CSPL: An Ada95-Like, Unix-Based Process Environment

Jen-Yen Jason Chen, *Member*, *IEEE*

Abstract—This paper presents a new process-centered environment called "concurrent software process language" (CSPL). CSPL takes a unique and innovative approach to integrate the object-oriented Ada95-like syntax (for its modeling power) with Unix shell semantics (for its enactment capability) in a software process language. This paper depicts the following new CSPL features: 1) object orientation, 2) multirole and multiuser, and 3) unified object modeling. Language constructs specially designed for software process such as work assignment statement, communication-related statements, role unit, tool unit, relation unit and so on, are, respectively, described. The related work of this diversified field is also surveyed in some depth. The CSPL environment prototype has been built. A CSPL process program for the IEEE Software Process Modeling Example Problem has been developed and enacted to demonstrate the capabilities of this environment.

Index Terms—Process-centered environment, software process, software engineering, software engineering environment, process modeling.

1 Introduction

Software development process (software process) has traditionally been modeled using the sequential, phased waterfall process model. This is an over-simplified model because it merely identifies the phases, and the documents and reviews associated with each phase. Actually, a process involves complexities resulting from multiple developers performing activities concurrently to produce a set of related documents. Moreover, the highly iterative nature of software process is not focused upon in this model. With software processes becoming extremely complex, sophisticated and nontraditional approaches are definitely needed by the software industry.

Various approaches to model software processes have been proposed. Each provides a formalism or language with a precise syntax and semantics (see Section 5 for discussions on the approaches). One approach is *process programming language* [24], [25] which represents a software process in a programming language as a *process program*. The process program can be automatically executed (enacted) in a process-centered environment. This automation is expected to enforce development policies, to relieve developers from tedious routine work, and consequently, to reduce potential human errors. It should be noted though that manual activities can only be assisted, rather than automated, by the environment.

Most developers have been using one programming language or another for quite some time. It thus appears natural for them to use this language approach. In regard to modeling software processes in detail, it seems that language approach is better than graph formalism approach such as Petri Net [28]. Moreover, programs in modern programming languages are normally readable and thus understandable. Particularly, languages like Ada [5] can be either programming-in-the-large languages that specify module interface or programming-in-the-small languages that specify module detail. They thus encourage developing modular programs. All these facilitate using the process programming language approach in the development, review, refinement, enactment, and maintenance of complicated software process models.

This research thus adopted Ada95 to design a software process language called concurrent software process language (CSPL). The associated CSPL process-centered environment prototype (CSPL environment) is also thus developed [9], [10]. CSPL takes a unique and innovative approach in integrating the object-oriented Ada95-like syntax (for its modeling power) with Unix shell semantics (for its enactment capability) within a software process language.

This paper depicts the new CSPL developments. It is organized as follows. Section 2 depicts the objectives. Section 3 gives an overview. Section 4 covers the CSPL syntax and semantics. Section 5 depicts related work. Finally, Section 6 draws the conclusions together. And a CSPL process program for the IEEE Software Process Modeling Example Problem is presented in Appendix A.

2 Objectives

The concurrent software process language (CSPL) has been recently enhanced to meet the following objectives to: 1) achieve an object-oriented language, 2) allow multiroles and multiusers systems, and 3) provide unified object modeling.

2.1 Need for an Object-Oriented Language

Recently quite a few object-oriented programming languages have emerged such as C++, Ada95, Java, etc. One important

• *J.-Y. Chen is with the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan. E-mail: jychen@csie.nctu.edu.tw.*

Manuscript received Oct. 5, 1995; revised Mar. 10, 1997.
Recommended for acceptance by D.E. Perry.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95092.

benefit of object orientation is that it leads to constructing well-structured complex systems comprised of classes [6]. Through class inheritance, programs are easier to reuse and are more resilient to change. In the domain of software process, because a software product (a document) can be easily mapped to an object, identifying a class (a document type or object type) appears to be straightforward.

Originally, CSPL is Ada83-like, which is not an object-oriented language. Unlike C++ that adds new syntax such as "class" to C [31], Ada95 simply extends the derived type concept of Ada83 so that it becomes an object-oriented language [3]. Similarly, CSPL is extended to be Ada95-like with the object-oriented features.

Nonprocedural process languages are not object-oriented. Neither are many procedural process languages. In this regard CSPL is rather distinctive. Example 1 shows the CSPL *inheritance feature*. In the example, object type "design_type" is a derived type of its parent type "doc_type". That is, the former inherits attribute "last_modified_time" of the latter and adds a new attribute "descriptor" (see Section 4.1.3).

EXAMPLE 1. (*Object Type Inheritance*)

```
type doc_type is tagged
  record
    last_modified_time: time;
  end record;

type design_type is new doc_type with
  record
    descriptor: string;
  end record;
```

2.2 Need for Multirole and Multiuser Systems

Under certain circumstances, one developer can assume multiple roles to perform several kinds of work. This reduces the number of developers needed in a process. CSPL provides a role unit (see Section 4.4.1) for that purpose. Example 2 depicts this *multirole feature*. In the example, developer "bktseng" can assume an "analyst" role as "analyst2" to analyze a requirement. He can also assume another "designer" role as "designer1" to design the software.

EXAMPLE 2. (*Multirole Feature*)

```
role analyst is
  analyst1 := "cywang";
  analyst2 := "bktseng";
end;
role designer is
  designer1 := "bktseng";
  designer2 := "cywang";
  designer3 := "jychen";
end;
```

Next, CSPL provides a work assignment statement (to be described) to support the *multiuser feature*. The statement assigns "manual work" (to be described) to a role, such as "designer," instead of actual developers. During process enactment, the role is "instantiated" to multiple developers of that role (e.g., three developers "bktseng," "cywang," and "jychen" of role "designer" in Example 2). Work is assigned to those developers through the CSPL communication system.

