Technical Report
CMU/SEI-93-TR-4
ESC-TR-93-181

Process-Centered Development Environments: An Exploration of Issues

Alan M. Christie

June 1993

Technical Report
CMU/SEI-93-TR-4
ESC-TR-93-181
June 1993

Process-Centered Development Environments: An Exploration of Issues

Alan M. Christie
CASE Environments Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

List of Figures

List of Tables

Process-Centered Development Environments: An Exploration of Issues

Abstract: Software development environments are beginning to move from research communities to commercial applications. As this occurs, the need to address process issues related to such environments is becoming increasingly apparent. Thus there is a growing awareness of the need for process-centered development environments (PCDEs). This report addresses process definition and enactment issues which pertain to the specification and design of a PCDE. The first part of the report explores some of the required characteristics of an enactable graphical language and the relationship between process definition and enactment. This process language naturally led to the ability to perform process verification, i.e., a verification that the actual process path taken throughout a project conforms to the defined process. The issue of process verification is thus also explored. The success of PCDEs rests heavily on end-user acceptance. Because of this, the report concludes with a review of user-oriented process and social issues relevant to the successful adoption of PCDEs.

1 Background

Software production has historically been a very labor-intensive, highly-skilled and costly business. In addition, the more technically advanced Western nations have led the way in the rapidly changing field of software engineering. However, other nations, whose labor costs are far below those of the West, are catching up [Firth 93]. These facts, coupled with the efficiency of global communications, suggests that the competitive position of Western nations in software development may soon erode. Software is a strategic technology, from both defense and commercial points of view, and loss of this competitive edge could have profound consequences.

How can this competitive edge be maintained? Several elements are necessary. First, a vibrant infrastructure (such as Silicon Valley) where intense competition drives technology ahead of outside competition is needed. Second, an educated workforce, capable of using new technologies and of rapidly adapting to new circumstances, must be available. Third, an emphasis on continuously improving software quality and hence organizational performance is required. Since the process with which a product is built has a direct bearing on quality, an understanding of the process is essential. Finally, once a well-defined process has been developed, the process may be automated as a means of assuring process consistency and of reducing cost. Automation of the process includes, but is not limited to, providing software organizations with appropriate tools to perform specific tasks, relieving developers and managers of as much tedium as possible, eliminating error-prone activities, and guiding process-critical tasks. Such support should allow developers and managers to concentrate on creative non-automatable tasks. If such cost-reducing, quality-enhancing measures are not taken, the competitive advantage currently held by those countries that have relatively high labor rates

is likely to disappear. One such cost-reducing, quality-enhancing measure is to introduce process-centered development environments (PCDEs).

A note of caution should, however, be observed with respect to automating software production. When producing mechanical or electrical products, the number of component variants is usually fewer than the number of software variants, logical complexity is less, and initial requirements are usually better understood than with software products. These factors often result in mechanical and electrical products having logically simpler manufacturing processes. As a result, while much of hardware manufacturing can be performed repetitively by machines, software production, having a higher intellectual (and non-standardized) content, tends to be manually produced. Thus any approach to software automation cannot simply rely on principles developed for hardware. In particular, software automation forces a very tight and complex relationship between the computer environment and the software developer. Consequently, an understanding of human-computer interaction is particularly important for PCDEs to be effective. Simply automating the software production process without giving due importance to end-user, process and even social issues could result in the erroneous conclusion that automation does not work.

2 A Process Development and Usage Scenario

This report addresses issues relevant to improving the software productivity and quality, as they relate to PCDEs. First, it emphasizes that both technology and user issues affect success. Second, it suggests that a graphical and enactable specification of the PCDE will contribute significantly to the successful implementation of the PCDE. Third, it discusses process verification. Process verification affects quality because it guarantees compliance with the defined process.

Figure 2-1 illustrates a proposed process development and usage scenario.[1] This figure sum-

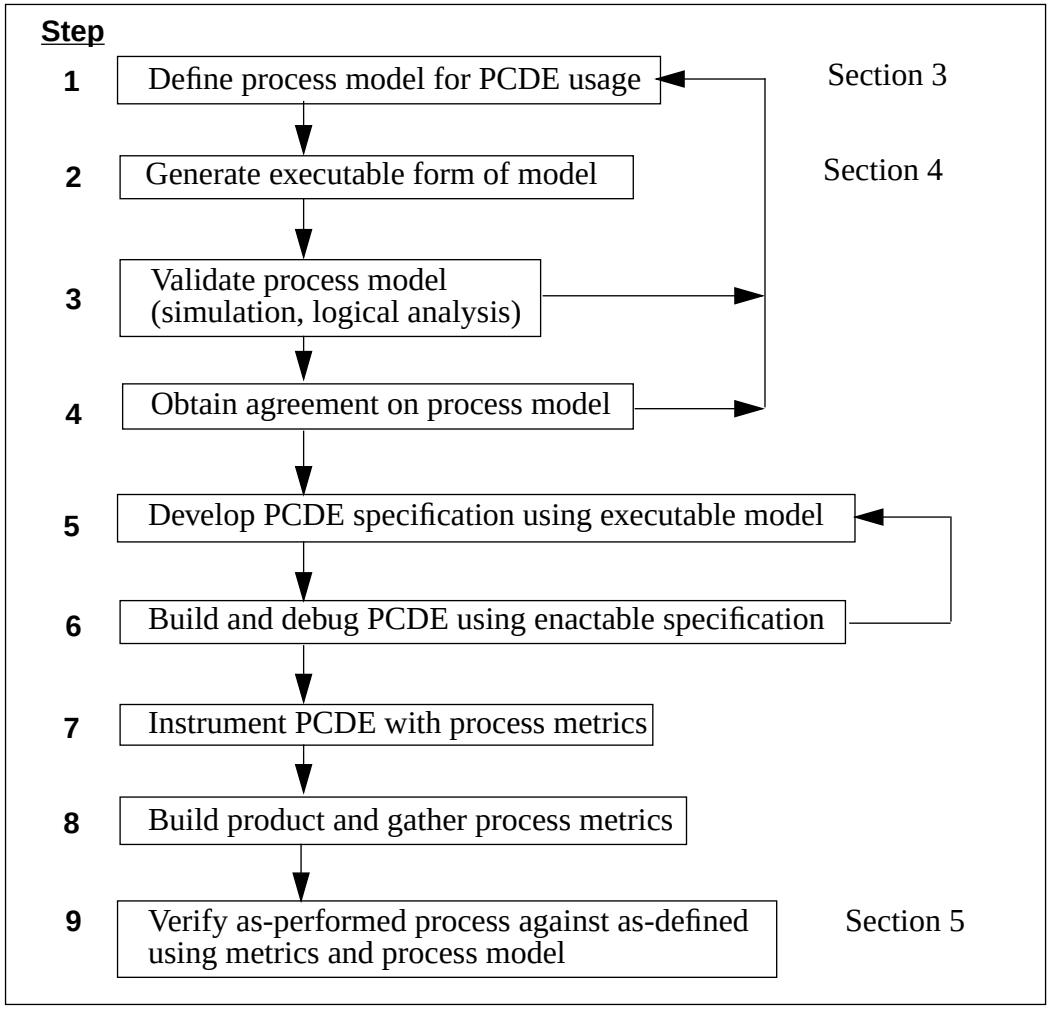


Figure 2-1 The Process Development and Usage Scenario

marizes the technical steps necessary to implement a PCDE. The first step in Figure 2-1 defines an appropriate process model with which to support the software development activity.

---

1. In the following text, the numbers down the right hand side of Figure 2-1 are referred to as steps.

The process supported could be a modest one such as peer reviews or it could be a wide-ranging one such as comprehensive project support. In any case, this process is defined using a graphical language. In Step 2, the graphical model is compiled into a symbolic and executable form. Through this executable form, the dynamics of the process can be studied at a high level, without being encumbered at this point by the low-level implementational detail. Validating the model (Step 3) will likely involve logical (static) analysis to check for deadlock and reachability and dynamic simulation to test the system's behavioral characteristics. After the model has been formally validated, managers and developers will be asked to agree upon its adequacy from a user perspective. This buy-in of managers and developers is essential if the process is to gain acceptance. By defining the model graphically, communication and agreement on the process model will be significantly enhanced. Such a visible representation is more readily comprehensible than a process defined through text or symbolic coding. In addition, modifying the defined process at this point is less expensive than to waiting until major development of the actual PCDE has been performed.

Steps 1 through 4 are analogous to the process through which the specification of a software product is developed using graphical techniques. As stated in [Osterweil 87]:

> *Because software processes are programs in what we now see as to be the classical sense of the term, we should expect that they are best thought of as being only part of a larger information aggregate. This information aggregate contains such other software objects as requirements specifications (for the process description itself)...*

This is indeed what we are doing by defining the process specification in Step 5. However, [Oserweil 87] also states that "the process itself is a dynamic entity and the process description is a static entity."

As we have seen above, our specification is also dynamic. This allows us not only to be precise and formally consistent in a static sense, but also to be more behaviorally correct. By allowing for behavior in the specification, we can address, in a preliminary way, some basic end-user issues prior to investing in the full-scale implementation. At this point, the enactable specification defines all significant high-level artifacts in the process, the agents or roles who will perform the activities, and the decisions which are made or acted upon throughout the process.

The architecture or implementation of the PCDE that occurs in Step 6 could take many forms. At one end of the spectrum, tools can be directly coupled to allow for simple process enactment (point-to-point integration). More complex process enactment is also possible, for example, using an underlying framework which provides control and data integration mechanisms for coupling diverse tools [Wallnau 91]. Recently, a number of commercial process support and enactment tools have become available; these are in addition to those which have academic origins. Such process support tools may prove to be critical to finding a total PCDE solution. However, they are not the main subject of this report.

Collecting process metrics is essential to process understanding and improvement. Having a defined model of one's process significantly simplifies the selection of which metrics to gather (Step 7). These metrics are gathered during the software product's manufacture. Such metrics may include decisions taken, intermediate product versions used, reviews completed, and which agents involved in each of the activities (Step 8). Upon completion of the project, these metrics can also be used to verify that the as-implemented process conforms to the as-defined process (Step 9). Note that in Step 3 we have used "validate" to imply correctness of the enacted process model, while in Step 7 we have used the word "verify" to mean compliance of the is-implemented process with the as-defined process. This convention will be used throughout the rest of the report.

This report focuses on Steps 1, 2, and 9 which are discussed in Sections 3, 4, and 5 respectively. These three areas; process definition, enactment, and verification can all rely on the same graphical notation, independent of the specific PCDE used for implementation; hence it makes sense to discuss them under the common heading of this report. Section 3 describes the graphical language ProNet, which was explicitly developed for process modeling with enactment in mind. Section 4 discusses how models within this language can be translated into an enactable form. Section 5 then explains how verification can be performed using the same ProNet notation, together with process data gathered during product development. Finally, Section 6 reviews some general issues associated with end-user needs, organizational factors, and process automation in light of the process modeling experience. Such issues as what can and should be automated, environment support versus control, and developer needs versus management needs are discussed.

Implementation issues associated with PCDEs will not be addressed. Thus we will not investigate what tools are needed to support software development, how these tools are integrated, or what use is made of environment frameworks. In summary, it is intended that the following work form a basis for

- exploring issues associated with process definition and enactment in the context of process model specification and validation,
- addressing the issue of software quality through process verification, and
- investigating end-user issues with respect to process automation.

It is intended that process models defined, enacted, and evaluated using the techniques described in this report will actually be implemented through recently available process support tools such as SynerVision [Cooley 92], ProcessWeaver [Ellen 92], or ProcessWise [Bruynooghe 91]. These investigations will be the subject of another report.

3 The ProNet Graphical Modeling Language

This section discusses the graphical modeling language ProNet (Step 1 in Figure 2-1) which forms a basis for process enaction. Section 3.1 first discusses why there should be close relationship between graphical process definition and process enactment. The ProNet notation itself is described in some detail in Section 3.2, while Section 3.3 compares this notation to that of several other well-known modeling approaches. Finally, Section 3.4 illustrates the notation with a small process modeling example.

One general definition of process is: *A set of partially ordered steps intended to reach a goal* [Feiler 92]. Given this definition, steps, which are interpreted to be activities, take a central position in ProNet. An activity may only occur if certain entrance conditions are met or if certain products become available. As a consequence of the activity, exit conditions may change from false to true (or vice versa) and certain products may be generated. Hence the firing of an activity changes the state of the system, generating and setting up necessary entrance conditions for other activities to fire. ProNet is thus a declarative model. Each activity and its entrance and exit conditions bear a strong relationship to the elements of a rule in a rule-based system, it is because of this characteristic that enactability is possible.

3.1 The Relationship Between Process Definition and Enactment

Why develop a graphical process notation with enactable characteristics? There are many valid reasons for doing so:

- Debugging an enactable process can be significantly easier if a graphical form of the process model is used. Automatic transformation of the graphical model to its enactable form then guarantees a degree of correctness not found when the model is developed analytically.
- By having enactable characteristics, a process model can be used to explore different process alternatives before any process is implemented. By providing a debugged, enactable, and agreed-to process model, the simulated process can be faithfully reproduced in the real world. This has implications not only for establishing the initial process but for process modification and improvement.
- Both the graphical model and its enactable form will provide guidance on tool integration issues. The process model specifies the input and output data and control information (through the products and conditions), thus providing a rational basis for communication between tools. (Of course, actual integration of the tools within a process support environment raises many implementation issues not addressed in this report.)

- There are significant advantages to defining the process graphically, independent of enactability considerations:

  -- Graphical specification provides a means to communicate to developers, management and others what the “new” process will look like. This generates early feedback from those involved in the future use of the process, thus encouraging its success.

  -- Managers can thus sign-off on a defined process without having to understand a complex symbolic notation.

  -- The graphical process model allows for accelerating training for new employees.

- If incorporated into a PCDE, the graphical model can be used for real-time process support, as a “front-end” for task selection and to obtain process-related status information during product development.

The ProNet graphical process notation can readily be mapped to a set of production rules through which the process can be enacted. The resulting symbolic model can then be used to investigate the dynamic characteristics of the process or, equally importantly in our case, to define the characteristics of a process driver for a PCDE.

3.2 Entity Classes

This section provides a definition of the notation. ProNet diagrams are based on a modified Entity-Relation model [Chen 83] in which entities fall into one of eight classes. The following list defines these entity classes:

- **Activities** provide the backbone to the process model. Other entity classes are attached to and support activities. The existence of products and conditions (which are defined below) provide the entrance conditions for activity initiation. Activities are responsible for generating exit products and conditions.

- **Products** can either be required to support an activity or be produced by an activity. Products may be the result of some activity internal to the model (e.g. a file containing source code) or maybe generated outside of it.

- **Conditions** can either be required to initiate an activity or result from an activity (e.g., “review completed”) and take the values TRUE or FALSE. The existence or non-existence of a product, agent, etc., can be equivalent to a condition.

- **Composites** are boolean combinations of conditions, products, agents, etc.

- **Agents** are specific entities which perform activities. Humans or non-human entities capable of performing activities (e.g., the software developer, Mary or the Vertex C Compiler, Ver 5.0) are considered to be agents. Agents may support activities or may be derived from, or identified by, activities.

- **Roles** are abstractions (i.e., a super-class) of the agents concept (e.g., *reviewer, editor*). If the process model is enactable, the role must be instantiated by an agent at run time. Like agents, roles may support multiple activities or may be derived from, or identified by, activities.

- **Stores** allow for persistence of instantiated entities. Products, condition values and even agents can be deposited in or retrieved from stores.

- **Constraints** are policy restrictions imposed on the performance of an activity. Unlike conditions these do not take on boolean values but may reflect guidance on how things are done (e.g., a quality assurance constraint such as on documenting written code).

Most relationships link the entities to the activities. These relationships types are of the form shown in Tables 1 and 2.

Table 1 Entrance Relationships

Entrance relationships	Inverse entrance relations
product_A *is entrance product for* activity_A	activity_A *has entrance product* product_A
condition_A *is entrance condition for* activity_A	activity_A *has entrance condition* condition_A
composite_A *is entrance composite for* activity_A	activity_A *has entrance composite* composite_A
agent_A *is entrance agent for* activity_A	activity_A *has entrance agent* agent_A
role_A *is entrance role for* activity_A	activity_A *has entrance role* role_A
store_A *is entrance store for* activity_A	activity_A *has entrance store* store_A

Table 2 Exit Relationships

Exit relationships	Inverse exit relations
product_A *is exit product for* activity_A	activity_A *has exit product* product_A
condition_A *is exit condition for* activity_A	activity_A *has exit condition* condition_A
composite_A *is exit composite for* activity_A	activity_A *has exit composite* composite_A
agent_A *is exit agent for* activity_A	activity_A *has exit agent* agent_A
role_A *is exit role for* activity_A	activity_A *has exit role* role_A
store_A *is exit store for* activity_A	activity_A *has exit store* store_A

In the process diagrams, these relationships are all written in italics. The information they contain allows the reader to identify the class of entity which is linked to the activity. Also, if a re-

lationship (such as “product_A *is entrance product for* activity_A”) holds, so does the inverse relationship (“activity_A *has entrance product* product_A”).

In the process diagrams, activities can be identified since they always have a shadowed box around the entity name. In addition, a black dot is placed next to the entity at the end of the relationship. For example, in the relationship “ABC *is entrance product for* XYZ”, the dot would appear in the graphical relationship close to the box surrounding the activity XYZ. Thus the dot plays the same role as the tip of an arrow does in entity-relationship diagrams. The final entity class, the constraint, is simply imposed on an activity, rather than being attached to the initiation or conclusion of an activity. Thus, a constraint’s relationships to an activity are of the forms:

- constraint_A *is constraint for* activity_A
- activity_A *has constraint* constraint_A

ProNet handles three other relationship types: inheritance, aggregation, and custom. With respect to inheritance, there are two named relationships: *is instance of* and *is generalization of*, each of which is the inverse of the other. Unlike the relationships discussed earlier, these relationships link two arbitrary entities, so long as they are of the same class. With respect to aggregation, there are also two named relationships: *is part of* and *includes*, each of which is the inverse of the other. These also link arbitrary entities together, but there is no constraint that the joined entities be of the same class. Occasionally there is a need to define a non-standard relationship between two entities which does not fall into one of the predefined categories. ProNet allows definition of a “custom” relationship. This feature allows creation of an arbitrary relationship (i.e., one not belonging to the pre-defined set) to link two entities. For example one might define the custom relationship *depends on* as in the example:

agent_A *depends on* resource_A[2]

ProNet provides a second way of substructuring entities besides use of the *is part of* relationship. While the *is part of* relationship allows the components of an entity to be displayed on the same graph as the entity itself, it is sometimes necessary to display the detailed structure of an entity (particularly an activity) on a separate graph. Most real-world process models have significant complexity, and this latter approach allows for multiple-levels of hierarchical decomposition. If a particular entity in a process diagram has an underlying structure, then the box representing that entity is enlarged so as to be twice as deep as the normal entity box.

[2.] Because custom relationships are not defined within the standard set of relationships types, they weaken the formalism. This option thus cannot be used in an enactable form of the model. However, from the point of view of describing process, as opposed to its enaction, custom relationships can be useful if used with discretion.

