SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **30**:175–197

Process program change control in a process environment

Shih-Chien Chou[1†] and Jen-Yen Jason Chen[2,*,‡]

[1]*Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan*
[2]*Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan*

SUMMARY

Process programs may change during enactment. The change of an enacting process program should be controlled. This paper describes process program change control in the CSPL (Concurrent Software Process Language) environment, which consists of techniques to (1) define change plans, (2) analyze and handle change impacts, and (3) resume process programs. Moreover, we provide an editor to guide process program change, with which the consistency between a change plan and the actual change can be enforced. We have used the environment in an experimental setting, and that experience shows that change control provided by the environment depends significantly upon process programs. If the dependencies among process components are well-specified in a process program, the change control mechanism will detect all the affected components when the process program is changed. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: Process-centered Software Engineering Environment (PSEE); software process; process evolution; process program change control; impact analysis

INTRODUCTION

A *software process* is a partially ordered set of activities to develop software [1,2]. Activities are assigned to roles (played by developers) to develop products (software products, including documents and program code). Therefore, activities, roles and products are primary components in a process.

Software processes are becoming more and more complicated. Process-centered Software Engineering Environments (PSEEs) have thus been developed [3–17] to facilitate controlling processes. A PSEE is normally composed of a process language and an environment. The language is used to model processes as *process programs*, which can then be enacted in the environment.

*Correspondence to: Jen-Yen Jason Chen, Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsuch Road, Hsinchu 30050, Taiwan.
†E-mail: scchou@csie.ndhu.edu.tw
‡E-mail: jychen@csie.nctu.edu.tw

Received 6 May 1999
Revised 19 October 1999
Accepted 28 October 1999

Software processes may evolve [18–23]. Process evolution support is thus essential in a PSEE. Generally, process evolution can be accomplished through meta-process [18,21] or process program change [21]. This paper focuses on process program change, which corresponds to online process change as described in Reference [20].

In changing a process program, all process components are subject to change. Since there are relationships between process components, changing a component may affect others. For example, changing an activity may affect the product it produced. As another example, changing a product may affect the products dependent upon it, which should be changed accordingly to keep product consistency [24]. Therefore, process program change may significantly affect process enactment. Handling that change may thus consume much manpower. Although that handling can be delayed using a lazy change policy [20], it does, however, need to be handled sooner or later.

To improve efficiency, changing a process program should be controlled. First, a change plan should be defined. Secondly, change impacts (i.e. effects of change) should be analyzed. Thirdly, time and developers needed in the change should be estimated. If that change costs too much, a new change plan should be defined. Fourthly, the process program should be changed according to the change plan. Finally, change impacts should be handled and the process program should be correctly resumed. According to the above description, issues in process change control are (1) defining change plans, (2) analyzing change impacts, (3) changing process programs, (4) handling change impacts, and (5) resuming process programs. A PSEE that controls process program change should provide solutions to the above issues.

Many PSEEs support process evolution [18,19,21,25–31]. However, most PSEEs do not facilitate controlling process program change. It is developers' responsibility for that control. Manually controlling process program change by developers, however, may cause problems. First, change impacts for large projects may be too complicated to analyze. An improper impact analysis may underestimate change efforts and result in an inappropriate change plan. Secondly, the actual change may be inconsistent with the change plan, which may in turn result in an incorrect process program. Thirdly, improper handling of change impacts may result in incorrect processes, product inconsistency, or garbage products.

We have developed a PSEE called the CSPL (Concurrent Software Process Language) environment [3,32–34] which supports process evolution [21]. From past experience, however, we found that its impact analysis technique is insufficient. Moreover, it does not facilitate controlling process program change. We thus enhanced CSPL to control the change. This paper describes process program change control in CSPL. In the remainder of this paper, an overview of the CSPL environment is first given. Then, CSPL process program change control is described.

CSPL OVERVIEW

CSPL is a Unix-based PSEE. It has been ported onto a PC, and will finally be ported onto the Internet. CSPL environment is composed of the CSPL language, a CSPL compiler, a CSPL server, an Object Management System (OMS) server, multiple CSPL clients, and multiple OMS clients. Figure 1 shows the CSPL architecture.

The CSPL language is used to develop process programs. The CSPL compiler translates process programs for enactment. During process enactment, the CSPL server interacts with CSPL clients to

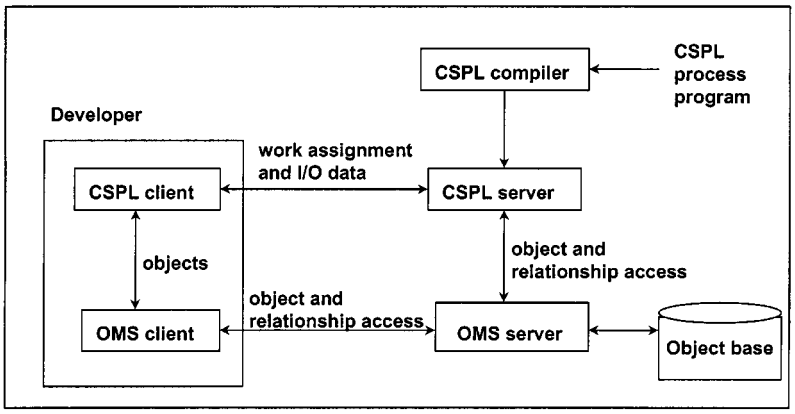

Figure 1. CSPL environment architecture.

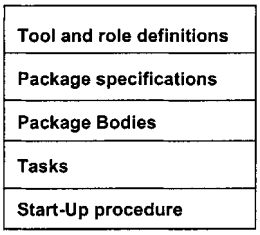

Figure 2. CSPL process program structure.

assign work to developers, and accesses objects (products) and their relationships via the OMS server. In the following sections, the CSPL language and process evolution support are described.

CSPL language

The CSPL language models software process components including roles, tools, products, packages, tasks, exceptions, and so on. The general structure of a CSPL process program is shown in Figure 2, in which packages are composed of their specifications and bodies.

Tool definitions define tools to be used. Role definitions define roles played by developers. Figure 3 shows some tool and role definitions. Note that in the figure, bold face words denote CSPL keywords.

Package specifications define *product types* and *operation interfaces*, where two kinds of operations, namely procedures and functions, can be defined. Figure 4 is a package specification defining the product types 'Requirement' and 'Specification', the procedure interface 'GenSpec', and the function interface 'ReviewSpec'. Those types inherit the built-in type 'DocType' and add new attributes. Product


```
-- tool definition
tool ToolSet is
   editor := "vi";
   AnalysisTool := "ROSE";
   ReviewTool := "RTool";
end;

-- role definition
role analyst is
   analyst1 := "jychen";
   analyst2 := "scchou";
end;
```

Figure 3. Tool and role definitions.

```
-- Packages specification
package sys_analysis is
   -- Product type definitions
   type Requirement is new DocType with record
      RequirementDescription: TextType;
   end record;
   type Specification is new DocType with record
      LanguageSpec: TextType;
     Notation: NonTextType;
   end record;

   -- Operation interfaces
   procedure GenSpec(requirement: in Requirement, specification: in out Specification);
   function ReviewSpec(requirement: in Requirement, specification: in Specification)
                                                          return integer;
end sys_analysis;
```

Figure 4. Package specification.

types defined in a package specification can be instantiated. For example, the following statement instantiates the product 'specification' from the type 'Specification':

```
specification: Specification := "spec.doc";
```

where 'spec.doc' is the name of the file for storing the product 'specification'.