The CSPL communication system supports the work assignment statement, communication-related statements (to be described), etc. It is implemented as a client-server architecture using SUNOS remote procedural calls. Due to limited space, the system will be discussed elsewhere.

2.3 Need for Unified Object Modeling

A large software system consists of a large number of documents (objects) and the relationships among those documents easily become uncontrollable. Thus, objects and their relationships should be properly modeled.

Object relationship often refers to object dependency relationship. Fig. 1 shows object code "A" is produced by compiling program module "B." Here, "A" is said to be dependent on "B." CSPL uses the relation unit to model such relationship. When a document is modified, a relation unit triggers some activities on its dependent documents.

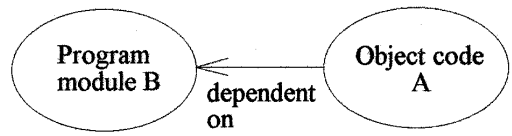

Fig. 1. A "dependent on" relationship.

In addition to modeling object and object relationship, process language needs to manage concurrent, long-duration transaction on object. For example, multiple developers may need to work on a design document concurrently for several weeks. CSPL object management system (OMS) uses a check-in/check-out model to handle this kind of transaction. Briefly, when a developer checks-in an object currently checked out by others, a new version of that object should be created. If this is not the case, the current version is simply overwritten. The created versions will be merged through review later. Due to limited space, CSPL OMS will not be discussed in detail in this paper.

One CSPL significance is that its object modeling capabilities just mentioned are unified with the process language. In other words, the CSPL object management system is fully integrated with the process environment (see Section 3.1).

3 OVERVIEW

To give the readers an overview of this paper, CSPL environment architecture and the ideas behind using Unix shell are, respectively, depicted below.

3.1 Environment Architecture

CSPL environment contains five components: 1) CSPL compiler, 2) CSPL server, 3) CSPL clients, 4) object management system (OMS) server, and 5) OMS clients, as sketched in Fig. 2.

In Fig. 2, CSPL compiler compiles a CSPL program to generate C shell (Unix shell) scripts [1] which are executed by CSPL server running Unix. Specifically, each Ada-like program unit in the CSPL program (such as subprogram and task) maps to one C shell script generated.

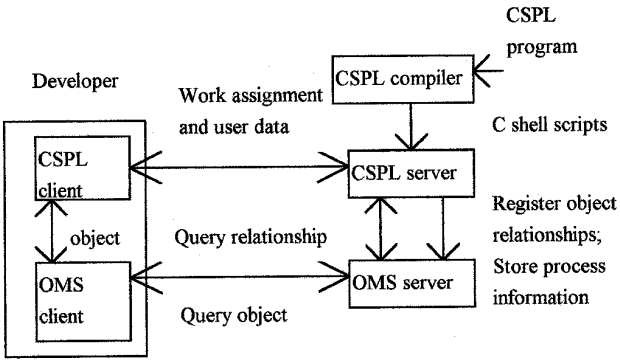

Fig. 2. CSPL environment architecture.

During process enactment, the CSPL server first registers object relationships and stores process information (such as interprocess communication) in the OMS server. Then, according to program flow of the CSPL program, the CSPL server assigns work to CSPL clients. Note that one client corresponds to one developer. Also, the CSPL server can query relationships from the OMS server and, based on the object relationships, may trigger necessary actions.

When a developer needs to access an object, his or her CSPL client will get the object by querying the OMS server via his or her OMS client. User data such as on-line users are also available to developers via CSPL clients.

Currently the CSPL server and the OMS server run on Sun Sparc20 workstation. And each client (a CSPL client and an OMS client) runs on a workstation with X windows. Our laboratory plans to port a client to a Pentium PC running Unix.

3.2 Why CSPL Uses Unix Shell

Operating systems manage hardware resources such as files, devices, etc. A striking similarity between the operating system and software engineering environment (environment) is that both are resource managers. Environment manages resources needed by software development and maintenance, including human resources (e.g., developers), software objects and hardware resources. Through the managerial supports, environment users need not worry about the details in utilizing those resources.

Since hardware resources have long been managed by operating systems, it is unnecessary for an environment to handle hardware resources directly. Instead, an environment can, and should, utilize an operating system to manage hardware resources through an interface language such as Unix shell. Furthermore, the entire environment can be built on top of an operating system, again through the operating system interface language. This relatively simple strategy may lower the barrier to adoption of the process environment. On the other hand, another Ada-like process language APPL/A uses a different strategy. An APPL/A program is first translated into a standard Ada program and then processed by an Ada compiler. This results in the difficulty of accessing run-time information such as run-time stack [32].

In a sense, the concept of the operating system has been extended to that of "operating environment" in this approach. This more or less reflects the trend of information technology moving from expensive hardware to expensive software development, and the consequent shift of management focus from hardware to software development.

In short, the CSPL environment is built on top of Unix operating system. The relationship between CSPL and Unix is illustrated in Fig. 3.

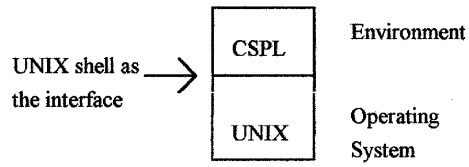

Fig. 3. Build CSPL environment on top of Unix.

4 CSPL Syntax and Semantics

Data, statements, and program units of CSPL are briefly described in this section. To specify program control flow, CSPL provides sequential statement, decision statement, iterative statement, and so on, that are primarily adopted from Ada83 and thus are not discussed. Rather, special statements for software process are discussed, such as the work assignment statement and communication-related statements. Additionally, special program units are discussed, such as role unit, tool unit and relation unit. Miscellaneous features, such as genetic unit and exception handling, are also briefly covered.

4.1 Data

There are two kinds of data in CSPL: variable and object. Variables are temporary data used in the flow control of process execution. Objects represent documents that could be intermediate or final software products, along with the attributes associated with the documents. Note that objects are persistent.

4.1.1 Variable

There are two types of variable: scalar variable and composite variable. Predefined scalar types are provided such as integer, real, char, Boolean, time, and event.

