# Software Process Representation to Support Multiple Views

David Jacobs Chris Marlin
Discipline of Computer Science
Flinders University of South Australia
GPO Box 2100 Adelaide, S.A. 5001, Australia
Email: (jacobs, marlin)@cs.flinders.edu.au

## Abstract

*Current interest in improving the effectiveness and predictability of software development has led to a recent focus on software process modeling and improvement. Process-centered software development environments (PCSDEs), have been examined as a useful adjunct to software process modeling. A number of PCSDEs have been designed and built; an examination of the range of potential users of such environments reveals a wide range of needs with respect to information about an enacted software process and how this information is presented. This paper describes one aspect of a PCSDE supporting multiple simultaneous views: the design of a representation of enacted software processes which is suitable for the generation of multiple simultaneous views.*

## 1 Introduction

In any group software development effort, difficulties arise in coordinating and controlling the work of the team members. Early attempts to tackle this problem led to software development organisations writing software process descriptions in the form of manuals; such a manual would then be used by software project managers, software engineers, quality assurance teams, and others, to ensure that the appropriate process was followed throughout the project. These manuals proved to be cumbersome and tended to place a large additional burden on those involved in software development. Also, such manuals are difficult to update; hence, the standard process defined by the manual and that carried out in practice would diverge over time.

Regarding a software process description as a kind of program[7] has led to the development of a kind of software development environment which "executes" the software process description. Such environments are called *process-centered software development environments* (PCSDEs) and attempt to overcome the difficulties described above by having a software development project follow a defined process. PCSDEs have a software development process described within them and they coordinate, control and guide the project team through this software development process. Furthermore, the software process described within the environment may be changed as needed and the users of such an environment are not burdened with having to use a large manual describing the process.

Current PCSDEs have been developed as research platforms to investigate methods for describing and enacting software development processes. The nature of the interaction of potential users of PCSDEs with the environment has generally been a low research priority. In fact, these potential users cover a wide range of needs in terms of information about the software process while it is being *enacted* (i.e., executed): software engineers will want to know about the tasks on which they are engaged and those that await them, project managers will want various information about the conduct of the project as a whole, and so on.

In the domain of the coding aspects of software development, some work has been carried out on developing multiple view software development environments (such as PECAN [10, 11], PIGS[9] and MultiView[1, 4]). These environments attempt to improve the productivity of individual software engineers by supporting multiple views of the software components under development. The Viewpoints[3, 6] and Melmac[2] (CORMAN) environments have demonstrated that similar advantages may be obtained in PCSDEs by providing more than a single view of the software development process being enacted by the software development team. Melmac provides multiple views to the software process developer, with each view being generated from a single representation of the software process; unfortunately, there is no support for multiple views of the enacted software pro-



0-8186-6960-8/94 $04.00 © 1994 IEEE

cess. Viewpoints provides each member of a software development team with a specialized view of the process known as a viewpoint; however, these views are produced independently of one another and thus consistency of the process between viewpoints can never be guaranteed.

This paper describes aspects of the design of a PCSDE which supports multiple simultaneous views of the enacted software process. It uses an adaptation of the approach used in the MultiView system, mentioned above, to ensure the consistency of the various views without having to support a combinatorial explosion of translation schemes between views as the number of view types grows. The next section describes, in general terms, the approach used in the MultiView system and how it is being adapted to the new context of PCSDEs. The following section describes the canonical representation of software process which is the key to the support for multiple simultaneous consistent views of the enacted process. The final section presents some conclusions and describes future work on this approach.

## 2 Supporting Multiple Views

The MultiView software development environment provides support for multiple simultaneous views during coding. From a user's point of view, it is possible to load a number of software components into a database local to the session and then to have various view instances created against these components. Among the types of views currently supported are textual and graphical (e.g., flowchart and tree) views. The interested reader is referred to [4] for more detail on the appearance and use of the system.

The architecture of the MultiView implementation is shown in Figure 1. The *database process* contains copies of the software components being manipulated by this MultiView session. These components are each stored in a canonical structured representation; the representation used by the MultiView environment for storing these components is abstract syntax trees. Each view instance in the MultiView session is managed by a *view process*. As depicted in Figure 1, each view process contains a view-specific copy of the structured representation of one software component; this representation will be isomorphic with the representation of that component stored in the database process, but will be decorated with additional information. Examples of such information include screen coordinates for a graphical view such as a flowchart view, or (line number, character position) tuples in a textual view, to record the positions of parts of the visible depiction of the component corresponding to nodes within the structured representation.

