ELSEVIER

Information and Software Technology 41 (1999) 507–524

INFORMATION AND SOFTWARE TECHNOLOGY

Process evolution support in concurrent software process language environment

S.-C. Chou[a], J.-Y.J. Chen[b,*]

[a]*Department of Information Management, Minghsin Institute of Technology, Hsinfong, Taiwan*
[b]*Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan*

Received 2 June 1998; received in revised form 3 February 1999; accepted 4 February 1999

Abstract

This article describes the process evolution support in concurrent software process language (CSPL) environment. Process evolution refers to dynamically changing process programs during process enactment. It is required in process-centered software engineering environment (PSEE) because: (1) parts of a software process may be unclear during process modeling, and (2) processes may change during enactment. In CSPL, process evolution can be achieved through (1) meta-process or (2) process program change. This article also describes object decomposition in CSPL, which relates to process evolution. CSPL allows decomposing large objects (software products) into sub-objects. With this, the schedule, budget, and developer of each sub-object development can be separately assigned and controlled. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Process-centered software engineering environment; Software process; Process evolution

1. Introduction

Software becomes more and more complicated, which in turn complicates software processes (software development processes). To facilitate controlling software processes, process-centered software engineering environments (PSEEs) have been developed [1–15].

A PSEE manages software process components such as activities, developers, tools, objects (software products), exceptions, and so on. It is generally composed of a process language to represent the processes, a process interpreter to enact (execute) the processes, and an object management system to manage the objects. A PSEE called concurrent software process language (CSPL) environment has been developed in our laboratory [3,16,17]. It provides an Ada95-like process language called CSPL. Processes are modeled in CSPL *process programs* that can be enacted in the CSPL environment. In the past years, we have applied CSPL to model and enact object-oriented methods [18]. From the past experiences, the following problems are identified:

1. Some parts of a software process may be difficult to define during project planning. For example, the testing strategy (e.g. black box or white box strategy) may be unclear during project planning. Therefore, the testing process is difficult to define at that stage.
2. A process may change during enactment. For example, suppose a specification is originally planned to be formally reviewed. However, when the review starts, customers have trouble with understanding the specification. The analysts, thus, decide to apply rapid prototyping technique such as that in [19] instead of reviewing the specification. This decision results in changing the review process.
3. The activity for developing a large object may take a long time. For example, implementing an operating system may take years. In software development, large objects are often decomposed into sub-objects, each developed by a developer. With this, the developer, schedule, and budget of each sub-object can be separately defined and controlled. Moreover, the sequence of sub-object development can be explicitly defined and controlled. For example, in a bottom-up unit test process, program modules at the bottom of a program structure should be tested first. Using a PSEE that does not model object decomposition may cause difficulties in controlling developers, schedules, budgets, and development sequences.

To solve the first two problems, processes should be

* Corresponding author.
E-mail address: jychen@csie.nctu.edu.tw (J.-Y.J. Chen)

0950-5849/99/$ - see front matter © 1999 Elsevier Science B.V. All rights reserved.
PII: S0950-5849(99)00013-0

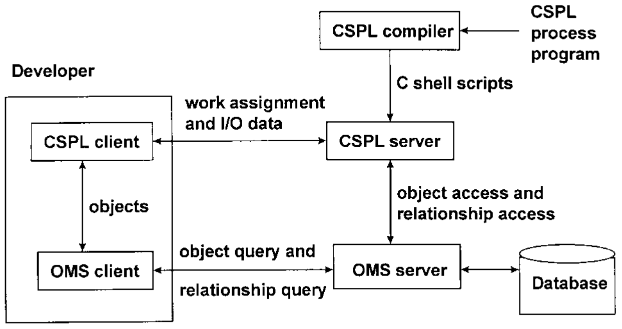

Fig. 1. CSPL environment architecture.

allowed to evolve during enactment. Accordingly, a process language should possess the reflective feature that allows process programs to be dynamically changed during process enactment [20]. To solve the third problem, a PSEE should model object decomposition. However, decomposing objects may be difficult to model during process planning, because developers cannot decide on the decomposition at that time. Therefore, object decomposition should be better modeled through process evolution. For example, a process program originally defines an analysis process only. Having enacted the process, a specification is developed and the decomposition of the system can be modeled. Thus, the process program can be evolved to contain the decomposition and the relevant design process.

The CSPL environment was enhanced to support the process evolution and object decomposition. This article describes these two new CSPL features. Section 2 gives a CSPL overview. Section 3 describes process evolution in CSPL. Section 4 describes object decomposition in CSPL. Section 5 depicts an example. Section 6 discusses related work. Finally, Section 7 gives the conclusions and future work.

2. Concurrent software process language overview

CSPL is a Unix-based PSEE with client/server architecture. It is composed of the following components: (1) the CSPL language, (2) a CSPL compiler, (3) a CSPL server, (4) an object management system (OMS) server, (5) multiple CSPL clients, and (6) multiple OMS clients. Fig. 1 shows

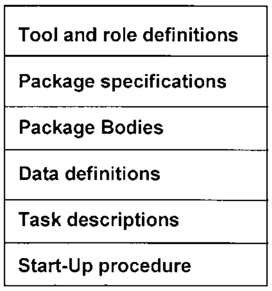

Fig. 2. CSPL process program structure.

the CSPL architecture. Among the components, a CSPL client and an OMS client constitute a developer (client) site. The CSPL compiler, CSPL server, and OMS server constitute the server site.

The CSPL language is used to write process programs. The CSPL compiler translates process programs into C shell scripts, which can be enacted in the CSPL server. During process enactment, CSPL server assigns work to developers by interacting with CSPL clients, and accesses objects and object relationships via the OMS server. Moreover, a CSPL client accesses objects (e.g. browses objects) by invoking an OMS client.

The CSPL language is an Ada95-like procedural language. It provides constructs to specify software process components including roles, tools, objects, object relationships, packages, tasks, task communication, exceptions, and so on. For encapsulation purpose, packages are composed of their specifications and bodies. The general structure of a CSPL process program is shown in Fig. 2.

Example 1. Tool and role definitions.

```
– tool definition
tool ToolSet is
  editor := ''vi'';
  AnalysisTool := ''ROSE'';
  ReviewTool := ''RTool'';
end;
– role definition
role analyst is
  analyst1 := ''scchou'';
  analyst2 := ''syjan'';
end;
```

Tools and *roles* are defined at the beginning of a CSPL process program. With tool definitions, the intended tools can be bound and invoked during enactment. For instance, in Example 1, ‘‘vi’’ will be used as an editor. With role definitions, work can be assigned to right developers. For instance, in Example 1, analysis work will be assigned to ‘‘scchou’’ and ‘‘syjan’’. Note that tools and roles defined here can be used in the entire process program.