One type worthy of noting is event type. An event represents a particular circumstance in a software process. Sometimes an event must be waited on before a process can proceed. For example, only after a design document has been approved can coding start. The approval of the design can thus be regarded as an event declared as "design_approved" given below (bold face denotes CSPL keyword):

```
design_approved: event;
```

An event is essentially a Boolean variable with the value "True" or "False." The value of an event can be assigned asynchronously by an inform statement (to be described). An event can appear in places expecting Boolean expressions. However, Boolean variables cannot be used in places expecting events such as an inform statement.

CSPL provides enumeration types (also a scalar type) to increase program readability. For example:

```
type Button_Status is (Down, Up);
```

defines an enumeration type "Button_Status."

Two composite types are offered: array and record. For instance, array type "List" is defined as follows:

```
type List is array (1..10) of Integer;
```

Also, string type is a predefined array type in CSPL.

A record type contains several components of different types. For example, record type "Workload_of_Day" is defined as follows:

```
type Workload_of_Day is
    record
        Day:        Day_type;
        Workload:   Integer;
    end record;
```

Variables can be used to monitor progress of processes. For example, a counter in a loop can record the iteration number of an activity. A Boolean variable can represent the status of a process. Also, an event variable can be set by another task using an inform statement to notify occurrence of the event.

Because a process language focuses on documents (objects), the variables are used for auxiliary purposes. It seems unnecessary to provide complex user-defined variable types like ordinary programming languages do. Thus, CSPL only provides several simple variable types. The variables declared in a program are managed by the object management system (OMS) server through which the CSPL server can access the contents of the variables.

4.1.2 Object

As just mentioned, a CSPL object represents a document and the associated attributes. Although the semantics of variable and object are different, objects can be typed (called "object type") just like variables. All the object types are directly or indirectly derived from a type called "DocType" supplied by the object management system. For example, an object type called "test_case" with no attributes can be defined as follows:

```
type test_case is new DocType with null record;
```

An object of this type can then be declared as follows:

```
test_doc : test_case;
```

Most of the time an object has several attributes to assist developers in managing the object. Example 3 depicts object type "design_type" with three attributes: "last_modify_time," "modifier," and "description."

EXAMPLE 3. (*Defines an Object Type with Attributes*)

```
type design_type is
    record
        last_modify_time:   time;
        modifier:           string;
        description:        string := "This describes the
                                      document.";
    end record;
```

Like Ada, CSPL provides keyword "constant" to control access to attributes. For instance:

```
Button1 : constant Button_Status := Up;
```

declares a constant "Button1" of type "Button_Status" which is initialized as "Up." A constant cannot be modified by a developer after its initialization.

Note that objects' attributes are variables, which are persistent and are managed by the object management system. They can be manipulated using operators just like ordinary variables. For example, value "Kane" can be assigned to attribute "modifier" of object "design_doc" as follows:

```
design_doc'modifier := "Kane";
```

Note that an object can be a composite object. Fig. 4 depicts that composite object A is composed of two objects B and C, each corresponding to a file. Again, each object has its associated attribute values. Example 4 in the next subsection will depict composite object definition.

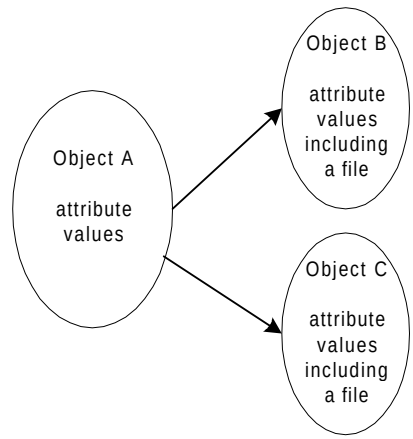

Fig. 4. Composite object.

4.1.3 Inheritance

Inheritance is very important in object-oriented programming languages. Ada83 cannot be regarded as object-oriented because it does not support inheritance. That is, Ada83 does not allow creating a new type that inherits an existing type. On the other hand, by marking a record type as tagged, Ada95 allows adding new components to an existing type to get a new type [3]. Ada95 thus supports inheritance.

In CSPL, users can define a composite object type that contains other object types as its attributes. In Example 4, type "SystemCharter" inherits the system-supplied type "DocType." Similarly, type "RequirementDoc" inherits "DocType." Moreover, type "RequirementDoc" contains an attribute of type "SystemCharter" just defined. That is, composite object type "RequirementDoc" is composed of type "SystemCharter" and other attributes. Note that tools cannot be inherited like this. Tools are separately defined in tool unit (to be described).

EXAMPLE 4. (*Composite Object Type "RequirementDoc"*)

```
type SystemCharter is new DocType with record
    statement: TextType;
    ....
end record;

type RequirementDoc is new DocType with record
    problemStatement:  TextType;
    systemCharter:     SystemCharter;
    visitingTime:      time;
end record;
```

4.2 Work Assignment Statement

Originally CSPL supported only single-user usage. Multi-user (multideveloper) requirements were not addressed. To remedy that, a new statement called "work assignment statement" is designed to assign work to multiple developers. There are three elements specified in this statement: 1) object manipulated, 2) tool used, and 3) role who manipulates object. For example, the following work assignment statement specifies a requirement document (an object) is edited by one analyst (a role) using an editor (a tool):

1 analyst ***edit*** req_doc ***using*** editor;

There are two kinds of work: manual work and automatic work. Manual work performed by a developer always requires creativity and experience of the human developer. For example, editing of the design document or source code is done by the developer, which cannot be done by computers automatically. On the other hand, automatic work such as compiling source code can be executed by computer without role (assumed by human developer).