Package bodies specify *operation details*. Figure 5 shows a package body. The most important statement used in a package body is the work assignment statement (i.e. the 'edit' statement), where each statement defines an activity. A work assignment statement assigns work to developers, binds tools, indicates product(s) for reference, and requires the developers to create a product. An example work assignment statement is as follows:

```
2 analyst edit specification referring to requirement using AnalysisTool;
```



```
-- Package body
package body sys_analysis is
   procedure GenSpec(requirement: in Requirement, specification: in out Specification) is
   begin
      2 analyst edit specification referring to requirement using AnalysisTool;
   end;

   function ReviewSpec(requirement: in Requirement, specification: in Specification) is
   begin
      review_result: integer;
      all reviewer review specification referring to requirement using ReviewTool
                                                                  resulted in review_result;
      return review_result;
   end;
end sys_analysis;
```

Figure 5. Package body.

```
task body RequirementAnalysis is
begin
   loop
      accept start;
      sys_analysis.GenSpec(sys_requirement, sys_specification);
      review_result := sys_analysis.ReviewSpec(sys_requirement,sys_specification);
      if review_result=1 then
         -- A meta-process to create and enact a design process
         1 ProcessProgrammer edit DesignProcess using editor;
         enact DesignProcess;
      end if;
   end loop;
end;
```

Figure 6. CSPL task.

A CSPL *task* groups related activities that are enacted sequentially. Generally, activities are accomplished by calling operations defined in package bodies. Activities, however, can also be accomplished by using work assignment statements. Figure 6 depicts a CSPL task.

Exceptions and their handlers, which are defined in tasks, are used to model activities that cannot be regularly controlled. The following statements depict the handler of the exception 'RequirementChange':

```
exception
   when RequirementChange =>
         output "Requirement change, redo the analysis work!!";
         SysAnalysis.start;
```

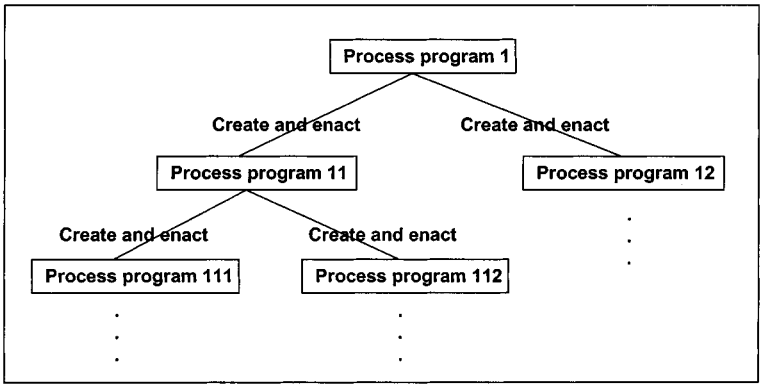

Figure 7. Process program enactment tree.

Process evolution support

CSPL supports process evolution through meta-process and process program change [21]. A meta-process is a process that creates and enacts other processes. CSPL uses the statements 'edit' and 'enact' to define meta-processes. The statement 'edit' informs a developer to create a process program, while the statement 'enact' enacts that newly created process program. With meta-process support, highly uncertain processes, such as processes that are currently unclear and those that will be enacted a long time later, need not be defined during project planning. They can be defined by enacting meta-processes when the uncertainty is resolved. Figure 6 depicts the use of meta-process, in which the task 'RequirementAnalysis' contains the analysis work and a meta-process. The analysis work includes two activities: 'sys_analysis.GenSpec' and 'sys_analysis.ReviewSpec', while the meta-process creates and enacts 'DesignProcess'.

In addition to meta-process, CSPL also supports process evolution through changing an enacting process program. To manage the change, CSPL records process program states during process enactment. When a process program is changed, CSPL analyzes and handles change impacts using the program's state. After change, CSPL resumes the process program. Process program state, impact analysis and handling, and process program resumption are respectively described below:

(i) Process program states are for identifying the affected process components during process program change. Since a process program may have created other process programs through meta-processes, changing a process program may affect the process programs it created. Moreover, since a process program will produce products, changing a process program may affect the products it produced. Therefore, process program states should facilitate identifying the affected process programs and products. To address these needs, CSPL records *process program enactment trees* (see Figure 7) and relationships between process programs and products (see Figure 8) as the process program state. The former is for identifying affected process programs, and the latter for identifying affected products.

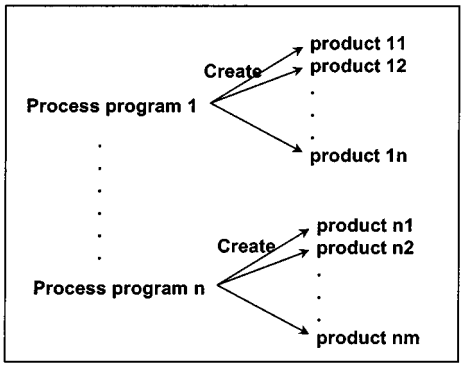

Figure 8. Relationship between process programs and products.

(ii) In analyzing change impacts, the state of the process program being changed is used. That is, CSPL identifies affected process programs by tracing process program enactment trees (Figure 7), and identifies affected products by tracing relationships between process programs and products (Figure 8). The affected components are then reported to CSPL users, who should decide upon the handling of the affected components. In handling the affected products, CSPL removes those that should be deleted, and informs developers to change those that should be changed. In handling the affected process programs, CSPL informs developers to change those that should be changed, and stops those that should be killed. CSPL also removes the products produced by the process programs that have already been killed.

(ii) To resume a changed process program, CSPL language provides a *resume block* for indicating which activities should be enacted after resumption. As shown in Figure 9, a resume block starts and ends with the statements 'resume block is' and 'end resume block', respectively. Statements inside resume blocks will be enacted after process resumption.

CSPL PROCESS PROGRAM CHANGE CONTROL

CSPL controls process program change by automatically enacting the following process. Note that the role 'project manager' in the following process can be replaced by a developer responsible for the change.