3.2.1 Basic Graphical Elements

As previously stated, the notion of activity is central. Generally the goal of the software process is to produce software products. Thus, one outcome of activities is to produce products. Another activity is making decisions, the outcomes of which are reflected in the values attached to conditions. Also activities cannot generally begin unless certain products and agents are available and certain conditions are satisfied. This view can be represented as shown in Figure 3-1.

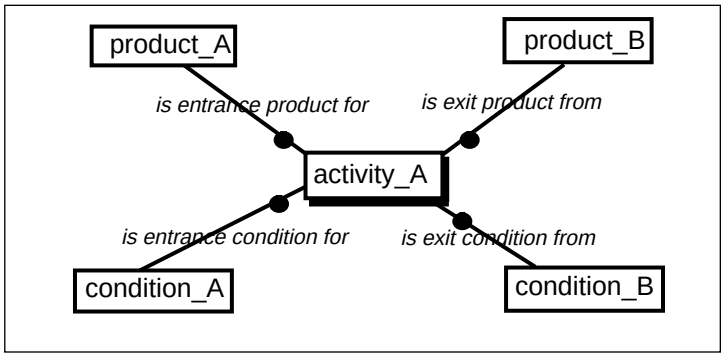


Figure 3-1 Basic Representation of a Process Element

Note the following about this representation:

- It is an entity-relationship diagram, with the entity classes *activity, product, condition, agent, role, junction,* and *constraint* being defined within the notation.
- As shown in Tables 1 and 2, relationships have predefined types. In the above diagram, *is entrance condition for* and *is exit condition for* relate conditions to activities. Similar relationships exist for products. Inverse relationships always hold.

Other entities must also support the notion of activity. At least one agent or role must be associated with any activity. A role represents an abstraction of an agent. For example *manager* is an example of a role while *Joe* is an example of an agent that may take on the attributes of a manager. The agent concept is thus a subclass of, and takes on the properties of, the role concept.

Agents and roles may or may not be human. Examples of non-human roles are compilers and editors. Agents not only support activities (through the relationship *is entrance agent for*), but may be identified by, or generated by, an activity. Generally human agents are “identified” while non-human agents are “generated”. The human/non-human distinction is not as critical for process definition as it is for enaction. Thus ProNet does not explicitly distinguish between human and non-human agents and, for both cases, the ProNet relationship is *is exit agent for*. The same is true for roles. For process enaction, however, the distinction clearly is important since automated agents may initiate activities without human intervention. These additions are shown in Figure 3-2.

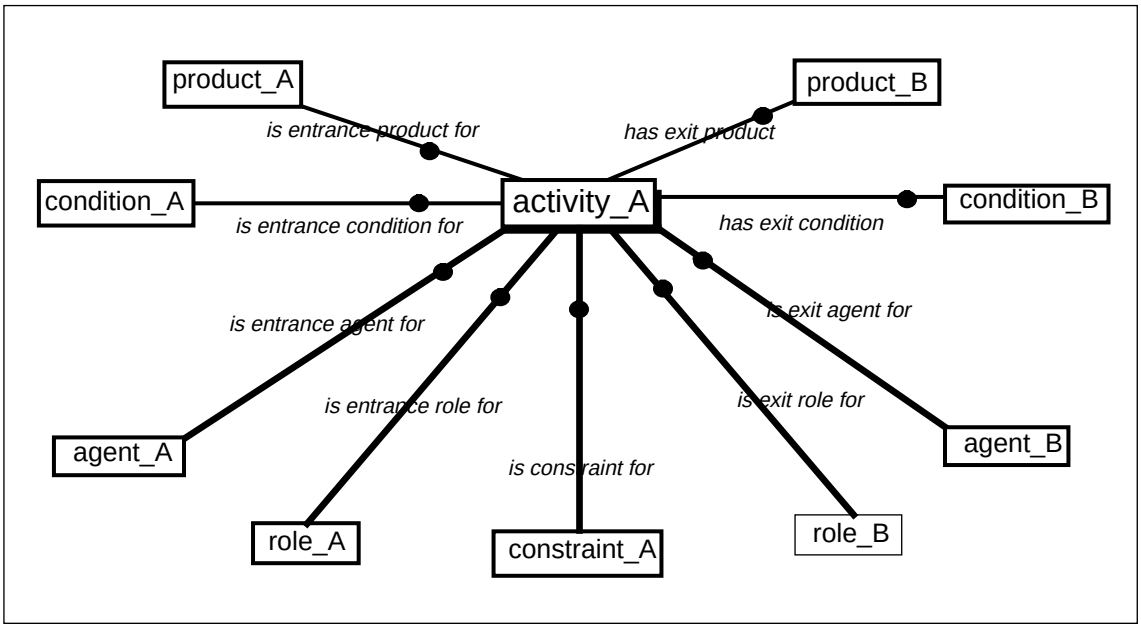


Figure 3-2 Augmented Representation of a Process Element

In defining an enactable process model, it may be necessary to establish some entities as role since the specific agent will not be not known prior to process enactment. In other cases the agent may be known. For example, it may be known that Fred will be manager of a project; it may not be known which developer will perform a specific technical task for Fred. Thus agents that are explicitly defined in the process model represent early bindings of roles**.**

Two entity-labeling conventions should also be noted. First, a numerical extension on classes of entities which are not activities indicates instances of the same entity. For example, *proj file 1* and *proj file 2* are different instances of the same physical product, *proj file*. As will be seen later, this can be convenient in defining the graphical process model. Such a labeling convention is also sometimes necessary for reasons of uniqueness during process enactment. On the other hand, numerical extensions on activities do indicate truly different and separate activities.

An important concept in the basic notation is the junction class. Junctions allow boolean combinations of entities to be logically related, either as input to an activity or as the output from an activity. There are four instances of the junction class: CA, CO, DA, and DO, where C stands for *convergent*, D stands for *divergent*, A stands for *AND* and O stands for *OR*. Since convergent junctions are always implemented prior to an activity they are called *entrance junctions*. Similarly, since divergent junctions are always implemented after an activity they are called *exit junctions*. These are described below:

- **CA**: This stands for a *convergent AND* junction. In this kind of junction, an example of which is shown in Figure 3-3, several entities must all be present before the output from the junction is activated.

- **CO**: This stands for a *convergent OR* junction and is similar in structure to the CA junction. In this case, however, only one branch of the inputs needs to be activated before the activity can be fired.
- **DA**: This stands for a *divergent AND* junction. As a result of an activity, a conjunction of conditions and products may result. While agents or constraints may by required as components of an entrance junction (e.g., Fred or Mary may be the agent), only conditions and products can be linked to exit junctions.
- **DO**: This stands for a *divergent OR* junction and is similar in structure to the DA junction. In this case, however, only one branch of the outputs will be activated. Thus in Figure 3-4, either *product_A*, *product_B*, or *condition_A* will activate.

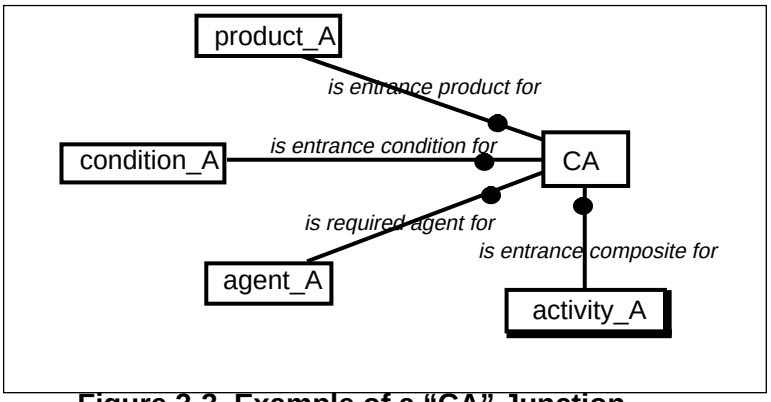


Figure 3-3 Example of a "CA" Junction

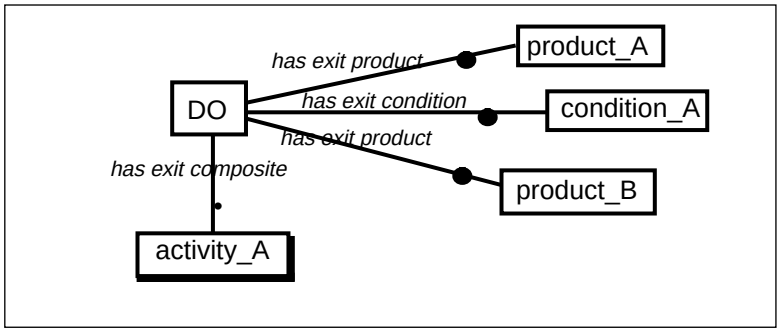


Figure 3-4 Example of a "DO" Junction

It should be noted that if multiple conditions and/or products tie directly into an activity, then by default these are all assumed to be conjoined (i.e., no CA box is needed). The same rule applies for conditions/products exiting an activity (i.e., no DA box is needed). This default can be seen in Figure 3-2. Finally Figure 3-5 illustrates a complex boolean expression as input to an activity which in turn generates a product and sets a condition.

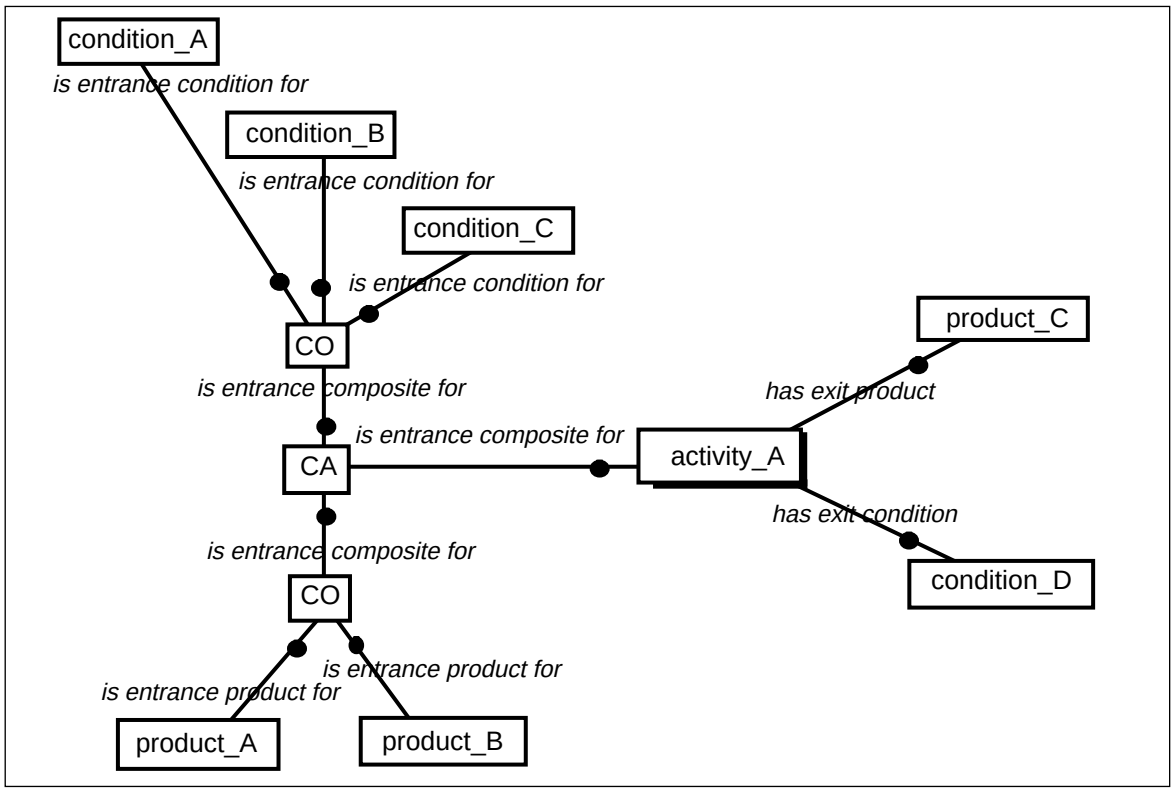


Figure 3-5 Example of a Complex Boolean Junction

The last required basic concept is that of iteration. This is easily accommodated within the existing notation. Iteration is required, for example, when a product undergoes a series of revisions, where each revision is different from the last. Figure 3-6 illustrates the implementation.

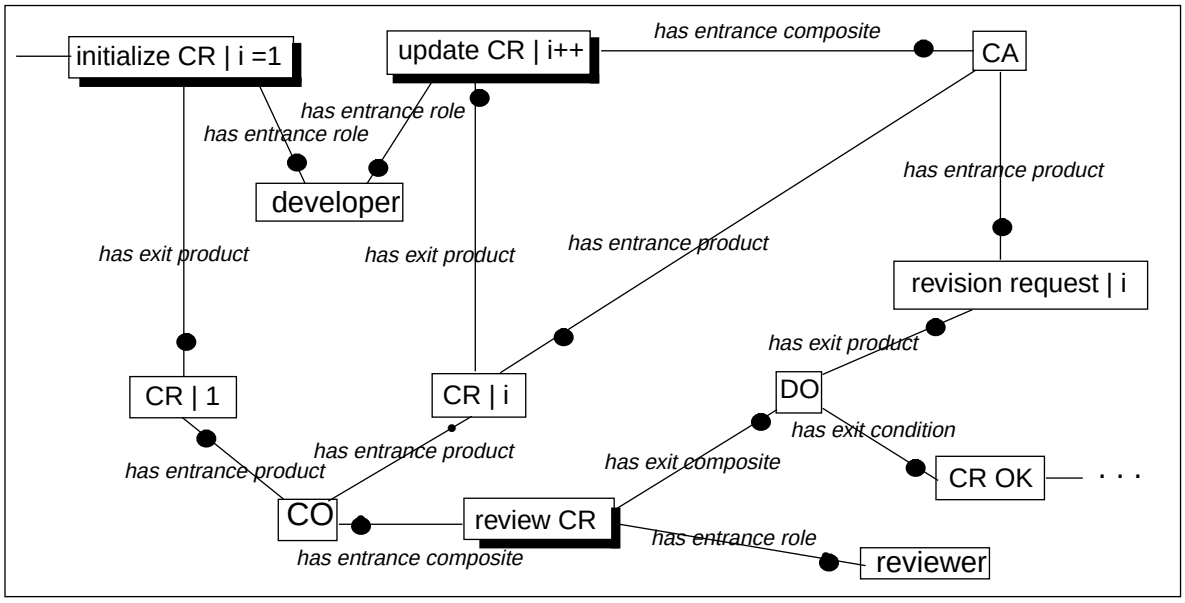


Figure 3-6 A Simple Change Request Model

A change request (CR) is initially composed at which time the incrementing variable is set to an initial value *(initialize CR | i=1*). Each time the change request is updated, the variable is incremented (*update CR | i++*). Versions of products are tagged with the variable *i*, and hence versions of the product are defined. Thus *revision request | i* represents the i^{th} version of the change request. Clearly, nested loops can be implemented using this notation. It should be noted that the nomenclature used to increment versions (i.e., *| i*, *| i++*, and *| i=1*) is not part of the ProNet tool. This is simply a notational convention. The terms *i++* etc. are simply added after the entity names when these entities are defined.

3.2.2 Some Properties of Stores

Stores support collections of product and condition values, and thus allow for the modeling of persistency of generated entities. Since we must be able to add these entities to or retrieve these entities from a store, we introduce two special activities. These do not generate products, conditions, etc., but add products to and remove products from a store. The activity types (*put*, *get*) are illustrated in Figure 3-7 and are added on to the front of the activity name as shown in the figure. Note that in the case of the *put* operation, there must always be an exit condition stating that the activity has taken place. Likewise, in the *get* operation, an entrance condition must be present in order for the operation to be initiated.

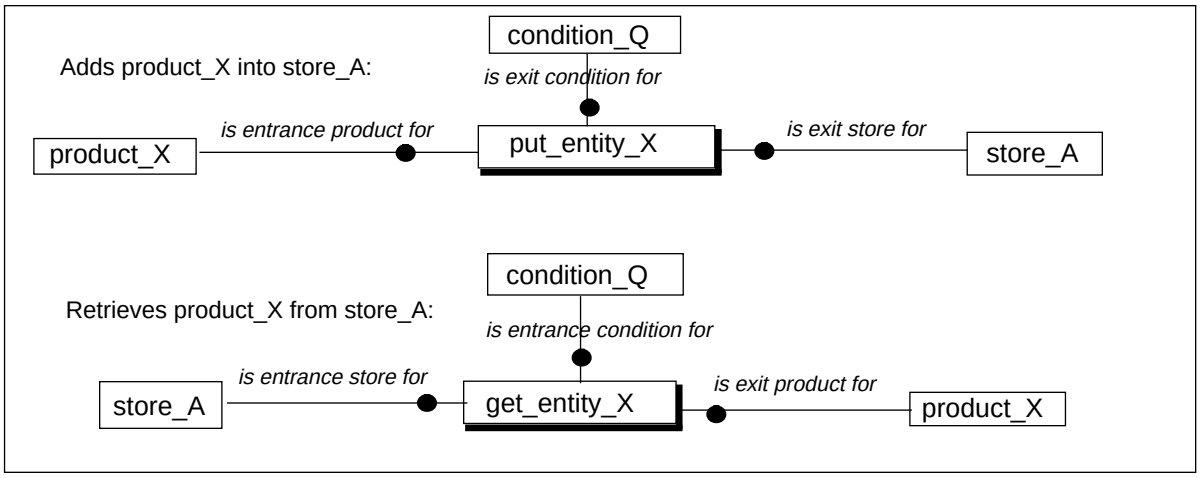


Figure 3-7 Definition of Activities Associated Only with "Store" Entities

3.3 Relationship to Other Modeling Techniques

The approach to process modeling taken by ProNet has connections to other modeling notations. In particular, there are similarities to Petri Nets, entity-relationship models, state transition diagrams, and data flow diagrams. ProNet can also be interpreted in a declarative rule-based sense. However, since this is discussed in some detail in Section 4, it is not dealt with here.

Petri Nets [Reisig 82] were initially developed to model synchronous and asynchronous events in communication systems. These nets consist of three types of elements which can be dis-

played graphically: *places* (denoted by circles), *transitions* (denoted by rectangles), and directed *arcs*. Arcs connect places to transitions and transitions to places. In addition, the concept of *tokens* (which are inserted into places) is used for process enaction to mark a place which has been asserted in some way (e.g., is made "true"). Thus, the Petri Net concept of *place* is a generalization of the ProNet concepts of *conditions*, *products*, or even *agents*. The Petri Net *transition* is comparable to the ProNet concept of *activity*. In Petri Nets, transitions are never connected directly to other transitions. In the same way, ProNet activities are not generally connected directly to other activities (the exceptions being through the inheritance, aggregation and custom relationships).

The connection between ProNet diagrams and entity-relationship (E-R) diagrams [Chen 83] is a fairly direct one. E-R diagrams define entities connected by directed arcs, and names describe the relationships (i.e., arcs) between the entities. Entities may have attributes or properties and, in the database world, these can stored in tables. However, it is the concept of *relationship*, rather that the concept of data structure, which is of importance to ProNet. (See Section 4.5 for a brief discussion of a data structure interpretation of ProNet.) While E-R relationships can, in general, take on arbitrary values, in a ProNet model the relationships, such as *is exit condition for*, come from a predefined small typed set. (An exception to this rule is the *custom* relationship discussed in Section 3.1.)