When a view is employed by a MultiView user to edit the software component viewed by it, the edits are interpreted by the view process and related to the structured representation of the component (using the view-specific information in the representation). The corresponding modifications are made to the representation of the component and then notified to the database process; the database process makes these modifications to the copy of the component's representation in that process and broadcasts the change to other view processes viewing the component concerned. These view processes update their view-specific copy of the component and redisplay it in the visual presentation corresponding to this view type.

In all of the above, the interprocess communication is in terms of the canonical structured representation and is independent of the views involved. This has a number of important advantages, including

- the definition of new views is facilitated, since introducing an new type of view is only a matter of making sure that the view process concerned conforms to the defined protocol for interprocess communication – this new view type will then work with all existing view types, and

- a potential combinatorial explosion of transformations between the view types is avoided, as already mentioned in the previous section – communication between views is via the database process and in terms of transformations on the structured representation of components.

More information on the interprocess communication mechanisms used in MultiView, and the tools provided to generate the communications-related aspects of the MultiView system, are described in [5].

The software architecture depicted in Figure 1 can be generalized to provide support for multiple views in contexts other than the manipulation of software components. This generalized software architecture to support multiple simultaneous views consists of a database process storing some canonical structured representation, communicating via message-passing with a collection of view processes which each cache some view-specific information. To apply this architecture to new contexts, it is first necessary to devise a suitable canonical structured representation which will support the generation of appropriate view types.

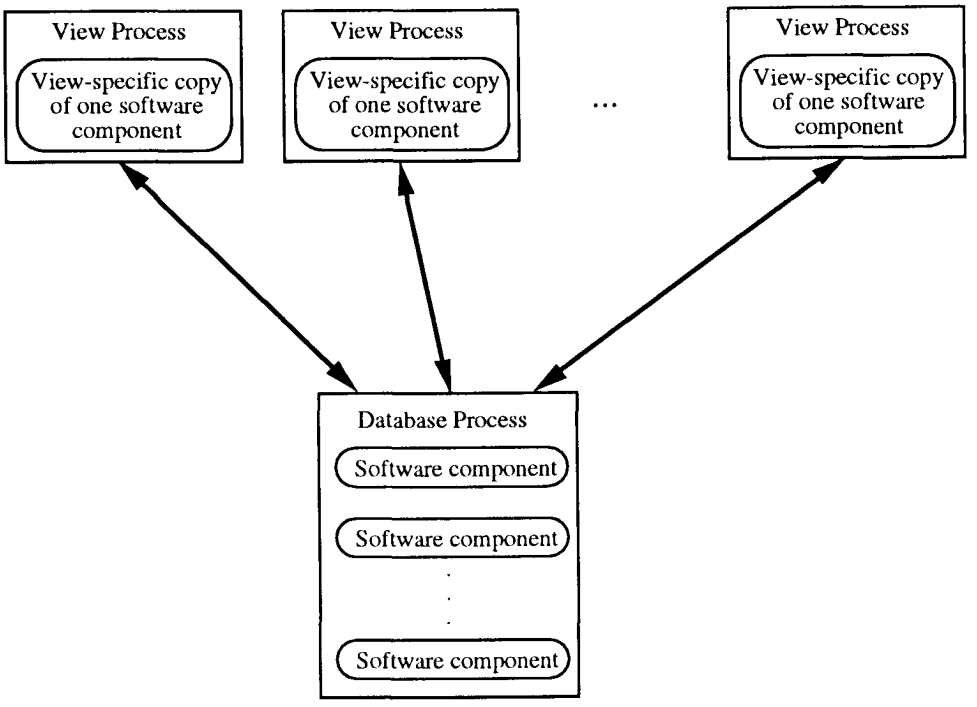


Figure 1: The architecture of the MultiView implementation.