Step 1: CSPL suspends the process program to be changed and records the program's state in its Object Management System (OMS) for impact analysis.
Step 2: CSPL guides the project manager to define a change plan.
Step 3: CSPL, together with the project manager, analyzes change impacts according to the change plan.


```
task body RequirementAnalysis is
begin
   loop
      accept start;
      sys_analysis.GenSpec(sys_requirement, sys_specification);
      resume block is
         review_result :=
              sys_analysis.ReviewSpec(sys_requirement,sys_specification);
         if review_result=1 then
            -- A meta-process to create and enact a design process
            1 ProcessProgrammer edit DesignProcess using editor;
            enact DesignProcess;
         end if;
      end resume block;
   end loop;
end;
```

Figure 9. Process program with resume block.

Step 4: The project manager estimates the efforts needed for the change, and then decides whether the change plan is acceptable. If not, go back to Step 2. Otherwise, proceed to the next step.
Step 5: CSPL guides process programmers to change the process program according to the change plan.
Step 6: CSPL guides developers to handle the change impacts.
Step 7: CSPL resumes the changed process program.

Although process program states and the technique for analyzing and handling change impacts have been defined in CSPL [21], they are insufficient for controlling process program change. For example, changing a product may affect other products. This effect cannot be identified from Figures 7 or 8. Therefore, we have redefined process program state, and redesigned the technique for impact analysis and handling. In the following sections, process program change control in CSPL are respectively described, including (1) the newly defined process program state, (2) change plan definition and impact analysis, (3) changing process program according to a change plan, and (4) impact handling. As for process program resumption, it is controlled by CSPL resume block described near the end of the previous section.

Process program state

Process program states are used to identify affected process components when a process program is changed. This identification can be accomplished through tracing relationships among process components. For example, tracing dependency relationships can identify the affected design document and program code when a specification is changed. Therefore, a process program state is composed

of process components linked by relationships. Generally, all CSPL process program constructs (see Figure 2) should be included in a process program state, because they are all subject to change. Effects of changing CSPL process program constructs are described below:

(i) Changing roles or tools will affect activities, because they are involved in activities to accomplish work. Moreover, since a product developed by a tool is bound to that tool, changing a tool may affect products developed by the tool.

(ii) A package specification is composed of product types and operation interfaces. Therefore, changing a package specification may affect its product types and operation interfaces. Product types are used to instantiate products, which will be used in activities. Therefore, changing a product type may affect products instantiated from it, and may also affect activities using those products. As for operation interfaces, changing them may affect their operation details specified in package bodies. Moreover, since an operation may be invoked by tasks or other operations, changing an operation interface may affect tasks and operation details invoking that operation.

(iii) A package body is composed of operation details. Therefore, changing a package body may affect its operation details. Since operation details are composed of activities, changing operation details may affect activities, which are represented in work assignment statements. A work assignment statement assigns work to roles, binds tools, indicates product(s) for reference, and requires the roles to create a product under possible schedule and/or budget constraints. Therefore, changing a work assignment statement (activity) may affect roles, tools, products and constraints. Moreover, an affected product may in turn affect other products because of possible dependency relationships among products [3].
Since a work assignment statement may create a process program (that is, the statement together with an 'enact' statement" constitute a meta-process), changing a work assignment statement may also affect other process programs.

(iv) Tasks, like package bodies, are also composed of activities. Therefore, changing a task and changing a package body have the same effect.

According to the above description, a process state is composed of the following components and their relationships: role, tool, package specification, product type, product, operation interface, package body, operation details, task, activity, constraint, and process program. Figure 10 shows those components linked by 'affect' relationships. Arrows in the figure show the affecting direction. For example, an arrow from 'Product type' to 'Product' means that changing a product type will affect the products instantiated from it.

During process enactment, CSPL keeps in its OMS a state for each enacting process program. Process program states can be examined to analyze the impact when an enacting process program is interrupted for change. Figure 11 shows the state for the process program in Appendix I when the statement 'enact DesignProcess' is enacting. Note that in the figure, rectangles represent process components and arrow-headed lines represent 'affect' relationships. The lines are used to identify affected process components. For example, if 'package body: sys_analysis' in Figure 11 is changed, then 'operation detail: GenSpec' and 'operation detail: ReviewSpec' will be directly affected. By following the 'affect' relationships linked to the latter two components, those that will be indirectly affected can also be identified.

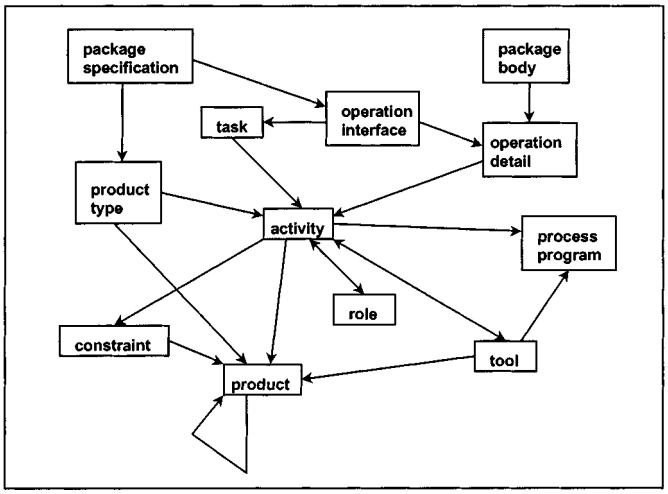

Figure 10. Components and relationships in a process state.

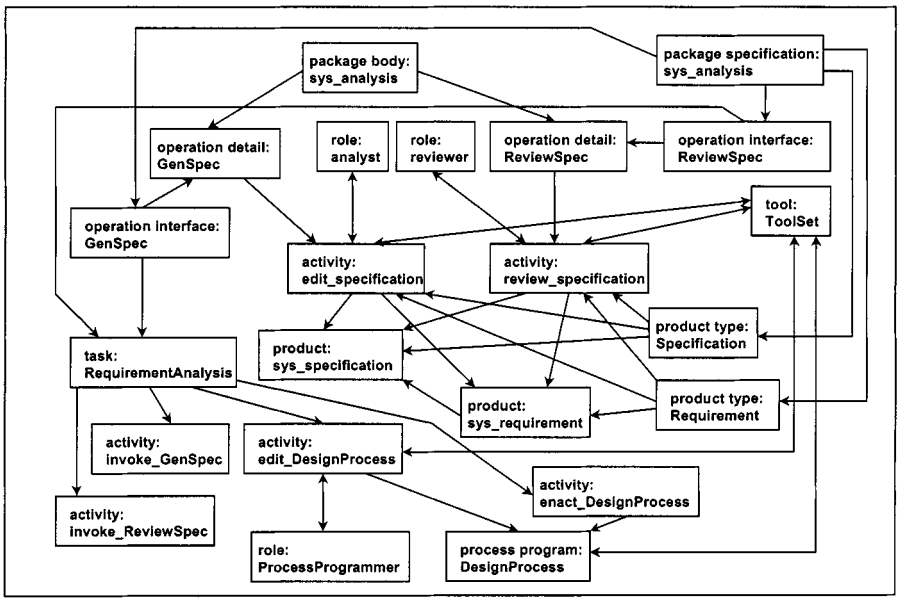

Figure 11. Process program state example.