Because manual activity (for manual work) and automatic activity (for automatic work) have different semantics over work assignment, two separate work assignment statements are designed for each:

For manual activity,

num_of_worker role_name activity_name object1
[***referring to*** object2 {objectN}] ***using*** tool_name
[***resulted in*** variable];

For automatic activity,

activity_name object1 [***to get*** object2] ***using*** tool_name
[***resulted in*** variable];

A manual activity is shown below where one designer designs a design document using an editor while referring to a requirement document:

1 designer ***edit*** design_doc ***referring to*** req_doc ***using*** editor;

A manual activity works on only one object, but it can refer to multiple objects. When multiple objects need work, multiple work assignment statements should be used.

An automatic activity does not specify a role in the work assignment statement. Thus, the work is not assigned to a client. Instead, it is assigned to the server. Two automatic activities are shown below: 1) a compiler is used to compile source code to get an executable unit and 2) a metrics tool is used to measure the quality of a design document.

1) ***compile*** source ***to get*** exec_unit ***using*** compiler;
2) ***measure*** design_doc ***using*** metrics-tool;

Some work may be executed (performed) by multiple developers, such as reviewing a design document. Under this circumstance, a manager specifies the number of developers (seven or all, respectively, in the statements shown below) for the work as:

7 reviewer ***review*** design_doc ***referring to*** req_doc ***using*** review_tool;

or

all reviewer ***review*** design_doc ***referring to*** req_doc ***using*** review_tool;

When multiple developers are involved in a review, each reviewer separately sees the same copy of an object. Those reviewers then communicate with each other through the user interface system to obtain a review result which can be a consensus or majority vote. The semantics of the review result is not defined here. After that, the work assignment statement as a whole terminates.

If a work assignment produces some result, the result can be specified in the work assignment statement. The statement below shows that the review result is stored in variable "result."

all reviewer ***review*** design_doc ***referring to*** req_doc ***using*** review_tool ***resulted in*** result;

Tools are managed by the user interface system of CSPL environment. After the tool unit of a CSPL program is compiled, a tool definition file is sent to the clients. When manual work is assigned to a client, the user interface system automatically checks the tool definition file to invoke the proper tool. Currently a developer is allowed to modify tool definition based on his or her preference through the user interface system (see Section 4.4.2). The CSPL user interface system will be discussed elsewhere.

For the sake of understandability, a work assignment can be encapsulated in a procedure, and then in a package. Example 5 shows a package that contains a work assignment statement. In the example, package "design" consists of object type "design" and an associated procedure "modify". The inform statement appearing in the procedure "modify" will be described shortly. Event "design_modified" in the inform statement is declared in the relation unit "design_and_source" not shown here (see Appendix A for the complete CSPL process program).

EXAMPLE 5. (*A Package That Contains a Work Assignment Statement*)

```
package design is
    type design_type is new DocType with record
        descriptor : string;
    end record;
    procedure modify(design_doc: in out design_type; req_doc:
        in req_type);
end design;
package body design is
    procedure modify(design_doc: in out design_type; req_doc:
        in req_type) is
    begin
        1 designer edit design_doc referring to req_doc using
          editor;
        inform design_and_source to set design_modified;
    end;
end design;
```

4.3 Communication-Related Statements

In CSPL, a task can issue an entry call for synchronous communication or inform (set/reset) an event for asynchronous communication. Normally the former is used to synchronize or start other tasks, while the latter notifies other tasks about the events. The scenario of a CSPL entry call is illustrated in Example 6 where task " ModifyDesign" issues an entry call to task "ReviewDesign." The semantics of Ada rendezvous is preserved in CSPL. This is implemented using the Unix named pipe [9], [18].

EXAMPLE 6. (*Task "ModifyDesign" Makes an Entry Call to Task "ReviewDesign"*)

"ModifyDesign"

```
task body ModifyDesign is
        …
begin
-- modify the design document
        …
ReviewDesign.start(design_doc, req_doc); -- rendezvous
      …
end;
```

"ReviewDesign"

```
task body ReviewDesign is
begin
 loop
  accept start(design_doc: in design_type;
               req_doc: in req_type) do
   -- reset event "design_approved"
   -- review the design document
   -- if review is ok, set event "design_approved"
  end;
 end loop;
end;
```

"Inform" and "waitfor" statements are provided in CSPL for asynchronous communication. An inform statement sets (or resets) an event in another task. A "waitfor" statement waits for the event to occur. When the event occurs, the "waitfor" statement stops waiting and clears the event. In Example 7, task "ModifyTestPlan" informs task "ModifyUnitTestPackage" to set event "start". Meanwhile, task "ModifyUnitTestPackage" waits for the event to be set before it can execute the following statements. The arrow in Example 7 indicates this communication.

EXAMPLE 7. (*Task "ModifyTestPlan" informs task "Modify Unit TestPackage*)"

"ModifyTestPlan"

```
task body ModifyTestPlan is

begin
 loop
 …
 testplan.modify(test_plan, design_doc);
 inform ModifyUnitTestPackage to set start;
end loop;
end;
```

"ModifyUnitTestPackage"

```
task body ModifyUnitTestPackage is
   start : event;
begin
 loop
  waitfor start;
  testpac.modify (test_unit, test_plan);
  inform TestUnit to set test_available;
 end loop;
end;
```

Example 8 illustrates another asynchronous communication. After the design review is approved, task "ReviewDesign" informs task "ModifyCode" to set event "design_approved". When the event is set to "True," the exit statement of task "ModifyCode" causes the process to jump out of the loop.

EXAMPLE 8. (*Task "ReviewDesign" Informs Task "ModifyCode" to Set an Event*)

"ReviewDesign"

```
task body ReviewDesign is
begin
 loop
    …
 -- reset event "design_approved"
 -- review the design document
 -- if the review is ok,
    inform ModifyCode
    to set design_approved;
 end loop;
end;
```

"ModifyCode"

```
task body ModifyCode is
 design_approved: event; …
begin
 loop
  loop
   -- loop to modify and compile source
   -- code while compilation is not ok
      …
   exit when design_approved = True;
  end loop;
  -- set event "code_available"
 end loop;
end;
```

4.4 Program Unit

A program unit is regarded as a unit which can be separately compiled. Incidentally, CSPL follows the Ada scope and visibility rules. Several kinds of the Ada-like program unit are supported by CSPL: 1) package unit which is used to collect resources such as data type, data, or program unit, 2) subprogram unit (procedure or function) which is a set of statements and can be invoked by another program unit, and 3) task unit which can be executed in parallel with other task units. Normally task units are used to model concurrent activities; while subprogram units to model sequential activities.