State-transition networks [Harel 87], based on finite-state machines, can be used to model the dynamic (or behavioral) aspects of process. Such networks model *states*, *events* and the relationships between them. An event occurs (in theory) instantaneously and represents control or stimulus information required to activate a new state. A state can represent a particular configuration of objects and may have an implied activity associated with it. Through this activity, new events may be activated. In process modeling, the state-transition concept of *state* corresponds to the ProNet concept of *activity* while the concept of *event* corresponds loosely to the ProNet concepts of *condition*. While a product may be interpreted as a physical object, its existence or lack of existence can also be interpreted as a condition.

Finally, ProNet has much in common with data-flow (or functional) modeling techniques [Yourdon 89]. Data-flow diagrams define activities and the artifacts which flow between them. They also allow for stores in which artifacts, generated by the activities, are kept, or retrieved. They do not, however, consider the temporal sequence in which these activities occur and thus do not consider behavior. ProNet borrows several concepts from data-flow diagrams:

- activities play a central role,
- stores are used to contain artifacts,
- activities can be hierarchically nested, and
- artifacts are explicitly generated by some activities and consumed by others.

However, ProNet has a combination of features which make it unique. First, it was designed explicitly for software process modeling, and its entity classes are tailored to this goal. Second, it was developed so that there is a direct and unique correspondence between graphical pro-

cess modeling in ProNet and the symbolic enactment model (as will be seen in Section 4). Third, ProNet provides for version management within a process definition/enactment context and provides for persistency of entities generated during enactment. Finally, the underlying model provides a basis for process verification (as will be seen in Section 5).

3.4 A ProNet Example

To illustrate the modeling concepts discussed in ProNet, the following simple change request example is given. While this example is small, it illustrates many, but not all, of the concepts which may go into a ProNet model. For a large and detailed ProNet model of a real software maintenance organization, [Slomer 92] should be consulted.

Figure 3-8 shows the model. In this figure, the major process flow lines have been highlighted for clarity. In addition, it should be noted that certain of the activities have circled numbers next to them. These activities are the main ones which will be discussed with respect to process enaction in the next section. One general point should be made about reading ProNet process diagrams: they do not have arrows indicating in which direction the process is flowing in time. Rather, the dots on the lines connecting the entities provide relationship information, not temporal process flow information. To initiate the process, entry products or entry conditions are placed along the left-hand edge of the diagram. In this case the process starts with either the condition *extern_prob_iden* or *intern_prob_iden*. Similarly, the process terminates with exit products or conditions being placed along the right-hand edge of the diagram. In this case there is one exit condition, *cr_appr*. Thus, the process can be followed by starting on the left of the diagram and working over to the right.

The process can start when either the condition *extern_prob_iden* or *intern_prob_iden* is initiated. This means that an initial problem can be identified either externally, such as by a field representative, or internally such as by a developer. When, for example, the condition *extern_prob_iden* is set to true, the activity *devel_extern_cr* can proceed. The resulting product is the initial version of the change request, *field_cr.* This version is then electronically transmitted to the central office where it is identified as *cr_extern*. CRs generated internally are designated by *cr_intern*. In either case the responsible developer formats the CR messages *(format_cr | k = 1*) suitable for inclusion in the CR repository. At this point, the CR version indicator *i* is initiated to 1. The output of this activity is *cr1|1*. The first “1” indicates from which activity the CR has come (This is necessary for process enaction, as will be described later.) The second “1” is the CR version number.

The next two activities are related to the repository. The *put* prefix on the activity indicates that the entrance product will be stored in the repository, *cr_repos*, linked to this activity. While the developer puts version *k* of the change request into the repository *cr_repos*, the reviewer removes (i.e., *gets*) a copy of the same version of the CR for review. If the change request passes the review then the condition *cr_appr* is generated; otherwise, version *k* of the product *rev_doc* is generated. It will be noticed that there are three activities all with the name

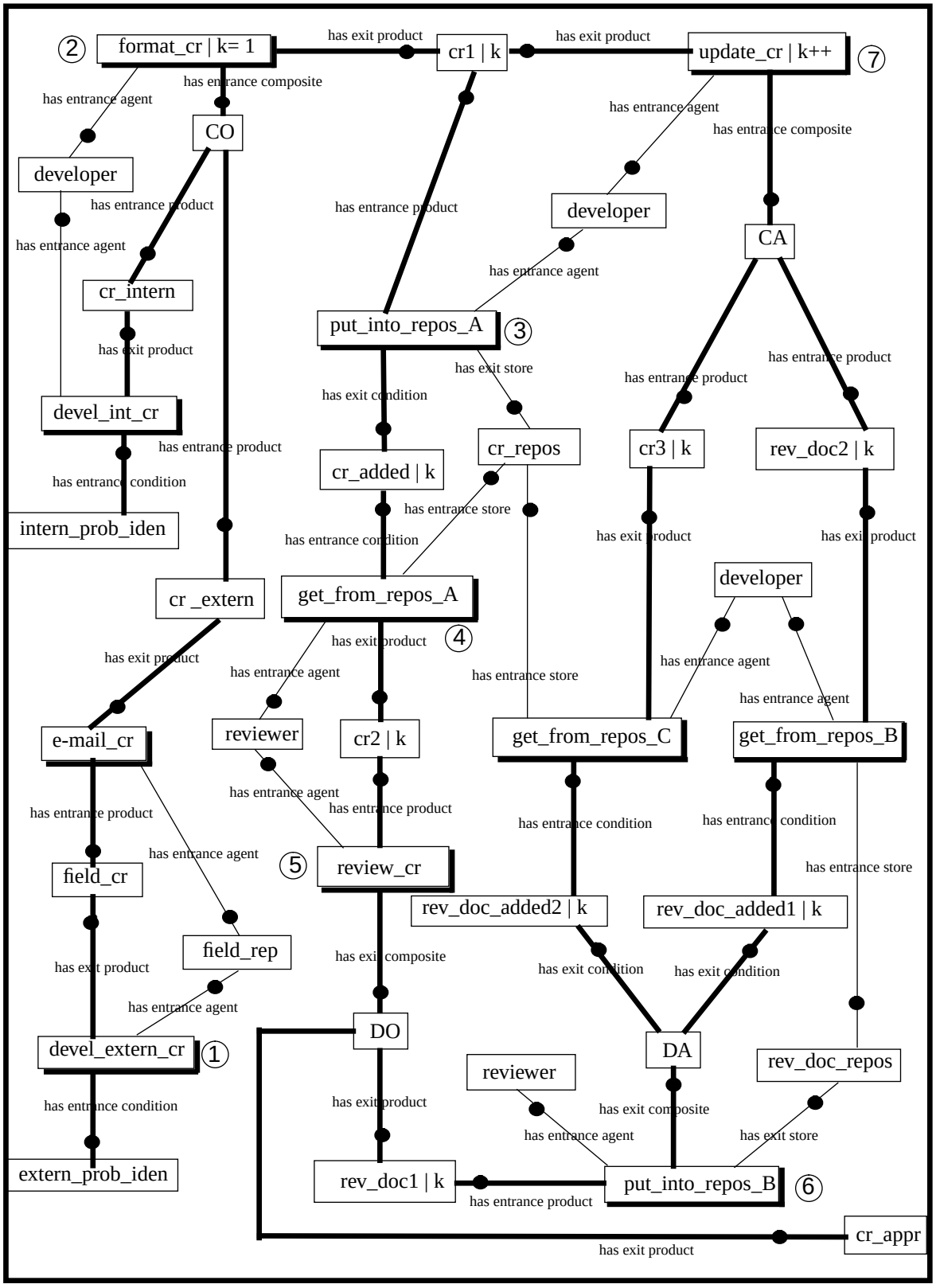


Figure 3-8 A More Complex Change Request Process Model

get_from_repos. These activities are distinguished with the suffixes *_A, _B*, and *_C* respectively.

Beyond this point, the developer retrieves the appropriate version of the change request and the corresponding review document, *rev_doc2*, makes updates to the change request, *cr3*, and puts the updated change request (*cr4* | *k*) back into the repository *cr_repos*. Note that the version number *k* is incremented before the exit product *cr1* | *k* is generated.

ProNet has been implemented in prototype form using HyperCard 2.0™ on the Apple Macintosh™.

4 Enacting the Process

This section describes how the ProNet language defined in Section 3 can be used as a basis for generating an enactable model (see Step 2 in Figure 2-1). As discussed in Section 2, the focus of this report is, in part, to investigate issues associated with developing enactable process specification. Thus the implementation of the PCDE (which would use the specification) will not be discussed.

The approach taken to enactment uses logic programming. Logic programming and its principle implementation Prolog [Bratko 86], have previously been used by various researchers [Heimbigner 90, Lee 91], and appear to be effective in capturing process data and enacting process models. Prolog's declarative characteristics not only allow for straightforward mapping between the graphic and symbolic forms of the model, but also provide an effective tool for developing the driver through which a process can be enacted.

4.1 Mapping Activities to Rules

The manner in which the defined process is made enactable is not rigorous in a mathematical sense, but requires 1) identifying appropriate Prolog rules for specific process model elements and 2) generalizing these rules. The mapping exercise has been conducted using the process elements of Figure 3-8. These elements do not cover all of ProNet's modeling features but do provide sufficient generality to indicate that the approach has a high chance of success.

Returning to Figure 3-8, it can be seen that before any activity is performed, certain entrance conditions must be true. For example, in order to review version 3 of the change request (*review_cr*), the product *cr2 | 3* must be available. In addition, neither the condition *cr_appr* (i.e., the change request has been approved) nor the product *rev_doc1 | 3* must exist prior to execution of the *review_cr* activity. If all of these conditions are met, then the activity *cr_review* can take place and either *cr_appr* or *rev_doc1|3* can be generated. During process enactment, a sequence of statements (called *log* statements) is generated. These leave a trace of what activities have been performed, and are principally required to assure that once activities have been completed they are not performed again. On completion of each activity, a *log* statement is generated and added to the database, thus indicating that the activity has been completed. A *log* statement contains information indicating which activity has been completed, what inputs were consumed and what outputs were generated. As will be discussed in Section 5, the trace generated by the *log* statements is also useful in supporting the audit trail required for process verification.

The above discussion provides an overview of the enactable model. It is a declarative model in which each activity forms the core of a rule. A variety of approaches to dynamically enacting rules exist. Examples include the production system OPS5 [Brownston 85] or Prolog [Bratko 86]. Prolog was chosen because an implementation was readily available and the language is appropriate to this type of problem. A knowledge of Prolog will be useful in the discussions below; however, an understanding of the general approach taken should not require a prior

background in the language. The activities in Figure 3-8 will be used to define the mapping procedure.

Before doing this, however, a simple process element will be first described to show what the structure of the corresponding rule looks like. This is shown in Figure 4-1. Here the activity

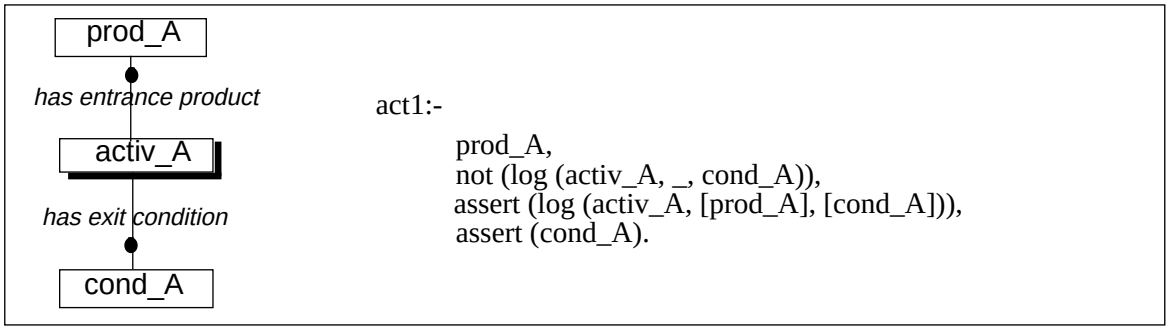


Figure 4-1 The Mapping Between a Simple Process Element and Its Prolog Expression

activ_A requires product *prod_A* as input and generates the output condition *cond_A*. The equivalent Prolog statement is shown to the right of the diagram. This statement indicates that 1) *prod_A* must exist in the Prolog database prior to performing the activity, and that 2) the activity *activ_A* has not yet been performed (through the *not (log (activ_A, _, cond_A))* statement). When those two conditions are true, the statement *(log (activ_A, prod_A, cond_A)*) is added to the Prolog database through the *assert* statement. This indicates that the activity has now been performed and, as a result, the output condition *cond_A* can be added to the database. The square brackets around the entities indicate lists in Prolog (albeit, in the above cases, one element lists).

The above example illustrates that each activity can be treated as a separate chunk of knowledge and is not explicitly coupled to any other activity. This allows for easier incremental growth of the model. The generation of outputs from the activity (e.g., *cond_A*) then allows other activities to be initiated. In order for an activity to be initiated, not only must the entrance conditions be satisfied, but the activity cannot have taken place already. This is tested for by the *not (log())* statement. We now look at a variety of typical process elements extracted from Figure 3-8. The examples to be discussed are associated with the numbered activities in Figure 3-8. As a result of examining these examples, we will be able to define a more general expression for the graphic-to-symbolic mapping. Once this is completed, we will then show how the resulting rules can be used as part of an enactable model.

Figure 4-2 (1 in Figure 3-8) shows the initial activity in the process. Note that the role involved (*field_rep*) is not specified in the rule. The manner of specifying the agents associated with activities will be discussed later. The symbolic form of this process element is conceptually very similar to the example of Figure 4-1. Figure 4-3 (2 in Figure 3-8) illustrates a more complex activity. In this activity, either the initial change request or a revised change request is inserted into the database. The *1* extension on *cr* (i.e., *cr1*) removes potential ambiguity from other, otherwise identically named *cr*'s generated by different activities (for example, see *cr2* in Figure

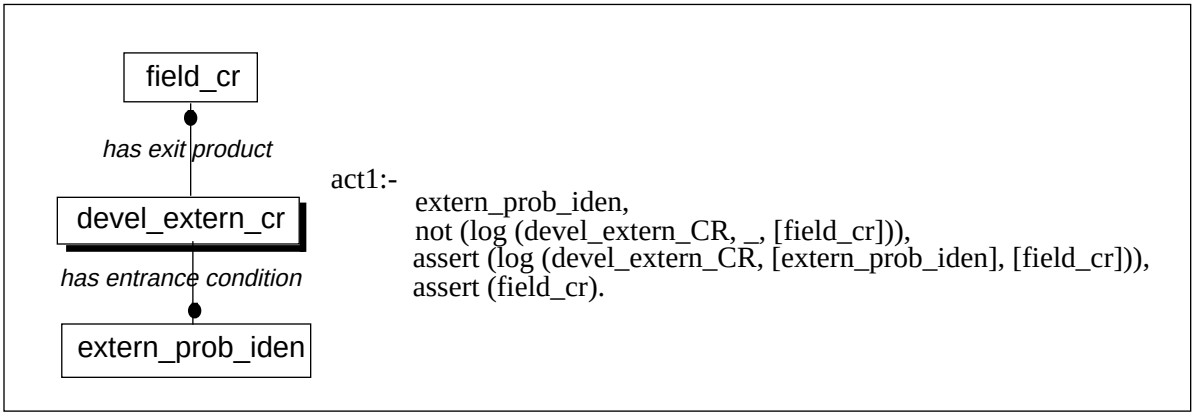


Figure 4-2 The Initial Step in the Process

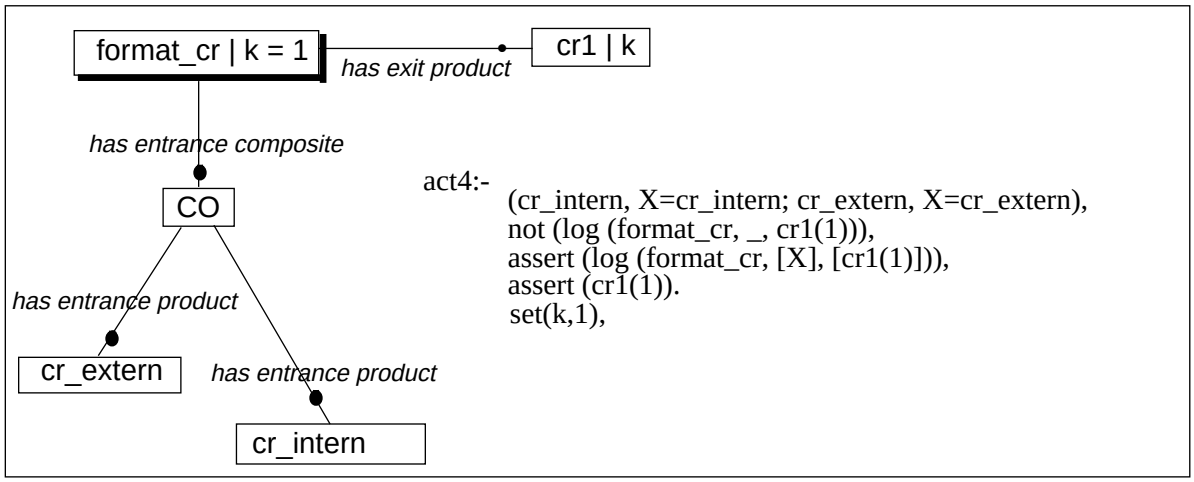


Figure 4-3 Combining Disjunctive Inputs

4-5). Version information is supplied by the version number after the vertical bar (for example, the *k* in *cr1 | k*).

The next two rules, *put_into_repos_A* and *get_from repos_A*, deal with database support. These process elements and their corresponding rules are shown in Figures 4-4 and 4-5 and can be identified in Figure 3-8 (3 and 4 respectively). In Figure 4-4, the k^{th} version of the change request is stored in the database *cr_repos.* (Within the Prolog implementation, this simply means that the name *cr(1)* is stored as an element of a list, called *cr_repos*, of cr versions.) The first statement in this rule (*ver(k,K)*) is added in order to instantiate the value of the variable k. To indicate to other interested activities that this task has been accomplished, the condition *cr_added(k)* is generated. Notice that this condition must also be given a version number. The function *put_ent* in the Prolog rule of Figure 4-4 is responsible for adding the change request to the database and also issuing the message that the transaction has taken place. Finally, it should be noted that the *put* in front of the activity name *put_into_repos_A* has a special significance in that this activity performs a *put* operation into the database. The *get* operation is the inverse of the *put* operation. Thus a condition is required to initiate a *get* and the *get* activity results in a copy of an entity in the database being released.