Example 2. Package specification.

```
– Package specification
package CAI_analysis is
  – object type definition
  type Specification is new DocType with record
```

```
    LanguageSpec: TextType;
    Notation: NonTextType;
    end record;
    procedure GenSpec(CAI_requirement: in Requirement,
    CAI_specification: in out Specification);
    function ReviewSpec(CAI_requirement: in Requirement,
    CAI_specification: in Specification) return integer;
  end CAI_analysis
```

Package specifications define object types and operations, in which two kinds of operations, namely procedures and functions, can be defined. Regarding object types, CSPL provides built-in ones, such as ‘‘DocType’’, ‘‘TextType’’, ‘‘NonTextType’’, and so on. CSPL allows new object types to be defined by inheriting existing ones. Example 2 is a package specification defining the object type ‘‘Specification’’, the procedure ‘‘GenSpec’’, and the function ‘‘ReviewSpec’’. The object type ‘‘Specification’’ inherits the built-in type ‘‘DocType’’ and adds two new attributes. Object types defined in a package specification can be instantiated. For example, the following statement instantiates an object from the type ‘‘Specification’’, where the object name is ‘‘CAI_specification’’ and the object is stored in the file ‘‘CAI_specification.doc’’:

```
  CAI_specification: Specification := ‘‘ CAI_specification.doc’’
```

Example 3. Package body.

```
  – Package body
  package body CAI_analysis is
    procedure GenSpec(CAI_requirement: in Requirement,
    CAI_specification: in out Specification) is
    begin
    2 analyst edit CAI_specification referring to
    CAI_requirement using AnalysisTool;
    end;
    function ReviewSpec(CAI_requirement: in Requirement,
    CAI_specification: in Specification) is
    begin
    review_result: integer;
    all reviewer review CAI_specification referring to
    CAI_requirement using ReviewTool resulted in
    review_result;
    return review_result;
    end;
  end CAI_analysis
```

Package bodies specify the details of procedures and functions (see Example 3). The most important statement used in a package body is the ‘‘edit’’ statement. It assigns work to developers, binds tools, indicates object(s) for reference, and requires the developers to create an object. An example ‘‘edit’’ statement is as follows:

```
  2 analyst edit CAI_specification referring to CAI_requirement using AnalysisTool;
```

this statement assigns the analysis work to two analysts. The bound tool is ‘‘AnalysisTool’’. The object for reference is ‘‘CAI_requirement’’. And the object to be created is ‘‘CAI_specification’’.

The ‘‘Data definitions’’ section (see Fig. 2) defines the data used in the entire process program. Various data types such as ‘‘integer’’ can be used. Moreover, object types defined in package specifications can also be used as data types. That is, objects can be defined by instantiating object types. The instantiation relationships between object types and objects are then managed by the OMS server.

Example 4. CSPL task.

```
  task body CAI_RequirementAnalysis is
  begin
    loop
    accept start;
    CAI_analysis.GenSpec(CAI_requirement, CAI_specification);
    review_result :=
    CAI_analysis.ReviewSpec(CAI_requirement, CAI_specification);
    If review_result = 1 then
    CAI_analysis.PlanEnactDesign(DesignProcess);
    end if;
    end loop;
  end;
```

A CSPL *task* groups related activities that are enacted

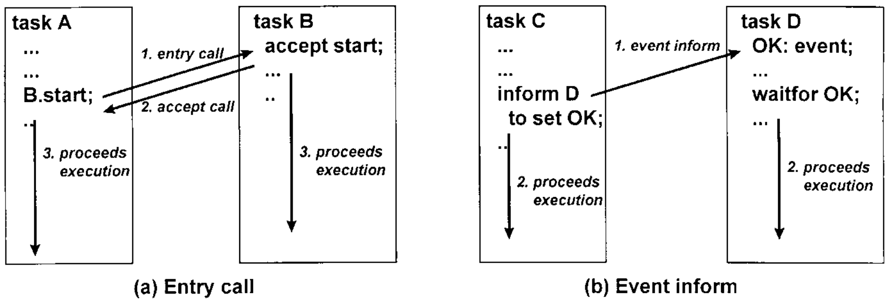

Fig. 3. Entry call and event inform.

sequentially, in which activities constitute a process. Tasks, though, can be enacted concurrently. Therefore, activities that can be enacted in parallel should be assigned to different tasks. Example 4 depicts a CSPL task.

During the enactment of concurrent tasks, communication among them is needed. For example, analysis task should be completed before design task starts. The tasks should thus communicate to synchronize their execution. CSPL provides *entry call* and *event inform* for synchronous and asynchronous communications, respectively (see Fig. 3). In Fig. 3(a), task ‘‘A’’ synchronously communicates with task ‘‘B’’ by the statement ‘‘B.start’’. Task ‘‘A’’ can proceed only after the entry call is accepted. In Fig. 3(b), task ‘‘C’’ asynchronously communicates with task ‘‘D’’ by the statement ‘‘inform D to set OK’’, where ‘‘OK’’ is an event defined in task ‘‘D’’. In this case, task ‘‘C’’ proceeds without waiting for task ‘‘D’’.

Exceptions and their handlers are used to model activities that cannot be regularly controlled. For example, requirement change may occur at any time during software development. It, thus, cannot be controlled regularly and should be modeled as an exception. The following statements depict the handler of the exception ‘‘RequirementChange’’:

```
exception
  when RequirementChange ⇒
  output ‘‘Requirement change, redo the analysis work!!’’;
  Analysis.start;
```

When the exception occurs, CSPL environment outputs ‘‘Requirement change, redo the analysis work!!’’, and then re-starts the analysis work.

During process enactment, CSPL environment first creates objects according to the declarations in the ‘‘Data definitions’’ section of a process program. Object creation is accomplished through instantiating object types. Next, CSPL lists the process program’s exceptions on the screen of the project manager, who can raise them if necessary. Then, CSPL enacts the process program’s tasks, which will assign work to developers and bind tools to objects. During enactment, CSPL keeps process program states in its OMS server. The states are primarily used for process evolution, details of which will be described in Section 3.2.1.

Unlike the languages with late binding feature [6,21–22], binding in CSPL is completed in compilation time. Therefore, CSPL does not support the concepts of process types and instances [6,21]. That is, only one copy of a process program exists during process enactment. This characteristic affects process evolution support in CSPL, as described in Section 3.2.

3. Process evolution in concurrent software process language

Process evolution in CSPL is described in this section. First, the requirement of a simplified computer aided instruction (CAI) system, which is used as an example throughout this article, is described below:

> The CAI system plays the role of a teacher. Class notes and their associated review questions and answers are stored in a database. The class notes are for teaching purposes. The questions and answers test whether a student understands the lesson he/she studied. If he/she passes the test, he/she can proceed to the next lesson. Otherwise, he/she should go back to the previous lesson(s).
>
> The system provides interface for storing new class notes and their associated review questions and answers. In addition, the interface allows teachers to update testing rules.

Bearing this requirement in mind, let us start describing process evolution, which can be classified as predictable or unpredictable as described bellow:

1. *Predictable evolution.* As described before, some parts of a software process may be unclear during project planning. Usually the parts will become clear during process enactment. For example, the testing strategy may be unclear during project planning and will become clear after programs are implemented. To write a process program, the unclear parts are difficult to specify correctly. Therefore, the process program is predicted to evolve during enactment.

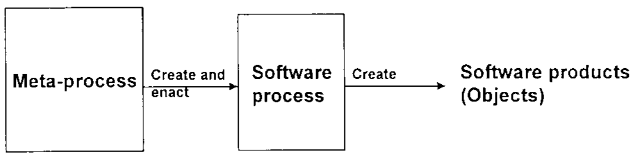

Fig. 4. Relationship between meta-process and software process.

2. *Unpredictable evolution.* An enacting process program may evolve according to exceptions. For example, schedule or budget overrun may result in exceptions which trigger process evolution. This kind of evolution is dynamically caused by developers during process enactment. Therefore, it is regarded as unpredictable.

In CSPL, predictable process evolution can be accomplished through meta-processes, which are processes that create and enact other processes. That is, when writing a process program, meta-processes are used to model the unclear parts. When these parts become clear, the meta-processes are enacted to create process programs for the parts. According to the preceding description, process evolution using meta-processes will not change the enacting process programs. It, however, will create and enact new process programs. Regarding the unpredictable process evolution, it can be accomplished through process program change, in which enacting process programs will be changed if necessary. In the following subsections, process evolution through meta-processes and that through process program change are respectively described.

3.1. Evolution through meta-process

At stated before, a meta-process is a process that creates and enacts processes. The relationship between a meta-process and the software process it creates is shown in Fig. 4. If necessary, multiple levels of meta-processes can be used as shown in Fig. 5. Also, levels of meta-processes can be extended to be a tree-like structure (see Fig. 6).