Furthermore, to support the multiuser and multirole feature and the dependency relationship, three kinds of program unit are designed: 1) role unit, 2) tool unit, and 3) relation unit. They are, respectively, depicted below.

4.4.1 Role Unit

It is not a good idea for managers to randomly assign work to developers without considering developers' experiences. "Role" provides an abstraction of the experiences. If a process language uses a role feature in work assignment, the feature will prevent process programmers from writing programs that negligently assigns work to unsuitable developers. A role unit defines the mapping of a role to the developers. Because a role can be assumed by multiple developers and a developer can also assume multiple roles, the mapping from role to developer is "many to many." The syntax of role unit is:

```
role role_name is
     user_variable := "user_name";
     {user_variable := "user_name";}
end;
```

The "user variable" shown above specifies abstract developer which maps to actual developer shown in "user_name". Role unit "designer" shown below specifies actual developers "cywang" and "bktseng," respectively, map to abstract developers "designer1" and "designer2" of "designer" role:

```
role designer is
    designer1 := "cywang";
    designer2 := "bktseng";
end;
```

Since a role unit is a separate compilation unit, it can be changed without affecting other program units. For instance, when designers are changed (which is quite often in a process) only the role unit, rather than the entire program, needs recoding and recompilation. No relinking is needed either.

Either abstract developer or role can be assigned work in work assignment statement. For example, abstract developer "designer1" is assigned the work to edit object "design_doc" in the statement below:

```
designer1 edit design_doc referring to req_doc using editor;
```

In this case, role unit "designer" just depicted will be consulted to map abstract developer "designer1" to actual developer "cywang."

On the other hand, role "designer" is assigned the work in the statement below:

```
1 designer edit design_doc referring to req_doc using editor;
```

In this case, there are two options to map a role to an actual developer: one is to monitor the developers and automatically assign the work to a developer who is underloaded, while the other is to prompt a message to the manager and let him or her manually assign the work. The two options are not features of the language. They are done manually with the support of the user interface system.

4.4.2 Tool Unit

There are two kinds of tool. Manual tool is used in manual activity which requires human intervention. Automatic tool, on the other hand, is used in automatic activity. For example, an editor is a manual tool; while a compiler is an automatic tool.

The syntax of tool unit is:

```
tool tool_name is
  tool_variable := "tool_name";
  {tool_variable := "tool_name";}
end;
```

A tool unit that contains a manual tool "editor" and an automatic tool "compiler" is depicted below where actual tools "vi" and "gcc" are assigned to abstract tools "editor" and "compiler," respectively.

```
tool IEEE_Example is
    editor := "vi";
    compiler := "gcc";
end;
```

Changing tools is quite common in a process because newer and more efficient tools may become available or developers may have their preferred tools. When the tool is changed (owing to abstract tool concept just mentioned), only the tool unit, instead of the entire program, needs modification. This improves program maintainability. Besides, an abstract tool name is usually easier to understand than an actual tool name. For example, "editor" (an abstract tool) appears to be clearer than "vi" (an actual tool).

Manual tools are usually chosen according to developers' preferences. For example, a developer might prefer "emacs" editor to "vi" editor. Given the centrality of developers in software process, developers' preferences have to be taken into account. CSPL thus allows developers to change their tools. For instance, a developer can change his or her editor to "emacs" (the editor was originally assigned "vi" in the tool unit shown above) through the user interface system.

4.4.3 Relation Unit

An object usually produces one or more objects in an activity. For example, the object code is produced from the source code in a compilation activity, and a design document is produced from a requirement specification in a design activity. The produced objects are thus dependent on the original object. This is called dependency relationship.

Dependency relationships among object types form a graph. Fig. 5 shows such a graph for the software process modeling of IEEE Example (see Appendix A). Three dependencies are identified in Fig.5. And each can be modeled as a relation unit.

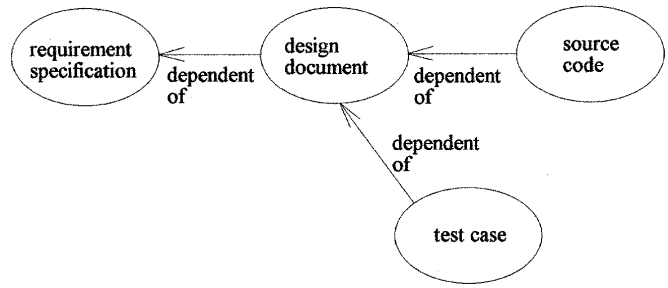

Fig. 5. Object type dependencies.

The syntax of relation unit is:

```
relation relation_name is
    object : object_type;
    object : object_type;
    upon modification_event do
        statements;
    end;
    {upon modification_event do
        statements;
    end;}
end;
```

Example 9 defines the dependency relationship between a requirement specification ("req_doc") and a design document ("design_doc"). Upon occurrence of event "req_modified," procedure "design.modify" is invoked to modify the dependent design document ("design_doc"). The dependency of design document upon requirement document is thus implicitly specified. Note that event "req_modified" needs no separate declaration for simplicity.

