quick translation between two local parties from a well-designed module intended for global use, and so on. However, rather than addressing these issues in this document, we will address them in a separate document as we gain more experience with PSV modules.

To date, two PSV Modules have been specified: Temporal-Relation-1 and IDEF-0 Modules. The Temporal-Relation-1 Module is specified in Appendix III. It extends the core PIF by adding all possible temporal relations that can hold between two activities (cf. Figure 3). The IDEF-0 Module adds the constructs necessary for translating between IDEF-0 descriptions and PIF. IDEF-0 is a functional decomposition model, which however has been historically used widely as a process model description language. IDEF-0 has been used in various ways with no single well-defined semantics. Hence, the IDEF-0 PSV Module supports translation between PIF and one particular version of IDEF-0. It introduces two additional relations, USES-AS-RESOURCE and USES-AS-CONTROL, as specializations of the USES relation. They are meant to capture the Control and Mechanism input of IDEF-0. The Input and Output relations of IDEF-0 may be translated into PIF by using the Precondition and Postcondition attribute of ACTIVITY. The IDEF-0 Module is specified in Appendix IV. The mapping between IDEF and PIF is shown in Figure 4. These modules

[3]If the target representation happens to be PIF (albeit Group A's variant of it), the uninterpretable attributes would be stored as text in the User-Attribute attribute, which all PIF entities have.

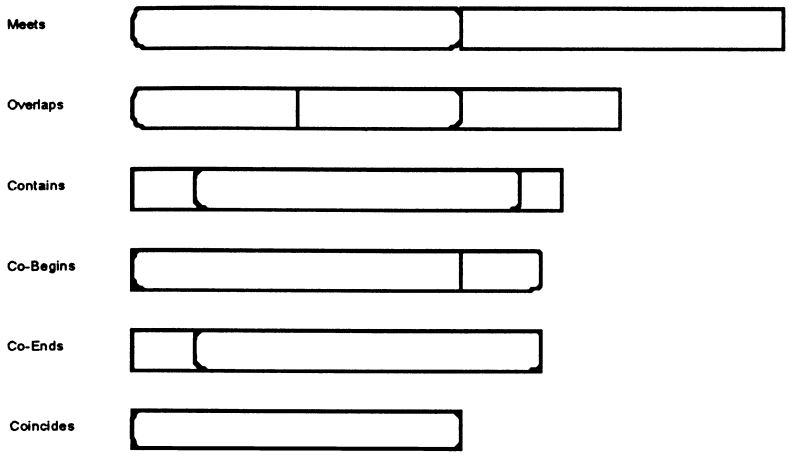

Figure 3 Possible temporal relations between two activities

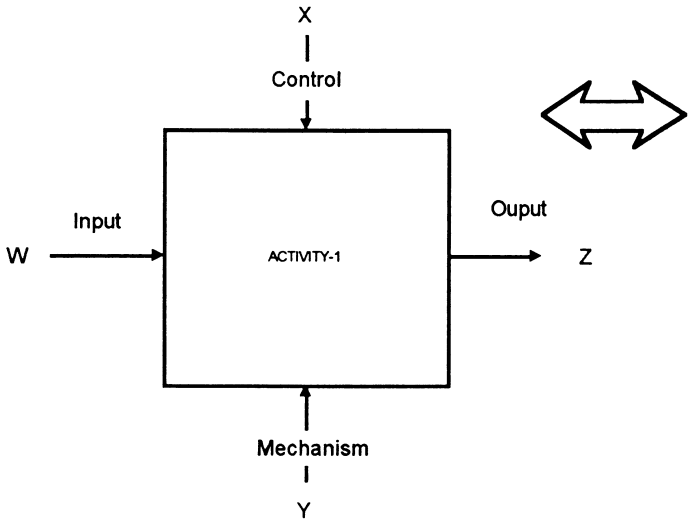

Figure 4 Mapping between IDEF-0 and PIF constructs

have not been officially registered. They are presented here only to provide examples of PSV modules. We are soliciting further inputs before we register them.

7 Future directions

Following the release of PIF version 1.2, PIF developments are expected to follow the following directions:

- We plan to coordinate further development of PIF with other knowledge sharing projects so as to produce compatibility, if not convergence, among the meta-models produced. We will continue working closely with the NIST PSL Group in order to make PSL and PIF compatible. We also plan to work with the International Workflow Management Coalition (`http://www.aiai.ed.ac.uk/WfMC`), whose goal is to produce interoperability among workflow applications. We have been also talking to the people in the Knowledge Sharing Initiatives (Neches et al., 1991), which has produced KIF (Knowledge Interchange Format) described earlier, tools and protocols for sharing knowledge bases and Web-based ontology libraries among other things. We plan to intensify these coordination efforts through more structured and active forms such as workshops and regular meetings.
- We plan to elaborate on the PIF extension mechanism. We need to discuss and work out the

details on such issues as Who can propose and accept PSV modules in which domain and how the modules should be named, registered, organized, and accessed. We also need to carefully lay out the space of PSV modules by identifying an initial set of generally useful ones extending the PIF-CORE. Again, this work will require close interactions with the other knowledge sharing groups as well as the experts in various domains. We hope to pursue this objective as a part of pursuing the first objective of coordination with other groups.

- To use PIF to share process descriptions automatically, we need a PIF-translator for each of the local process representations involved. For example, each of the groups represented in the PIF Working Group built a translator for translating between PIF 1.0 and its own representation. Building PIF-translators are ultimately the responsibility of individual groups who want to use PIF. However, we would like to provide a toolkit that will help future groups build PIF-translators.