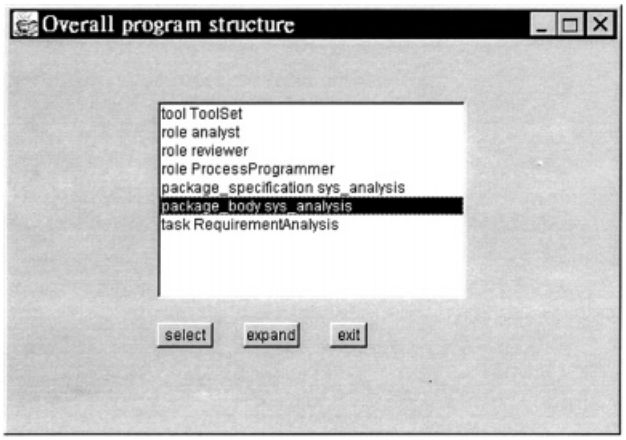

Figure 12. Window for overall program structure.

Change plan definition and impact analysis

To facilitate defining the change plan, CSPL displays the process program structure to guide the project manager. For example, to define a plan to change the process program in Appendix I, the program structure is shown in the window of Figure 12. When a construct in the window is selected to be changed, its components will be displayed for further selections. For example, Figure 13 shows the window displayed after 'package_body sys_analysis' in Figure 12 is selected. The iterative selection procedure goes on until all components that should be changed have been selected. The selected components constitute the primary part of a change plan. To support this selection procedure, the CSPL compiler constructs a process program structure for each process program. The structure looks like that in Figure 14, where arrow-headed lines represent composition relationships.

After a change plan is defined, the CSPL environment interacts with the project manager to analyze change impacts. In that analysis, CSPL traces the process program's state (see Figure 10) to automatically identify the process components that may be affected, and then reports them to the project manager, who should determine which of them are indeed affected. The analysis begins with identifying the *inherently affected components*, which include the components to be changed and the activities that are enacting when the process program is interrupted for change. For example, if 'operation detail: ReviewSpec' in Figure 11 is to be changed, then it is an inherently affected component.

Starting from the inherently affected components, CSPL traces the 'affect' relationships in the process program state to identify process components that may be directly affected. For example, with the process program state in Figure 11, suppose that 'operation detail: ReviewSpec' is an inherently affected component. Then, 'activity: review_specification' may be directly affected.

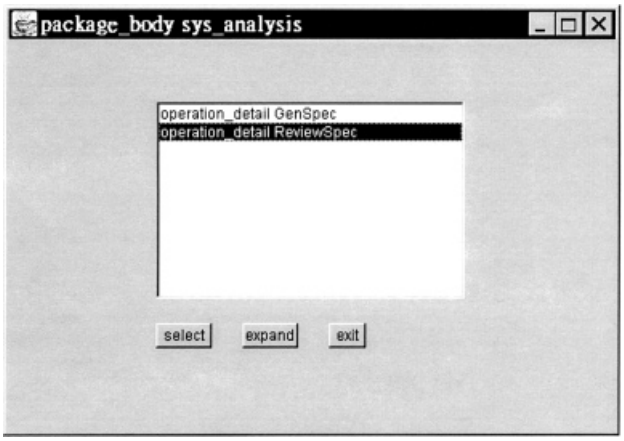

Figure 13. Window for sub-program structure of 'package_body sys_analysis'.

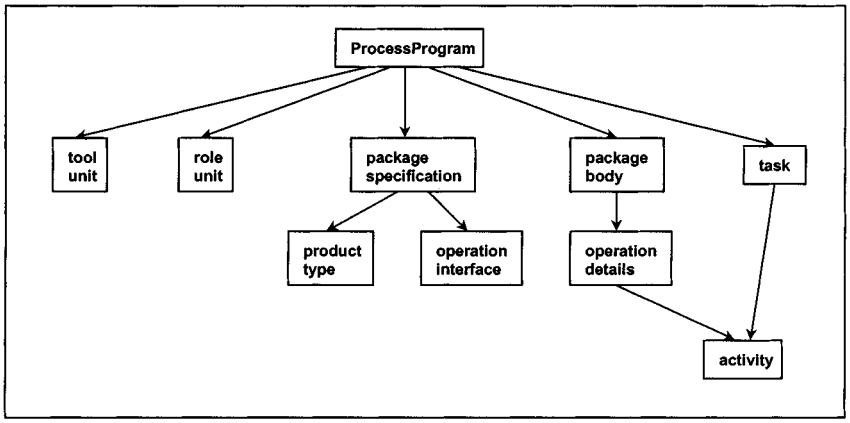

Figure 14. Process program structure.

Having identified those that may be directly affected, CSPL reports them to the project manager, who should determine which are indeed affected. CSPL then identifies those that may be indirectly affected by tracing 'affect' relationships linked to the affected components. For example, in Figure 11, suppose that the project manager determines that 'activity: review_specification' is indeed affected. Then, CSPL identifies that 'product: sys_requirement' and 'product: sys_specification' may be indirectly affected. Those that may be indirectly affected will then be reported to the project manager, who should again determine which ones are affected. The identification by CSPL and determination by the

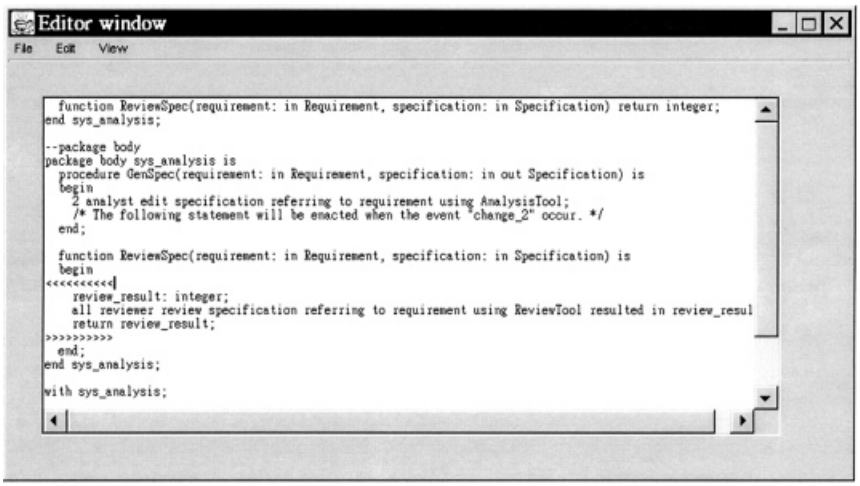

Figure 15. Editor for process program change.

project manager constitute an impact analysis process, which iterates until all of the affected process components are identified.