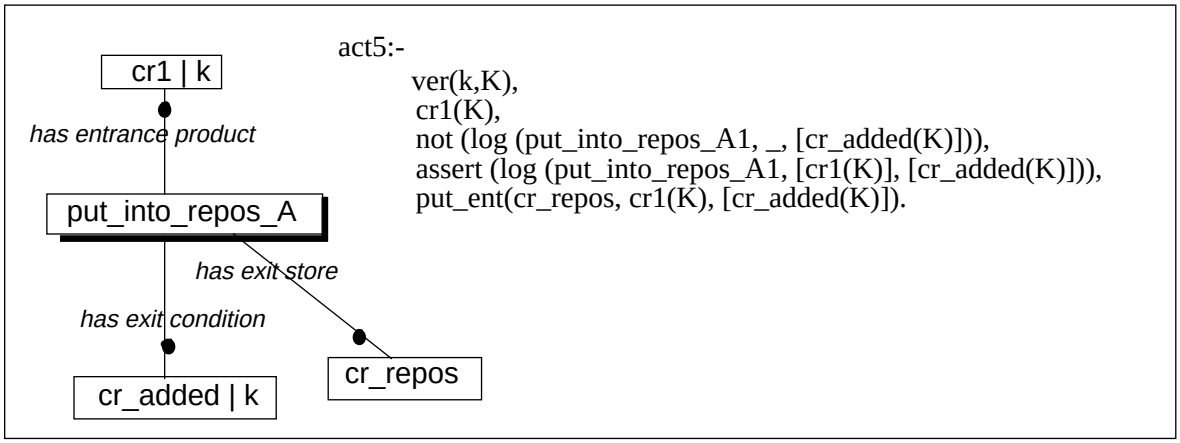


Figure 4-4 Inserting a Product into a Database

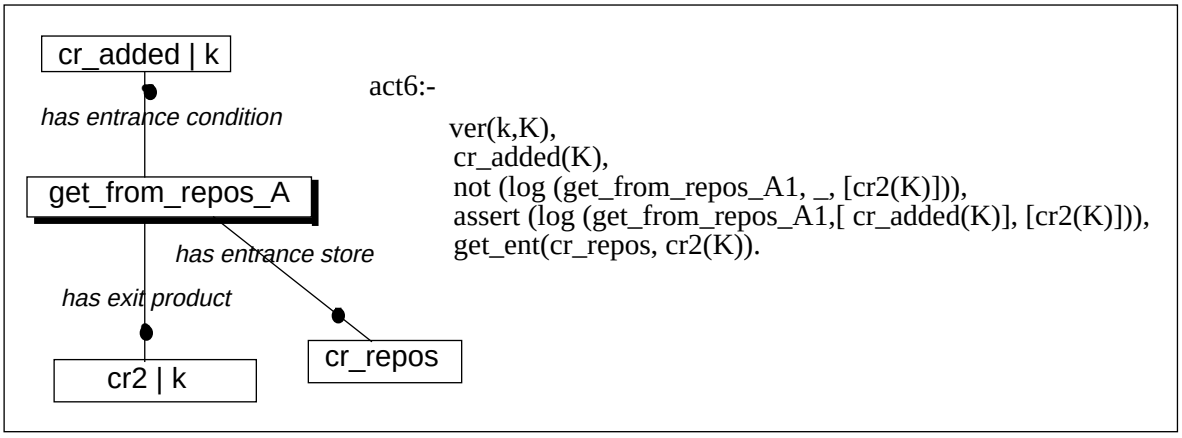


Figure 4-5 Removing a Product from a Database

The *review activity* (5) can have two outcomes: either the review succeeds (and it generates the condition *cr_appr*, or it fails to pass the change request, in which a review document *rev_doc1 | k* is written. The process element for this is shown in Figure 4-6. Note that, as with

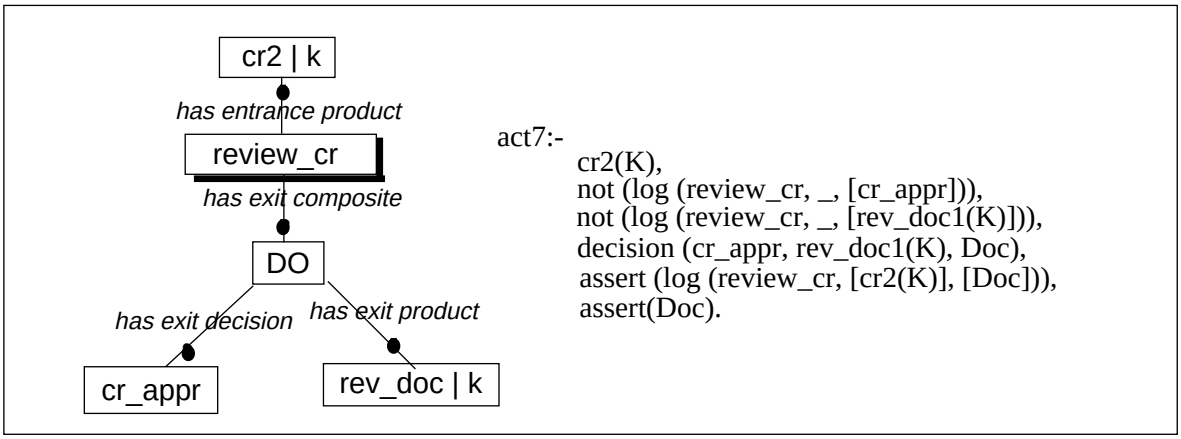


Figure 4-6 Making a Decision

any divergent OR statement (the DO box), a decision must be made as to which path to take. For simulation purposes, the *decision* function makes this choice based on a random number and, through this function, the probability of which decision is made can be varied.

The process element for the activity *put_into_repos_B* (6) is shown in Figure 4-7. This is very

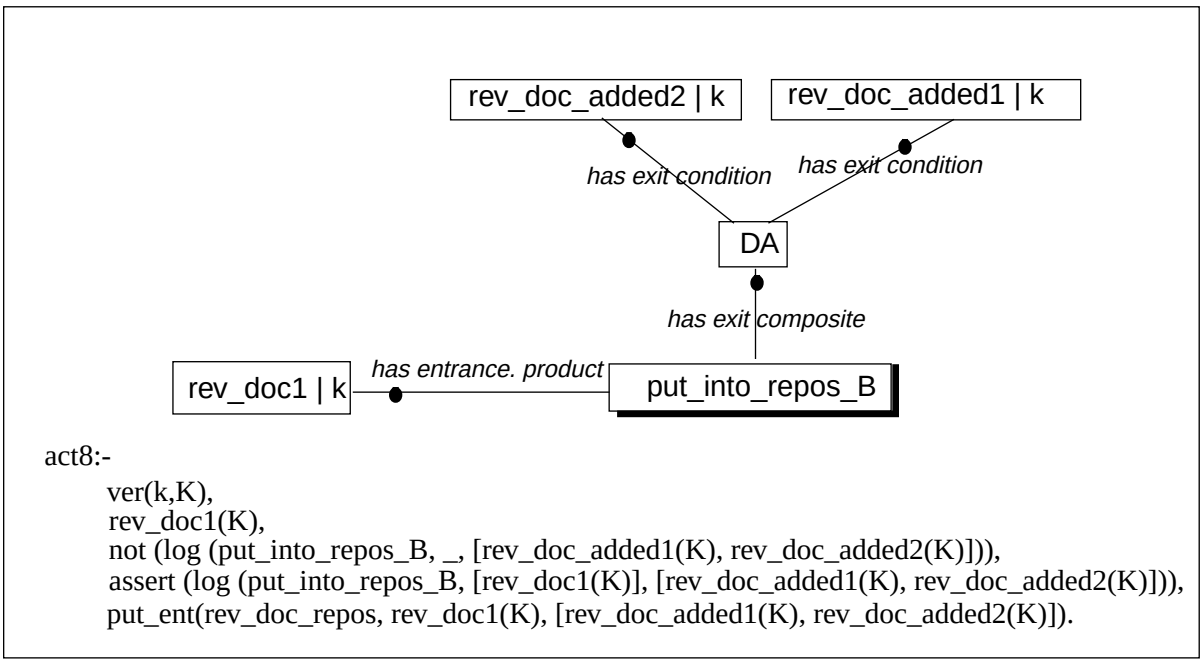


Figure 4-7 Generating Multiple Output Decisions

similar to the insertion of a product into a database as shown in Figure 4-4. However, this element generates two identical output conditions, *rev_doc_added1* and *rev_doc_added2*. This redundancy is not required for process enactment, since *get_from_repos_B* and *get_from_repos_C* could use the same output condition during process enactment, but it is required for process verification (as will be described in Section 5). The final activity to be illustrated is *update_cr* (7 in Figure 3-8). This is shown in Figure 4-8 and demonstrates two modeling concepts. First it shows how two entrance products (in this case, *cr3* and *rev_doc2*) support an activity through the use of the convergent AND (CA) junction. Secondly, it shows how a version number is incremented. The products entering the activity are associated with the unincremented number while the exit product is associated with the incremented number.

4.2 Generalizing the Rules

The above example and the associated Figures 4-2 through 4-8 provide insights into how to define the graphics-to-symbolic mapping. Table 3 provides a general procedure for performing this mapping. It can either be used as a basis for manually mapping a graphically-defined ProNet process or as a mechanism for developing a computer-based approach. The latter has not, to date, been done. This procedure is applied to each activity and the entities (i.e., products and conditions) upon which the activity is dependent. Table 3 has three columns. The first

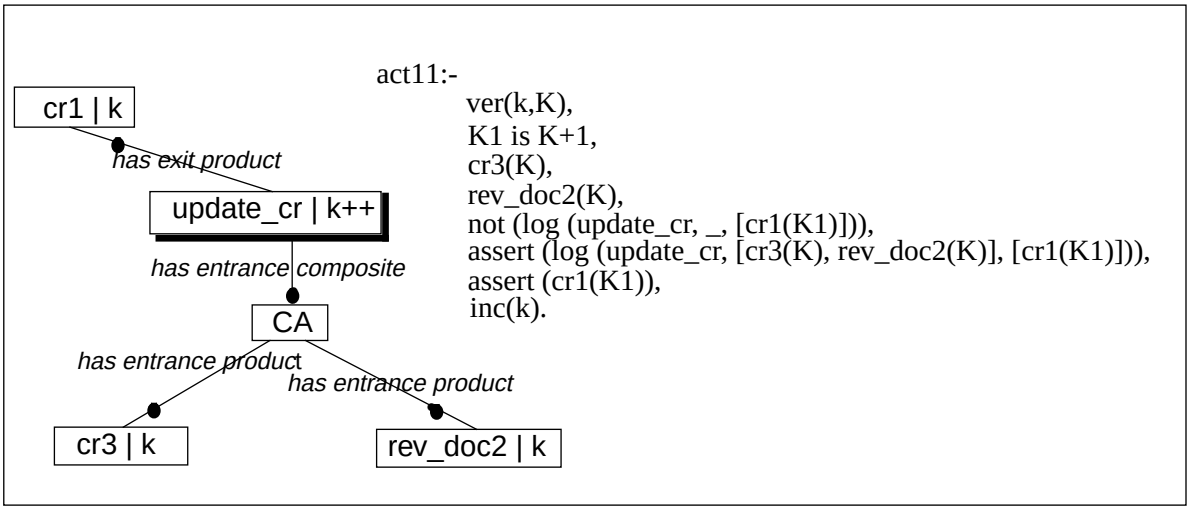


Figure 4-8 Combining Inputs and Incrementing a Version Number

Table 3 Mapping a Graphical ProNet Model to Its Enactable Form

#	Mapping Task	Example Statements
1	Identify input entities	prod_a_i, prod_b, cond_c
2	Identify output entities	prod_x, cond_y_j
3	Add *ver* statements for versions	ver(i, I), ver(j, J),
4	Add temporary increment statements	I1 is I + 1
*5	Assert input entities:	
5a	convergent AND (CA)	prod_a_i, prod_b, cond_c,
5b	convergent OR (CO)	(prod_a_i, X=prod_a_i; prod_b, X=prod_b; cond_c, X=prod_c),
*6	Test for existence of *log* statements:	
6a	divergent AND (DA)	not (log (act_A, _, [prod_a_i, prod_b, cond_c])),
6b	divergent OR (DO)	not (log (act_A, _, [prod_a_i])), not (log (act_A, _, [prod_b])), not (log (act_A, _, [cond_c])),
*7	Assert *log* statements:	
7a	convergent AND (CA), divergent AND (DA)	assert (log (act_A, [prod_a_i, prod_b, cond_c], [prod_x, cond_y_j])),
7b	convergent AND (CA), divergent OR (DO)	decision (prod_x, cond_y_j, Y), assert (log (act_A, [prod_a_i, prod_b, cond_c], Y)),
7c	convergent OR (CO), divergent AND (CA)	assert (log (act_A, X, [prod_x, cond_y_j])), --- (X as in 5b above)

Table 3 Mapping a Graphical ProNet Model to Its Enactable Form

#	Mapping Task	Example Statements
7d	convergent OR (CO) divergent OR (DO)	decision (prod_x, cond_y_j, Y), assert (log (act_A, X, Y)), ---------- (X as in 5b above)
8	Add item to database (*put_*...)	put_ent (repository_A, prod_b, [b_added]),
9	Remove item from database (*get_*...)	get_ent (repository_A, prod_x,),
*10	Assert output entities:	
10a	divergent AND (DA)	assert (prod_x), assert (cond_y_j),
10b	divergent OR (DO)	assert (Y), ---------- Y as in 7b above
11	initialize version numbers	set(i,1),
12	increment version numbers	inc(i),

column lists the item number. Some of these items are mandatory and must be included in every Prolog rule (these are identified with an asterisk). Other items have sub-items (a, b,...) which may be chosen depending on the nature of the current process element. For example, if the current process element has a DO as an output, items 6b, 7b or 7d, and 10b are chosen. Note that in Table 3, the only functions defined within the Prolog language are *not* and *assert*. All other statements and functions (e.g. *decision* and *put-ent*) are defined as part of the rule-driver.

To complete the discussion of the process model, three additional topics are addressed. These deal with:

- the actual rule structure,
- modeling the agents which perform the activities, and
- the rule driver through which the rules are enacted.

In the actual implementation, each rule is separated into two parts. In the first part, the entrance conditions, which test whether the rule should fire, are grouped into a ‘test’ category (under the heading *test*). These are items 5 and 6 in Table 3. Then the assertions, stating that the activity has taken place and that the output products and conditions have been generated, are placed in a second “action” category (under the heading *act*). These are items 7 through 12 in Table 4.1.This separation, which can be seen in the program listing provided in Appendix A.2, is required in order to find all satisfied entrance conditions prior to performing any activity. This provides a list of current actions which are then presented to the user. Statements linking each entrance condition to its action are defined and have the form *actRole(test3, act3, field_rep, 'e-mail external CR')*, where the test *test3* is linked to the action *act3*. The third field in this *actRole* data statement defines which role *(e.g., field_rep*) can perform this activity,

while the last field provides textual information about the activity (*'e-mail external CR'*). It should be noted that when an activity is performed, some implicit events may take place. For example, tools may be invoked to perform the actual operations, or subprocesses may be invoked to define lower levels of process granularity. Invocation of tools is not investigated in this paper. However, in Appendix A.3 an extension of the model defined in Section 4 illustrates how the approach can be generalized to account for hierarchies of sub-processes.

The second topic relates to how agents are modeled. Each activity should have one or more attached roles (or agents) responsible for the implementation of that activity. Prior to the actual performance of an activity, we must assign actual agents to these roles. While these agents are often persons, they can also be non-living entities such as a specific compiler version with specific flags. The mapping between the role and the actual agent is given by statements which have the form: *hasRole(field_rep, howard)*. Hence a list of *hasRole* statements must be part of the process definition in the enactable model.

Using Table 3, the rules which define the process can be generated. However, in order to make the process enactable, a rule driver is required. While the rules themselves are dependent on the process being defined, the rule driver is process independent and controls the order in which rules are fired as the process is enacted. The rule driver is discussed in somewhat more detail in Section 4.3, as this pertains to how the enactable model is used in a process context.

Details of the full model, as described in Section 4, can be found in Appendix A. This provides sample output from the demo process controller and lists the Prolog program, including the *test* entrance conditions, the *act* activities, the *actRole* and *hasRole* definitions and the rule driver.

4.3 User Interaction with the Process Model

Let us first look at how a user would interact with our simple enactable process model and then address the issue of managing a large rule-base. We have referred above to the *rule driver*, that is, the part of the system which selects and displays appropriate information (e.g., current activities) to the user. From an implementation point of view, the name *rule driver* is appropriate, but from the perspective of a user of the process, it is not. In this section it is more appropriate to rename it as the *process controller.*

The structure of the process controller is shown in Figure 4-9. It is (of course) defined in ProNet notation. After the project is initiated, or after any project activity has been completed, a list of all candidate activities which can currently be performed is generated. If this list is empty then the project has been completed. If the list is not empty then a subset of the list is generated which applies to the current user. The user then specifies which activity should now be performed. Thus the process controller cycles round the activities defined in the rule-base and on each cycle performs one of the activities. These cycles continue until the candidate list no longer contains any activities. Thus the controller acts like an rule-based production system

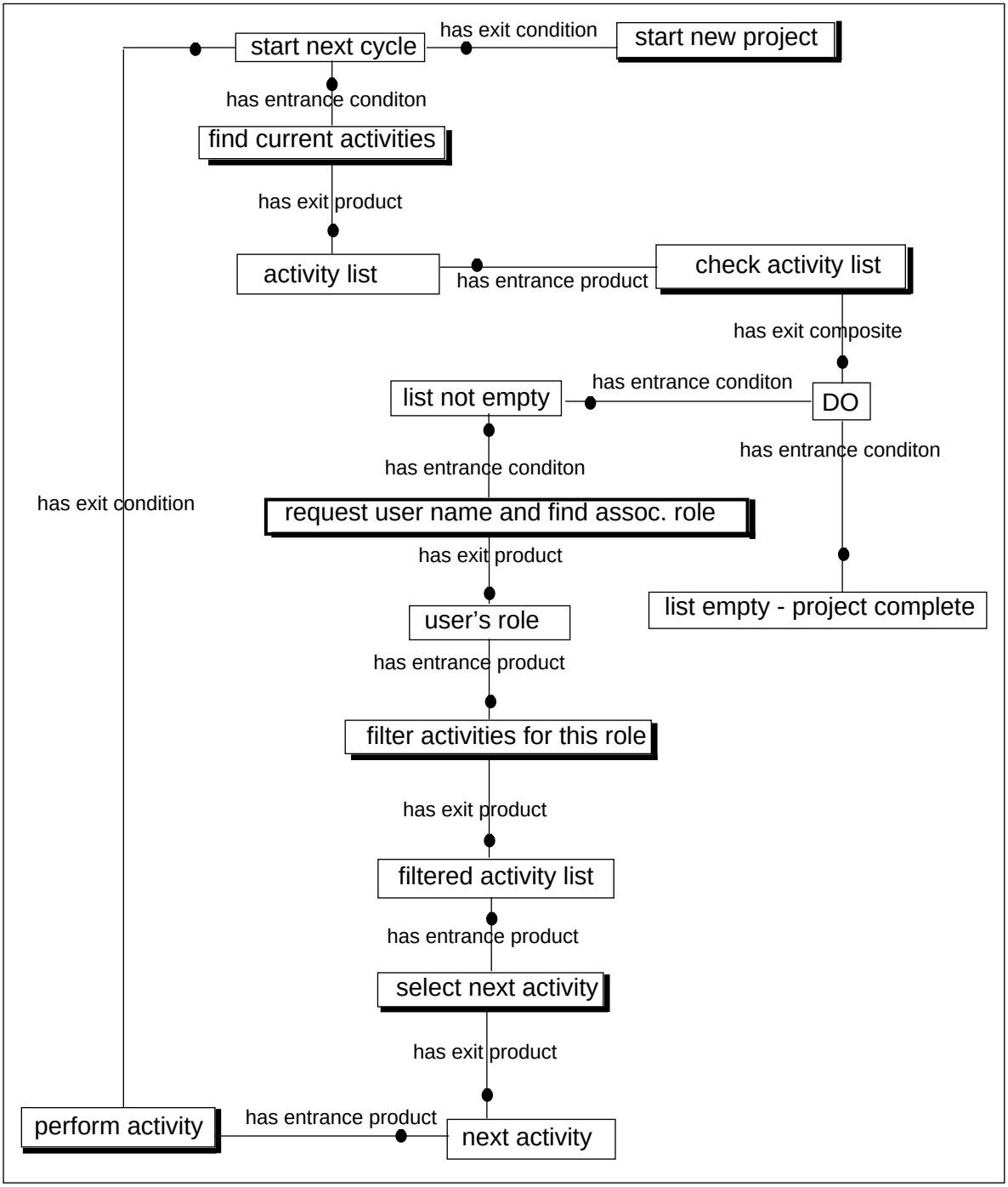


Figure 4-9 Process Controller for the Enactable ProNet Model

[Brownston 85]. From the above discussion and the model shown in Figure 4-9, it should now be clear why the activity entrance conditions are separated from the actual activities. The entrance conditions must all be checked in order to generate the activity list prior to candidate activity selection. (This selection process is the manual equivalent to conflict resolution in a production system.) While the process is being enacted, products and conditions are being generated, product versions are being added to and removed from databases, and a history

of the user-defined process path used is being stored. Appendix A.1 provides sample output from a user interacting with the process controller.