Example 5. Meta-process example.

```
– Tool definition
tool ToolSet is
  editor := ‘‘vi’’;
end;
– Role definition
role ProcessProgrammer is
  ProcessProgrammer1 := ‘‘scchou’’;
end;
– Data definition
AnalysisProcess: TextType := ‘‘analysis.cspl’’;
DesignProcess:TextType := ‘‘design.cspl’’;
…
task meta_waterfall is
  entry start;
end;
task body meta_waterfall is
begin
  loop
  accept start;
  ProcessProgrammer1 edit AnalysisProcess using
  editor;
  enact AnalysisProcess;
  ProcessProgrammer1 edit DesignProcess using editor;
  enact DesignProcess;
  …
  end loop;
end;
– Start the process
begin
  meta_waterfall.start;
end;
```

CSPL meta-processes are defined using the statements

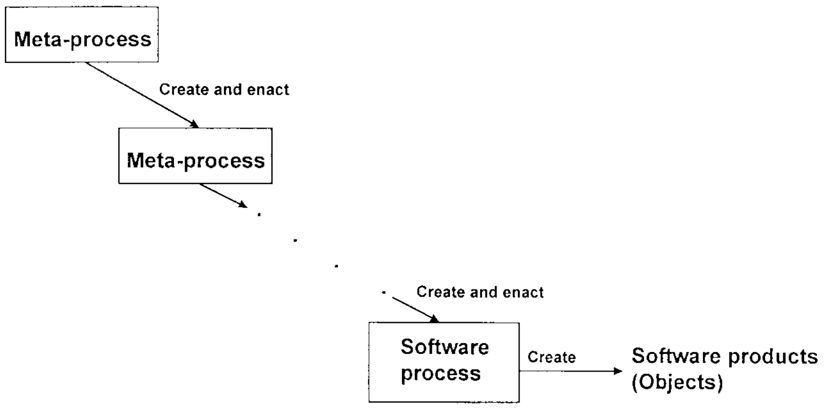

Fig. 5. Multiple levels of meta-processes.

''edit'' and ''enact''. The statement ''edit'' requires a developer to create a process program. The statement ''enact'' then informs CSPL environment to enact the newly created process program. In that enactment, the CSPL compiler first checks the syntax of the process program. If syntax errors occur, the environment will require a developer to correct the errors. If semantic errors are identified during enactment, the project manager should raise an exception to change the process program (see Section 3.2 for details about process program change). Example 4 shows a meta-process that creates and enacts the (partial) waterfall software process model. The meta-process task ''meta_waterfall'' in Example 5 creates and enacts an analysis process and other processes in sequence.

Example 6. Mixing software process with meta-process.

```
– Tool definition
tool ToolSet is
  editor := ''vi'';
  AnalysisTool := ''ATool'';
  ReviewTool := ''RTool'';
end;
– Role definition
role ProcessProgrammer is
  ProcessProgrammer1 := ''scchou'';
end;
role analyst is
  analyst1 := ''scchou'';
  analyst2 := ''syjan'';
end;
– Data definition
Requirement: TextType := ''requirement.doc'';
Specification: TextType := ''specification.doc'';
DesignProcess: TextType := ''design.cspl'';
task mix_process is
  entry start;
end;
task body mix_process is
begin
  loop
  accept start;
  all analyst edit specification using AnalysisTool refer-
  ring to requirement;
  all analyst review specification using ReviewTool;
  ProcessProgrammer1 edit DesignProcess using editor;
  enact DesignProcess;
  end loop;
end;
– Start the process
begin
  mix_process.start;
end;
```

A process program can include meta-processes only (as shown in Example 5) or mix software processes with meta-processes (as shown in Example 6). The task ''mix_process'' in Example 6 mixes an analysis process with a meta-process. The analysis process includes two activities: ''edit specification'' and ''review specification'', while the meta-process creates and enacts the process ''DesignProcess''. Mixing software processes with meta-processes increases CSPL flexibility. With meta-process support, process programs can be evolved according to the following guidelines:

1. Use normal software processes to model well-known processes.
2. Use meta-processes to model processes with high uncertainty, such as processes that are currently unclear, those that are changeable, those that will be enacted a long time later, and so on.

During process enactment, process programs and those they create are enacted simultaneously. For example, suppose process program ''A'' creates and enacts process program ''B''. Then, ''A'' and ''B'' will be enacted simultaneously. This reflects the needs in software development as described in the following:

1. Tasks defined in both the creating and the created process programs should execute concurrently. For example, when analysis tasks are partially completed, the design process can be created and enacted by a meta-process. The relationship between the creating and the created process programs is shown in Fig. 7. Here, the design tasks in the created process program and other analysis tasks in the creating process program are enacted concurrently.
2. Tasks in the created process programs may invoke tasks in the creating process programs. For example, in Fig. 8, when the design tasks in the created process program are enacting, an exception for requirement change may occur. Under this situation, analysis tasks in the creating process program will be invoked to re-analyze system requirements.

The CAI example is used here to explain evolution through meta-process. Fig. 9 depicts the CAI system development process. During process planning, developers have no idea about how the CAI system will be decomposed, or about which modules should be implemented. Therefore, only the analysis process (i.e. Step1 in Fig. 9) can be

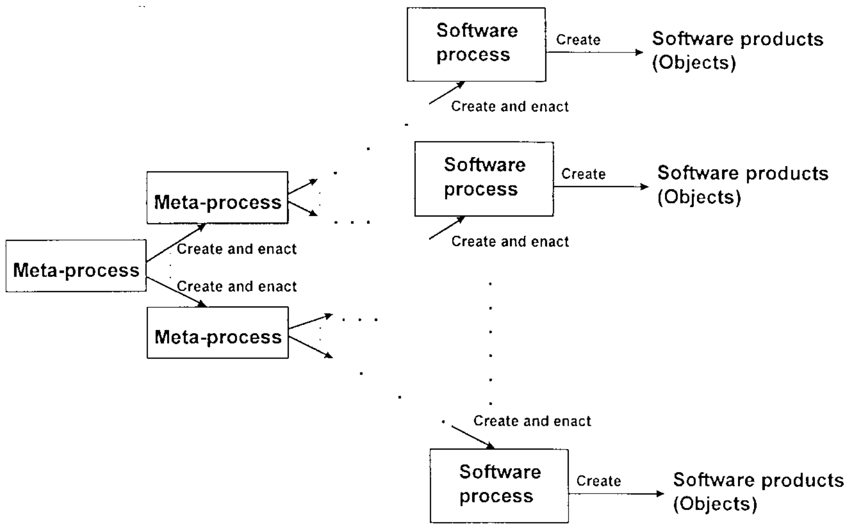

Fig. 6. Tree-like meta-processes.

planned. Other processes should be planned after the analysis task is completed. Meta-processes can be used here. The process program template will look like that in Fig. 10 (see Appendix A for the process program).

After the analysis process, the CAI system specification was created and reviewed. Suppose the analysts decompose the system specification into three subsystem specifications: (1) ‘‘UI’’ for user interface, (2) ‘‘Reasoning’’ for checking whether a student passes a test, and (3) ‘‘DBAccess’’ for database access. Based on this decomposition, the design process is planned. At this point, however, the implementation and test process cannot be planned, because the software modules were not yet designed. Therefore, the process program created by the meta-process in Fig. 10 will look like the one shown in Fig. 11 (see Appendix C for the process program). After the design process, the design document was produced and reviewed. Suppose the CAI system architecture is designed as shown in Fig. 12, where:

1. ‘‘UI’’ is for user interface,
2. ‘‘Storing’’ is for storing class notes, questions, answers, and testing rules,
3. ‘‘Retrieving Course’’ is for retrieving class notes and questions,
4. ‘‘Retrieving Answers’’ is for retrieving answers for the questions,
5. ‘‘Grading’’ is for grading a test,
6. ‘‘Retrieving Rules’’ is for retrieving testing rules,
7. ‘‘Reasoning’’ is for checking whether a student passes a test, and
8. ‘‘DBI’’ is the interface to the database management system.