Appendix A PIF syntax

The syntax of PIF adopts that of KIF (Knowledge Interchange Format—see Genesereth and Fikes, 1992). KIF is a language that has been developed by the Interlingua Working Group, under the DARPA (Defense Advanced Research Projects Agency) Knowledge Sharing Initiative (Neches et al., 1991) to facilitate knowledge sharing. Its features include: formally defined declarative semantics, expressive power to represent knowledge required for a typical application knowledge base, and a structure that enables semi-automatic translation into and out of typical representation languages. PIF also adopts the frame syntax, which is an extension of the KIF syntax for representing object-based knowledge. Figure 5 shows the BNF for the frame syntax.

There are several reasons why PIF adopts KIF syntax:

- There is little overhead involved in using KIF for our current purpose. Any interchange format we choose should provide a structured way of specifying classes, instances, values, and value restrictions. KIF has a relatively simple and well-developed syntax for all of these. Currently, we make little use of KIF beyond its syntax. For example, we do not currently use any of KIF's extensive provisions for describing the formal semantics of a set of object classes.
- KIF is already a proposed interchange format and much effort has gone into making it a good general-purpose interchange format. In particular, KIF has a well-defined formal semantics, which helps reduce the ambiguity that might arise in translating between PIF and other process description formats.

```
(define-frame <frame-name>
   :own-slots ((<own-slot-spec>)*)
   :template-slots ((<template-slot-spec>)*)
   )

   <own-slot-spec> := = (<slot-name> <value-spec> +)
   <template-slot-spec> := = (<slot-name> <facet-or-value-spec> +)
   <value-spec> := = <PIF-Value>

<PIF-Value> := = <number> | <string> | <symbol> | <list> | <PIF-sentence>
   <list> := = ({ (<string> <string>)} +)
   <PIF-sentence> := = cf. Figure 6.

<frame-name> := = symbol naming a class, relation, function, or object
<slot-name> := = symbol naming a binary relation or unary function
<facet> := = symbol naming a slot constraint relation, such as SLOT-VALUE-TYPE
```

Figure 5 The BNF for the PIF frame syntax (taken and modified from the Ontolingua frame syntax)

- Using KIF allows us to exploit existing resources. For example, the KIF repository of generic and domain-specific knowledge structures has been growing. These knowledge bases include formalizations of configuration design, engineering mathematics, job assignment, and bibliographic information), all of which would be accessible to any PIF translators that were extended to process the more general KIF structures that are not included in basic PIF.
- There are tools available, such as Ontolingua (Gruber, 1993), that can facilitate encoding knowledge in KIF and translation to and from other representations. Ontolingua provides higher-level knowledge-description constructs than standard KIF, as well as translation services between KIF and other knowledge representations (for example, LOOM, KEE, Epikit).

KIF syntax is based on Lisp (Steele, 1990), but little in KIF requires subscription to the Lisp philosophy. We could view the KIF syntax simply as a standard way of specifying a structured list of information. PIF uses a simplified version of the KIF syntax (cf. Appendix A).

A PIF file begins with a version number of the PIF being used, followed by a description of the class hierarchy for objects in the file, and then by descriptions of all of the object instances. Figure 5 shows the BNF grammar for PIF expressions. The grammar uses the common conventions that non-terminals are enclosed in angle brackets, * denotes zero or more repetitions, + denotes one or more repetitions, optional items are enclosed in square brackets, vertical bars separate alternatives, place holders for primitive literals are in uppercase (for example, NUMBER), and everything else is a literal constant. A PIF expression is case-insensitive. Appendix C contains a very simple example PIF file.

:OWN-SLOTS—slots on the object itself, as opposed to the instances of a class. If the object is a class, then own slots describe relationships and properties of the class as a whole, such as its superclasses and documentation. If the object is an instance, then own slots describe properties of the object, including the relation instance-of.

Own slots are binary relations applied to frames, with the frame inserted as a first argument. For example:

```
(define-frame frame-1
  :own-slots ((instance-of class-2)))
```

translates to the KIF sentence

```
(instance-of frame-1 class-2)
```

:TEMPLATE-SLOTS—only make sense if the frame is an instance of CLASS, because template slots describe properties of instances of the class. For example, the template slot spec

```
(slot-2 (SLOT-VALUE-TYPE type-3)) for the frame class-1 translates to the KIF sentence
(slot-value-type class-1 slot-2 type-3)
```

which is a second-order way of saying

```
(forall ?c (=> (and (instance-of ?c class-1)
                    (defined (slot-2 ?c)))
               (instance-of (slot-2 ?c) type-3)))
```

A value of a template slot is a downward inherited value (it is a slot value for all instances of the class). For frame class-1, the template slot spec

```
(slot-2 value-3) translates into the KIF sentence
(inherited-slot-value class-1 slot-2 value-3)
```

The following set of facets are recognized by PIF: SLOT-CARDINALITY SLOT-VALUE-TYPE.

PIF allows two kinds of comments:

- Comments that begin with a semicolon and end at the end of the same line.
- Comments that begin with #| and end with |#. This kind of comment can be nested.

So, for example, #| ...#| ... |# ... |# is a valid comment.

The primitive literal types in the grammar are NUMBER, STRING, SYMBOL and PIF-SENTENCE. NUMBER, STRING and SYMBOL are defined very much like the corresponding concepts in the Common Lisp programming language (Steele, 1990).

An object variable is of the form, OBJECT[.SLOT*]. If there is no slot specified, i.e. OBJECT, then it refers to the object by that name. If there is a single slot specified, i.e. OBJECT.SLOT, then it denotes the slot value of the object. If there are two slots specified, i.e. OBJECT.SLOT-1.SLOT-2, then it denotes the slot value of an object which is the slot value of the object if there are two slots specified. And so on with multiple slots specified. If the object is SELF, it refers to the object within which the object variable is used.