4.4 Managing the Rules

Declarative rule-based modeling, as discussed earlier, has appealing properties with respect to building models incrementally. Since each rule defines an isolated activity, it is relatively easy to build on or modify models as confidence and insight increase. In order to manage the large number of process model variants which are likely to arise, configuration management of these models will be essential.

One concern with the declarative approach is that of verification. With any complex process, the numbers of rules may run into the hundreds. Thus the complexity of interactions can be high, and verifying behavioral correctness or completeness may be difficult. Performing tests, such as for deadlock and reachability, will help to reduce errors. In addition, process enactment simulation will help to assure that the additions or improvements are well behaved before they are fielded. A second verification issue is related to assuring that the agents that produced the end-products actually use the defined process. The *log* statements defined above help provide a trace of the process used to develop the products. Through this trace, this historical process can be identified and verified against the defined process. This is the main topic of Section 5.

4.5 User Interaction with the Automated Process

A problem discussed in Section 3 is that of unnecessarily restricting the start of an activity until all the necessary entrance conditions are met. In reality there are many cases where preliminary work can be performed on an activity prior to that point when all the formally specified entrance conditions are satisfied. This restrictive behavior is unfortunately exhibited by the above model. However, the problem can be overcome in the following simple way. Rather than prevent the start of any activity, the process controller only requires that the exit products and conditions of the activity become available after all the activity's inputs exist. Thus an activity can be started at any time during the process; the only imposed constraint is that the activity cannot terminate until all the inputs are accounted for (and of course, the activity itself has been performed).

Another restriction on the current model is that the process driver only accounts for forward chaining and not backward chaining. Forward chaining implies that activities are performed only when their entrance conditions are met (as described in the previous paragraph). However, it could be that an agent wishes to perform an activity, some of whose entrance conditions are not met. The agent may wish to satisfy that constraint by "backtracking" through the sequence of as-yet unperformed activities in order to generate the unmet entrance condition. Figure 4-10 provides a simple example. Assume that *activity A*, *activity E*, and *activity F* have been completed.

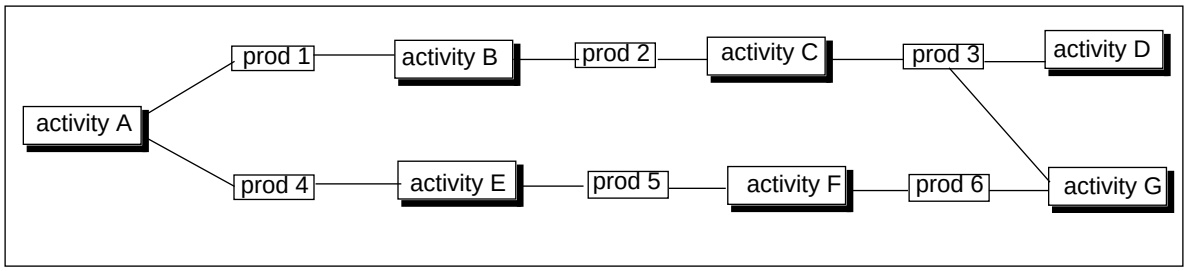


Figure 4-10 Backtracking to Define Activity Inputs

The agent next wants to start on *activity G*. However, *prod 3* is not yet available. To generate *prod 3*, *activity C* must be performed. This in turn requires *activity B* be performed. Hence to satisfy the entrance conditions to *activity G*, one option is to "backtrack" through these unperformed activities necessary to support *prod 3*. This control scheme is tighter than the one discussed immediately above, in that is still requires all input entities to be available before an activity can be started. However, it does allow an agent to start an arbitrary activity at any time; the system will then guide the user to perform the necessary precursor activities.

A limitation of process enaction, as it has been described, is related to the manner in which activities are displayed to the user. The simple user model presented provides a list of possible "next" activities to be performed (see Appendix A.1). However, this provides no context in which to make the selection. A significantly improved interface would show the graphical process model, driven by real-time execution data, and indicating project status [Mi 1992]. Color or symbolic coding of such information as the degree of activity completion or ownership of activities, would provide instant feedback on and control of the project. By clicking on an activity, the agent would thus open that task for development.

4.6 A Relational Definition of the ProNet Notation[3]

As discussed in Section 3, a model defined using ProNet notation has elements in common with entity-relation models. However, the underlying ProNet modeling and enaction concepts can be defined at a meta-level, also using a relational model [Monarchi 92]. This is briefly described below.

Using Figure 3-6 as the basis for a process simulation, we might cycle through two unsuccessful reviews (*review CR*) before satisfying the reviewer and generating the condition *CR OK*. With this scenario, we would produce two versions of the revision request and three versions of the change request before generating the *CR OK* condition. Table 4 lists the information that both defines the process model and uniquely specifies this execution. Note that executed activities (the *Xi*'s), product and condition versions, and agents assigned to executed activities are required to complete the specification of the enacted process. Thus, it can be seen that the essential entities required for ProNet enaction are: activities, artifacts (i.e., products and conditions), executed activities (X1, X2, etc.), artifact versions (1,2,3, etc.), roles and agents.

3. I'd like to thank Alan Brown for suggesting this concept.

Activities	Artifacts
initialize CR	CR
update CR	rev req
review CR	CR OK

Input artifacts	
update CR	rev req
update CR	CR
review CR	CR

Output artifacts	
initialize CR	CR
update CR	CR
review CR	rev req
review CR	CR OK

Artifacts consumed		
X2	CR	1
X3	rev req	1
X3	CR	1
X4	CR	2
X5	rev req	2
X5	CR	2
X6	CR	3

Artifacts produced		
X1	CR	1
X2	rev req	1
X3	CR	2
X4	rev req	2
X5	CR	3
X6	CR OK	1

Artifact versions	
CR	1
CR	2
CR	3
rev req	1
rev req	2
CR OK	1

Activities executed		
X1	initialize CR	Susan
X2	review CR	Ed
X3	update CR	Paul
X4	review CR	Ed
X5	update CR	Paul
X6	review CR	Jock

Roles	
developer	initialize CR
developer	update CR
reviewer	review CR

Agents	
Jock	reviewer
Susan	developer
Paul	developer
Ed	reviewer

Table 4 Relationships Among Model Entities

Figure 4-11 then generalizes the model of the relationships between these entities, using Table 4 as a basis. Note that the five entity structures on the left-hand side are required for process definition, while the five on the right reflect the additional structures required for process enaction. The symmetry of concepts on the definition side and the enactment side is quite striking. This can be summarized as follows:

definition <--> enaction

roles <--> agents

artifacts <--> artifact versions

activities <--> activities executed

This relational database approach to process enaction is not pursued any further here, but may have interesting implications.

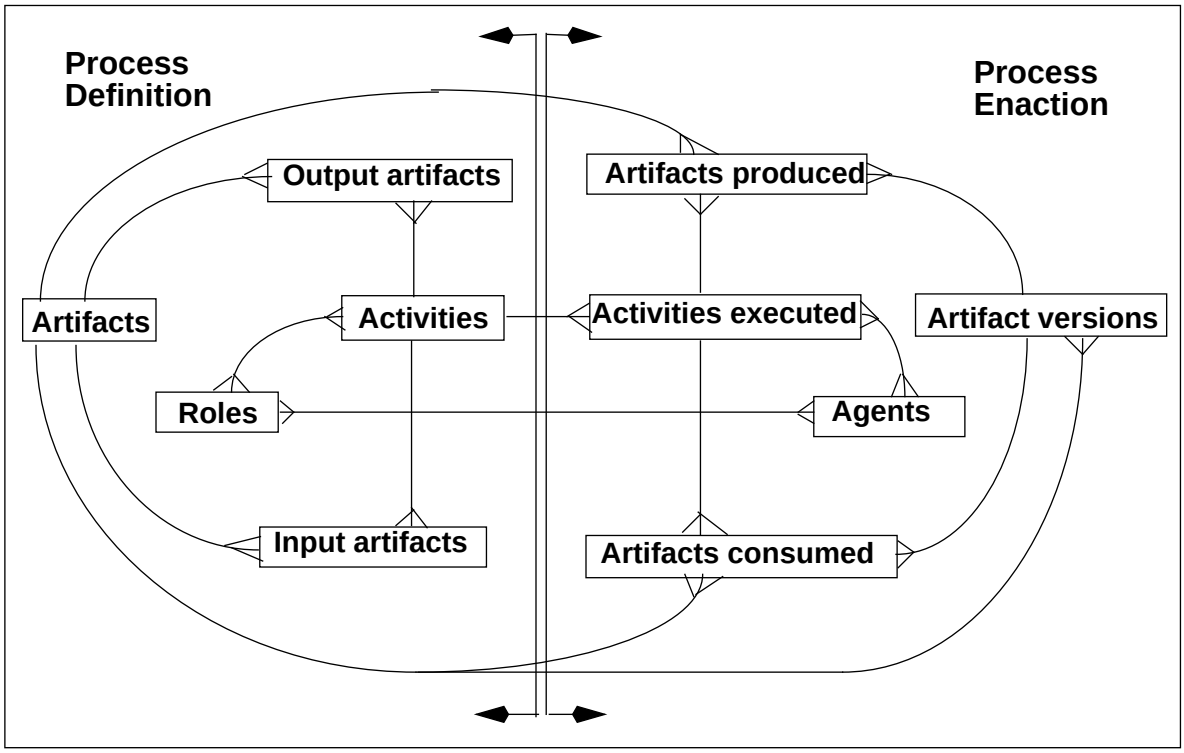


Figure 4-11 A Relational Model Linking ProNet Entities

5 Software Process Verification

This section of the paper presents an approach to assuring that any software product meets the requirements of the defined process, while at the same time accounting for deviations from the prescribed process [Stanley 92]. Process verification, as defined here, is a post-analysis of process data gathered during product manufacture (down to an appropriate level of granularity). The act of verification will identify where deviations from the agreed-to process occurred and allow a determination of the severity of those deviations. (See Step 9 in Figure 2-1).

Deviations need not invalidate a non-conforming process path. Indeed, they could be benign or even beneficial. Some process changes may be due to unforeseen circumstances such as accounting for the replacement of an assigned developer by another developer in the execution of certain tasks. Others may be made to allow for creative improvements such as the replacement of a specified compiler by a more efficient one. Whatever deviations occur, a post-project evaluation can be performed to assess their impact on the resulting software quality, provided that the enacted process is tracked. The approach to verification follows. Throughout a project, the artifacts which are produced by the developers are automatically recorded, along with information on who produced them and how they were produced. On project completion, the manner in which these artifacts were produced is then compared to the requirements of the defined process.

The approach taken rests heavily on the process modeling concepts described in previous sections. In particular, the *log* statements used to record process history are central to verifying the correctness of the as-implemented process. If the process has been followed exactly as specified in the enactment model, then a set of *log* statements, consistent with the defined process, must exist. However, as described above, there are likely to be situations where no path through the implemented process matches the defined process.

Process verification affects several aspects of software development. First, the tools used to develop the software will have to be instrumented to record what is being performed. Second, there are implications for metrics tools since process data will have to be recorded. Third, consistency of intermediate product versions is essential to assure final product quality. Hence, there are implications for configuration management. Finally, there are implications for process, process specification, and process modeling.

It should be noted that process verification can be the basis for process certification. In this context, “process-certified” software implies that either the final products are guaranteed to have been generated using an agreed upon process, or non-conformances are identified and justified. This does not guarantee that the software performs to specification, but certification may be regarded as a necessary if not sufficient guarantor of software quality. Such certification may be of interest to any organization which subcontracts out software development.

5.1 The Basis for Process Verification

The basis for verification has already been laid through the ProNet notation presented in Section 3 and the subsequent discussion on process enactment in Section 4. The verification process is the inverse of the enactment process. In enactment, activities follow the forward flow of time and the events are recorded (through the *log* data entities as described Section 4). However, verification takes place by starting with the end product, applying the *log* statements generated during process enactment, and working backwards. The object here is to prove that the process, as defined through the *log* statements, is consistent with the defined process as reflected through the process model. The final product establishes the "initial conditions" for verification, and one *log* statement is consumed as the corresponding activity joins those in the set of verified activities. While this approach is not as formal as proof of correctness of programs [Greis 81], it is interesting to note that both perform their verification proofs in this backward manner.

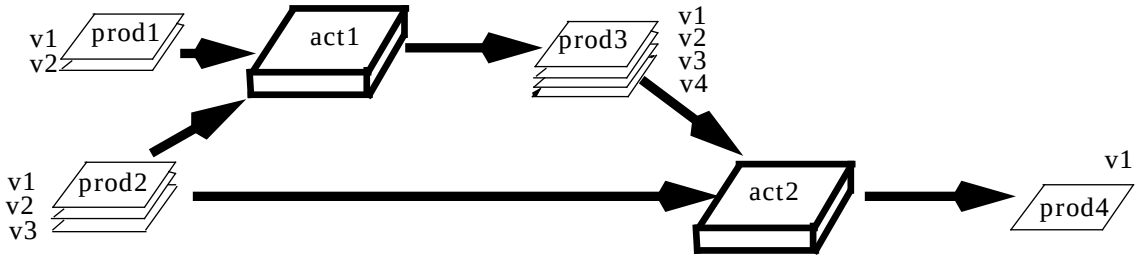
In summary, the following sequence of activities is performed for process verification:

- Prior to initiation of the software project, the software development process is established using an appropriate process modeling technique such as ProNet. The level of process detail should be such that important products can be tracked.
- During the project, all critical activities are instrumented such that their input and output products and conditions (and their version identifiers) are recorded. Conditions must be tracked, since these are important process indicators.
- When the final product has been generated, comparison is made between the process model and the *log* statements generated during project execution. For verification of the final product, complete consistency must exist between a subset of the intermediate artifacts produced and those expected by the process model.

This sequence need not be applied only to a complete project. If the project is sufficiently large, then intermediate products from sub-projects can be individually verified using the appropriate sub-processes. These intermediate products then contribute to the verification of products at a higher level in the overall project.

5.2 Implementing the Approach

In implementing process verification, the goal is to prove that at least one subset of the collected *log* data statements is consistent with the defined process. Figure 5-1 illustrates the elements of this theorem-proving approach. The diagram in Figure 5-1 shows a process model element in which products *prod1* and *prod2* support activities *act1* and *act2*, and where these activities generate products *prod3* and *prod4*. This process is defined by the two *verify* statements also shown. The *log* statements represent the historical data gathered during execution



Process model data:

```
verify:-
        ver(k,K),
        prod4(K),
        retract (log (act2, [prod2(L), prod3(M)], [prod4(K)])),
        assert (prod3(M)),
        assert (prod2(L)),
        retract (prod4(K)),
        verify.


verify:-
        ver(m,M),
        prod3(M),
        retract (log (act1, [prod1(K), prod2(L)], [prod3(M)])),
        assert (prod1(K)),
        retract (prod2(L)),
        retract (prod3(M)),
        verify.
```

Tracking Data:

```
log (act1, [prod1(1), prod2(1)], [prod3(1)])
log (act1, [prod1(1), prod2(2)], [prod3(2)])
log (act1, [prod1(2), prod2(2)], [prod3(3)])
log (act2, [prod2(3), prod3(4)], [prod4(1)])
```

Figure 5-1 A Simple Process Model with Process Data

of the process. The first *log* statement, for *act1*, uses versions *v1* for both input products *prod1* and *prod2*. The resulting version of product *prod3* is also version *v1*. As with the enactment model, the verification model is written in Prolog notation. In fact, as will be seen later, the implementation of the verification procedure is very similar to that of the process enactment procedure.

Let us now compare the process model data (the *verify* statements) with the tracking data (the *log* statements). The first *verify* statement asserts:

IF
    prod4, version K exists
AND

activity *act2* generates *prod4, version K* as output and uses *prod 2, version L* and *prod3, version M* as input

THEN

delete the *log* statement

add *prod3, version M* and *prod2, version L* to the database

delete *prod4, version K* from the database

find the next applicable *verify* statement to execute

The initial *ver* statement in the verify rule simply identifies the current value *K* of the version number *k*. The second *verify* statement in Figure5-1 has a construction very similar to the first. The values *L* and *M* of version numbers *l* and *m*, are determined when the appropriate *log* statement is instantiated. In this example, *prod4* takes on the version value 1, thus forcing (through the *log* statement for *act2*), the version numbers *l* and *m* to take values of 3 and 4 respectively. Hence *prod2*, *version 3* and *prod3, version 4* are added to the database. Since *prod3* is now in the data base, the second verify rule is now activated. However, there is no *log* statement which has *prod3*, *version 4* in its output list. Hence there is a disconnect in the process, and the verification fails.

This simple example also illustrates how the process verification procedure does not care what activities have been performed (e.g., there may be many redundant *log* activities), so long as there is a core set which can be threaded together consistently as defined by the process model.

5.3 The Verification Demo Program

As with process enactment, process verification is implemented through a rule-based, forward chaining production system. However, unlike the enaction program, verification is not, and does not need to be, interactive. The “initial conditions” for verification are the name of the final product and the *log* statements generated during execution of the project. As verification takes place, intermediate products appear and disappear and log statements are consumed, as the valid path extends further backwards in time. At the end of the verification procedure (assuming the process is verified), all the *log* statements associated with the verified path will have been be consumed and the entry products/conditions for the process will be generated. The symmetry between process enactment and verification is shown in Figure 5-2.