Having identified all of the affected process components, the project manager estimates the budget and schedule needed in changing the affected components. If the schedule or budget of the project do not allow for the change, CSPL guides the project manager to define a new plan and then analyzes the change impacts again. The change plan definition and impact analysis procedure iterates until an acceptable change plan is obtained.

Changing process program

Changing an enacting process program may affect the following components: (1) process program constructs such as activities, tools, and so on; (2) products that have been produced or are being produced; and (3) process programs that have been created by the changed process program through meta-processes. Handling of the second and third items above will be described in the next section. To handle the first item above, which corresponds to changing the program, CSPL provides an editor to guide the change. Figure 15 shows a window of that editor, in which a process program construct to be changed (the part that is affected) is enclosed with the marks '<<<<<<<<<<' and '>>>>>>>>>>'. Only statements between those marks are allowed to change.

Impact handling

Impact handling refers to handling the affected components in a process program change. As stated before, the affected components may be (1) process program constructs, (2) products that have been

produced or are being produced, and (3) process programs that have been created. The handling of affected program constructs, which corresponds to changing the process program, has been described in the previous section. As for the affected products, CSPL interacts with developers to decide upon the handling. Generally, an affected product should be deleted or changed. For a product that should be deleted, CSPL removes it from the OMS. For a product that should be changed, CSPL first identifies the developers responsible for the product and the tool bound to the product. It then tells the developers to change the product using the tool.

To handle the affected process programs, CSPL interacts with developers to decide upon the handling. Generally, an affected process program should be killed or changed. For an affected process program that should be killed, CSPL removes the products it produced. If the process program is still enacting, CSPL tells developers to stop the program immediately. For an affected process program that should be changed, CSPL guides developers to change it by following the change process described in this section.

AN EXAMPLE

The process program in Appendix I is used as an example. It is composed of an analysis process and a meta-process for creating a design process (see the task body 'RequirementAnalysis'). The analysis process is composed of an activity to generate a specification (which is accomplished by calling the operation 'sys_analysis.GenSpec') and an activity to review the specification (which is accomplished by calling the operation 'sys_analysis.ReviewSpec'). Details of the operations are specified in the package body 'sys_analysis'. Here the specification is verified by a formal review.

Suppose that when the statement 'enact DesignProcess' near the end of the process program is enacting, the project manager finds that errors exist in the specification. He/she then justifies that the situation occurs because the customer 'syjan', who is not a computer expert, has trouble with following the formal review process. The project manager thus decides to change the review approach from formal review to rapid prototyping. This causes the enacting process program to be changed.

To change the process program, CSPL saves its state (as shown in Figure 11). It then shows the process program's structure for guiding the project manager to define a change plan. CSPL first displays the window in Figure 12. The project manager then selects the 'package_body sys_analysis' to change. CSPL then displays the contents of the package body in the window of Figure 13. At this time, the project manager selects the 'operation_detail ReviewSpec'. CSPL then displays the contents of the operation details in the window of Figure 16. Here, the project manager selects the 'activity review_specification' to change, because the review approach should be changed. After the selection, CSPL re-displays the window in Figure 12 for selecting another process component to change. By following a similar procedure, the project manager selects the package specification 'sys_analysis' and the task 'RequirementAnalysis' to change. The former should be changed because a product type 'RapidPrototype' should be added. The latter should be changed because a resume block should be added to direct process program resumption. After the selection procedure above, the change plan has been defined, in which the activity 'review_specification', the task 'RequirementAnalysis' and the package specification 'sys_analysis' are required to be changed.

According to the change plan, CSPL interacts with the project manager to identify the affected process components. First, the inherently affected process components are identified. They

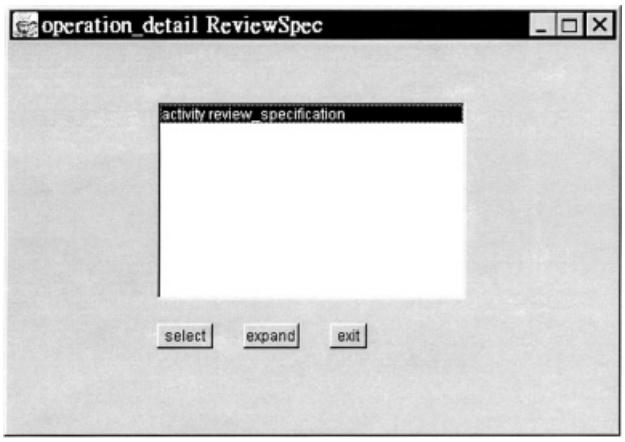

Figure 16. Window for sub-program structure of 'operation_detail ReviewSpec'.

are (see Figure 11): 'package specification: sys_analysis', 'task: RequirementAnalysis', 'activity: review_specification', and 'activity: enact_DesignProcess', in which the first three are planned to be changed and the last one is an activity that is enacting when the process program is interrupted. The impact analysis procedure starts with the above four components. Tracing the 'affect' relationships will identify those that may be affected. Those that may be affected are then reported to the project manager, who should determine which are indeed affected. After the impact analysis procedure, the process components 'product: sys_specification' and 'process program: DesignProcess' are identified as affected.

After impact analysis, suppose that the project manager accepts the change plan. Then, according to the change plan, CSPL uses an editor to guide the change. Figure 15 shows the window for the editor, in which 'activity: review_specification' is enclosed with the marks '<<<<<<<<<<' and '>>>>>>>>>>' for change.

Having changed the process program, CSPL interacts with the project manager to handle the affected process components (i.e. 'product: sys_specification' and 'process program: DesignProcess'). CSPL then resumes the process program by enacting the statements inside resume blocks. For example, suppose that the task 'RequirementAnalysis' has been changed as shown in Figure 17. Then, the activities 'sys_analysis.ReviewSpec', 'edit DesignProcess', and 'enact DesignProcess' will be enacted after process resumption.

EXPERIENCES

Some experiences obtained are discussed in this section. To prevent possible confusion, we clarify that the CSPL environment has been upgraded to the third version. The first CSPL version does not support process evolution. The second CSPL version supports process evolution through meta-process

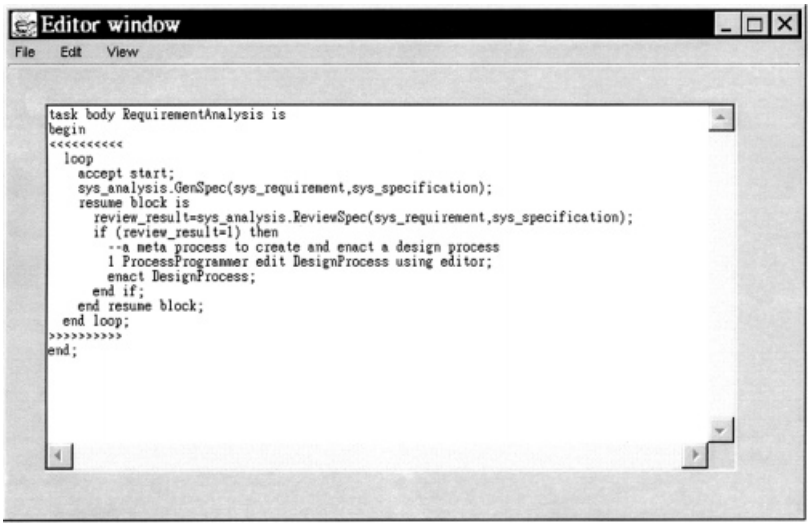

Figure 17. Editor window.

and process program change [21]. However, it does not control process program change. The third CSPL version controls process program change as described in this paper.