A PIF-SENTENCE is a logical expression for representing different constraints for PIF objects and relations. Within the PIF-CORE, a PIF-SENTENCE is used in the following ways:

- Constraint slot of ENTITY.
- Precondition and Postcondition slots of ACTIVITY.
- If slot of a DECISION activity.

For the PIF-CORE, this class is restricted to sentences composed of terms with variables and logical connectives. Syntactically, a PIF-SENTENCE is a restricted class of KIF sentences. Figure 6 shows the BNF specification of PIF-SENTENCE.

The PIF-CORE makes a specific assumption about the quantifiers and the scope of variables in a PIF-SENTENCE. This assumption is characterized below along three dimensions in the treatment of variables. Each of these dimensions can be considered to be a set of design choices that are adopted within The PIF-CORE or some PSV module. The class of KIF sentences corresponding to PIF-SENTENCE within a given PSV module is defined by the particular design choices adopted within the module.

1. All variables within a PIF-SENTENCE must be quantified either universally (in which case the sentence must be satisfied for all values of the variable) or existentially (in which case, the sentence must be satisfied for some value of the variable). The issue to be addressed in the syntactic specification of PIF-SENTENCE is whether or not to explicitly include quantifiers, since many process ontologies do not include explicit quantifiers. In addition, the presence of

```
<pif-sentence> ::= <relsent> |
          <logsent>
<relsent> ::= (<relconst> <term>*) |
          (<funconst> <term>* <term>)
<logsent> ::= (not <sentence>) |
          (and <sentence>*) |
          (or <sentence>*) |
          (=> <sentence>* <sentence>) |
          (<=> <sentence> <sentence>)
<term> ::= <indvar> | <constant>
<indvar> ::= ?<objconst> | <objconst>[.<slotconst>]*
<objconst> ::= SYMBOL
<slotconst>::= SYMBOL
<funconst> ::= ; There is no function constant in the PIF-CORE but extensions of the PIF-CORE are expected to introduce their own constants.
<relconst> ::= + | - | = | <> | > | < | <= | <= ; These eight symbols exhaust the relation constants in the PIF-CORE but extensions of the PIF-CORE are expected to introduce their own constants.
<constant> ::= <objconst> | <slotconst> | <funconst> | <relconst>
```

Figure 6 BNF specification of PIF-SENTENCE

quantifiers within an expression would require more sophisticated translators for parsing arbitrary KIF sentences.

Within the PIF-CORE, we adopt conventions for the use of quantifiers in a PIF-SENTENCE. Variables that appear in the Precondition and Postcondition slots of ACTIVITY or in the If slot of a DECISION activity are assumed to be implicitly existentially quantified. Variables that appear in the Constraints slot of an ENTITY are assumed to be implicitly universally quantified.

Additional PSV modules can allow for explicit quantification and richer expressiveness.

2. The second dimension defines the scope of the variable within the PIF-SENTENCE. For example, three options along this dimension are:

   (i) the scope of a variable is restricted to the object in whose slot it appears;
   (ii) the scope of the variable is (syntactically) specified;
   (iii) the scope of the variable is global over the set of objects in a PIF file.

   Within the PIF-CORE, we assume that the scope of a variable is the object, that is, that the variables of the same name within a frame definition are bound to the same values, whereas variables of the same name may be bound to different values if they appear in different frames.

3. The third dimension defines how variables are allowed to be used:

   (i) variables only refer to slot values;
   (ii) variables can refer to arbitrary objects in the PIF file; examples of this is a constraint such as "All agents must clean their work area after completing their activities," and "All purchase orders must be approved by the finance supervisor before being sent out."

   Within the PIF-CORE, a PIF-SENTENCE is a constraint that is local to the frame definition in which it appears.

The define-hierarchy construct that appears at the beginning of every PIF file is used by the Partially Shared Views (PSV) translation scheme described in section 6. The PSV scheme must be able to determine the parent classes of any classes that a given translator does not know how to translate. A define-hierarchy construct has the form (define-hierarchy LIST) where LIST is a nested list of class ids. The first id in the list is the id of the root class (ENTITY, in PIF). The remaining elements are sub-hierarchy definitions, in the same form. So, for example,

```
(define-hierarchy
(A
  B
  (C
    D
    (E))
   (F
    G)))
```

defines the class tree shown in Figure 7.

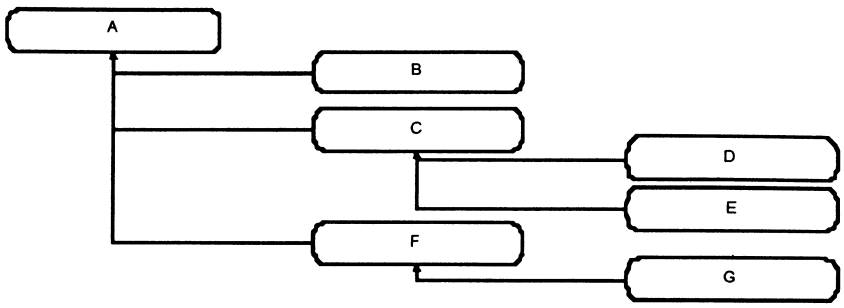

Figure 7 A leaf class can be denoted either by a symbol or by a list with no subhierarchy definitions (for example, E)