Based on the architecture, the implementation and test process can be planned.

3.2. Evolution through process program change

Process evolution through meta-process depends on the assumption: *the enacting process programs never change*. However, this is not always true. For example, suppose that in the analysis process of the CAI system, the specification is originally planned to be formally reviewed. However, when the review starts, customers have trouble in understanding the specification. The analysts thus decide to use a rapid prototyping technique for the review. The process program containing the analysis process should thus be changed.

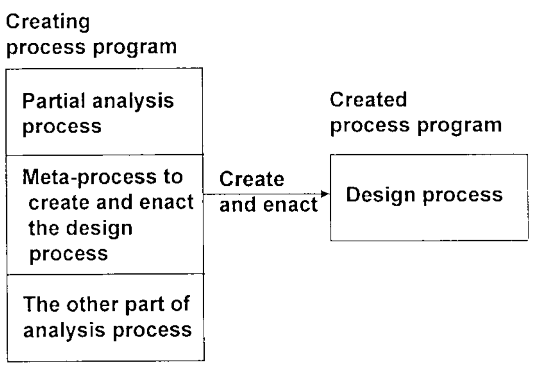

Fig. 7. Concurrent enactment of creating and created process programs.

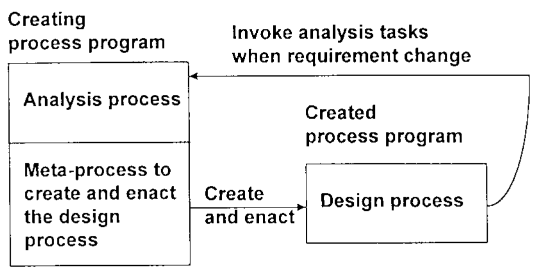
Creating process program
Invoke analysis tasks when requirement change
Analysis process
Created process program
Meta-process to create and enact the design process
Create and enact
Design process

Fig. 8. Invoking tasks in the creating process program.

Step1: Two analysts analyze system requirements, define its specification, and review the specification.
Step2: The analysts decompose the system specification into subsystem specifications.
Step3: Each designer designs one CAI subsystem. The designers work concurrently.
Step4: The designers combine subsystem design documents into the CAI system design document. They then review the design document.
Step5: The programmers implement the CAI system. Each implements some program modules. The programmers work concurrently.
Step6: The testers test the CAI system.

Fig. 9. CAI system development process.

CSPL allows changing process programs during enactment. It uses exceptions to control the change. It automatically creates an exception for each enacting process program, which can be raised by the project manager. Raising the exception allows changing the process program and resuming it later. In managing process program change, the following features should be defined: (1) change policy [23], (2) process program state, and (3) impact analysis and handling [24]. These features are closely related to the following CSPL characteristics:

1. Owing to the constraints of language paradigm, CSPL does not support late binding [6,21,22]. That is, concepts of process types and instances [6,21] are not utilized in CSPL.
2. The name scope of CSPL process program seems rather global. For example, tools and roles defined in the beginning of a process program can be used all over the program. And, data defined in the ‘‘Data definitions’’ section (see Fig. 2) can be used by all tasks in the entire process program. Changing a process program with global name scope appears to be more difficult than changing one with localized name scope.

According to the characteristics, the aforementioned three features are managed as described in the following:

1. Change policy. A change policy may be eager, lazy, or somewhere in between [23]. In an eager policy, the effects of change are handled immediately. However, a lazy policy delays the handling. A lazy policy can be used in PSEEs that provide late binding features [6,21,22], in which processes are defined as types for instantiation. In these PSEEs, changing a process type and then instantiating it will not affect the enacting instances of that type [6,21]. CSPL, however, does not support late binding. That is, only one copy of a process program exists during enactment. Therefore, changing a part of an enacting process program will immediately affect the entire program. CSPL thus uses an eager policy for process program change.
2. Process program state. According to global name scope, changing a part of a CSPL process program is expected to affect the entire program. The change should thus be under careful control. CSPL controls the change by managing process program states, which primarily show: (a) enactment status of process programs, and (b) products that have been created.
3. Impact analysis and handling. Before changing a process program, some activities in the program may have been completed and some objects created. What activities and objects will be affected by the change? Which activities should be enacted after the program is resumed? Which objects should be changed or discarded? All these questions relate to impacts caused by the change, which should be analyzed and handled before resuming the program. CSPL utilizes process states for this purpose.

CSPL keeps process program states during process enactment. When an exception for changing a process program occurs, CSPL immediately interrupts the program for change. CSPL then analyzes and handles change impacts using the program’s state. After the change, CSPL resumes the process program. In the following subsections, (1) process program state, (2) impact analysis and handling, and (3) process program resumption are described.

3.2.1. Process program state

Process program states are primarily used in managing process evolution. To decide what should be included in a process program state, the followings are taken into consideration:

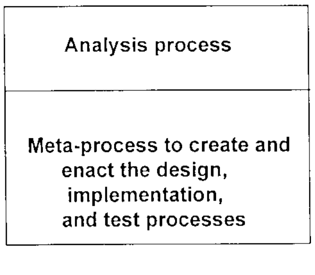
Analysis process
Meta-process to create and enact the design, implementation, and test processes

Fig. 10. Process program template for the CAI system development.

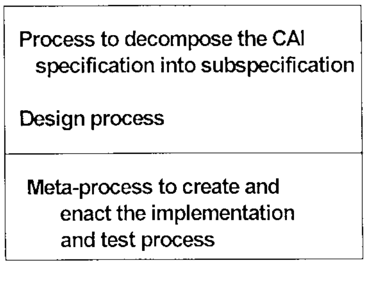
Process to decompose the CAI specification into subspecification
Design process
Meta-process to create and enact the implementation and test process

Fig. 11. Process program created by the meta-processes in Fig. 10.

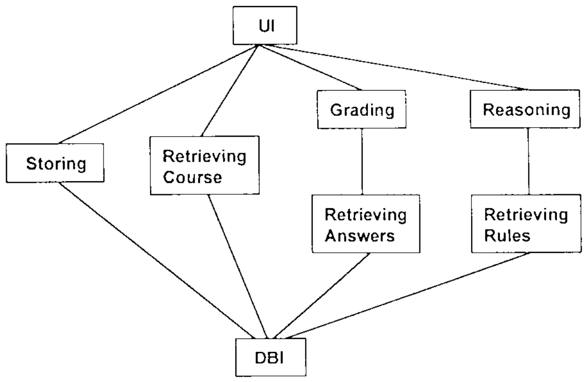

Fig. 12. CAI system architecture.

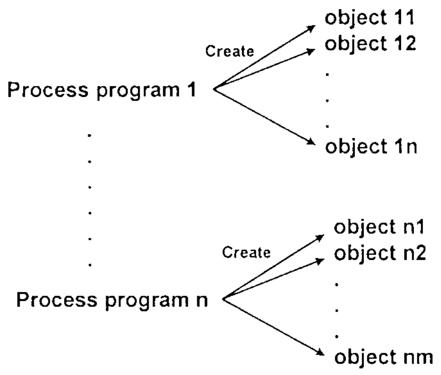

Fig. 14. Relationship between process programs and objects.

1. A changed process program may have created and enacted other process programs through meta-processes. Process change is expected to affect those directly or indirectly created, which may need to be killed, changed, or enacted. A process state should facilitate managing the affected process programs when a process program is changed.
2. Before changing a process program, some activities in the program may have been completed and some objects created. After resumption, some completed activities should be redone, but not the others. Moreover, some created objects should be discarded, some reused, and some others changed. A process state should facilitate deciding which activities should be enacted and which objects should be discarded or changed after process resumption.