Since the development of the first CSPL version, we have required our students to use it on their term projects. From their usage, we can see that the processes of student projects change frequently, which seems to be caused by poor project planning. Although commercial software processes may not change as frequently, we do believe that processes will change according to schedule or budget overrun, requirement change, and so on.

In handling process program change, we found that heavy human engagement is unavoidable. In that change, developers are involved in defining a change plan, estimating change efforts, analyzing impacts, changing the process program, handling change impacts, and resuming the process program. In the first CSPL version, no process evolution support is available, therefore developers have to manually do all the work just mentioned to change a process program. We thus upgraded CSPL to the second version, which supports process evolution. From the use of the second CSPL version, we found that impact analysis and handling, and the process of change play important roles in process program change, as described below.

(i) Impact analysis, as stated before, identifies the process components affected when an enacting process program is changed. To analyze change impacts, the second CSPL version identifies process programs created and products produced by the changed process program. This support, however, is insufficient, because impact analysis may be incomplete. For example, one of our students developed a Computer-Aided Instruction (CAI) system using CSPL. After the implementation process started for a period of time, he identified that the analysis process was

incorrect. He thus decided to change that process. In this case, the CSPL environment informed him that the CAI specification, which was created by the analysis process, was affected. With this information, he modified the specification. However, apparently, the design document and program code were also affected, because they were produced based on the specification. These effects were not identified by the second CSPL version, and therefore resulted in an incomplete impact analysis.

Since impact analysis results are used to estimate process change efforts, an incomplete impact analysis may underestimate change efforts. With the underestimated efforts, an incorrect change plan may be defined. Moreover, an incomplete impact analysis may also cause incomplete impact handling, and other problems. For example, as just mentioned, CSPL informed the student who developed the CAI system to change the CAI specification when the analysis process was changed. CSPL, however, did not tell him to change the corresponding design document and program code. If the student did not change the design document and program code, inconsistency between products will result.

(ii) As to the process of change, we found that the actual change may be inconsistent with the change plan, because the second CSPL version does not control the change. For example, four students cooperated to develop an inventory management system on the Internet. They originally decomposed the system into a client subsystem and a server subsystem. During process enactment, they decided to decompose the system into more detail and then develop the subsystems *in parallel*. One student was assigned to change the process program. He, however, incorrectly changed the process program to require developers to develop the subsystems *in sequence*. The change was apparently inconsistent with the change plan, and thus resulted in an incorrect process. In addition to inconsistency, an uncontrolled change may cause other problems. For example, in changing a process program, some students did not define a change plan, and some did not handle change impacts. Changing process programs in this uncontrolled manner may result in incorrect processes, product inconsistency, budget or schedule overrun, and so on.

The experiences obtained from the second CSPL version stated above motivated the development of the third CSPL version, which controls process program change. The change control is accomplished through providing tools to (1) analyze and handle change impacts, (2) facilitate defining change plans, (3) guide process program change, and (4) resume process programs. We expect that problems encountered in the second CSPL version will be solved in the third CSPL version. Currently, we have required our students to use the third CSPL version, from which we found that the change control support in this CSPL version is as good as the process program itself. That is, if some dependencies are not specified in the program, the change control mechanism will not automatically detect such dependencies. For example, suppose that the dependency between a specification and a design document is not specified in the process program. Then, when the specification is changed, the change control mechanism will not detect that the design document should be changed accordingly.

RELATED WORK

Process change is a kind of process evolution [21]. Therefore, this section discusses related work in process evolution. Generally, process evolution research can be roughly classified into two categories:

(1) those collecting and analyzing data, and then evolving processes according to analysis results [35–40]; and (2) those supporting process program evolution during enactment [18,19,21,25–31,41]. Process change support in CSPL belongs to the second category. Therefore, only research in this category is discussed.

EPOS provides a reflective, rule-based process language called SPELL [6]. During process enactment, EPOS stores meta-classes, classes, and instances of process models in a database called EPOSDB [19]. All the contents in EPOSDB are subject to change. Changing meta-classes, classes or instances of process models results in process evolution. In a change, the Process Model (PM) manager is responsible for controlling the change. First, the PM manager checks whether the change is allowed. If so, the PM manager analyzes the change impacts and then interacts with users to handle the impacts. According to the above survey, we identify that EPOS supports impact analysis and handling. Regarding change plan definition and change process control, EPOS does not provide such support.

PROSYT [30,31] allows the observed process to diverge from the modeled process. By tolerating process deviation, minor process evolution can be accomplished. In the environment, a process enactment is associated with preconditions and constraints. An enactment fulfilling its preconditions corresponds to a process that does not diverge. An enactment that fails to meet its preconditions corresponds to a process that diverges. For a diverged process, if the constraints are not violated, the deviation is tolerable. Otherwise, the deviation is intolerable. The occurrence of an intolerable deviation will trigger a 'pollution analysis' activity to discard garbage data (products). Here the activity is similar to impact analysis. From this survey, we can see that PROSYT does not allow process change during enactment. Instead, it tolerates minor process evolution and analyzes impacts when intolerable deviations occur.

SPADE [18] provides a reflective, Petrinet-based process language called SLANG [25,26], and uses multiple process engines to enact a process model. Before enacting an activity, an active copy of the activity should be prepared. The environment thus allows late binding between activity definitions and invocations. With the late binding feature, process evolution in SPADE is accomplished by changing active copies of activities. From the survey of SPADE we identify that the environment, like the second CSPL version, just supports process evolution. As for the issues of change plan definition, impact analysis and handling, and process program change control, SPADE does not provide solutions. SPADE, however, is not the only PSEE that does not provide solutions to those issues. The following paragraphs describe process evolution support in some PSEEs that also do not provide solutions to the issues.

Process Weaver [8] allows process models (process programs) to be instantiated into process instances. The environment provides a mechanism for dynamically changing the process models being enacted. With the mechanism, the process instances that are scheduled for future enactment can be changed through the late binding feature. Accordingly, a process model need not be completely defined before enactment. Instead, the model can be changed dynamically during enactment to complete the definition of the model.

In OPSIS [28], process models are decomposed into views such as role, product, and so on. It provides techniques to extract views from a process model and compose views. Process evolution in OPSIS focuses on views instead of on the whole process model. Evolving processes on views seems more accurate and convenient than on the whole process model, as the authors stated.