Appendix B PIF-CORE specification

```
(define-frame ENTITY
:own-slots
((Name ''ENTITY'')
(Documentation ''The PIF Root Class''))
:template-slots
((Component (slot-value-type ENTITY) (slot-cardinality MULTIPLE))
(Constraint.(slot-value-type PIF-SENTENCE))
(User-Attributes (slot-value-type LIST) (slot-cardinality MULTIPLE)))
)

(define-frame ACTIVITY
:own-slots
((Name ''ACTIVITY'')
(Subclass-Of ENTITY)
(Documentation ''ACTIVITY represents anything that happens over time. The PIF-CORE
makes no distinction among process, procedure, or event.''))
:template-slots
((Component (slot-value-type ACTIVITY (slot-cardinality MULTIPLE))
(Begin (slot-value-type TIMEPOINT))
(End (slot-value-type TIMEPOINT))
(Precondition (slot-value-type PIF-SENTENCE))
(Postcondition (slot-value-type PIF-SENTENCE)))

(define-frame TIMEPOINT
:own-slots
((Name ''TIMEPOINT'')
(Subclass-Of ENTITY)
(Documentation ''TIMEPOINT represents a particular point in time, for example ''Oct.
2, 2.32 p.m. 1995'' or ''the time at which the notice is received.''))
:template-slots
((Component (slot-value-type TIMEPOINT) (slot-cardinality MULTIPLE)))
)

(define-frame OBJECT
:own-slots
((Name ''OBJECT'')
(Subclass-Of ENTITY)
(Documentation ''An OBJECT is intended to represent all the types of entities involved
in a process description beyond the other three primitive ones of ACTIVITY, TIMEPOINT,
and RELATION.''))
:template-slots
((Component (slot-value-type OBJECT) (slot-cardinality MULTIPLE)))
)

(define-frame RELATION
:own-slots
((Name ''RELATION'')
(Subclass-Of ENTITY)
(Documentation ''RELATION represents relations among the other constructs.''))
:template-slots
((Component (slot-value-type RELATION) (slot-cardinality MULTIPLE)))
)

(define-frame DECISION
:own-slots
((Name ''DECISION'')
(Subclass-Of ACTIVITY)
(Documentation ''DECISION, which represent conditional activities, is the only special
type of ACTIVITY that the PIF-CORE recognizes.''))
:template-slots
((Component (slot-value-type DECISION) (slot-cardinality MULTIPLE))
(If (slot-value-type PIF-SENTENCE))
(Then (slot-value-type ACTIVITY) (slot-cardinality MULTIPLE))
```

```
(Else (slot-value-type ACTIVITY) (slot-cardinality MULTIPLE)))
)

(define-frame AGENT
:own-slots
((Name ''AGENT'')
(Subclass-Of OBJECT)
(Documentation ''AGENT is a special type of OBJECT which has some capability such as
of making decisions.''))
:template-slots
((Component (slot-value-type AGENT) (slot-cardinality MULTIPLE))
(Capability (slot-value-type ENTITY) (slot-cardinality MULTIPLE)))
)

(define-frame CREATES
:own-slots
((Name ''CREATES'')
(Subclass-Of RELATION)
(Documentation ''creation relation between an activity and an object.''))
:template-slots
((Component (slot-value-type CREATES) (slot-cardinality MULTIPLE))
(Activity (slot-value-type ACTIVITY))
(Object (slot-value-type OBJECT) (slot-cardinality MULTIPLE)))
)

(define-frame MODIFIES
:own-slots
((Name ''MODIFIES'')
(Subclass-Of RELATION)
(Documentation ''modification relation between an activity and an object.''))
:template-slots
((Component (slot-value-type MODIFIES) (slot-cardinality MULTIPLE))
(Activity (slot-value-type ACTIVITY)
(Object (slot-value-type OBJECT) (slot-cardinality MULTIPLE)))
)

(define-frame USES
:own-slots
((Name ''USES'')
(Subclass-Of RELATION)
(Documentation ''use relation between an activity and an object.''))
:template-slots
((Component (slot-value-type USES) (slot-cardinality MULTIPLE))
(Activity (slot-value-type ACTIVITY)
(Object (slot-value-type OBJECT) (slot-cardinality MULTIPLE)))
)

(define-frame PERFORMS
:own-slots
((Name ''PERFORMS'')
(Subclass-Of RELATION)
(Documentation ''perform relation between an actor and an object.''))
:template-slots
((Component (slot-value-type PERFORMS) (slot-cardinality MULTIPLE))
(Actor (slot-value-type OBJECT))
(Activity (slot-value-type ACTIVITY)))
)

(define-frame BEFORE
:own-slots
((Name ''BEFORE'')
(Subclass-Of RELATION)
(Documentation ''Precedence relation between two timepoints''))
:template-slots
((Component (slot-value-type BEFORE) (slot-cardinality MULTIPLE))
```

```
(Preceding-Timepoint (slot-value-type TIMEPOINT))
(Succeeding-Timepoint (slot-value-type TIMEPOINT)))
)

(define-frame SUCCESSOR
:own-slots
((Name ''SUCCESSOR'')
(Subclass-Of RELATION)
(Documentation ''Precedence relation between two activities, i.e. the End timepoint of
the Preceding Activity comes before the Begin timepoint of the Succeeding
Activity.''))
:template-slots
((Component (slot-value-type SUCCESSOR) (slot-cardinality MULTIPLE))
(Preceding-Activity (slot-value-type ACTIVITY))
(Succeeding-Activity (slot-value-type ACTIVITY)))
)

(define-frame ACTIVITY-STATUS
:own-slots
((Name ''ACTIVITY-STATUS'')
(Subclass-Of RELATION)
(Documentation ''ACTIVITY-STATUS is a RELATION which specifies the status of a process
at a timepoint.''))
:template-slots
((Activity (slot-value-type ACTIVITY))
(Status (slot-value-type ENTITY) (slot-cardinality MULTIPLE)
(When (slot-value-type TIMEPOINT))
)
```