With these considerations, CSPL keeps in its OMS server the following as process program state: (1) *process program enactment trees* to manage the affected process programs, and (2) relationships between process programs and objects to manage objects created before process change. Moreover, CSPL language provides *resume blocks* to indicate which activities should be enacted after process resumption. Process program enactment tress and relationships between process programs and objects are respectively described in the following. The resume block will be described in Section 3.2.3.

1. Process program enactment tree. A process program enactment tree shows the creating and enacting relationships between process programs. For example, Fig. 13 shows that ‘‘Process program A’’ creates and enacts ‘‘Process program A1’’ and ‘‘Process program A2’’. Moreover, ‘‘Process program A1’’ creates and enacts ‘‘Process program A11’’ and ‘‘Process program A12’’. A process program enactment tree can be traced to identify the affected process programs when a process program is changed. The affected can then be reported to CSPL users, who should decide which of the affected must be killed, changed, or kept enacting.
2. Relationship between process program and object. Relationships between process programs and objects indicate which process program creates which objects. For example, Fig. 14 shows that ‘‘Process program1’’ creates ‘‘object11’’, ‘‘object12’’, ‘‘object1*n*’’, and so on. Moreover, ‘‘Process program*n*’’ creates ‘‘object*n*1’’, ‘‘object*n*2’’, ‘‘object*nm*’’, and so on. The relationships can be traced to identify objects created by a changed process program. The objects can then be reported to

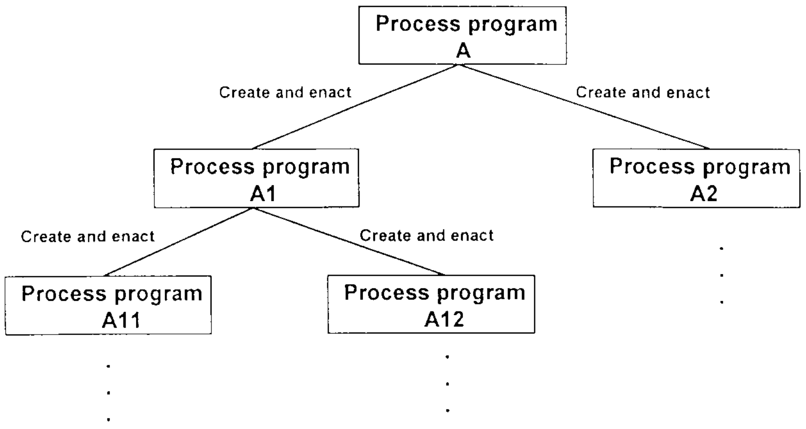

Fig. 13. Process program enactment tree.

CSPL users, who should decide which objects should be reused, changed, or discarded.

3.2.2. Impact analysis and handling

Changing a process program will affect: (1) the process programs created, (2) the objects created, and (3) the activities in the process program. To analyze the change impact, CSPL identifies (1) the affected process programs by tracing process program enactment trees (see Fig. 13), and (2) the affected objects by tracing relationships between process programs and objects (see Fig. 14). CSPL then reports the affected process programs and objects to CSPL users. The users then decide the handling of the affected process programs and objects. Regarding activities in the changed process program, the CSPL users should decide which must be enacted after process resumption by using CSPL resume block, as described in Section 3.2.3.

In handling the affected objects, CSPL removes those that should be discarded and informs the corresponding developers to change those that should be changed. In handling an affected process program, if it should be changed accordingly, CSPL initiates the change by raising an exception. If an affected process program should be killed, CSPL stops its enactment and removes the objects it created.

3.2.3. Process program resumption

After change, the changed process program can be resumed. However, as some activities may have been completed before the change, a mechanism should be available to indicate which activities should be enacted after the resumption. CSPL language provides *resume block* for this purpose. As shown in Example 7, 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.

Example 7. Process program with resume block.

```
task body CAI_RequirementAnalysis is
begin
  loop
  accept start;
  CAI_analysis.GenSpec(CAI_requirement, CAI_specification);
  – Statements inside resume block will be enacted by the ‘‘resume’’ command
  resume block is
  review_result := CAI_analysis.ReviewSpec(CAI_requirement,CAI_specification);
  if review_result = 1 then
  CAI_analysis.PlanEnactDesign(DesignProcess);
  end if;
    end resume block;
    end loop;
  end;
  – Begin of the process
  begin
    resume block is
    CAI_RequirementAnalysis.start;
    end resume block;
  end;
```

4. Object decomposition in CSPL

An activity to develop a large object could take a long time. To well control large object development, the decomposition of object and the assignment of sub-objects to developers should be explicitly modeled in a process program. Decomposing large objects has the following advantages:

1. The schedule and budget of each sub-object development can be separately assigned and controlled. This facilitates project monitoring and control.
2. Clearly defining responsibilities between developers and sub-objects facilitates document maintenance. For example, suppose that the sub-objects of an object are explicitly assigned to three developers. Then, if the object should be changed, each developer will be informed to change the sub-object for which he/she is responsible.
3. The sequence for developing sub-objects can be defined and controlled. For example (see Fig. 12), to test the CAI system with a bottom-up testing procedure, the module ‘‘DBI’’ should be tested before others.

Example 8. Structure block

```
structure is
  CAI_specification subsume UI_specification, DBAccess_specification,
  Reasoning_specification;
end structure;
```

To model object decomposition, CSPL language provides *structure block*, which starts and ends with the statements ‘‘structure is’’ and ‘‘end structure’’, respectively. Inside the block, the ‘‘subsume’’ statement is used to define the decomposition relationships. For example, the structure block in Example 8 decomposes the CAI system specification into three subspecifications: (1) ''UI_specification'', (2) ''DBAccess_specification'', and (3) ''Reasoning_specification''.

As object decomposition may be unclear during process modeling, it is better modeled through process evolution. For example, while modeling the CAI system development process, the developers have no idea about how to decompose the system. Therefore, the original process program is composed of an analysis process and a meta-process, which looks like Fig. 10. After the analysis, the CAI system specification was defined and the system can be decomposed. The process program can thus be evolved to contain the decomposition and design processes, which may look like Fig. 11.

5. The CAI system example

In this section, the CAI system example is used to show process evolution and the corresponding object decomposition. During project planning, the developers cannot decide on the decomposition of the system. Therefore, only an analysis process, which includes the activities to define and review specification, is planned. The design process will be created and enacted by a meta-process after analysis. The process program containing the analysis process and the meta-process is shown in Appendix A. Note that in order to improve understandability of the appendix, parts irrelevant to process evolution are replaced by ''…''.

The process program in Appendix A is then enacted. Suppose that after the CAI specification has been created and the review starts, the reviewers decide to change the specification review strategy from formal review to a rapid prototyping technique. The process program should thus be changed. Appendix B shows the process program after change, in which the function ''ReviewSpec'' is changed. In the resume blocks near the end of Appendix B, one can see that the review activity and the meta-process for creating the design process should be enacted after process resumption.

After the specification is reviewed, decomposition of the CAI system can be determined. Therefore, the design process can be created through a meta-process. Appendix C shows the process program containing the design process, in which the CAI system specification is decomposed into these three subspecifications: (1) ''UI_specification'', (2) ''DBAccess_specification'', and (3) ''Reasoning_specification''. The subspecifications are then designed by three concurrent tasks. Having finished the design, subdesign documents are combined and reviewed. Then, according to the system architecture as shown in Fig. 12, the implementation and test process can be created and enacted using a meta-process. As the creation of implementation and test process is similar to that of the design process, it is not described here.

6. Related work

The need for process evolution is widely recognized [23,25]. Researches of process evolution can be roughly classified into the following two categories: (1) those providing techniques to collect and analyze data, and then evolve a process according to the analysis results, and (2) those providing techniques to support process evolution during process enactment in a PSEE. Some related works in the two categories are briefly discussed in the ensuing paragraphs.