In the OBM environment [29], processes operating on a product are defined as the product's methods. Products in the environment are instantiated from product types, which are modeled as

abstract objects. Process evolution is accomplished by changing the methods of abstract objects. Method change is accomplished by invoking the method 'changeself'. The OBM language can thus be considered a reflective language that facilitates process change.

Tempo [41] models processes as types for instantiating. Process evolution is thus accomplished through late binding. Tempo uses a lazy change policy [20], which means that the process instances being enacted will not be affected by the change. Tempo also supports process evolution through dynamically changing role during enactment. Since different roles perform different activities on the same product, changing role corresponds to changing process.

The Agile Software Process (ASP) model is composed of a number of lightweight processes [42]. With them, an ASP model can quickly adapt the change in requirements, therefore ASP does not really support process evolution. However, it reduces the need for process evolution.

As stated in this paper, CSPL supports the process program change by facilitating change plan definition, analyzing change impacts, controlling process program change, handling change impacts, and resuming process programs. Comparing the CSPL approach with the PSEEs discussed above, we have the following findings:

(i) It seems that none of the above PSEEs controls process program change; it is developers' responsibility. Therefore, a change process may take much time, and even worse, the change may be out of control.
(ii) It seems that only EPOS and PROSYT support impact analysis and handling. Without impact analysis, the cost of change may be underestimated. Without impact handling, garbage products may exist and products may become inconsistent.
(iii) It seems that none of the above PSEEs facilitates defining a change plan; it is the project manager's responsibility. An ill-defined change plan may be costly or may be incorrect.
(iv) It seems that none of the above PSEEs guides developers in changing process programs, which may result in inconsistency between a change plan and the actual change.

CONCLUSIONS AND FUTURE WORK

This paper proposes a technique to control process program change in the CSPL (Concurrent Software Process Language) environment. The technique records a state for each enacting process program, which is composed of process components linked by 'affect' relationships. When a process program is changed, CSPL traces the program's state to analyze the impacts of change (i.e. to identify the process components affected). From the results of impact analysis, a change plan can be defined. According to the plan, a project manager estimates the change efforts. If the change costs too much, the change plan should be modified through the assistance of impact analysis. If the plan seems appropriate, an editor guides users to change the process program. CSPL then guides users to handle the affected process components and resumes the changed process program. According to the above description, CSPL controls process program change by (1) facilitating defining change plans, (2) analyzing change impacts, (3) guiding process program change, (4) handling change impacts, and (5) resuming process programs.

In the proposed technique, process program change follows a process controlled by the CSPL environment. This relieves the load of the project manager, because he/she no longer needs to


```
-- An example CSPL process program containing an analysis activity, a specification review
-- activity, and a meta-process to create a design process is listed below.

-- tool definition
tool ToolSet is
        editor := "vi";
        AnalysisTool := "ROSE";
        ReviewTool := "RTool";
end;
-- role definition
role analyst is
        analyst1 := "jychen";
        analyst2 := "scchou";
end;
role reviewer is
        reviewer1: = "jychen";
        reviewer2 := "scchou";
        reviewer3 := "syjan";
        -- "syjan" is a customer
end;
role ProcessProgrammer is
        process_programmer1 := "scchou";
end;

-- Packages specification
package sys_analysis is
        -- Product type definitions
        type Requirement is new DocType with record
                RequirementDescription: TextType;
        end record;
        type Specification is new DocType with record
                LanguageSpec: TextType;
                Notation: NonTextType;
        end record;
        -- Operation interfaces
        procedure GenSpec(requirement: in Requirement, specification: in out Specification);
        function ReviewSpec(requirement: in Requirement, specification: in Specification) return integer;
end sys_analysis;
-- Package body
package body sys_analysis is
        procedure GenSpec(requirement: in Requirement, specification: in out Specification) is
        begin
                2 analyst edit specification referring to requirement using AnalysisTool;
        end;
        function ReviewSpec(requirement: in Requirement, specification: in Specification) is
        begin
                review_result: integer;
                all reviewer review specification referring to requirement using ReviewTool resulted in
                        review_result;
                return review_result;
        end;
end sys_analysis;

with sys_analysis;
procedure StartTask is
-- Data definitions
sys_requirement: Requirement := "sys_requirement.doc";
sys_specification: Specification := "sys_specification.doc";
DesignProcess: TextType := "design.cspl";
review_result: integer;

-- Task specifications
task RequirementAnalysis is
        entry start;
end;
-- Task bodies
task body RequirementAnalysis is
begin
        loop
                accept start;
                sys_analysis.GenSpec(sys_requirement, sys_specification);
                review_result := sys_analysis.ReviewSpec(sys_requirement,sys_specification);
                if review_result=1 then
                        -- A meta-process to create and enact a design process
                        1 ProcessProgrammer edit DesignProcess using editor;
                        enact DesignProcess;
                end if;
        end loop;
end;

-- Begin of the process
begin
        RequirementAnalysis.start;
end;
```

Figure A.1.

coordinate developers for the change. The proposed change control technique offers the following features:

(i) Process program states facilitate impact analysis, from which the affected components can be identified to estimate change efforts. Process program states (and hence impact analysis) are thus helpful in planning a change. Moreover, if a change plan is decided, impact analysis results indicate which process components should be handled.
(ii) The program structure of an enacting process program guides the project manager to identify process components that should be changed. Process components to be changed are then collected to form the primary part of a change plan.
(iii) An editor is provided for changing process programs, in which process program constructs that should be changed are marked. Only the marked parts are allowed to change. Therefore, consistency between a change plan and the actual change can be enforced.
(iv) The change process is controlled by the CSPL environment, which coordinates developers and the necessary activities. This approach decreases the probability of errors, and relieves the load of the project manager.

As stated in the 'Experiences' section, we found that processes change frequently during enactment. We also found that process program change requires heavy human engagement. Although the CSPL environment controls process program change, the change is still costly. Therefore, in future we will provide a mechanism to reduce the frequency of change and justify whether a change is necessary. To reduce the frequency of change, CSPL will be enhanced to tolerate minor evolution. For example, the change of tools or roles will be tolerated by the enhanced CSPL version, so no process program change will be needed. To justify a change, we will identify proper metrics to evaluate process enactment. According to the evaluation, a poorly performing process should be changed. On the other hand, a normal or well-performing process need not be changed.

APPENDIX I: EXAMPLE CSPL PROCESS PROGRAM

See Figure A.1.

ACKNOWLEDGEMENT

This research is sponsored by the National Science Council in Taiwan under grant number NSC88-2213-E-009-012.

REFERENCES

1. Garg PK, Jazayeri M. *Process-Centered Software Engineering Environments*; IEEE Press, 1996; 17.
2. Feiler PH, Humphrey WS. Software process development and enactment: concepts and definitions. *Proc. 2nd Int. Conf. Software Process*, Los Alamitos, CA, 1993; 28–40.
3. Jason Chen J-Y. CSPL: An Ada95-like, Unix-based process environment. *IEEE Transactions on Software Engineering* 1997; **23**:171–184.