EXAMPLE 9. (*A Relation Unit*)

```
relation req_and_design is
    req_doc: req_type;
    design_doc: design_type;
    upon req_modified do
        design.modify(design_doc, req_doc);
    end;
end;
```

Roughly speaking, a relation unit can be regarded as a class with members (such as "(design_doc, req_doc)" in Example 9) and with operations (such as insertion and deletion) predefined in the object management system. For instance, relation unit "req_and_design" models the dependency relationship of design documents on requirement documents. A program segment below:

```
    req_doc : req_type;
design_doc1 : design_type;
design_doc2 : design_type;
......
req_and_design.insert(req_doc, design_doc1);
req_and_design.insert(req_doc, design_doc2);
```

shows two insertions are made to this relation unit, probably because two design documents "design_doc1" and "design_doc2" are developed based on, and thus dependent on, requirement document "req_doc." After the insertions, this relation unit (a class) contains the members (req_doc, design_doc1) and (req_doc, design_doc2). Note that a relation unit defines the relationship between two object types (e.g., design documents depend on requirement documents), while the members of this class (the relation unit) define the relationship between two objects (e.g., object "design_doc1" depends on object "req_doc").

4.5 Miscellaneous

Sometimes objects of different types are manipulated similarly in an activity. CSPL thus provides a generic program unit to avoid writing similar programs simply because of different object types [9]. The CSPL generic unit and exception handling are similar to those in Ada. They are briefly described below.

Example 10 depicts generic package "Inspection" is instantiated, respectively, by two different object types ("req_type" and "design_type") to obtain two packages—"Requirement_Inspection" and "Design_Inspection."

EXAMPLE 10. (*A Generic Unit*)

```
generic
type doc is private;
package INSPECTION is
procedure modify(d : in out doc);
end INSPECTION;

-- instantiate new packages
package REQUIREMENT_INSPECTION is new INSPECTION(req_type);
package DESIGN_INSPECTION is new INSPECTION(design_type);
```

CSPL allows process programmers to define their own exceptions and the handlers [9]. Exceptions usually occur due to abnormality in process enactment such as schedule delay, developer replacement, and so on. Exception "time out" (when the review time is up) is depicted in Example 11. It is raised by the ***raise*** statement in procedure "Review."

EXAMPLE 11. (*Exception Handling*)

```
procedure Review is
timeout : exception;
    …
begin
    …
  raise timeout;
    …
exception
when timeout =>
    output "Time out, start review now.";
    ReviewDesign.start(design_doc, req_doc);
end;
```

5 RELATED WORK

This section compares the related work with CSPL in this diversified field. The process model, multiuser and multirole features, and object modeling of several environments are thus compared.

5.1 Process Model

This section focuses on several aspects of process model: form of expression, object, and activity.

5.1.1 Form of Expression

In process-centered environment, software process must be modeled in some *form*. The forms of the existing environments are categorized as follows:

1) *Regular Expression*. Regular expression is used in Hakoniwa to model software processes [20]. This form appears clear and easy to recognize by computers. However, it is not very easy to read and understand, compared with program languages.
2) *Petri Net*. Petri Net is used in Process Weaver [7]. Because it has high degree of parallelism, it is very suitable to model concurrent tasks in a process. Since Petri Net is a graphical form, it is easy to read and understand—if a Petri-Net is neither too large nor too complicated. A state in a process is represented as a place in Petri Net. Thus, it is easy to use Petri Net to monitor the progress of software processes. However, Petri Net is weak in the capability of describing object models. For example, how can we describe which product serves as the input or output of a transition? Incidentally, Process Weaver uses another language (i.e., co-shell) to solve this problem. FUNSOFT in Kernel/2r is also Petri Net based [17].
3) *Programming Language*. One important advantage of using programming language to model software process is that a process in this form is relatively easier to enact using computers. Diversified programming language paradigms have been used. Examples are:
   a) *procedural* languages, such as APPL/A [32], CSPL [9], [10], MDL [8];
   b) *functional* languages, such as HFSP [22];
   c) *rule-based* languages, such as Merlin [29], Marvel [21];
   d) *goal-directed or planning* languages, such as Intermediate [26], [27], Grapple [19]; and
   e) *triggered* languages, such as Adele-2 [4], EPOS [12].

Each paradigm above exhibits some strengths, but also suffers some weaknesses. Note that the languages cited above may also represent other paradigms. The paradigms are briefly compared below:

Procedural languages in a) appear to be the paradigm that developers are most familiar with. Developers often use this kind of language, such as C++, Ada and Java, to develop a product. Therefore it is natural for the developers to use a procedural language to develop a process program. However, the abstraction level of this paradigm is relatively low.

Functional languages in b) such as HFSP applies top-down functional decomposition to obtain an activity hierarchy. The hierarchy is very clear and easy-to-understand. However, unless developers understand a software process very well, it seems difficult to carry out such a decomposition without too much backtracking.

Rule-based languages in c) achieve a higher abstraction level than that of procedural languages [30]. The latter specifies exact control flow of the processing. The former, on the other hand, specifies the form of results without revealing the lower level procedural control flow knowledge. This paradigm provides declarative meaning to a process. However, if developers do not understand adequately about the meaning of the process, it would be difficult for them to construct a rule-based process program.

Let us take this one step further. Some rules about a process can be constructed to serve as constraints or goals for the process. This idea brings about the *goal-based* or *planning* language in d), where mixed paradigms can be observed. For example, Intermediate uses preconditions, obligations, and postconditions to construct an "implicit, internal locus of control" which is a partial ordering. Within the constraint of that partial ordering, the developer is free to control the execution at his or her will. Similarly, Grapple uses some rules to build constraints. A recent language Julia [34] provides rich semantics. For instance, a Julia "step" specifies objects, resources (such as developer and tool), substeps, step constraints, proactive control, reactive control, preconditions, postconditions, and exception handlers. These mixed-paradigm languages seem promising. However, it is needed to strike the balance between language theory and process practicality.

In e), *triggered* languages, a database transaction may signal an event which triggers an activity. For instance, in Adele-2 and EPOS, a transaction may set an event which triggers an action (activity). That action may set another event which in turn triggers another action, and so on. This kind of language is good for object modeling, such as dependency, version, and so on. However, it lacks a general view of an activity model. Thus, complex process programs appear extremely difficult to construct using this paradigm.