Appendix C The Temporal-Relations-1 PSV module

```
Name: Temporal-Relations-1 PSV Module
Version: 0.1
Uses: (PIF-CORE, 1.2)
;; The Name of the Module together with its Version number currently provides a unique
identifier.

(define-frame MEETS
:own-slots
((subclass-of RELATION)
(Documentation ''The Succeeding Activity begins at the moment when the Preceding
Activity ends.''))
:template-slots
((Preceding-Activity ?act-1)
(Succeeding-Activity ?act-2)
(Constraint (= ?act-1.End ?act-2.Begin)))
)

(define-frame OVERLAPS
:own-slots
((subclass-of RELATION)
(Documentation ''The Succeeding Activity begins at the moment before the Preceding
Activity ends.''))
:template-slots
((Preceding-Activity ?act-1)
(Succeeding-Activity ?act-2)
(Constraint (< ?act-2.Begin ?act-1.End )))
)

(define-frame COINCIDES
:own-slots
((subclass-of RELATION)
```

```
(Documentation ''The two activities begin and end at the same moments.''))
:template-slots
((Activity-1 ?act-1)
(Activity-2 ?act-2)
(Constraint (AND (= ?act-1.Begin ?act-2.Begin) (= ?act-1.End ?act-2.End))))
)

(define-frame CONTAINED
:own-slots
((subclass-of RELATION)
(Documentation ''Contained Activity begins after the Containing Activity begins and
ends before the Containing Activity ends.''))
:template-slots
((Contained-Activity ?act-1)
(Containing-Activity ?act-2)
(Constraint (AND (< ?act-2.Begin ?act-1.Begin) (< ?act-1.End ?act-2.End))))
)

(define-frame CO-BEGINS
:own-slots
((subclass-of RELATION)
(Documentation ''The two activities begin together.''))
:template-slots
((Activity-1 ?act-1)
(Activity-2 ?act-2)
(Constraint (= ?act-1.Begin ?act-2.Begin))
)

(define-frame CO-ENDS
:own-slots
((subclass-of RELATION)
(Documentation ''The two activities end together.''))
:template-slots
((Activity-1 ?act-1)
(Activity-2 ?act-2)
(Constraint (= ?act-1.End ?act-2.End))
)
```

Appendix D The IDEF-0 PSV module

```
Name: IDEF-0
Version: 0.1
Uses: (PIF-CORE, 1.2)

(define-frame USES-AS-RESOURCE
:own-slots
((Subclass-Of USES)
(Documentation ''The relation that should capture the Mechanism Input arrow of the
IDEF-0 diagram.''))
:template-slots
((Activity (slot-value-type ACTIVITY)) ; inherited from USES
(Object (slot-value-type OBJECT))) ; inherited from USES
)

(define-frame USES-AS-CONTROL
:own-slots
((Subclass-Of USES)
(Documentation ''The relation that should capture the Control Input arrow of the IDEF-
0 diagram.''))
:template-slots
((Activity (slot-value-type ACTIVITY)) ; inherited from USES
(Object (slot-value-type OBJECT))) ; inherited from USES
)
```

Appendix E An example PIF file

This appendix gives an example PIF file for a very over-simplified design process. This example design project is composed of five activities and a design team with four designers. The precedence network of the activities and the task responsibilities of the AGENTs are illustrated in Figure 8.

```
(version 1.2)
(define-hierarchy
 (entity
     (activity
          decision)
     (TIMEPOINT)
     (object)
     (agent)
     (relation
          before
          creates
          uses
          modifies
          performs
          successor)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Project and Team definitions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-frame EXAMPLE-PROJECT
:own-slots
((Instance-Of ACTIVITY)
(Documentation ''A project is the top-level activity of
this activity elaboration hierarchy. The
Component attribute lists the sub-activities of
the project.'')
(Name ''The Example Project Process'')
(Component ARCHITECTURE-DESIGN-1 ELECTRICAL-DESIGN-2
MECHANICAL-DESIGN-3 DESIGN-REVIEW-4)
))
(define-frame DESIGN-TEAM-1
:own-slots
```