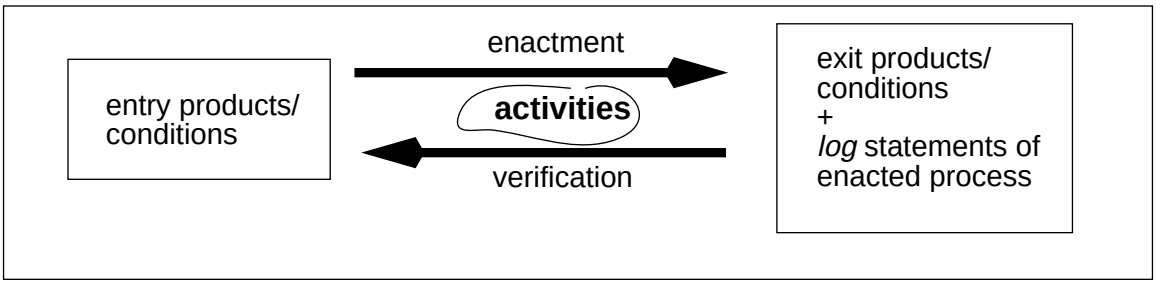


Figure 5-2 Symmetry Between Process Enactment and Verification

The *verify* program is listed in Appendix B. This program consists of:

- a set of clauses to support needed functionality, such as for adding entities to, and removing entities from stores,
- a set of clauses which define the verification rules, each rule being associated with an activity in the defined process, and
- a set of *log* statements which were previously generated during process enactment.

The process used in verification is identical to that defined in Figure 3-8, and the *log* statements were physically removed from the output of an enactment run. Thus there is complete consistency between the enactment model and the verification model. It should be noted that, while we have not discussed automatic generation of the rules for a verification model, such automation could be performed using a mapping quite similar to that shown in Table 3.

5.4 Implications for Verification

The exploratory work described above has some significant implications. These fall under the main headings of software quality and process certification, configuration management, tool integration, user interaction, metrics and finally, process improvement. These are briefly discussed below.

- **Software quality and process certification**. On completion of a project, a verification analysis can formally be made to assure that all steps have been completed and that they have been completed in the right order. Such formal verification may have implications for contractor process certification and the ISO 9000 standard [Kalinosky 90].
- **User interaction**. As discussed in the introduction, a major concern in enforcing effective process is the imposed restrictions which software developers feel either from a supervisor, or from an automated environment. The approach to verification in no way restricts the manner in which work is accomplished in terms of detailed oversight. It only requires that, when the project is completed at least one consistent path through the agreed upon process has been taken, or if deviations from the defined path have been taken, they can be justified.
- **Process improvement and metrics**. Throughout a project which uses process verification, significant quantities of historical data will be gathered. This data can be used, along with like data from other projects, for long-term process improvement. Data from each project can be analyzed to identify, for example, where bottlenecks, inefficiencies, and inaccurate estimates occur. By having formally-defined process models and data from their enactment, considerable insight can be thus obtained. This provides all the necessary ingredients for both qualitative and quantitative process improvement.

6 User-Oriented Issues with PCDEs[4]

The previous sections have dealt primarily with technically-oriented issues associated with automating software production. However, implementing a process-centered development environment involves much more than just addressing the technology. Indeed the success of adoption rests at least as heavily on personnel, organizational and cultural elements. This section thus looks briefly at a variety of more user-oriented problems facing organizations wishing to adopt a PCDE. Many of these issues are also relevant to software development environments in general.

A significant fraction of software today becomes “shelfware” [Page 92], in part because it may not meet users’ functional requirements, because of lack of appropriate user training, or because of lack of compatibility with the project’s process. Other elements may also contribute. [Boone 91] discusses factors which have been impediments to the successful introduction of CASE technology. Such factors include the difficulty which developers have in adapting to the different philosophy which CASE tools impose upon them, and the discomfort which managers have when coding doesn’t begin until much later in the development cycle. These changes in working habits will only be magnified when complex PCDEs are introduced.

A PCDE will be costly to install, in terms of the initial investment in software, in terms of the required training, and in terms of its adaptation to the needs of the projects using it. There will also be financial and technical investments in its long-term maintenance. The likelihood of failure in using such a system will be higher than with individual products. Given the high cost of this kind of investment, an organization may shy away from making more than one attempt at installing a PCDE if the first attempt is perceived as a failure. However, using an approach which relies heavily on graphical support for process definition, process enactment, user interface, and process maintenance will increase the likelihood of success of such systems.

In order to develop an effective PCDE, a review of some relevant user-oriented, adoption and technology transition issues is thus appropriate. These issues are the subject of the rest of this section.

6.1 The Application of Automated Support

If not designed properly, automated support may get in the way rather than help. Automating a process just because the technology exists can be like building a radar-activated mouse trap -- it is too complex, too costly, and does not meet the user’s simple needs. What should be automated? First, it is clear that automating those processes that are well understood (i.e., defined) makes sense. Thus islands of automation may be established, with the later possibility of building bridges between these islands. Second, tasks that are tedious should be automated, thus allowing the developer to concentrate on the more creative aspects of the job. Third,

---

[4] Parts of this section have been adapted from an article appearing in IOPener, Volume 1 No. 5, November, 1992.

tasks where manual involvement frequently introduces error should be candidates for automation.

Areas where automation can be of significant help include configuration management (CM), change tracking and metrics. Others include code reviews, document reviews, requirements management and project management. Today CM is already well automated and certain CM tools [e.g., CaseWare 92] increasingly allow for a customized process to be defined by the end-user. Much work still needs to be done in examining the overlap between CM and process, but systems like ISTAR [Dowson 86] and others [Bernard 89, Mahler 90] have investigated this topic. Process models can provide insight into what artifacts should be placed under CM control and what process metrics should be gathered. In the CM case, our simple process model provides some guidance. In a well-formed model, those artifacts which connect activities to activities are candidates for CM control. Thus, both product versions and condition versions should be placed under configuration management. In this context, tracking condition versions allows us to roll back in time to any prior part of the process and reconstruct what has occurred. This process data may best be maintained in a data structure similar to the *log* statement of our process enaction notation. The required process information also provides guidance on what metrics should be gathered from a process point of view. This information is similar to the CM information just discussed.

Not only must process models be sufficiently flexible to account for a wide variety of scenarios, but they must also allow for the adaptation of existing scenarios. Process modification can take place in at least in two ways: 1) as part of a process improvement effort, or 2) “on the fly” while in the middle of a project. The former is based on a long-term strategy, supported by metrics gathering and analysis, and implemented through process redesign and verification. While this is challenging, with sufficient lead-time the inherent risks can be minimized. However, real-time modifications to an on-going process contain significant risk. Nevertheless such changes may have to be made for a variety of reasons. For example, the process may not be performing as expected (i.e., it was not adequately debugged), or as mentioned earlier, circumstances such as new schedule constraints may force process changes. With a manually implemented process, changes can be made on an informal basis. However, if the process is automated, the ability to adapt the process may be significantly restricted.

Process support tools must therefore have the flexibility to support rapid process modification. Tools to support such needs as graphical process definition, automatic translation of graphically defined process to enactable form, verification of the completeness and correctness of the process model, or part of it, and process simulation capability will be very useful in this regard. These tools will of course also be of significant help in long-term process improvement efforts. Finally, it may be necessary to design these tools so that a project can degrade gracefully into manual process control if the automated mode does not perform adequately.

6.2 Organizational Factors

A PCDE must consider the needs of multiple roles, such as upper management, project management, and technical development. Each of these roles has different support and information requirements; in general lower levels in the hierarchy will support the higher levels. For example, developers will provide their project management with technical and status information related to tasks. Metric information may or may not be gathered automatically for management review and analysis. In addition, planning and schedule information will flow from project managers to upper management. A PCDE must have an understanding of entities such as activities, roles, products and of the relationships connecting those entities. Information about the entities and relationships should be reflected in the data objects underlying the environment.

Because process support environments are likely to be large and relatively complex, they are most appropriate for large structured organizations involved with major software systems. For such organizations, having common data schemas is important for communications, project integration, metrics gathering and for process improvement. Hence it seems appropriate that these schemas should be defined at the organizational level. For similar reasons, defining a user interface that is as consistent as possible is appropriate at this level. Different technical departments within an organization will need different tool sets and may use processes tailored to their specific needs. Hence, specification of tools sets and department-specific processes may be needed at a lower level of organizational granularity.

6.3 Process Needs of Managers and Developers

In general, project management needs will include the ability to access information such as quality assurance reports, test reports and status reports on products and tasks. On the basis of this information, the manager can make decisions on future courses of action. These decisions set broad constraints on how developers perform their technical work. The means of imposing management decisions on technical personnel should be through a well-defined process; that is, the developers should understand the ground rules within which they can perform their work. This may be called the defined management process and is an enabling mechanism for a productive working environment.

While the manager's view of process tends to be one of control, the developer's view is one of support. Developer's processes are related to product development and involve such activities as design, development, and testing of software products [Humphrey 92]; they are thus quite different from management processes. The developer wishes to use automated process support to provide guidance when it is requested and relieve menial chores when possible. Automated process support tools need to provide a sufficiently broad range of functionality to express these different needs. One area where automated support is critical is metrics collection, since developers may be reluctant to invest time in an activity that they may perceive to be of primary benefit to management.

Within these two extremes, there is a third category of process. In larger projects, groups of developers must cooperate [Ellis 91] in product development. In this case, a cooperative process should be defined which involves aspects of both support for and control of products. Clearly, when multiple developers participate closely in a task, issues of control at the developer level must also be addressed. At a minimum, configuration management processes are important here [Dart 91]. In the future, special tools which allow developers to interact, for example in concurrent debugging [Dewan 93] may become more prevalent. Thus the potential influence of groupware products in PCDEs needs to be investigated.

If a software technology does not meet the user's needs, it will not be accepted. In a similar way, if a PCDE is not well adapted to the way people work and interact, it is likely to be subverted. By enforcing rigid conventions on the software process, an automated environment may not only take away the feeling of personal control but may prevent a developer from showing initiatives that improve efficiency. For example, in a rigidly automated process, a developer may be unreasonably prevented from initiating a subsequent task because an earlier task has not been completed. Thus, either the level of granularity of the modeled process was too coarse (i.e., there was insufficient resolution to account for small tasks), or it was too intrusive (it should not even have attempted to control tasks in this way). Section 4.5 suggests an alternative solution to this problem.

Automation should primarily liberate the software development team from chores which can be automated (e.g., configuration control, gathering of metrics). Second, it should enforce process in a broad sense i.e., provide a framework into which individual processes can operate with significant freedom. Other constraints are probably appropriate for specific projects, but in adding them, a trade-off between the perceived benefits (productivity, management control, etc.) and the human impact of the constraint should be carefully assessed. As Humphrey [Humphrey 89] states: "[T]he environment should provide strict enforcement of liberal process." Humans can be very creative in circumventing systems which do not support their needs.

6.4 Adapting the Process to Unforeseen Circumstances

Producing a computer program with an effective understanding of user-oriented process is a challenge. Like all knowledge-based computer models, process programs rely on reasoning (for example, rule-based or state-transition-based) which explicitly takes into account only specified and understood changes in the system behavior. While a well-regulated process will have a significant degree of predictability; unaccounted circumstances may occur. These are much more difficult to handle within an automated environment.

The problem of what mental models humans apply to particular situations is one which the Danish industrial psychologist Jens Rasmussen has addressed. He perceives three levels of cognitive abstraction. At the lowest level there is the reflexive "if this happens...then do that" reasoning that is applied under normal conditions. This he calls the "skill" level. Computers modeling this level have shown significant success using, for example, expert systems. At a

second level, the expert may have to rely on past experiences to reason about a condition which is not normally encountered; from these past experiences he will then formulate a plan. An example might be how to modify the process if a hard deadline is approaching. This Rasmussen calls the “rule” level; while amenable to computer analysis, this is more challenging than the skill-based regime. Finally the expert may encounter an altogether new situation where reasoning from first principles is required. Examples of this might be loss of critical project data through natural disaster or because of a virus-infected repository. This Rasmussen calls the “knowledge” level; this level probably requires complete human intervention.

These examples indicate how the human expert modifies reasoning levels according to on the complexity and novelty of the situation [Rasmussen 79]. It is difficult to build this kind of flexibility in reasoning into a software system. This is not to say that automating the process is inappropriate, rather that the limitations must be understood. Addressing the limitations of PCDEs may mean, for example, that they should have the facility to be modified “on the fly” (so that incidents at the “rule” level can be accommodated), and should be designed to gracefully degrade to a manual process (so that incidents at the “knowledge” level can be accommodated).

6.5 Lessons from Groupware

Human-computer interaction has been extensively analyzed by people in the field of computer-supported cooperative work (CSCW) [Ellis 91]. The products developed (usually called groupware) are aimed at supporting teams of people who use networked computers to facilitate joint efforts. Generally groupware supports three functional categories: communication, collaboration, and coordination. A common example of a communication package is e-mail; an example of a collaboration package might be an editor that allows simultaneous update from multiple terminals; an example of a coordination package might be a group scheduler. Most of these tools are of a generic nature to support non-technical or management goals. However, because they have to deal with how humans interact in tightly coupled settings, research in this field offers insights into issues relevant to the process constraints on environments.

Within the context of PCDEs, there are several areas where groupware technology is likely to be important:

- specification/design
- documentation
- code inspections
- project planning
- reviews (e.g., of change requests)

Some of the above (e.g., code inspections) are truly synchronous activities, that is, the group participates at the same time, either at different terminals at the same location or geographically dispersed. In others (e.g., documentation) the mode of interaction is more likely to be

asynchronous. Some lessons learned from groupware evaluations which bear on software PCDEs are:

- Don't try to get too elaborate with sophisticated functionality. Instead focus on a system which emphasizes basic functionality and ease of access and does these well from a user point of view.
- If non-electronic analogies or metaphors can be used to describe entities/ attributes, use them.
- The system's architecture should be designed from the beginning to reflect seamless data and control integration. The difference between a system where one can move effortlessly between applications and one where such movement is awkward can be crucial to success.
- Be careful in adding features which may have advantages for one group while imposing the work on another group. (This is the “what's in it for me?” syndrome.)
- Don't try to over-automate the process as all possible configurations that the system may encounter cannot be predicted. The need for manual control over process execution will therefore be necessary.
- Design the environment for changing requirements. Human needs are very difficult to predict, and the system must be adaptable to account for user experience.

These points are also discussed in [Bull 90, Grud 88, Sing 91].

6.6 Configuration Management, Conflict, and Cooperation

Those aspects of ProNet dealing with 1) versioned products or conditions and 2) the *put* and *get* commands, allow the notation to explicitly model some basic configuration management operations. In CM terms, *put* and *get* correspond to check-in and check-out. Currently, operations such as branch and merge have not been not addressed, nor has the ability to lock the store after a product has been checked out. At a minimum, a locking mechanism could be added to the notation with little difficulty. However, to be fully responsive to users, any realistic process enactment language needs to go beyond simple locking.

In any practical PCDE, the ability to handle products within a cooperative group environment could be important, particularly at the software development level. Currently configuration management tools do support cooperative development. For example, tools such as SHAPE, SMS, and NSE provide isolated workspaces, and the ability to resolve conflicts when merges are made between workspace product versions and the public repository [Dart 91]. In particular, NSE [Feiler 90] provides for multiple workspaces. Each workspace can have a virtual copy of the original parent environment; only when a file is checked out in the workspace is a physical copy of it made. If files in the parent environment have been updated by one developer, then modifications from another developer (one who has also removed a copy of the unmodified parent) cannot be merged into the environment. The second developer must perform

a local merge of the files and debug the two sets of changes before updating the parent environment.

The above approach to parallel development does relax the constraint that development is strictly serial. However, it still does not support close cooperative development. For example, two persons may not be able to work in an interactive way on a section of code. As stated in [Bargouti 90],

> *Having the development process explicitly encoded does not alone solve the problem of supporting multiple users, a requirement for any large-scale software development effort. The basic problem is the inability to allow concurrent access to project components while still maintaining the consistency of these components.*

The above issues of concurrency and synchronization have been addressed neither within the ProNet modeling language nor in its enactable form. However, work which investigates graphical modeling in these areas is discussed by [Singh 92] using a role-centered approach supported by Petri Nets. Such work may help illuminate some of the problems associated with developing enactable PCDE specifications when, for example, close developer cooperation is required.

7 Summary and Conclusions

This report has explored a variety of issues at the intersection of process definition, process enactment, and process verification. The central focus has been to describe a unified approach to these topics. It was suggested that construction of a process-centered development environment should be preceded with the development of a graphically-based specification of the process to be automated. This specification should not only be graphically based, but should allow for automatic compilation into an enactable form. The reasons for having such a graphical, enactable specification are five-fold:

- The enactable specification allows a PCDE to be validated through dynamic simulation and logical analysis prior to its construction. This approach will be much less costly than developing and debugging a fully-fledged PCDE using other approaches.
- Any real-world software development process will exhibit significant complexity. This process needs to be understood by all interested parties, from the managers who sign off on its acceptability to the personnel who will use it. If the only people who understand the language in which the process is defined are the process “gurus,” then it is unlikely to be adopted. This buy-in is critical to success.
- The ability of the organization’s members to discuss the process and to suggest improvements or “bug fixes” will depend on their understanding of the process. This will be helped significantly if the process is documented graphically.
- The graphical representation will be of considerable help to new employees as they learn the elements of their jobs and how their jobs relate to other activities.
- A record of the history of the process improvement efforts can be captured automatically through the use of graphical process descriptions.

A graphical process notation was introduced (Section 3) which exhibits many of the needed characteristics for process definition. This notation accounts for

- the agents, roles, artifacts, and activities,
- control flows and product version management, and
- hierarchical nesting of models.

To investigate issues associated with enactment, the ProNet notation was used. Fortunately, this graphical notation is appropriately structured for translation into an enactable form. Examples of how different graphical process elements can be mapped to corresponding symbolic forms was provided in Section 4. This mapping was done by hand and used the example of a change request process. However, as was shown, a well-defined set of rules exists whereby the mapping could be automated. The symbolic form is declarative and implemented in Prolog. It results in a production system in which each of the activities in the process model trans-

lates into a rule. These rules are managed by a “process driver” which allows for process interaction with the user. The actual program is listed in Appendix A.

Process verification, as defined here, is a post-analysis of process data gathered during project execution (down to an appropriate level of granularity). This was discussed in Section 5. The object of verification is to determine if a subset of all the activities that have been performed and products that have been produced conform to the prescribed process. With process automation in place, gathering process data is essential (otherwise the system does not know what needs to be performed next). Thus the data needed for verification comes at no extra charge. If a project strictly adheres to the defined process during process execution, then the data should, by definition, be valid. However, in real situations manual interventions, which may compromise strict adherence to the defined process, are likely. Thus the verification procedure provides a final assurance that the process has been followed. It was found that a symmetry between process enactment and verification exists. While enactment moves forwards in time and generates a trace of the process history, verification starts at the end and works backwards in time, consuming the trace generated by the enactment activity. The program by which process verification was investigated is provided in Appendix B.