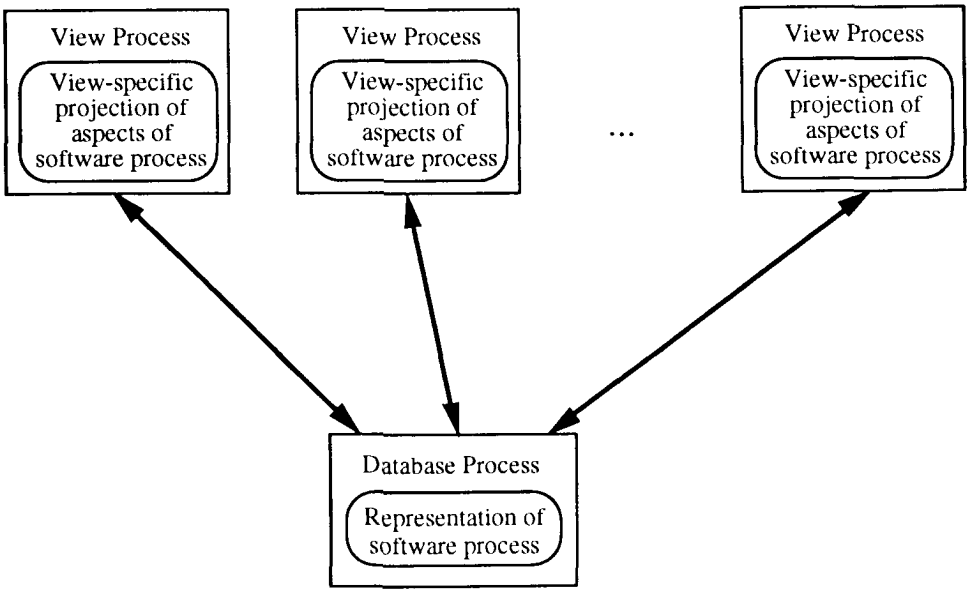


Figure 2: An architecture to support multiple views of the dynamic software process.

Figure 2 shows how the architecture in Figure 1 has been adapted to provide support for the multiple simultaneous views of the dynamic (enacted) software process. The database process now contains a canonical structured representation of the dynamic software process. Each view process, which will be notified by the database process of any changes to the process representation that are relevant to this view type, maintains a view-specific projection of those aspects of the process representation which are related to this view type. A view instance may be used to make some change to the process representation (e.g., a manager assigns a task to an individual software engineer); edits in the user interface of the view concerned are interpreted in terms of changes in the view-specific projection of aspects of the process representation and this view-specific projection is updated. These changes are then transmitted to the database process, where they are used to update the software process representation residing there. Finally, the changes are broadcast to other view processes of types which relate to the parts of the process representation which have changed.

The key to the success of the above approach to constructing a multiple view process-centered software development environment is the design of a suitable canonical structured representation of the dynamic software process. This representation must not only be sufficiently complete and detailed to cover all aspects of the software process which relate to likely view types, but must also be structured in such a way that it is amenable to the generation of the anticipated view types. The following section describes the design of such a dynamic process representation.

# 3 Dynamic Process Representation

As already discussed in Section 1, the various different members of a project team will have different needs in terms of the aspects of the process which they will wish to see in a process-centered software development environment. Section 2 has described why it is vital that all views of the dynamic process be generated from a single canonical structured representation of the software process, thus ensuring the consistency of these views without the need for many transformations between the view types.

The views likely to be of use in a multiple-view process-centered software development environment include:

- a process overview suitable for project managers to see the current status of the project,
- a "to-do" list for each project team member,
- a view showing all design documents and the relationships between them, and
- a current working context for a particular team member as is provided, for example in the Merlin environment [8, 12].

The dynamic process representation described here was designed from the start with the support for multiple views of a software process the first priority. The process engine for the corresponding PCSDE will use an interactive interpretive approach to process enactment. The system will be event-driven, with the majority of the events being produced by the actions of the environments users. The users will interact with the process engine via multiple concurrent views of the process being enacted.

Each aspect of a process to be described within the environment is represented by an object. These objects are created, destroyed, and manipulated dynamically by the process engine according to the requests of users and governed by the prescribed process which the engine is enacting. The objects are divided into two groups. The first group is the collection of *passive objects* of the system; these objects are the data upon which decisions within the software process are made. The second collection of objects comprises the *active objects* of the system; these objects describe what activities are to be carried during the process and when they are to be carried out. The following sections will describe each of these object types in turn.

All objects in the object base of the process engine are equal. This, coupled with the interpretive nature of the process engine, means that objects of any type can be created, destroyed, or modified as the software process is being enacted. A consequence of this fact is that the process being enacted can itself be modified dynamically by modifying the objects which describe the process.