6.1. Techniques that collect and analyze data, and then evolve processes

The quality improvement paradigm (QIP) technique [26] is an evolutionary concept for learning and improvement. It utilizes the goal/question/metric (GQM) [27–28] paradigm to set project goals and define metrics. During process execution, metric data are collected and analyzed. The analysis results are then used for process improvement. And, the experiences obtained are packaged for future projects.

In the helical model [29], an existing process is first evaluated by: (a) subjective and/or qualitative information, and (b) objective measurement and feedback data. The evaluation results are then used to develop the desired process, which is then executed and monitored. The helical model can be recursively applied to a process for continuing improvement.

In the method by Bhandari et al. [30], the defect-based process improvement is used. The method is composed of these activities: (a) defect classification, and (b) defect analysis and feedback. The feedback can then be used to improve processes.

The improvement method by Basilli et al. [31] is also based on defect analysis. It is composed of the steps to: (a) characterize approaches/environments, (b) set up goals, questions, and data for successful project development and improve over previous project development, (c) choose appropriate methods and tools for the project, (d) perform software development, collect and validate data, and feedback, and (e) analyze data for process improvement.

6.2. Techniques to support process evolution during process enactment in a PSEE

SPADE [22] provides a reflective, Petri-net based process language SLANG [32–33]. During process enactment, a process model is enacted by multiple process engines, in which an engine enacts an activity. Before the enactment of an activity, an active copy should be prepared based on the activity and the object types it uses. This allows late binding between activity definitions and invocations. Process evolution can thus be accomplished by changing active copies of activities. The change is normally defined as a meta-process.

EPOS [24,34] provides a rule-based process language SPELL [4]. During process enactment, meta-classes, classes, and instances of objects and process models are stored in a database called EPOSDB [24]. Contents in EPOSDB are all subject to change. Process evolution can be accomplished by changing meta-classes, classes, or instances of process models. The process model (PM) manager manages process change. When a change request occurs, the PM manager checks whether the change is allowed. If so, the PM manager analyzes the impacts of the change. It then interacts with users to solve the problems resulting from the impacts.

OPSIS [35] decomposes a process model into various views such as role, product, and so on. It provides techniques to: (1) extract views from a process model, and (2) compose views. In OPSIS, process evolution focuses on a set of views instead of on the whole process model. The authors believed that expressing process evolution on a set of views is more accurate and convenient than on a single huge process model.

In Process Weaver [6], process models can be instantiated as instances for enactment. Through the feature of late binding, a process model can be changed dynamically during enactment. Therefore, process models need not be completely defined during enactment. Process Weaver provides a mechanism for dynamic changing the process models being enacted. Through the mechanism, the process instances that are scheduled for future enactment can be changed.

In the Tempo technique [21], processes are modeled as types which can be instantiated as instances for enactment. Process evolution is accomplished by changing a process type, and then instantiating the type. The process instances being enacted will not be affected by the change, which corresponds to a lazy change policy [23]. Tempo also supports process evolution through dynamically changing role during enactment. As different roles perform different activities on the same product, changing role corresponds to changing process.

In the Hdev process of the OBM environment [36], processes operating on a product is defined as the product's methods, where product types are modeled as abstract objects using the object-oriented paradigm. Process evolution is accomplished by changing the methods of abstract objects. Method change is accomplished by invoking the method ‘‘changeself’’. Therefore, the OBM language can be considered a reflective language that facilitates process change.

In the research by Gugola et al. [37–38], the observed process is allowed to diverge from the modeled process. With this, process evolution can be accomplished by tolerating process deviation. The technique models process enactment as state transitions. Each transition is associated with pre-conditions, and each state is associated with constraints. A state transition fulfilling its pre-conditions can fire. This corresponds to a process that does not diverge. A state transition fails to meet its pre-conditions that can also fire. After this firing, if the state constraints are not violated, the state is safe. This corresponds to process deviation that is tolerable. If any constraint is violated, the state is unsafe. This corresponds to process deviation that is not tolerable. On the occurrence of an unsafe state, data (objects) produced should be discarded. This results in an activity ‘‘pollution analysis’’, which is similar to impact analysis.

The Agile software process (ASP) model [39] is based on an incremental-delivery and evolutionary model by which products are incrementally delivered. An ASP model is thus composed of a number of light-weight processes. With this, an ASP model can quickly adapt the change in requirements. The need for process evolution can thus be reduced.

Process evolution support in CSPL belongs to the second category described before. That is, CSPL environment supports process evolution during process enactment. When comparing our technique with techniques in the same category, the followings are obtained:

1. Unlike Process Weaver and Tempo, CSPL process programs cannot be instantiated. That is, only a single copy of a process program is enacted. Therefore, process evolution in CSPL is not accomplished by late binding. Instead, it is accomplished by meta-processes and process change.
2. Process evolution in CSPL takes an eager policy [23]. That is, CSPL immediately propagates the change effects to the affected process programs. This policy is expected to prevent the changed process programs from producing too many incorrect products. Regarding the technique mentioned before, Tempo and Process Weaver seem to take a lazy policy. Others may take a policy somewhere between the eager and lazy policies.
3. CSPL supports both meta-process and process program change for process evolution. Regarding the techniques mentioned before, SPADE support meta-processes. EPOS, OPSIS, Process Weaver, Tempo, and Hdev allow process program change, where EPOS limits the change to those being allowed [24]. Note that process change in Process Weaver and Tempo is accomplished by the late binding feature.
4. After process change, CSPL facilitates: (a) identifying which process programs should be killed or changed accordingly, (b) identifying which products should be modified or even discarded, and (c) indicating which activities should be enacted after process resumption. This corresponds to impact analysis and handling. Among the techniques mentioned before, it seems that only EPOS and the research by Gugola et al. support this analysis and handling.

7. Conclusions and future work

Process evolution refers to dynamically changing process programs during process enactment. It is needed in PSEE because: (1) parts of a software process may be unclear during process modeling, and (2) processes may change during enactment. CSPL environment provides the following two mechanisms for process evolution:

1. Evolution through meta-process. CSPL provides meta-processes to create and enact other process programs. Software processes with high uncertainty, such as those that are currently unclear, those that are changeable, and those that will be enacted a long time later, need not be defined during process modeling. Instead, they can be created and enacted by meta-processes after the uncertainty is resolved.
2. Evolution through process program change. CSPL allows an enacting process program to change. When an exception for changing a process program occurs, CSPL immediately interrupts the program for change. It then identifies the affected process programs and objects by examining the process program's state. In handling the affected objects, CSPL removes those that should be discarded and informs corresponding developers to change those that should be changed. In handling the affected process programs, CSPL raises exceptions to change those that should be changed and stops those that should be killed. It also removes the objects created by the process programs that should be killed. After change, CSPL resumes the process program. CSPL provides resume block to indicate which activities should be enacted after resumption.

CSPL also allows modeling object decomposition. With this, the schedule, budget, and developer for each sub-object can be separately assigned and controlled. Moreover, the sequence for sub-object development can be defined and controlled. As object decomposition of a system may be unclear during process modeling, it is better modeled through process evolution.

We have advised our students to use CSPL in developing their term projects. From their usage, the following experiences about process evolution were obtained:

1. Processes do evolve. Process evolution in the student projects occurred frequently. A process program might evolve according to changing roles or tools, modifying activities or tasks, and so on. The frequent evolution seems to be caused by poor project planning. Therefore, we do not suppose that processes in commercial software projects will evolve frequently. However, we do believe that processes will evolve according to schedule or budget overrun, requirement change, and so on.
2. Meta-processes can be used to develop and enact process programs incrementally. In monitoring student projects, we were surprised that some students did not plan their project. Their strategy was to put the easy-to-identify activities in a process program and used a meta-process for other parts. Having completed the easy-to-identify activities, the meta-process was enacted, in which the students applied the same strategy mentioned earlier to create a process program. The strategy was recursively applied until occasionally the project was finished.
   This kind of usage notified us that process programs can be created and enacted incrementally through meta-processes. This usage is especially valuable in a project with high uncertainty.
3. Process program change requires heavy human engagement. The change and resumption of a process program is accomplished through the interaction of developers and CSPL environment. When a process program is changed, CSPL reports the affected process programs and objects to developers. The developers then direct CSPL to handle those affected. Meanwhile, developers should modify the process program and use resume blocks to indicate CSPL about the enactment of activities after resumption. According to the preceding description, process program change requires heavy human engagement.