In Section 6 we addressed a wide spectrum of issues related to the use of PCDEs. We started out by reviewing areas that are good candidates for process automation, and discussed reasons why the associated process models should be built with flexibility in mind. We then looked at some organizational issues related to 1) information flows between developers, project management and upper management, and 2) the need for the consistency of both data structures and user interface across projects. The different needs of developers and managers was then addressed; how developers desire a PCDE to support their development needs while managers desire a PCDE to provide mechanisms for project control and information access. It was noted that both groups have a common need for a PCDE to alleviate chores. We then moved on to discuss models of problem solving and how a PCDE must be have the adaptability to address unexpected process-related problems. This was related to the work of Jens Rasmussen. Of increasing importance in large projects, is cooperative development in communication, collaboration and coordination. This field is generally referred to as “groupware” and, because of its increasing importance to software development, was also reviewed. Finally, it was noted than any PCDE which satisfies the needs of software developers working in a cooperative arena should include notions on configuration management, concurrency and synchronization.

It is hoped that this report has shed some light on issues which need to be addressed before process enactment becomes a practical and successful reality. Clearly, PCDE developers will have to focus increasingly on process, first with respect to the usability of their products and second with respect to the impact process modeling and enactment concepts will make on current software development environments.

Appendix A The Process Enactment Program

Appendix A provides information on the Prolog program which demonstrates simple process enactment. Section A.1 first illustrates typical output from the program. The program (Section A.2) is self-contained and does not require any input other than the requested name of the user and the activity that the user has selected (as seen in Section A.1). Section A.3 provides a modification to the program to show how hierarchical nesting of activities may be implemented. All program listings in Appendices A and B were developed using AIS Prolog [AAIS 90] on the Macintosh.

A.1 Typical Program Output

The following listing illustrates an interactive session with the automated process controller. Once the user's name is entered, the controller 1) checks the validity of the name, 2) identifies the roles associated with this name, 3) finds the activities which can next be performed by that role and then 4) presents the available activities to the user. The user selects the next activity to be performed. Clearly at this point the process controller must communicate with the tools (editor, compiler, etc.) necessary to perform the task, but the addition of such functionality is beyond the scope of this investigation. The example below is extracted from the process defined in Figure 3-8.

```
************************************************

What is your name: alanC.
Available activities:
     act10   get CR from CR repos - for update
     act9   get review doc from doc repos - for update
Enter the ID of the activity you wish to perform: act10.
get CR from CR repos - for update -- done

************************************************

What is your name: susan.
There are no activities currently available for this role.

************************************************

What is your name: alan.
 Nobody by that name or no designated role
abort? (y/n): n.

What is your name: alanC.
Available activities:
     act9   get review doc from doc repos - for update
Enter the ID of the activity you wish to perform: act9.
get review doc from doc repos - for update -- done

************************************************

What is your name: alanC.
Available activities:
```

```
     act11   update CR
Enter the ID of the activity you wish to perform: act12.
act12 -- is not in the activity list.

Available activities:
     act11   update CR
Enter the ID of the activity you wish to perform: act11.
update CR -- done

************************************************
```

A.2 The Process Controller Listing

The following Prolog listing is for the process controller. An overview of the program's functionality is described in Section 4.3.

```
/******************** process initiator **********************/
initProcess(InitEnt):-
  initSystemVars,
  initUserVars,
  makeNameList,
  assert(InitEnt),
  doNextAct.

/********************** driver functions ********************/
doNextAct:-
  /* sets up the recursive loop for each activity in the process */
  retractall(active(_)),
  write('************************************************'), nl, nl,
  /* get the agents's name and check if it exists */
  checkName(Role),
  /* find activities which can be performed next*/
  findActs(ActList),
  /* remove these activities not appropriate for this role */
  delActs(Role, ActList, [], ActList1),
  /* select the activity which will be performed */
  ask(Role, ActList1),
  doNextAct.

findActs(_):-
  /* test all activity perconditions to find currently valid activities */
  actRole(Test, _, _, _),
  TestX =.. [Test],
  TestX,
  fail.

findActs(ActList):-
  /* group all currently valid activities into a list */
  bagof(Act, active(Act), ActList).

findActs(_):-
  /* no activities in list - process completed */
  write('All activities have been completed'), nl,
  abort.

nullActs(_).
```

```
delActs(_, [], ActList, ActList).

delActs(Role, [Act|ActList], ActList1, X):-
  /* eliminate these activities which cannot be performed by current role */
  active(Act),
  actRole(_, Act, Role, _),
  delActs(Role, ActList, [Act|ActList1], X).

delActs(Role, [Act|ActList], ActList1, X):-
  /* eliminate these activities which cannot be performed by current role */
  retract(active(Act)),
  delActs(Role, ActList, ActList1, X).

ask(_, []):-
  /* present user with options */
  write('There are no activities currently avaliable for this role.'),nl, nl,
  doNextAct.

ask(_, _):-
  /* present user with options */
  write('Available activities:'), nl,
  active(Act),
  actRole(_, Act, _, Text),
  write('     '), write(Act), write('   '), write(Text), nl,
  fail.

ask(Role, ActList):-
  /* present user with options */
  write('Enter the ID of the activity you wish to perform: '),
  read(ActID),
  nextAct(ActID, Role, ActList).

nextAct(abort, _, _):-
  abort.

nextAct(ActID, Role, ActList):-
  /* activity selected is illegal */
  not(member(ActID, ActList)),
  write(ActID), write(' -- is not in the activity list.  '), nl, nl,
  ask(Role, ActList).

nextAct(ActID, Role, ActList):-
  /* activity selected is illegal */
  not(actRole(_, ActID, _, Desc)),
  write(Desc), write(' -- cannot be performed by the role:  '),
  write(Role), nl, nl,
  ask(Role, ActList).

nextAct(ActID, _, _):-
  /* perform activity selected by user */
  ActX =.. [ActID],
  ActX.

initSystemVars:-
  /* clear program of all garbage left from previous run */
  retractall(ver(_,_)),
  asserta(ver(0,0)),
  retractall(log(_,_,_)),
  asserta(log(nul,nul,nul)),
```

```
  retractall(subact(_)),
  retractall(nameList(_)),
  retractall(actList).

/********************** utility functions ********************/
set(X,K):-
  /* initialize version number X to value K */
  retractall(ver(X,_)),
  assert(ver(X,K)).

inc(K):-
  /* increment version number K by 1 */
  retract(ver(K,J)),
  J1 is J+1,
  assertz(ver(K,J1)).

put_ent(Store, Val, OutList):-
  /* put an entity into a store and assert exit entities  */
  testfor_(Val,Val1),
  retract(store(Store, Val_list)),
  assertz(store(Store, [Val1|Val_list])),
  assertList(OutList).

assertList([]).

assertList([First|Rest]):-
  assert(First),
  assertList(Rest).

get_ent(Store, Val):-
  /* retrieve an entity from a store */
  testfor_(Val, Val1),
  store(Store, Val_list),
  member(Val1, Val_list),
  assertz(Val).

get_ent(Store, Val):-
  /* retrieve an entity from a store - entity not found */
  write(Val),
  write(' is not contained in '),
  write(Store),nl,
  !,
  abort.

testfor_(Val, Val1):-
  /* remove digit from end of entity name - if entity has one */
  Val =.. [Name,Var],
  explode(Name, CharsList),
  reverse(CharsList, [Last|List]),
  string(Last, Str),
  int2string(Int, Str),
  integer(Int),
  reverse(List, List1),
  explode(Name1, List1),
  Val1 =.. [Name1, Var].

testfor_(Val, Val).

decision(Doc1,Doc2,Doc):-
```

```
  /* needed to decide between two paths og an output OR */
  random(10,Y),
  (Y > 5, Doc=Doc1; Doc=Doc2).

random(R,N):-
  retract(seed(S)),
  N is (S mod R) + 1,
  NewSeed is (125*S+1) mod 4096,
  assertz(seed(NewSeed)), !.

seed(13).

textOut(Act):-
  /* write out text associated with an activity */
  actRole(_, Act, _, Text),
  write(Text), write(' -- done'), nl, nl.

makeNameList:-
  /* make a list of all agent names */
  hasRole(_, Name),
  asserta(oneName(Name)),
  fail.

makeNameList:-
  /* make a list of all agent names */
  setof(Name, oneName(Name), Names),
  retractall(oneName(_)),
  assert(nameList(Names)).

checkName(Role):-
  /* enter and check user name */
  write('What is your name: '),
  read(Name),
  nameList(Names),
  member(Name, Names),
  hasRole(Role, Name).

checkName(_):-
  /* invalid user name */
  write(' Nobody by that name or no designated role'), nl,
  write('abort? (y/n): '),
  read('n'), nl,
  checkName(_).

checkName(_):-
  abort.

/******************** activity entrance conditions *************************/
/*** these are all tested against at each cycle to current activities ***/
test1:-
  ?(intern_prob_iden),
  not(log(devel_intern_cr, _, [cr_intern])),
  asserta(active(act1)).

test2:-
  ?(extern_prob_iden),
  not(log(devel_extern_cr, _, [field_cr])),
  asserta(active(act2)).
```

```
test3:-
  ?(field_cr),
  not(log(e-mail_cr, _, [cr_extern])),
  asserta(active(act3)).

test4:-
  (?(cr_intern); ?(cr_extern)),
  not(log(format_cr, _, [cr1(1)])),
  asserta(active(act4)).

test5:-
  ver(k,K),
  ?(cr1(K)),
  not(log(put_into_repos_A, _, [cr_added(K)])),
  asserta(active(act5)).

test6:-
  ver(k,K),
  ?(cr_added(K)),
  not(log(get_from_repos_A, _, [cr2(K)])),
  asserta(active(act6)).

test7:-
  ver(k,K),
  ?(cr2(K)),
  not(log(review_cr, _, [cr_appr])),
  not(log(review_cr, _, [rev_doc1(K)])),
  asserta(active(act7)).

test8:-
  ver(k,K),
  ?(rev_doc1(K)),
  not(log(put_into_repos_B, _, [rev_doc_added1(K), rev_doc_added2(K)])),
  asserta(active(act8)).

test9:-
  ver(k,K),
  ?(rev_doc_added1(K)),
  not(log(get_from_repos_B, _, [rev_doc2(K)])),
  asserta(active(act9)).

test10:-
  ver(k,K),
  ?(rev_doc_added2(K)),
  not(log(get_from_repos_C, _, [cr3(K)])),
  asserta(active(act10)).

test11:-
  ver(k,K),
  K1 is K+1,
  ?(cr3(K)),
  ?(rev_doc2(K)),
  not(log(update_cr, _, [cr1(K1)])),
  asserta(active(act11)).

test12:-
  ver(k,K),
  ?(act7in(K)),
  not(log(read_cr, _, [cr_read(K)])),
```

```
  asserta(active(act12)).

test13:-
  ver(k,K),
  ?(cr_read(K)),
  not(log(approve_cr, _, [rev_doc1(K)])),
  not(log(approve_cr, _, [cr_appr])),
  asserta(active(act13)).


/****************** activities and exit conditions ********************/
act1:-
  assertz(log(devel_intern_cr, [intern_prob_iden], [cr_intern])),
  assert(cr_intern),
  textOut(act1).

act2:-
  assertz(log(devel_extern_cr, [extern_prob_iden], [field_cr])),
  assert(field_cr),
  textOut(act2).

act3:-
  assertz(log(e-mail_cr, [field_cr], [cr_extern])),
  assert(cr_extern),
  textOut(act3).

act4:-
  (?(cr_intern), X=cr_intern; ?(cr_extern), X=cr_extern),
  assertz(log(format_cr, [X], [cr1(1)])),
  assertz(cr1(1)),
  set(k,1),
  textOut(act4).

act5:-
  ver(k,K),
  assertz(log(put_into_repos_A, [cr1(K)], [cr_added(K)])),
  put_ent(cr_repos,cr1(K), [cr_added(K)]),
  textOut(act5).

act6:-
  ver(k,K),
  assertz(log(get_from_repos_A, [cr_added(K)], [cr2(K)])),
  get_ent(cr_repos,cr2(K)),
  textOut(act6).

act7:-
  ver(k,K),
  decision(cr_appr,rev_doc1(K),Doc),
  assertz(log(review_cr, [cr2(K)], [Doc])),
  assertz(Doc),
  textOut(act7).

act8:-
  ver(k,K),
  assertz(log(put_into_repos_B,          [rev_doc1(K)],          [rev_doc_added1(K),
rev_doc_added2(K)])),
  put_ent(rev_doc_repos, rev_doc1(K), [rev_doc_added1(K), rev_doc_added2(K)]),
  textOut(act8).
```

```
act9:-
  ver(k,K),
  assertz(log(get_from_repos_B, [rev_doc_added1(K)], [rev_doc2(K)])),
  get_ent(rev_doc_repos,rev_doc2(K)),
  textOut(act9).

act10:-
  ver(k,K),
  assertz(log(get_from_repos_C, [rev_doc_added2(K)], [cr3(K)])),
  get_ent(cr_repos, cr3(K)),
  textOut(act10).

act11:-
  ver(k,K),
  K1 is K+1,
  assertz(log(update_cr, [cr3(K), rev_doc2(K)], [cr1(K1)])),
  assertz(cr1(K1)),
  textOut(act11),
  inc(k).

act12:-
  ver(k,K),
  assert(log(read_cr, [cr2(K)], [cr_read(K)])),
  assert(cr_read(K)),
  textOut(act12).
act13:-
  ver(k,K),
  decision(cr_appr,rev_doc1(K),Doc),
  assertz(log(approve_cr, [cr_read(K)], [Doc])),
  asserta(Doc),
  textOut(act13).

/********************** supporting model data *********************/
actRole(test1, act1, developer, 'initialize internal CR').
actRole(test2, act2, field_rep, 'initialize external CR').
actRole(test3, act3, field_rep, 'e-mail external CR').
actRole(test4, act4, developer, 'format CR for repository').
actRole(test5, act5, developer, 'put CR revision into CR repos').
actRole(test6, act6, reviewer, 'get CR from CR repos - for review').
actRole(test7, act7, reviewer, 'CR review').
actRole(test8, act8, reviewer, 'put review doc into doc repos').
actRole(test9, act9, developer, 'get review doc from doc repos - for update').
actRole(test10, act10, developer, 'get CR from CR repos - for update').
actRole(test11, act11, developer, 'update CR').
actRole(test12, act12, reviewer, 'read CR').
actRole(test13, act13, reviewer, 'approve CR').

/********************** supporting model data *********************/
hasRole(developer, ed).
hasRole(developer, paul).
hasRole(developer, alanB).
hasRole(developer, alanC).
hasRole(reviewer, susan).
hasRole(reviewer, dennis).
hasRole(reviewer, jock).
hasRole(developer, mike).
hasRole(reviewer, cliff).
hasRole(field_rep, howard).
```

```
/********************** initialize variables *********************/
initUserVars:-
  retractall(cr1(_)),
  retractall(cr2(_)),
  retractall(cr3(_)),
  retractall(cr_added(_)),
  retractall(rev_doc1(_)),
  retractall(rev_doc2(_)),
  retractall(rev_doc_added1(_)),
  retractall(rev_doc_added2(_)),
  retractall(cr_appr),
  retractall(cr_read(_)),
  retractall(active(_)),
  retractall(intern_prob_iden),
  retractall(extern_prob_iden),
  retractall(field_cr),
  retractall(cr_intern),
  retractall(cr_extern),
  asserta(store(cr_repos, [])),
  asserta(store(rev_doc_repos, [])).
```

A.3 An Extension to Account for Nested Activities

The following program excerpts modify the listing of Section A.2 to account for hierarchical nesting of activities. The changes take place in the rules and not the process driver routines. The activity *review_cr* in Figure 3-8 is given two serial sub-activities: *read_cr* and *approve_cr.* The resulting process fragment is illustrated in Figure A-1.

```
/************* modified activity ‘test’ clauses ******************/
test7in:-
  ver(k,K),
  ?(cr2(K)),
  not(log(review_cr_in, _, [act7in(K)])),
  asserta(active(act7in)).

test7out:-
  ver(k,K),
  ?(cr_appr); ?(rev_doc1(K)),
  not(log(review_cr_out, _, [act7out(K)])),
  assert(active(act7out)).

test8:-
  ver(k,K),
  ?(act7out(K)),
  not(log(put_into_repos_B, _, [rev_doc_added1(K), rev_doc_added2(K)])),
  asserta(active(act8)).

/************* additional activity ‘test’ clauses ******************/
test12:-
  ver(k,K),
  ?(act7in(K)),
  not(log(read_cr, _, [cr_read(K)])),
  asserta(active(act12)).
```

```
test13:-
  ver(k,K),
  ?(cr_read(K)),
  not(log(approve_cr, _, [rev_doc1(K)])),
  not(log(approve_cr, _, [cr_appr])),
  asserta(active(act13)).

/************* modified activity ‘act’ clauses ******************/
act7in:-
  ver(k,K),
  assertz(log(review_cr_in, [cr2(K)], [act7in(K)])),
  assert(act7in(K)),
  textOut(act7in).

act7out:-
  ver(k,K),
  (?(cr_appr), X=cr_appr; ?(rev_doc1(K)), X=rev_doc1(K)),
  assertz(log(review_cr_out, [X], [act7out(K)])),
  assert(act7out(K)),
  textOut(act7out).

/************* additional activity ‘act’ clauses ******************/
act12:-
  ver(k,K),
  assert(log(read_cr, [cr2(K)], [cr_read(K)])),
  assert(cr_read(K)),
  textOut(act12).

act13:-
  ver(k,K),
  decision(cr_appr,rev_doc1(K),Doc),
  assertz(log(approve_cr, [cr_read(K)], [Doc])),
  asserta(Doc),
  textOut(act13).

/************* modified ‘actRole’ clauses ******************/

actRole(test7in, act7in, reviewer, 'start CR review').
actRole(test7out, act7out, reviewer, 'end CR review').
```