## 3.1 Passive Objects of the Process

Figure 3 shows the passive object types in the environment. A dashed line adjacent to an attribute within an object type indicates that the attribute is a link to other objects.

A *document object*, whose template is shown in Figure 3(a), is an artifact of the process being enacted. Typically, it will be a file in the corresponding file system. Each file created in the file system as part of the process will be described by a document object within the process-centered environment. The attributes of

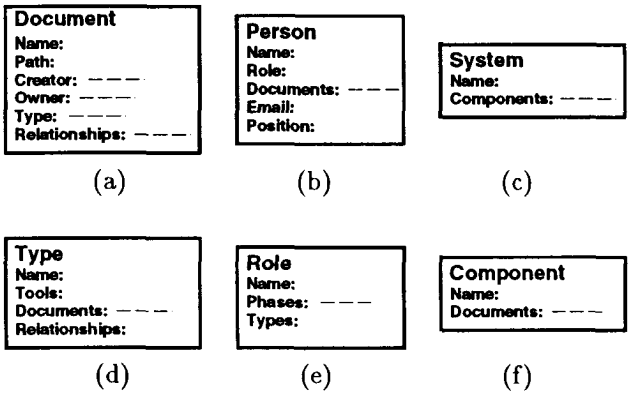


Figure 3: Passive objects of the process representation.

an object of type document are: the name of the document (*name*), the complete directory path to the document (*path*), a link to the person who created the document (*creator*), a link to the person who is currently responsible for the document (*owner*), a link to the description of the type of the document (*type*), and links to all related documents (*relationships*).

A *person object* is a description of a member of the project team enacting the process; the template for a person object is shown in Figure 3(b). The attributes of a person object are: the name of the person (*name*), a link to the current role of the person (*role*), links to the documents the person is currently responsible for (*documents*), the email address of the person (*email*), and the position of the person within the project team (*position*).

A *system object* is the collection of all components of the system under development. The template for a system object is shown in Figure 3(c). The attributes of a system object are: the name of the system (*name*) and links to the components which make up the system (*components*).

A *type object* is a description of a particular type of document; the template for this kind of object is shown in Figure 3(d). The attributes of a type object are: the name of the type (*name*), the tools to be used on documents of this type (*tools*), to all documents of this type (*documents*) and allowable relationships for documents of this type (*relationships*).

A *role object*, whose template is depicted in Figure 3(e), is a description of the duties to be performed by a person acting in a particular role. The attributes of a role object are: the name of the role (*name*), links to the documents currently associated with this role (*documents*) and links to the people who are currently performing this role (*people*).

A *component object* is a description of some logically related collection of documents; the template for such an object is shown in Figure 3(f). The attributes of this kind of object are: the name of the component (*name*) and links to the documents which are part of this component (*documents*).

## 3.2 Structures of Passive Objects

Within the multiple view process-centered environment, there are also *master objects*; these maintain links to all objects of a particular type. For example, in Figure 4 all objects of type "type" are linked to the type master object on the left-hand side of the diagram. Similarly, all objects of type "document" are linked to the doument master object on the left-hand side of the diagram.

These master objects simplify answering user requests about objects of a particular type, as there is always a simple path to the object. Other links are also maintained for similar reasons; for example, a type object maintains links to all documents of its type and all documents maintain links to the object which describes their type. This arrangement is illustrated in Figure 4. In this figure, there are the master objects for types and documents; one of the type objects (the middle one of the three) is for a type called "Code". All of the documents shown in Figure 4 are defined to be of this type.

From such a structure, it is possible to efficiently obtain information about all documents by working from the document master object, or about documents of a particular type by going via the type master object. In a similar fashion, master objects exist for objects of type person, system, role and component. The existence of the master objects and the highly linked nature of the objects allows for a general query mechanism to be implemented over the objects in the environment. Also, as information relating to the objects may be obtained via many different paths through the environment, the information may easily be interpreted in different ways.

## 3.3 Active Objects within the Process

The *active objects* within the environment describe and control the sequencing of steps through the software process. Figure 5 shows the types of these active objects; there are three active object types and these will now be described in turn.