In the future, our research is expected to complete the following works:

1. Enhance CSPL to tolerate minor evolution. From the third experience mentioned before, we believe that the frequency of process program change should be reduced. Therefore, in the future we will enhance CSPL to tolerate minor process evolution. For example, the change of tools or roles can be tolerated by CSPL, and therefore no process program change is needed in these cases.
2. Design metrics for process program performance and a technique to collect and analyze data. During process enactment, feedback can be obtained from developers or customers, with which process performance can be evaluated. According to the evaluation, a poor performed process program should be evolved. Therefore, feedback and its evaluation are key factors in effective process evolution. To facilitate the evaluation, in the future we will: (a) design metrics for process program performance, and (b) design a technique to collect and analyze metrics data.
3. Enhance CSPL to facilitate process program change control. Currently, process program change in CSPL is controlled by a project manager. That is, the manager raises exceptions to change process programs according to developer feedback. This, however, may increase the workload of the manager, because he/she should coordinate developers for that change. In the future, we will enhance CSPL to facilitate controlling the change.

Acknowledgements

This research is sponsored by the National Science Council in Taiwan under grant number NSC86-2213-E-009-025.

Appendix A

The CAI system analysis process is listed below:

```
– To improve understandability, parts irrelevant to evolution are replaced by ‘‘…’’.
– tool definition
…
– role definition
…
– Packages specification
package CAI_analysis is
  – Type definitions
  …
  procedure GenSpec(CAI_requirement: in Requirement, CAI_specification: in
  out Specification);
  function ReviewSpec(CAI_requirement: in Requirement, CAI_specification: in
  Specification) return integer;
  procedure PlanEnactDesign(DesignProcess: out TextType);
  – The procedure ‘‘PlanEnactDesign’’ is a meta-process. It plans and enacts
  – the design process.
end CAI_analysis

– Package body
package body CAI_analysis is
  procedure GenSpec(CAI_requirement: in Requirement, CAI_specification: in
  out Specification) is
  begin
  …
  end;
  function ReviewSpec(CAI_requirement: in Requirement, CAI_specification: in
  Specification) is
  begin
  review_result: integer;
  all reviewer review CAI_specification referring to CAI_requirement using
  ReviewTool resulted in review_result;
  return review_result;
  end;
procedure PlanEnactDesign(DesignProcess: out TextType) is
  begin
  1 ProcessProgrammer edit DesignProcess using editor;
  enact DesignProcess;
  end;
end CAI_analysis

with CAI_analysis
procedure StartTask is
– Data definitions
…
– Task specifications
task CAI_RequirementAnalysis is
  entry start;
end;

– Task bodies
task body CAI_RequirementAnalysis is
begin
  loop
  accept start;
  CAI_analysis. GenSpec(CAI_requirement, CAI_specification);
  review_result :=
  CAI_analysis. ReviewSpec(CAI_requirement, CAI_specification);
  if review_result = 1 then
  CAI_analysis.PlanEnactDesign(DesignProcess);
  end if;
  end loop;
end;

– Begin of the process
begin
  CAI_RequirementAnalysis.start;
end;
```

Appendix B

After changing the specification review process, the process program is listed below:

```
– tool definition
…
– role definition
…
```

```
– Packages specification
package CAI_analysis is
  – Type definitions
  …
  type Prototype is new DocType with record
  FormalSpecification: TextType;
  end record;
  – ‘‘Prototype’’ is a new object type
  procedure GenSpec(CAI_requirement: in Requirement,
  CAI_specification: in
  out Specification);
  function ReviewSpec(CAI_requirement: in Requirement, CAI_specification: in
  Specification) return integer;
  procedure PlanEnactDesign(DesignProcess: out TextType);
  – The procedure ‘‘ PlanEnactDesign’’ is a meta-process. It plans and enacts the
  – design process.
end CAI_analysis

– Package body
package body CAI_analysis is
  procedure GenSpec(CAI_requirement: in Requirement,
  CAI_specification: in
  out Specification) is
  begin
  …
  end;

– The function ‘‘ReviewSpec’’ has been changed
function ReviewSpec(CAI_requirement: in Requirement,
CAI_specification: in
  Specification) is
  begin
  review_result: integer;
  CAI_prototype: Prototype := ‘‘CAI_prototype.doc’’;
  1 analyst edit CAI_prototype referring to CAI_specification using
  PrototypeTool;
  all reviewer review CAI_prototype referring to CAI_requirement using
  PrototypeTool resulted in review_result;
  return review_result;
  end;
  procedure PlanEnactDesign(DesignProcess: out TextType) is
  begin
  …
  end;

  end CAI_analysis
with CAI_analysis;
procedure StartTask is
– Data definitions
…
– Task specifications
task CAI_RequirementAnalysis is
  entry start;
end;

– Task bodies
task body CAI_RequirementAnalysis is
begin
  loop
  accept start;
  CAI_analysis.GenSpec(CAI_requirement, CAI_specification);
  – Statements inside resume block will be enacted after process
  – resumption.
  resume block is
  review_result :=
  CAI_analysis.ReviewSpec(CAI_requirement, CAI_specification);
  if review_result = 1 then
  CAI_analysis. PlanEnactDesign(DesignProcess);
  end if;
  end resume block;
  end loop;
end;

– Begin of the process
begin
  resume block is
  CAI_RequirementAnalysis.start;
  end resume block;
end;
```

Appendix C

The design process created by the meta-process in Appendix A is listed below:

```
– tool definition
…
– role definition
…
package CAI_design is
– Type definitions
…
procedure GenDesign(subspecification: in Specification, subdesign: in out
  DesignDocu, specific_developer: in designer, deadline: in time);
procedure CombineDesign(CAI_design: out DesignDocu, UI_design: in
  DesignDocu, DBAccess_design: in DesignDocu, Reasoning_design: in
  DesignDocu);
function ReviewDesign(CAI_specification: in Specification, CAI_design: in
  DesignDocu) return integer;
procedure PlanEnactImplTest(ImplTestProcess: out TextType);
– The meta-process ‘‘PlanEnactImplTest’’ plans and enacts the
– implementation and test process.
end CAI_design;


package body CAI_design is
  procedure GenDesign(subspecification: in Specification, subdesign: in out
  DesignDocu, specific_developer: in designer, deadline: in time) is
  begin
  …
  end;
  procedure CombineDesign(CAI_design: out DesignDocu, UI_design: in
  DesignDocu, DBAccess_design: in DesignDocu, Reasoning_design: in
  DesignDocu) is
  begin
  …
  end;
  function ReviewDesign(CAI_specification: in Specification, CAI_design: in
  DesignDocu) is
  begin
  …
  end;

  procedure PlanEnactImplTest(ImplTestProcess: out TextType) is
  begin
  1 ProcessProgrammer edit ImplTestProcess using editor;
  enact ImplTestProcess;
  end;

end CAI_design;


with CAI_design;
procedure StartTask is
– Data definitions
…
– Definition of object decomposition
structure is
  CAI_specification subsume UI_specification, DBAccess_specification,
  Reasoning_specification;
end structure;


…
task bodyUI_design_task is
begin
  …
end;


task body DBAccess_design_task is
begin
  …
end;


task body Reasoning_design_task is
begin
  …
end;
```

```
task body review_design_task is
UI_design_end: event;
DBAccess_design_end: event;
Reasoning_design_end: event;
begin
  loop
  waitfor UI_design_end and DBAccess_design_end and
  Reasoning_design_end;
  CAI_design.CombineDesign(CAI_design, UI_design,
  DBAccess_design,
  Reasoning_design);
  review_result := CAI_design.ReviewDesign (CAI_spe-
  cification,
  CAI_design);
  if review_result = 1 then
  CAI_design.PlanEnactImplTest(ImplTestProcess);
  end if;
  end loop;
end;

– Begin of the process
begin
  – Concurrently start the three design tasks
  …
end;
```