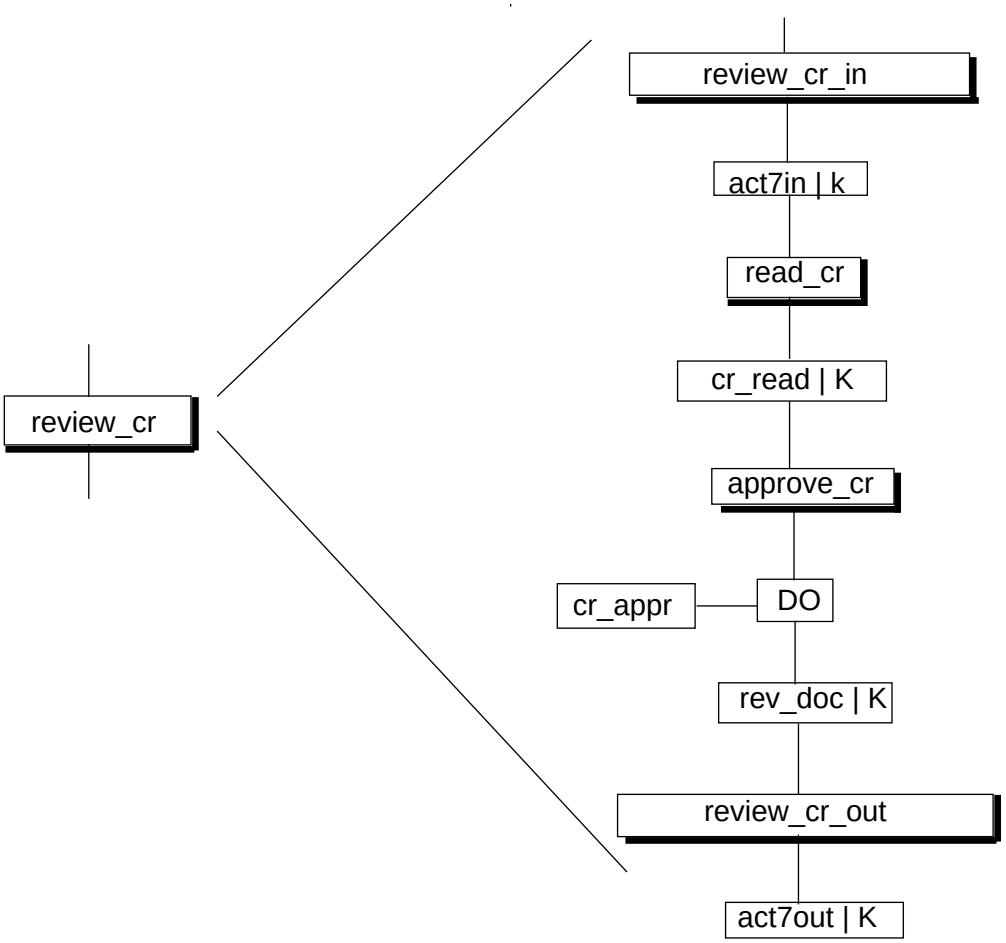


Figure A-1 Expansion of the Activity *review_cr*

Appendix B The Process Verification Program

The following is a short example of the output from the verification program. As verification proceeds, the program checks the validity of the activities and consistency of their inputs and outputs. When it finds an inconsistency, this inconsistency is identified and the program terminates. As an example of a process error, the *log* statement below was commented out of the set of *log* statements which define the process history. (See Section 4.1 for description of *log* statements.)

```
/* log(get_from_repos_C,[rev_doc_added2(1)],[cr3(1)]). */
```

The resulting verification output looks like this:

```
******* review CR **********
******* get CR from CR repos - for review **********
******* put CR revision into CR repos **********
******* update CR **********
"get CR from CR repos - for update" - not performed
```

The following is a listing for the process verification program which is, like the process enactment program, written in Prolog. While the structure of the program is similar to that of the enactment program (Appendix A), the verification program, as can be seen in the above example, above is non-interactive.

```
startVerify(FinalEnt):-
  initUserVars,
  assert(FinalEnt),
  verify.

/********************* utility functions *********************/
set(X,I):-
  /* set the version variable X to the value I */
  retractall(ver(X,_)),
  assert(ver(X,I)).

dec(I,J):-
  /* decrement the version variable I by 1 */
  retract(ver(I,J)),
  J1 is J-1,
  assert(ver(I,J1)).

del_ent(Store, Val, Desc):-
  /* inverse of 'put' - deletes entity from store, if it is in the store */
  /* otherwise verification fails */
  testfor_(Val, Val1),
  retract(store(Store, Val_list)),
  (member(Val1, Val_list); escape(Store, Val1, Desc)),
  delete(Val1, Val_list, [], Val_list1),
```

```
  assert(store(Store, Val_list1)),
  assert(Val).

delete(Val1, [Val1|Val_list], Val_listA, X):-
  /* removes entity from list */
  delete(Val1, Val_list, Val_listA, X).

delete(Val1, [Val|Val_list], Val_listA, X):-
  delete(Val1, Val_list, [Val|Val_listA],X).

delete(_, [], Val_list, Val_list).

rem_ent(Store, Val, Val2, Desc):-
  /* inverse of 'get' - tests if in store then deletes retrieved copy, if it is in
store */
  /* otherwise verification fails */
  Val,
  testfor_(Val, Val1),
  store(Store, Val_list),
  (member(Val1, Val_list); escape(Store, Val1, Desc)),
  retract(Val),
  assert(Val2).

escape(Store, Val, Desc):-
  /* verification fails */
  write(Val),
  write(' is not contained in '),
  write(Store),nl,
  write('Cannot perform operation: '),
  write(Desc), nl,
  !,
  abort.

testfor_(Val, Val1):-
  /* if last char on variable Var is an integer, remove it */
  /* this removes numbered extensions on products/conditions */
  /* before they are put into a store */
  Val =.. [Name,Var],
  explode(Name, CharsList),
  reverse(CharsList, [Last|List]),
  string(Last, Str),
  int2string(Int, Str),
  integer(Int),
  reverse(List, List1),
  explode(Name1, List1),
  Val1 =.. [Name1, Var].

testfor_(Val, Val).

/******************** verify  entrance conditions **************************/

verify:-
  ?(cr_intern),
  retract(log(devel_intern_cr, [intern_prob_iden], [cr_intern])),
  assert(intern_prob_iden),
  retract(cr_intern),
  write('******* develop internal change request **********'), nl.

verify:-
```

```
  ?(cr_intern),
  not(log(devel_intern_cr, [intern_prob_iden], [cr_intern])),
  write('"develop internal change request" - not performed'), nl,
  abort.

verify:-
  ?(field_cr),
  retract(log(devel_extern_cr, [extern_prob_iden], [field_cr])),
  assert(extern_prob_iden),
  retract(field_cr),
  write('******* develop external change request **********'), nl.

verify:-
  ?(field_cr),
  not(log(devel_ext_cr, [extern_prob_iden], [field_cr])),
  write('"develop external change request" - not performed'), nl,
  abort.

verify:-
  ?(cr_extern),
  retract(log(e-mail_cr, [field_cr], [cr_extern])),
  assert(field_cr),
  retract(cr_extern),
  write('******* mail external change request **********'), nl,
  verify.

verify:-
  ?(cr_extern),
  not(log(e-mail_cr, [field_cr], [cr_extern])),
  write('"mail external change request" - not performed'), nl,
  abort.

verify:-
  cr1(1),
  retract(log(format_cr, [Var], [cr1(1)])),
  assert(Var),
  retract(cr1(1)),
  write('******* format change request **********'), nl ,
  verify.

verify:-
  cr1(1),
  not(log(format_cr, _, [cr1(1)])),
  write('"format change request" - not performed'), nl,
  abort.

verify:-
  ver(k,K),
  cr_added(K),
  retract(log(put_into_repos_A, [cr1(K)], [cr_added(K)])),
  del_ent(cr_repos, cr1(K), 'put CR revision into CR repos'),
  retract(cr_added(K)),
  write('******* put CR revision into CR repos **********'), nl,
  verify.

verify:-
  ver(k,K),
  cr_added(K),
  not(log(put_into_repos_A, [cr1(K)], [cr_added(K)])),
```

```
  write('"put CR revision into CR repos" - not performed'), nl,
  abort.

verify:-
  ver(k,K),
  cr2(K),
  retract(log(get_from_repos_A, [cr_added(K)], [cr2(K)])),
   /* tests if cr is in DB then removes copy retrieved */
  rem_ent(cr_repos, cr2(K), cr_added(K), 'get CR from CR repos - for review'),
  write('******* get CR from CR repos - for review **********'), nl,
  verify.

verify:-
  ver(k,K),
  cr2(K),
  not(log(get_from_repos_A, [cr_added(K)], [cr2(K)])),
  write('"get CR from CR repos - for review" - not performed'), nl,
  abort.

verify:-
  ver(k,K),
  (cr_appr, Var = cr_appr; rev_doc1(K), Var = rev_doc1(K)),
  retract(log(review_cr, [cr2(K)], [Var])),
  assert(cr2(K)),
  retract(Var),
  write('******* review CR **********'), nl,
  verify.

verify:-
  ver(k,K),
  (cr_appr, Var = cr_appr; rev_doc1(K), Var = rev_doc1(K)),
  not(log(review_cr, [cr2(K)], [Var])),
  write('"review CR" - not performed'),nl,
  abort.

verify:-
  ver(k,K),
  rev_doc_added1(K),
  rev_doc_added2(K),
  retract(log(put_into_repos_B,          [rev_doc1(K)],          [rev_doc_added1(K),
rev_doc_added2(K)])),
  del_ent(rev_doc_repos,rev_doc1(K),'put review doc into doc repos' ),
  retract(rev_doc_added1(K)),
  retract(rev_doc_added2(K)),
  write('******* put review doc into doc repos **********'), nl,
  verify.

verify:-
  ver(k,K),
  rev_doc_added1(K),
  rev_doc_added2(K),
  (not(log(put_into_repos_B,           [rev_doc1(K)],           [rev_doc_added1(K),
rev_doc_added2(K)])),
     write('"put review doc into doc repos" - not performed'),nl),
  abort.

verify:-
  ver(k,K),
  cr3(K),
```

```
  retract(log(get_from_repos_C, [rev_doc_added2(K)], [cr3(K)])),
  rem_ent(cr_repos, cr3(K), rev_doc_added2(K), "get CR from CR repos - for update"),
  write('******* get CR from CR repos - for update **********'), nl,
  verify.

verify:-
  ver(k,K),
  cr3(K),
  not(log(get_from_repos_C, [rev_doc_added2(K)], [cr3(K)])),
  write('"get CR from CR repos - for update" - not performed'),nl,
  abort.

verify:-
  ver(k,K),
  rev_doc2(K),
  retract(log(get_from_repos_B, [rev_doc_added1(K)], [rev_doc2(K)])),
  rem_ent(rev_doc_repos, rev_doc2(K), rev_doc_added1(K), 'get review doc from doc re-
pos - for update'),
  write('******* get review doc from doc repos - for update **********'), nl,
  verify.

verify:-
  ver(k,K),
  rev_doc2(K),
  not(log(get_from_repos_B, [rev_doc_added1(K)], [rev_doc2(K)])),
  write('"get review doc from doc repos - for update" - not performed'), nl,
  abort.

verify:-
  ver(k,K),
  K1 is K-1,
  cr1(K),
  retract(log(update_cr, [cr3(K1), rev_doc2(K1)], [cr1(K)])),
  assert(rev_doc2(K1)),
  assert(cr3(K1)),
  retract(cr1(K)),
  dec(k,K),
  write('******* update CR **********'), nl,
  verify.
verify:-
  ver(k,K),
  cr1(K),
  (not(log(update_cr, [cr3(I1), rev_doc2(I1)], [cr1(K)])),
     write('"update CR" - not performed'),nl),
   abort.

/********************** initialize variables *********************/

initUserVars:-
  retractall(cr1(_)),
  retractall(cr2(_)),
  retractall(cr3(_)),
  retractall(cr_appr),
  retractall(cr_added(_)),
  retractall(rev_doc1(_)),
  retractall(rev_doc2(_)),
  retractall(rev_doc_added1(_)),
  retractall(rev_doc_added2(_)),
  retractall(cr_read(_)),
```

```
  retractall(active(_)),
  retractall(intern_prob_iden),
  retractall(extern_prob_iden),
  retractall(field_cr),
  retractall(cr_intern),
  retractall(cr_extern),
  set(k,2).

cr1(0).
cr2(0).
cr3(0).
rev_doc1(0).
rev_doc2(0).
rev_doc_added1(0).
rev_doc_added2(0).
cr_added(0).

/********************** process history ***********************/
/* these were generated by enacting the process - see Appendix A */

log(devel_intern_cr,[intern_prob_iden],[cr_intern]).
log(format_cr,[cr_intern],[cr1(1)]).
log(put_into_repos_A,[cr1(1)],[cr_added(1)]).
log(get_from_repos_A,[cr_added(1)],[cr2(1)]).
log(review_cr,[cr2(1)],[rev_doc1(1)]).
log(put_into_repos_B,[rev_doc1(1)],
    [rev_doc_added1(1),rev_doc_added2(1)]).
log(get_from_repos_B,[rev_doc_added1(1)],[rev_doc2(1)]).
log(get_from_repos_C,[rev_doc_added2(1)],[cr3(1)]).
log(update_cr,[cr3(1),rev_doc2(1)],[cr1(2)]).
log(put_into_repos_A,[cr1(2)],[cr_added(2)]).
log(get_from_repos_A,[cr_added(2)],[cr2(2)]).
log(review_cr,[cr2(2)],[cr_appr]).

store(rev_doc_repos,[rev_doc(1)]).
store(cr_repos,[cr(2),cr(1)]).
```

Acknowledgments

I would like to thank Edward Averill, Alan Brown, Michael Caldwell, David Carney, Susan Dart, and Dennis Smith for their insightful reviews of the report. They have done much to improve its quality. I would also like to thank Sandra Bond and Julia Deems for their excellent editorial review of the document. However, I take full responsibility for any errors, omissions or lack of clarity that may be found.

Bibliography

[AAIS 88] *AAIS Reference Manual*, Version M-2.0, Advanced A. I. System's Prolog, Inc., P.O. Box 39-0360, Mountain View CA 94039-0360

[Boone 91] Boone, G., "CASE and its Challenge for Change", *International Journal of Software Engineering and Knowledge Engineering*, Vol 1, No 2, pp 151-163, 1991

[Barghouti 90] Barghouti, N.S. et al, "Modeling Concurrency in Rule-Based Development Environments", *IEEE Expert*, December, 1990

[Bernard 89] Bernard, Y., Lavency, P., "A Process-Oriented Approach to Configuration Management", Trans ACM, pp 320-327, 1989

[Bratko 86] Bratko, I., *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1986

[Brownston 85] Brownston, L., et al, *Programming Expert Systems in OPS5*, Addison-Wesley, 1985

[Bruynooghe 91] Bruynooghe, R. F. et al, "PSS: A System for Process Enactment", *First International Conference on Software Process*, Redondo Beach, CA, Oct, 1990

[Bullen 90] Bullen, C. V., Bennett, J.J., "Learning from User Experience with Groupware", *Proceedings of the Conference on Computer-Supported Cooperative Work*, October, 1990, Los Angeles, California

[CaseWare 92] "CaseWare User's Guide", CaseWare, Inc., Irvine, CA, 1992

[Chen 83] Chen, P.P., (Editor) *Entity-Relationship Approach to Information Modeling and Analysis*, North-Holland, Amsterdam, The Netherlands, 1983

[Cooley 93] Cooley, J., "Adding Process Management to the Paramax SEE", Volume 3 of Proceedings of STARS '92

[Dart 91] Dart, S., "Concepts in Configuration Management Systems", Third International Software Configuration Management Conference, ACM Press, June 1991

[Dewan 93] Dewan, P., "Toward Computer Supported Software Engineering", *IEEE Computer*, January 1993

[Ellen 93, Ellen, L. W., "IBM Support Environment for Megaprogramming", Volume 3 of *Proceedings of STARS '92*

[Ellis 91] Ellis, C. A. et al, "GroupWare, Some Issues and Experiences", *Communications of the ACM*, Vol 43, No 1, January 1991

[Feiler 90] Feiler, P., Downey, G., "Transaction-Oriented Configuration Management: A Case Study", SEI Technical Report CMU/SEI-90-TR-23, ADA 235510, 1990

[Greis 81] Greis, D., *The Science of Programming*, Springer-Verlag, 1981

[Grudin 88] Grudin, J., “Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces”, *Proceedings of the Conference on Computer-Supported Cooperative Work*, September, 1988, Portland, Oregon

[Harel 87] Harel, D., “Statecharts: A Visual Formalism for Complex Systems”, Science of Computer Programming 8, pp 231-274, 1987

[Heimbigner 90] Heimbigner, D., “P^4: A Logic Language for Process Programming”, *Proceedings of the 5th International Software Process Workshop*, Kennebunkport, Maine, 1989

[Humphrey 89] Humphrey, W. S., *Managing the Software Process*, Addison-Wesley, 1989.

[Humphrey 92] Humphrey, W., “Toward a Discipline for Software Engineering”, *Sixth SEI Conference on Software Engineering Education*, San Diego CA, 1992

[Kaiser 90] Kaiser, G., et al, “Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel”, *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*. 1990

[Lee 91] Lee, S., Sluizer, S., “An Executable Language for Modeling Simple Behavior”, *IEEE Transactions on Software Engineering*, Vol 17, No. 6, June 1991

[Mahler 90] Mahler, A., Lampen, A., “Integrating Configuration Management into a Generic Environment”, SIGSOFT, Volume 15, No. 6, *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, 1990

[Mi 1992] Mi, P., Scacchi, W., “Process Integration in CASE Environments”, *IEEE Software*, pp 45-53, March 1992

[Monarchi 92] Monarchi, D.E., Smith, J. R., “The Representation of Rules in the ER Model”, *Data and Knowledge Engineering*, Volume 9, No. 1, pp 45-61, October, 1992

[Osterweil 87] Osterweil, J. “Software Processes Are Software Too”, *9th Conference on Software Engineering, Monterey*, California, 1987

[Page 92] Page-Jones, M., “The CASE Manifesto”, *CASE Outlook*, January-February, 1992

[Rasmussen 79] Rasmussen, J., “On the Structure of Knowledge: A Morphology of Mental Models in a Man-Machine System Context”, Technical Report, November 1979, Riso National Laboratory, Riso, Denmark

[Singh 92] Singh, B., “Interconnected Roles (IR): A Coordination Model”, Technical Report CT-084-92, Microelectronics and Computer Technology Corp., Austin Texas, July 1992

[Reisig 82] Reisig, W. *Petri Nets*, Springer-Verlag,1982

[Slomer 92] Slomer, H. M., Christie, A.M., “Analysis of a Software Maintenance System”, *SEI*

Technical Report CMU/SEI-92-TR-31, 1992

[Stanley 92] Stanley, M. E., “Verifying the Design Process”, *AI Expert*, pp 42-49, September 1992

[Wallnau 91] Wallnau, K.C., Feiler, P.H., “Tool Integration and Environment Architectures”, *SEI Technical Report CMU/SEI-91-TR-11*, ADA 237810,1991

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-93-TR-4

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-93-181

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/ENS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Process-Centered Development Environments: An Exploration of Issues

12. PERSONAL AUTHOR(S)
Alan M. Christie

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	June 1993	73 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
development environments software development environment
environments
modeling

19. ABSTRACT (continue on reverse if necessary and identify by block number)

Software development environments are beginning to move from research communities to commercial applications. As this occurs, the need to address process issues related to such environments is becoming increasingly apparent. Thus there is a growing awareness of the need for process-centered development environments (PCDEs). This report addresses process definition and enactment issues which pertain to the specification and design of a PCDE. The first part of the report explores some of the required characteristics of an enactable graphical language and the relationship between process definition and enactment. This process language naturally led to the ability to per-

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/ENS (SEI)

form process verification, i.e., a verification that the actual process path taken throughout a project conforms to the defined process. The issue of process verification is thus also explored. The success of PCDEs rests heavily on end-user acceptance. Because of this, the report concludes with a review of user-oriented process and social issues relevant to the successful adoption of PCDEs.