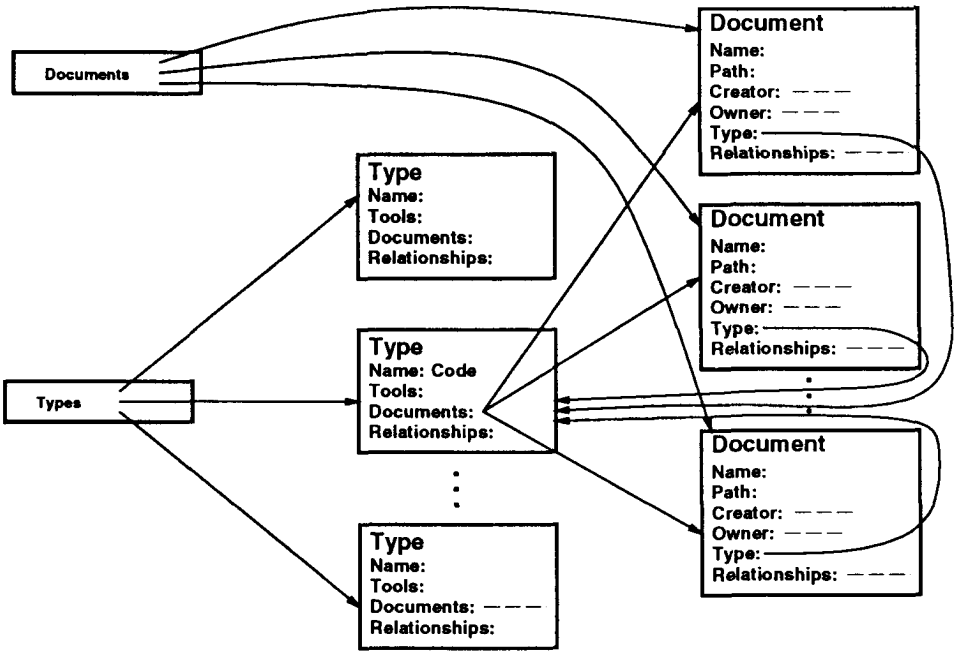


Figure 4: Type-document object links.

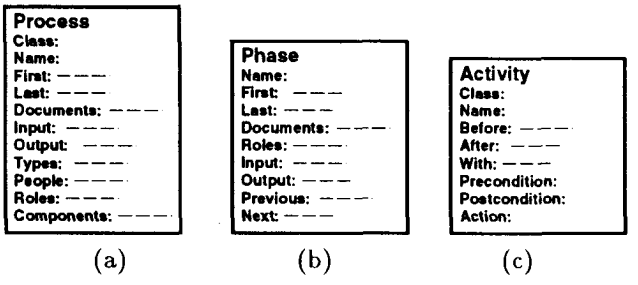


Figure 5: Active objects of the process representation.

The development of a software product is controlled by a *process object*, which is the repository for all information pertaining to the project. The template for a process object is shown in Figure 5(a). The attributes of a process object are: the type of process being enacted (*class*), the name of the particular project (*name*), a link to the first phase of the process (*first* — as illustrated in Figure 6), a link to the final phase of the process (*last* — also illustrated in Figure 6), links to all of the documents of the process (*documents*), the input to the process (*input*), the output of the process (*output*), links to the descriptions of all document types in the process (*types*), links to all people working on this project (*people*), links to the descriptions of all roles involved in the process (*roles*), links to all components of the product under development (*components*).

*Phase objects*, whose template is given in Figure 5(b), are used to group together logical sequences of activities. Figure 7 is an example of a phase object; this phase object describes the sequence of activities which must be undertaken in the implementation phase of a project. The use of phase objects allows the software process to be broken down into manageable pieces. Users of the environment can thus look at the process to a level of detail which is best suited to their particular needs. The attributes of a phase object are: the name of this phase of the process (*name*), a link to the first activity of the phase (*first* — as illustrated in Figure 7), a link to the final activity of the phase (*last* — also illustrated in Figure 7), links to the documents associated with this phase of the process (*documents*), links to the roles required to complete the phase (*roles*), the necessary input to the phase (*roles*), the output from the phase (*output*), a link to the previous phase (*previous* — illustrated in Figure 6), and a link to the next phase (*next* — also illustrated in Figure 6).