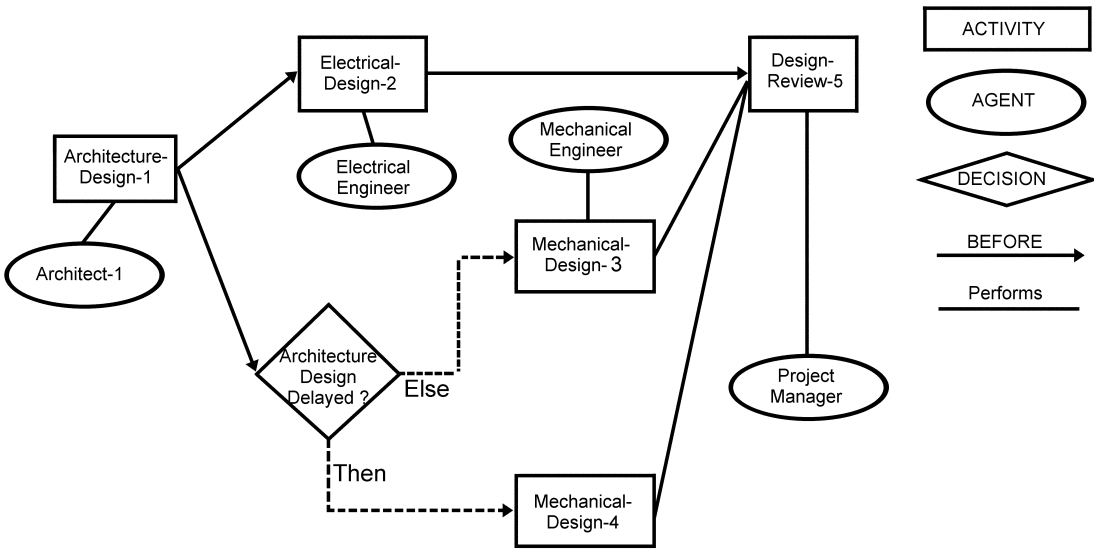

Figure 8 Precedence network of activities and AGENT responsibilities

```
((Instance-Of AGENT)
(Documentation ''A project team is composed of AGENTS, which as a whole can be viewed
as an AGENT itself.'')
(Name ''Project Design Team'')
(Component ARCHITECT-1 ELECTRICAL-ENGINEER-2
MECHANICAL-ENGINEER-3 PROJECT-MANAGER-4)
))

(define-frame TEAM-PERFORMS-PROJECT
:own-slots
((Instance-Of PERFORMS)
(Actor DESIGN-TEAM-1)
(Activity EXAMPLE-PROJECT)
))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Architectural Design and Architect-1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-frame ARCHITECTURE-DESIGN-1
:own-slots
((Instance-Of ACTIVITY)
(Documentation ''This is the first activity of the
example project. It starts when a contract is made.
It produces an architectural design which will be
followed by electrical and mechanical design.'')
(Name ''Architecture Design'')
(End ARCHITECTURE-DESIGN-1-END-TIMEPOINT)
))

(define-frame ARCHITECTURE-DESIGN-1-END-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
(Documentation ''The end TIMEPOINT of the ARCHITECTURE-DESIGN-1 activity, among other
things)
(Name ''End Timepoint for Architecture Design'')
))

(define-frame ARCHITECT-1
:own-slots
((Instance-Of AGENT)
(Name ''Robert Callahan'')
(Capability ARCHITECTURE-SKILL)
))

(define-frame PERFORMS-1
:own-slots
((Instance-Of PERFORMS)
(Actor ARCHITECT-1)
(Activity ARCHITECTURE-DESIGN-1)
))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Electrical Design and Electrical-Engineer-2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-frame ELECTRICAL-DESIGN-2
:own-slots
((Instance-Of ACTIVITY)
(Documentation ''This is the second activity of the project. This activity can begin
only after ARCHITECTURE-DESIGN-1 is completed. It can (but does not necessarily) occur
in parallel with MECHANICAL-DESIGN-3.'')
(Name ''Electrical Design'')
(Begin ELECTRICAL-DESIGN-2-BEGIN-TIMEPOINT)
(End ELECTRICAL-DESIGN-2-END-TIMEPOINT)
))
```

```
(define-frame ELECTRICAL-DESIGN-2-BEGIN-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
(Documentation ''The begin TIMEPOINT of the ELECTRICAL-DESIGN-2 activity, among other
things)
(Name ''Begin Timepoint for ELECTRICAL-DESIGN-2'')
))

(define-frame ELECTRICAL-DESIGN-2-END-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
(Documentation ''The end timepoint of the ELECTRICAL-DESIGN-2 activity, among other
things)
(Name ''End Timepoint for ELECTRICAL-DESIGN-2'')
))

(define-frame ARCH-BEFORE-ELECTRIC
:own-slots
((Instance-Of BEFORE)
(Preceding-Timepoint ARCHITECTURE-DESIGN-1-END-TIMEPOINT)
(Succeeding-Timepoint ELECTRICAL-DESIGN-2-BEGIN-TIMEPOINT)
))

(define-frame ELECTRICAL-ENGINEER-2
:own-slots
((Instance-Of AGENT)
(Documentation ''This engineer has two skills, one is electrical and the other is
mechanical. She is responsible for both electrical design and design review.'')
(Name ''Cristina Marconi'')
(Capability ELECTRICAL-SKILL MECHANICAL-SKILL)
))

(define-frame PERFORMS-2
:own-slots
((Instance-Of PERFORMS)
(Actor ELECTRICAL-ENGINEER-2)
(Activity ELECTRICAL-DESIGN-2)
))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Mechanical Design Mechanical-Engineer-3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-frame IF-ARCHITECTURE-DESIGN-1-DELAYED
:own-slots
((Instance-Of DECISION)
(Documentation ''The activity that decides what to do after checking if ARCHITECTURE-
DESIGN-1 is delayed or not.'')
(Begin IF-ARCHITECTURE-DESIGN-DELAYED-BEGIN-TIMEPOINT)
(If ARCHITECTURE-DESIGN-1-DELAYED)
(Then MECHANICAL-DESIGN-4)
(Else MECHANICAL-DESIGN-3))
)

(define-frame ARCHITECTURE-DESIGN-1-DELAYED
:own-slots
((Instance-Of ACTIVITY-STATUS)
(Documentation ''The ARCHITECTURE-DESIGN-1 is delayed at the beginning of the DECISION
activity, IF-ARCHITECTURE-DESIGN-DELAYED)
(Activity ARCHITECTURE-DESIGN-1)
(Status DELAYED)
(When IF-ARCHITECTURE-DESIGN-DELAYED-BEGIN-TIMEPOINT))
)

(define-frame IF-ARCHITECTURE-DESIGN-DELAYED-BEGIN-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
```

```
(Documentation ''The begin timepoint of the IF-ARCHITECTURE-DESIGN-DELAYED activity)
(Name ''Begin Timepoint for IF-ARCHITECTURE-DESIGN-DELAYED'')
))

(define-frame MECHANICAL-DESIGN-3
:own-slots
((Instance-Of ACTIVITY)
(Documentation ''This activity can begin only if ARCHITECTURE-DESIGN-1 is completed in
time. It can (but not necessarily) occur in parallel with ELECTRICAL-DESIGN-2.'')
(Name ''Mechanical Design'')
(Begin MECHANICAL-DESIGN-3-BEGIN-TIMEPOINT)
(End MECHANICAL-DESIGN-3-END-TIMEPOINT)
))