References

[1] N. Belkhatir, W.L. Melo, Supporting software development process in Adele 2, The Computer Journal 37 (2) (1994) 621–628.
[2] J.Y. Chen, P. Hsia, MDL (Methodology Definition Language): a language for defining and automating software development process, Journal of Computer Language 17 (3) (1992) 199–211.
[3] J.-Y.J. Chen, CSPL: an ada95-like, unix-based process environment, IEEE Transactions on Software Engineering 23 (3) (1997) 171–184.
[4] R. Conradi, et al., Design, use and implementation of SPELL, a language for software process modeling and evolution, in: Proceedings of the Second European Workshop on Software Process Technology, 1992, pp. 167–177.
[5] J.C. Doppke, D. Heimbigner, A.L. Wolf, Software process modeling and execution within virtual environments, ACM Trans. on Software Engineering and Methodology 7 (1) (1998) 1–40.
[6] C. Fernstrom, Process Weaver: adding process support to Unix, in: Proceedings of the Second international conference on the software process, IEEE Computer Society, 1993, pp. 12–26.
[7] H. Iida, K.-I. Mimura, K. Inoue, K. Torii, Hakoniwa: monitor and navigation system for cooperative development based on activity sequence model, in: Proceedings of the Second international conference on the software process, IEEE Computer Society, 1993, pp. 64–74.
[8] P. Heimann, C.-A. Krapp, B. Westfechtel, Graph-based software process management, International Journal of Software Engineering and Knowledge Engineering 7 (4) (1997) 431–455.
[9] B. Holtkamp, H. Weber, Kernel/2r-A software infrastructure for building distributed applications, in: Proceedings of the Fourth International Conference on Future Trends in Distributed Computing Systems, Lisboa, September 1993.
[10] K.E. Huff, Probing limits to automation: towards deeper process models, in: Proceedings of the Fourth International Software Process Workshop, New York, 1988, pp. 79–81.
[11] T. Katayama, A hierarchical, functional approach to software process description, in: Proceedings of the Fourth International Software Process Workshop, New York, 1989, pp. 87–92.
[12] D.E. Perry, Policy-directed coordination and cooperation, in: Proceedings of the Seventh Software Process Workshop, Yountville, CA, 1991, pp. 111–113.
[13] D.E. Perry, Enactment control in interact/intermediate Lecture Notes in Computer Science, 772, Springer Verlag, 1994, in: B.C. Warboys (Ed.), Proceedings of the Third European Workshop on Software Process, EWSPT 94, Villard de Lans, France, 1994 pp. 107–113.
[14] B. Peuschel, W. Schafer, Concepts and implementation of rule-based process engine, in: Proceedings of the Fourteenth International Conference on Software Engineering, 1992, pp. 262–279.
[15] S.M. Sutton Jr, D. Heimbigner, L.J. Osterweil, APPL/A: a language for software process programming, ACM Transaction on Software Engineering and Methodology 4 (3) (1995) 221–286.
[16] J.-Y. Chen, C.-M. Tu, An Ada-like software process language, J. System and Software 27 (1) (1994) 17–25.
[17] J.-Y. Chen, C.-M. Tu, CSPL: a process-centered environment, Information and Software Technology 36 (1) (1994) 3–10.
[18] J.-Y.J. Chen, S.-C. Chou, Enacting object-oriented methods by a process environment, Information and Software Technology 40 (5-6) (1998) 311–325.
[19] A. Luqi, V. Berzins, R.T. Yeh, A prototyping language for real-time software, IEEE Transactions on Software Engineering 14 (10) (1988) 1409–1423.
[20] V. Ambriola, R. Conadi, A. Fuggetta, Assessing process-centered software engineering environments, ACM Trans. on Software Engineering and Methodology 6 (3) (1997) 283–328.
[21] N. Belkhatir, W.L. Melo, Evolving software processes by tailoring the behavior of software objects, in: Proceedings of International Conference on Software Maintenance, 1994, pp. 212–221.
[22] S.C. Bandinelli, A. Fuggetta, C. Ghezzi, Software process model evolution in the SPADE environment, IEEE Transactions on Software Engineering 19 (12) (1993) 1128–1144.
[23] S. Bandinelli, E.D. Nitto, A. Fuggetta, Policies and mechanisms to support process evolution in PSEEs, in: Proceedings of the Third International Conference on the Software Process, Los Alamitos, CA, 1994, pp. 9–20.
[24] M.L. Jaccheri, R. Conradi, Technoques for process model evolution in EPOS, IEEE Transactions on Software Engineering 19 (12) (1993) 1145–1156.
[25] R. Conradi, C. Fernstrom, A. Fuggetta, Concepts for Evolving Software Processes, in Software Process Modeling and Technology, Research Studies, Taunton, Somerset, England, 1994, pp. 9–31.
[26] V. Basili, S. Green, Software process evolution at the SEL, IEEE Software 11 (4) (1994) 58–66.
[27] V.R. Basili, D.M. Weiss, A methodology for collecting valid software engineering data, IEEE Transactions on Software Engineering 10 (6) (1984) 728–738.
[28] V.R. Basili, H.E. Rombach, The TAME project: towards improvement-oriented software environment, IEEE Transactions on Software Engineering 14 (6) (1988) 758–773.
[29] M.I. Kellner, L. Briand, J.W. Over, A method for designing, defining, and evolving software processes, in: Proceedings of the Fourth International Conference on Software Process, Los Alamitos, CA, 1996, pp. 37–48.
[30] I. Bhandari, M. Halliday, E. Tarver, D. Brown, J. Chaar, R. Chillarege, A

case study of software process improvement during development, IEEE Transactions on Software Engineering 19 (12) (1993) 1157–1170.

[31] V.R. Basili, H.D. Rombach, Tailoring the software process to project goals and environments, in: Proceedings of the Ninth ICSE, 1987, pp. 345–357.

[32] S. Bandinelli, A. Fuggetta, S. Grigolli, Process modeling in the large with SLANG, in: Proceedings of the Second International Conference Software Process, Berlin, 1993, pp. 75–83.

[33] S. Bandinelli, A. Fuggetta, Computational reflection in software process modeling: the SLANG approach, in: Proceedings of the Fifteenth ICSE, 1993, pp. 144–154.

[34] M.N. Nguyen, A.I. Wang, R. Conradi, Total software process model evolution in EPOS experience report, in: Proceedings of the Nineteenth ICSE, 1997, pp. 390–399.

[35] D. Avrilionis, P.-Y. Cunin, C. Fernstrom, OPSIS: A view mechanism for software processes which supports their evolution and reuse, in: Proceedings of the Eighteenth ICSE, 1996, pp. 38–47.

[36] R.M. Greenwood, B.C. Warboys, Cooperating evolving components—a rigorous approach to evolving large software systems, in: Proceedings of the Eighteenth ICSE, 1996, pp. 428–437.

[37] G. Gugola, E. Di Nitto, G. Ghezzi, M. Mantione, How to deal with deviations during process model enactment, in: Proceedings of the Seventeenth ICSE, 1995, pp. 265–273.

[38] G. Cugola, Tolerating deviations in process support systems via flexible enactment of process models, IEEE Transaction on Software Engineering 24 (11) (1998) 982–1001.

[39] M. Aoyama, Agile software process and its experience, in: Proceedings of the Twentieth ICSE, 1998, pp. 3–12.