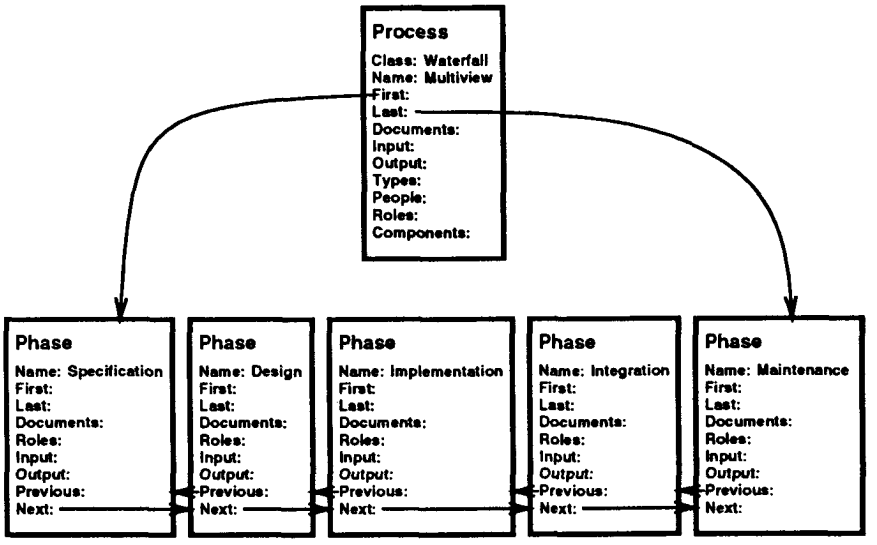


Figure 6: Example process overview.

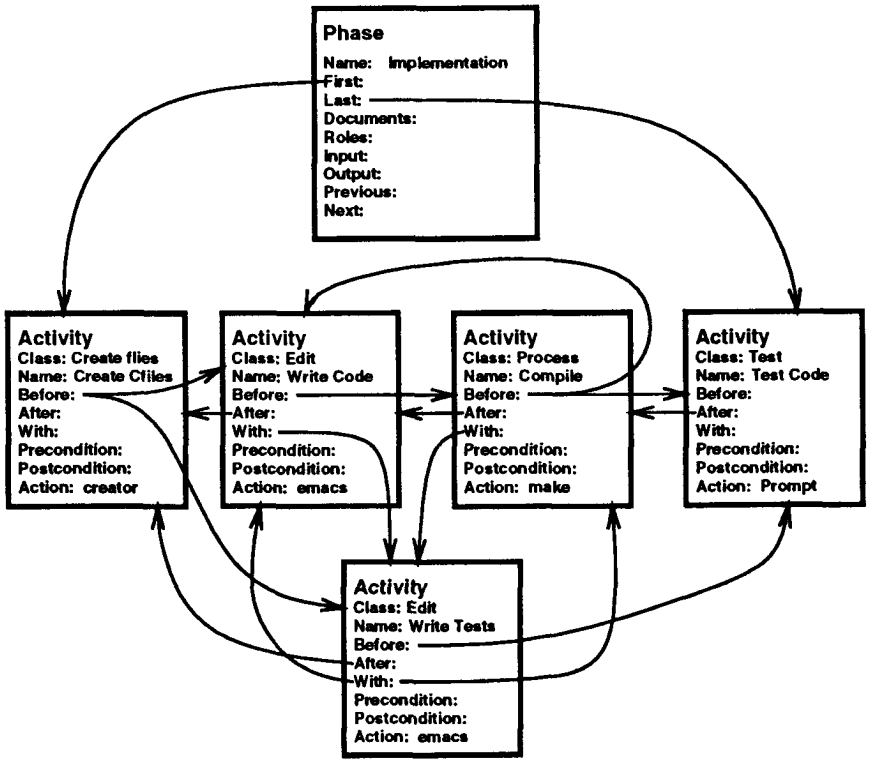


Figure 7: A description of a phase.

An *activity object* is a description of an action which is a step in the software development process. Figure 8 is an example of an activity description, that for the compilation activity of an implementation phase of a development project. The template for a phase object is shown in Figure 5(c). The attributes of an activity object are: the class of this activity (*class*), the name of this activity (*name*), links to the previous activities (*before*), links to the next activities (*after*), links to activities which may occur concurrently (*with*), the precondition for this activity to occur (*precondition*), the postcondition to be satisfied in order to complete this activity (*postcondition*) and the actions to be carried out in order to complete this activity (*action*).

Figure 7 illustrates the use of the before, after, and with links of activity objects to prescribe a partial ordering on the steps to be taken within a phase. The combination of this partial ordering and the pre- and post-conditions on each activity gives a mixture of declarative and imperative process programming which provides great flexibility while still allowing possible sequences to be easily determined by users of the environment.