(define-frame MECHANICAL-DESIGN-3-BEGIN-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
(Documentation ''The begin timepoint of the MECHANICAL-DESIGN-3 activity, among other
things)
(Name ''Begin Timepoint for Mechanical Design 3'')
))

(define-frame MECHANICAL-DESIGN-3-END-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
(Documentation ''The End timepoint of the MECHANICAL-DESIGN-3 activity, among other
things)
(Name ''End Timepoint for Mechanical Design 3'')
))

(define-frame MECHANICAL-DESIGN-4
:own-slots
((Instance-Of ACTIVITY)
(Documentation ''This activity can begin only if ARCHITECTURE-DESIGN-1 is delayed. It
can (but not necessarily) occur in parallel with ELECTRICAL-DESIGN-2.'')
(Name '' Mechanical Design 4'')
(Begin MECHANICAL-DESIGN-4-BEGIN-TIMEPOINT)
(End MECHANICAL-DESIGN-4-END-TIMEPOINT)
))

(define-frame MECHANICAL-DESIGN-4-BEGIN-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
(Documentation ''The begin TIMEPOINT of the MECHANICAL-DESIGN-4 activity, among other
things)
(Name ''Begin Timepoint for Mechanical Design 4'')
))

(define-frame MECHANICAL-DESIGN-4-END-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
(Documentation ''The End Timepoint of the MECHANICAL-DESIGN-4 activity, among other
things)
(Name ''End Timepoint for Mechanical Design 4'')
))

(define-frame ARCHITECTURE-DESIGN-1-BEFORE-MECHANICAL-DESIGN-3
:own-slots
((Instance-Of BEFORE)
(Preceding-Timepoint ARCHITECTURE-DESIGN-1-END-TIMEPOINT)
(Succeeding-Timepoint MECHANICAL-DESIGN-3-BEGIN-TIMEPOINT)
))

(define-frame ARCHITECTURE-DESIGN-1-BEFORE-MECHANICAL-DESIGN-4
:own-slots
((Instance-Of BEFORE)
```

```
(Preceding-Timepoint ARCHITECTURE-DESIGN-1-END-TIMEPOINT)
(Succeeding-Timepoint MECHANICAL-DESIGN-4-BEGIN-TIMEPOINT)
))

(define-frame MECHANICAL-SKILL
:own-slots
((Instance-Of ENTITY)
(Name ''Mechanical Skill)
))

(define-frame SPECIAL-MECHANICAL-SKILL
:own-slots
((Instance-Of ENTITY)
(Name ''Special Mechanical Skill'')
))

(define-frame ELECTRICAL-SKILL
:own-slots
((Instance-Of ENTITY)
(Name ''Electrical Skill'')
))

(define-frame MANAGEMENT-SKILL
:own-slots
((Instance-Of ENTITY)
(Name ''Management Skill'')
))

(define-frame MECHANICAL-ENGINEER-3
:own-slots
((Instance-Of AGENT)
(Name ''Gary Fassbinder'')
(Capability MECHANICAL-SKILL)
))

(define-frame PERFORMS-3
:own-slots
((Instance-Of PERFORMS)
(Actor MECHANICAL-ENGINEER-3)
(Activity MECHANICAL-DESIGN-3)
))

;; Nobody has been assigned yet to the MECHANICAL-DESIGN-4, but whoever does it has to
have the
;; special-mechanical-skill-1.
(define-frame PERFORMS-4
:own-slots
(Instance-Of PERFORMS)
(Activity MECHANICAL-DESIGN-4)
(Constraint (= ?SELF.Actor.Capability SPECIAL-MECHANICAL-SKILL-1)
))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Design Review and Project Manager
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-frame DESIGN-REVIEW-5
:own-slots
((Instance-Of ACTIVITY)
(Documentation ''This is the last activity of the project. This activity can begin
only after both ELECTRICAL-DESIGN-2 and MECHANICAL-DESIGN-3 (or MECHANICAL-DESIGN-4)
are completed. It has four responsible AGENTs, and this activity can be viewed as a
design-review meeting. All team members must participate.'')
(Name ''Design Review'')
(Begin DESIGN-REVIEW-5-BEGIN-TIMEPOINT)
))
```

```
(define-frame DESIGN-REVIEW-5-BEGIN-TIMEPOINT
:own-slots
((Instance-Of TIMEPOINT)
(Documentation ''The Begin timepoint of the DESIGN-REVIEW-5 activity, among other
things)
(Name ''Begin Timepoint for Design Review 5'')
))

(define-frame ELECTRICAL-BEFORE-DESIGN-REVIEW
:own-slots
((Instance-Of BEFORE)
(Preceding-Timepoint ELECTRICAL-DESIGN-2-END-TIMEPOINT)
(Succeeding-Timepoint DESIGN-REVIEW-5-BEGIN-TIMEPOINT)
))

(define-frame MECHANICAL-3-BEFORE-DESIGN-REVIEW
:own-slots
((Instance-Of BEFORE)
(Preceding-Timepoint MECHANICAL-DESIGN-3-END-TIMEPOINT)
(Succeeding-Timepoint DESIGN-REVIEW-5-BEGIN-TIMEPOINT)
))

(define-frame MECHANICAL-4-BEFORE-DESIGN-REVIEW
:own-slots
((Instance-Of BEFORE)
(Preceding-Timepoint MECHANICAL-DESIGN-4-END-TIMEPOINT)
(Succeeding-Timepoint DESIGN-REVIEW-5-BEGIN-TIMEPOINT)
))

(define-frame PROJECT-MANAGER-4
:own-slots
((Instance-Of AGENT)
(Documentation ''This AGENT is the manager of this project. She is responsible for
decision-making whenever a exception occurs during the process of the project. She is
also co-responsible for the design-review activity.'')
(Name ''Ann Rollins'')
(Capability MECHANICAL-SKILL ELECTRICAL-SKILL MANAGEMENT-SKILL)
))

(define-frame PERFORMS-5
:own-slots
((Instance-Of PERFORMS)
(Agent DESIGN-TEAM-1)
(Activity DESIGN-REVIEW-5)
))
```