5.1.2 Object

"Object" in CSPL, "work object" in AD/Cycle [11], or "document" in MDL [8] represents product that can be input or output of activity. Many process-centered environments provide predefined object types. Some environments allow users to define new types. For example, ALF [14] users can define new types as subtypes of existing types. SLANG [2] allows users to define an object like a record. Marvel users can define object types as classes in a "strategy". A strategy defines classes, tools and rules, which is similar to a simple CSPL program. Furthermore, EPOS provides a predefined class hierarchy called EPOS types. Users can define new object types by inheriting the attributes, procedures, and triggers from the hierarchy. PLEIADES [36] uses a unique approach to construct object types. It provides type constructors such as node, edge, relation, relationship and sequence. Like APPL/A, PLEIADES is a part of the Arcadia project. Various ways can be used by APPL/A to generate relations such as using Triton or PLEIADES [32].

CSPL adopts record type and derived type from Ada [5]. CSPL users can derive a new type based on an existing type. Moreover, CSPL allows users to define a new object type which is not based on a defined type. For example, if an activity is to process a picture file, an object type for that can be defined, provided that the tool for that type is specified.

5.1.3 Activity

Completing a process usually takes many activities. Some issues of activity are: *primitive activity*, *activity flow*, and *task communication*.

Primitive Activity. A primitive activity is an elementary activity. It represents one abstraction such as editing source code or compiling source code. In this paper it is simply called an activity.

In a Petri Net based process model such as Process Weaver, a primitive activity can be represented as a transition [7]. Because Petri Net cannot model the work to be done in a transition, Process Weaver uses another co-shell language to model that. In CSPL, a primitive activity is modeled as a work assignment statement. CSPL need not use another language as Process Weaver does.

Activity Flow. Activity flow relates to activity dynamics of process such as execution sequence, concurrency, and synchronization.

In Process Weaver, activity flow can be easily represented by Petri Net "procedure model." In Hakoniwa [20], it can be modeled by regular expression which, however, is difficult to write and understand. In HFSP, activities in the functional hierarchy can be concurrently executed—if there is no data dependency between their attributes.

In rule-based languages such as Marvel and Merlin, preconditions and postconditions of activities serve as the basis for backward and forward reasoning. In triggered languages such as Adele-2, a database transaction may signal an event, which then triggers an activity. That activity may still signal another event that triggers another activity, etc.

Because Ada provides sequential statement, alternative statement and iterative statement to control the flow, Ada-like languages such as CSPL or APPL/A can directly describe activity flow.

Task Communication. Task is more abstract than activity. It executes activities according to activity flow. For instance, task "ModifyCode" executes activities of editing source code, compiling it, etc.

Because multiple tasks are concurrently executed in a process, task communication is needed. Environments provide their distinctive methods to handle this. For instance, Process Weaver provides two functions: "SendEvent" and "WaitForEvent" to communicate with all external systems. Hakoniwa provides message transfer and task control primitives. Adele-2 provides events to trigger other tasks. CSPL provides high level language statements for the purpose, such as Ada rendezvous related statements and inform and waitfor statements. That is, a CSPL task can either issue an entry call for synchronized communication or inform another task about an event for asynchronous communication.

5.2 MultiUser and MultiRole

Many environments support multiuser usage in which a user is usually provided with an agenda for work assigned to him or her. In Hakoniwa, each user has a "task organizer" to control several "task drivers" which guide the user to do his or her work [20]. Process Weaver provides an agenda for each user [7]. When executing a "procedure," a "work assignment" is sent to an agenda. Because Process Weaver's agenda is shown in a window and a piece of work is represented by an icon, users can easily know what work they should do. In Merlin, a "working context" consists of all objects a developer can manipulate. For each object, all the available activities are shown on the screen. And the developer can choose any one of the activities to execute. The relationships between objects are also shown [29]. In CSPL, a client is provided for each user, collecting the assigned work during process enactment. A CSPL client runs a windowed user interface system that provides an agenda.

As mentioned earlier, one distinctive CSPL multiuser feature is that the work assignment statement assigns work to multiple developers of the same role. For instance, multiple reviewers can be assigned with a review using this feature. Moreover, concurrent object access by multiple developers is supported by the object management system.

Next, multirole feature is covered. In Kernel/2r, role is defined in project management view [13]. It not only defines mapping of role to developer, but also defines what activities are performed by a role and what object can be manipulated by an activity. In APDM [15], there are responsibility charts for similar purposes. APPL/A does not seem to support multiuser and multirole feature at the moment. In CSPL, role unit defines role mapping to the developer. Besides, work assignment statement specifies work and related role.

5.3 Object Modeling

Object dependency relationship is important because it facilitates automation of triggering the modification of dependent objects when an object is modified. Both Marvel [21] and Merlin [29] use rules to describe dependency relationship. Adele-2 [4] uses events instantiated on relations to model the dependency. EPOS [12] provides several relation types such as "DependOn" and "ImplementedBy" to model the dependency. In APPL/A [32], dependency relationship is represented by relation. The dependency specification in relation defines how dependent objects are to be generated. Also, the "trigger unit" can be used to handle such relationship and to cause a transitive change. That is, if object C is dependent on object B, and object B is dependent on object A, when object A is modified, a modification on object B will be triggered, which in turn trigger a modification on object C. This is reactive control. On the other hand, proactive control actively calls a modification procedure when needed.

In process program, reactive control is intuitively appealing, but it can get complicated if used extensively, according to APPL/A experiences [33]. Use of reactive and proactive controls should thus be balanced. In Adele-2 [4], events can be instantiated on: object or object relationship. Their experience depicted the two types of instantiation making it difficult to have a general view of control.

In managing object relationship, CSPL provides the relation unit which defines dependency between two object types and statements to be executed when modification event is raised. Transitive change can be accomplished by using multiple relation units.