**Activity**
**Class: Process**
**Name: Compile**
**Before:**
**After:**
**With:**
**Precondition: <file>*state = to_be_compiled**
**Postcondition: <file>*state = complete & ∃<file>*name[.o]**
**| <file>*state = to_be_coded**
**Action: make**

Figure 8: A description of an activity.

## 4 Future Work

The software process representation described in this paper allows multiple views of the software process to be produced, in order to meet the differing needs of the project team members. Provision of multiple views of the process being enacted by a project team is a powerful mechanism for improving the team members' understanding of the process and thus the overall quality of the software produced. The interactive interpretive approach to software process enactment which is proposed for the environment allows the process to be refined on the fly, as all objects are treated equally in the system and can be created, modified, or destroyed as required.

The process representation described in this paper has made some progress towards finding solutions to two of the major problems in PCSDEs. Firstly, it addresses the need to produce tailored views of the process for each team member, which are all generated from a canonical representation of the process. Secondly, it provides for changes to the process dynamically as it is being enacted.

The next step in this project is to define a protocol for the communication between the process views and the database/process engine. For this task, a number of the tools produced within the MultiView project will be used; specifically, a communications protocol compiler[5] will be employed which performs some analysis on a defined protocol and then produces Ada packages to implement this protocol. Once the communications protocol has been defined, the environment will be implemented. As with the MultiView system, this implementation will use Ada and the X Window System.

The final phase of this project will be the test and evaluation phase. In this phase, many views of the software process will be implemented and evaluated in order to determine their usefulness to different members of software project teams. As part of this, attempts will be made to mimic the interface of some existing PCSDEs in order to test the generality of the process representation and the communications protocol. In order to test the environment's process modeling capability, software processes used by industrial project teams will be modeled and enacted.

## Acknowledgements

The work described in this paper forms part of a long-term collaborative software engineering research programme involving the Discipline of Computer Science at Flinders University and the CSIRO-Macquarie University Joint Research Centre for Advanced Systems Engineering; funding from the CSIRO Institute of Information Science and Engineering is gratefully acknowledged.

## References

[1] R. A. Altmann, A. N. Hawke, and C. D. Marlin. An integrated programming environment based on multiple concurrent views. *The Australian Computer Journal*, 20(2):65–72, 1988.

[2] W. Deiters and V. Gruhn. Managing software processes in the environment Melmac. In Richard N. Taylor, editor, *Proceeding of the 4th ACM SIGSOFT Symposium on Software Development Environments SIGSOFT Software Engineering Notes*, volume 15, pages 193–205. ACM Press, 1990.

[3] A. Finkelstein, J. Kramer, and B. Nuseibeh. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

[4] C. D. Marlin. A distributed implementation of a multiple view integrated software development environment. In *Proceeding of the 5th Conference on Knowledge-Based Software Assistant*, pages 388–402, 1990.

[5] M. J. McCarthy and C. D. Marlin. Interprocess communication protocol support in a distributed integrated software development environment. In *Proceedings of the Seventeenth Annual Computer Science Conference, Part B*, volume 16, pages 363–371, 1994.

[6] B. Nuseibeh, J. Kramer, and A. Finkelstein. Expressing the relationship between multiple views in requirements specification. Unpublished Report, Department of Computing, Imperial College, London, UK, 1992.

[7] L. Osterweil. Software processes are software too. In *Proceeding of the $9^{th}$ International Conference on Software Engineering*, pages 2–13, 1987.

[8] B. Peuschel, W. Schafer, and S. Wolf. A consistency model for team cooperation. In *Proceedings of the $7^{th}$ International Software Process Workshop*, pages 114–116, 1991.

[9] M. C. Pong and N. Ng. Pigs - a system for programming with interactive graphical support. *Software Practice and Experience*, volume 13, pages 847–855. 1983.

[10] S. P. Reiss. *Graphical Program Development with PECAN Program Development Systems.* ACM, 1984.

[11] S. P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, 11(3):276–285, 1985.

[12] W. Schäfer and S. Wolf. *Multi-User Support in the Process-Centered Software Engineering Environment Merlin.* In W. Riddle, editor, *Proc. of the Workshop on Process sensitive Environment Architectures, Boulder, Colo.*, 1992.