4. Belkhatir N, Melo WL. Supporting software development process in Adele 2. *The Computer J.* 1994; **37**:621–628.
5. Chen JY, Hsia P. MDL (Methodology Definition Language): A language for defining and automating software development process *J. Comput. Lang.* 1992; **17**:199–211.
6. Conradi R, Jaccheri ML, Mazzi C, Aarsten A and Nguyen MN. Design, use and implementation of SPELL, a language for software process modeling and evolution. *Proc. Second European Workshop on Software Process Technology*, 1992; 167–177.
7. Doppke JC, Heimbigner D, Wolf AL. Software process modeling and execution within virtual environments. *ACM Trans. on Software Engineering and Methodology* 1998; **7**:1–40.
8. Fernstrom C. Process Weaver: Adding process support to Unix. *Proceedings of the 2nd International Conference on the Software Process*; IEEE Press, 1993; 12–26.
9. Iida H, Mimura K, Inoue K, Torii K. Hakoniwa: Monitor and navigation system for cooperative development based on activity sequence model. *Proceedings of the 2nd International Conference on the Software Process*; IEEE Press, 1993; 64–74.
10. Heimann P, Krapp C-A, Westfechtel B. Graph-based software process management. *Int. J. Software Eng. Knowledge Eng.* 1997; **7**:431–455.
11. Holtkamp B, Weber H. Kernel/2r – A software infrastructure for building distributed applications. *Proc. 4th Int. Conf. on Future Trends in Distributed Computing Systems*, Lisbon, September 1993.
12. Huff KE. Probing limits to automation: towards deeper process models. *Proc. 4th Int. Software Process Workshop*, New York, NY, 1988; 79–81.
13. Katayama T. A hierarchical and functional approach to software process description. *Proc. 4th Int. Software Process Workshop*, New York, NY, 1989; 87–92.
14. Perry DE. Policy-directed coordination and cooperation. *Proc. 7th Software Process Workshop*, Yountville, CA, October 1991; 111–113.
15. Perry DE. Enactment control in interact/intermediate. *Proc. 3rd European Workshop on Software Process, EWSPT 94*, Villard de Lans, France, February 1994.
16. Peuschel B, Schafer W. Concepts and implementation of rule-based process engine. *Proc. 14th Int. Conf. Software Engineering*, 1992; 262–279.
17. Sutton Jr. SM, Heimbigner D, Osterweil LJ. APPL/A: A language for software process programming. *ACM Trans. Software Engineering and Methodology* 1995; **4**:221–286.
18. Bandinelli SC, Fuggetta A, Ghezzi C. Software process model evolution in the SPADE environment. *IEEE Trans. Software Engineering* 1993; **19**:1128–1144.
19. Jaccheri ML, Conradi R. Techniques for process model evolution in EPOS. *IEEE Trans. Software Engineering* 1993; **19**:1145–1156.
20. Bandinelli S, Nitto ED, Fuggetta A. Policies and mechanisms to support process evolution in PSEEs. *Proceedings of the Third International Conference on the Software Process*, Los Alamitos, CA, 1994; 9–20.
21. Chou S-C, Chen J-YJ. Process evolution support in concurrent software process language environment. *Information and Software Technology* 1999; **41**:507–524.
22. Ambriola V, Conadi R, Fuggetta A. Assessing process-centered software engineering environments. *ACM Trans. Software Engineering and Methodology* 1997; **6**:283–328.
23. Conradi R, Fernstrom C, Fuggetta A. Concepts for evolving software processes. *Software Process Modeling and Technology*; Research Studies Press: Taunton, 1994; 9–31.
24. Chen J-YJ, Chou S-C. Consistency management in a process environment. *J. Systems and Software* 1999; **47**:105–110.
25. Bandinelli S, Fuggetta A, Grigolli S. Process modeling in the large with SLANG. *Proc. 2nd Int. Conf. Software Process*, Berlin, 1993; 75–83.
26. Bandinelli S, Fuggetta A. Computational reflection in software process modeling: the SLANG approach. *Proceedings of the 15th International Conference on Software Engineering*, 1993; 144–154.
27. Nguyen MN, Wang AI, Conradi R. Total software process model evolution in EPOS experience report. *Proceedings of the 19th International Conference on Software Engineering*, 1997; 390–399.
28. Avrilionis D, Cunin P-Y, Fernstrom C. OPSIS: a view mechanism for software processes which supports their evolution and reuse. *Proceedings of the 18th International Conference on Software Engineering*, 1996; 38–47.
29. Greenwood RM, Warboys BC. Cooperating evolving components – a rigorous approach to evolving large software systems. *Proceeding of the 18th International Conference on Software Engineering*, 1996; 428–437.
30. Cugola G, Di Nitto E, Ghezzi G, Mantione M. How to deal with deviations during process model enactment. *Proceedings of the 17th ICSE*, 1995; 265–273.
31. Cugola G. Tolerating deviations in process support systems via flexiable enactment of process models. *IEEE Trans. Software Engineering* 1998; **24**:982–1001.
32. Chen J-Y, Tu C-M. An Ada-like software process language. *J. System and Software* 1994; **27**:17–25.
33. Chen J-Y, Tu C-M. CSPL: a process-centered environment. *Information and Software Technology* 1994; **36**:3–10.

34. Chen J-YJ, Chou S-C. Enacting object-oriented methods by a process environment. *Information and Software Technology* 1998; **40**:311–325.
35. Basili V, Green S. Software process evolution at the SEL. *IEEE Software* 1994; **11**:58–66.
36. Basili V, Weiss DM. A methodology for collecting valid software engineering data. *IEEE Trans. Software Engineering* 1984; **10**:728–738.
37. Basili V, Rombach HE. The TAME project: towards improvement-oriented software environment. *IEEE Trans. Software Engineering* 1988; **14**:758–773.
38. Kellner MI, Briand L, Over JW. A method for designing, defining, and evolving software processes. *Proceedings of the 4th International Conference on Software Process*, Los Alamitos, CA, 1996; 37–48.
39. Bhandari I, Halliday M, Tarver E, Brown D, Chaar J, Chillarege R. A case study of software process improvement during development. *IEEE Trans. Software Engineering*, 1993; **19**:1157–1170.
40. Basili VR, Rombach HD. Tailoring the software process to project goals and environments. *Proceedings of the 9th International Conference on Software Engineering*, 1987; 345–357.
41. Belkhatir N, Melo WL. Evolving software processes by tailoring the behavior of software objects. *Proceedings of International Conference on Software Maintenance*, 1994; 212–221.
42. Aoyama M. Agile software process and its experience. *Proceedings of the 20th International Conference on Software Engineering*, 1998; 3–12.