Appendix F Changes from the PIF 1.0

The following describes the rationales for removing some of the PIF 1.0 constructs from the PIF 1.2 Core. ACTOR has been renamed to AGENT because ACTOR, we believe, is a role (cf. The discussion of ROLE in section 4), and thus should not be represented by an explicit construct. AGENT, on the other hand, is an entity type that can be characterized by, for example, decision making capability.

PREREQUISITE has been removed from PIF 1.0 because it is a mixture of descriptive and prescriptive specification. Any of the constructs in PIF can be used either descriptively or prescriptively. When used descriptively, the PIF constructs describe what actually happen. When used prescriptively, it says what should happen. The BEFORE relation, in particular, can be used descriptively or prescriptively. It is not clear what PREREQUISITE is other than a prescriptive use of BEFORE. Actually, BEFORE is a relation between TIMEPOINTS whereas PREREQUISITE is a relation between ACTIVITIES. Hence, strictly speaking, PREREQUISITE would mean BEFORE(ACTIVITY-1.End, ACTIVITY-2.BEGIN) in the prescriptive sense.

SKILL has been removed because SKILL is not useful without agreeing on its subclasses or its different instances (e.g. MECHANICAL-SKILL, MARKETING-SKILL, etc.). However, the determination of these subclasses would be meaningful only in the context of a specific domain or task. Hence, we decided that in the PIF-CORE a skill should be represented simply as an ENTITY in the Capability attribute of AGENT and let PSV Modules introduce its specialization based on their needs.

RESOURCE (in the proper sense of something used by a given activity) has been removed because it's something that can and probably should be defined dynamically. That is, whether a given object is a resource cannot be defined statically because the same object can be a resource for one activity but not for another. Instead, the RESOURCE membership of an object should be defined in terms of the relation it satisfies (i.e. USED by an ACTIVITY) and tested dynamically in the given context.

GROUP has been removed because a group can now be represented by the Component attribute of ENTITY, which represents a homogeneous collection of its own type. For example, a group of AGENT can be represented by creating an object of type AGENT and specifying its Component to be the individual members of the group.

The Status attribute of ACTIVITY is removed. Instead ACTIVITY-STATUS (Activity, Symbol, Timepoint) is introduced in order to associate with it a timepoint (cf. the discussion of the attribute value earlier in this section).

References

Chan, FY, 1995. "The round trip problem: a solution for the process handbook" *Unpublished Master's Thesis*, MIT Department of Electrical Engineering and Computer Science, May.

Genesereth, M and Fikes, R, 1992. "Knowledge Interchange Format v.3 Reference Manual" Available as a postscript file via anonymous ftp from `www-ksl.stanford.edu:/pub/knowledge-sharing/papers/kif.ps`

Gruber, T, 1993. "Ontolingua: A translation approach to portable ontology specifications" *Knowledge Acquisition* **5**(2), 199–200. (Available via anonymous ftp from `www-ksl.stanford.edu:/pub/knowledge-sharing/papers/ongolingua-intro.ps`)

Lee, J and Malone, T, 1990. "Partially shared views: A scheme for communicating between groups using different type hierarchies" *ACM Transactions on Information Systems* **8**(1), 1–26.

Malone, T, Crowston, K, Lee, J and Pentland, B, 1993. "Tools for inventing organizations: toward a handbook of organizational processes" *Proceedings of the 2nd IEEE Workshop on Enabling Technologies Infrastructure for Collaborative* IEEE Press.

Neches, R, Fikes, R, Finin, T, Gruber, T, Patil, R, Senator, T and Swartout, WR, 1991. "Enabling technology for knowledge sharing" *AI Magazine* 12(3), 36–56.

Steele, G, 1990. *Common Lisp: The Language. Second edition*. Digital Press.

Tate, A, 1995. "Characterizing plans as a set of constraints – the <I-N-OVA> model – a framework for comparative analysis" *ACM SIGART Bulletin*, Special Issue on "Evaluation of Plans, Planners, and Planning Agents", **6**(1), January. (Available as a postscript file via `ftp://ftp.aiai.ed.ac.uk/pub/documents/1995/95-sigart-inova.ps`)

Uschold, M, King, M, Moralee, S and Zorgios, Y, 1995. "The enterprise ontology" (Available via WWW URL `http://www.aiai.ed.ac.uk/entprise/enterprise/ontology.html`)