ALF-based environment [14] uses a PCTE object management system (OMS). APPL/A ultilizes a relational database management system Triton [16] as its OMS. CSPL develops its OMS on top of the Unix file system. As mentioned earlier, a distinctive CSPL feature is that its object modeling capabilities are unified with the CSPL process environment.

6 Conclusions

It is concluded that CSPL process environment has provided facilities to support: 1) object orientation, 2) multirole and multiuser, and 3) unified object modeling. Finally, a CSPL program solving the Software Process Modeling Example Problem is presented.

6.1 Object Orientation

CSPL originally adopted concepts from Ada83. Ada83 has some capabilities which are normally associated with object orientation [35]. However, Ada83 could not inherit a type. Neither could the original CSPL. By using the tag of a tagged type as discriminant, Ada95 can inherit a record type. Ada95 is thus enhanced to be an object-oriented language [3]. This enhancement technique is adopted by CSPL to support inheritance feature. Therefore the current CSPL is an object-oriented language.

6.2 Multirole and Multiuser

First, CSPL uses role unit to define the role that the developer can assume. Notice that a developer can appear in multiple role units to assume multiple roles. This supports multirole feature. Second, CSPL uses work assignment statement to assign work to multiple developers of the same role who are working concurrently. CSPL also provides communication-related statements to support both synchronous and asynchronous communications among developers. Moreover, concurrent object access is managed by the object management system. These support multiuser feature.

6.3 Unified Object Modeling

Object modeling includes modeling object, object type, object relationship, and so on. CSPL models object relationship by using a relation unit which provides capability to trigger activity on dependent objects when modification event is raised. In addition, CSPL object management system manages long transaction, version, access, and so on that are not covered in this paper due to limited space. Of particular significance is that CSPL object modeling capabilities are unified with CSPL process environment.

To evaluate the CSPL environment, a CSPL program solving the "Software Process Modeling Example Problem" [23] is developed and enacted. In this example, there are six tasks contained in main procedure "StartTasks":

- "ModifyDesign"
- "ReviewDesign"
- "ModifyCode"
- "ModifyTestPlan"
- "ModifyUnitTestPackage"
- "TestUnit".

An object type and the associated procedures to manipulate the objects of that type are encapsulated in a package. Seven packages are developed in this program. The program size is about 250 lines of CSPL code (see Appendix A).

APPENDIX A – CSPL PROCESS PROGRAM

A CSPL process program to solve the "IEEE Software Process Modeling Example Problem" is developed. The main procedure "StartTasks" starts six concurrent tasks described below (see Fig. 6):

Task 1. Task "ModifyDesign" modifies a design document. When the modification is finished or review time is up, it makes an entry call to task "ReviewDesign" to review the design document.

The two tasks will be synchronized. Readers will notice that all other task communications in this example are asynchronous where a task just passes some information to another task - no need to synchronize the tasks.

Task 2. When the document is approved in the review, task "ReviewDesign" sends an event to inform task "ModifyCode."

Task 3. Task "ModifyCode" modifies a source code according to whether the design document is approved or not.

Task 4. When the modification of code is finished and the design document is approved, task "ModifyCode" informs task "TestUnit" that the code is available.

Task 5. Task "ModifyTestPlan" modifies test plan and sends an event to start task "ModifyUnitTestPackage" which then modifies unit test package and sends an event to inform "TestUnit" that the test unit is available.

Task 6. Task "TestUnit" tests the source code when the events "code_available" and "test_available" are true. If the testing fails, task "TestUnit" restarts "ModifyCode" and "ModifyUnitTestPackage" until testing is successful.

The CSPL process program follows.

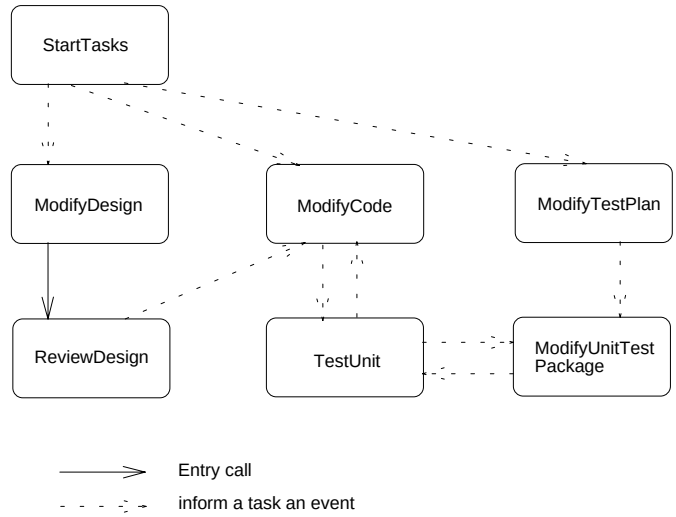

Fig. 6. The activity model for the IEEE example problem.

```
tool IEEE is
  compiler := "gcc";
end;

role analyst is
  analyst1 := "cywang";
  analyst2 := "bktseng";
end;

role designer is
  designer1 := "cywang";
  designer2 := "bktseng";
end;

role reviewer is
  reviewer1 := "cywang";
  reviewer2 := "bktseng";
end;

role coder is
  coder1 := "cywang";
  coder2 := "bktseng";
end;

role tester is
  tester1 := "cywang";
  tester2 := "bktseng";
end;

package IEEE is

  review_time : time;

  package requirement is
    type req_type is new DocType with null record;
    procedure modify(req_doc: in out req_type);
  end requirement;

  package design is
    type design_type is new DocType with
      record
       descriptor : string;
      end record;
      procedure modify(design_doc: in out design_type;
                       req_doc: in req_type);
      function reviewing(design_doc: in design_type;
                         req_doc: in req_type)
```

```
    return Boolean;
  end design;

  package source_code is
    type source_type is new design_type with null record;
    procedure modify(source: in out source_type;
                     design_doc: in design_type);
  end source_code;

  package exec is
    type exec_type is new DocType with null record;
    function compilation(exec_unit: in out exec_type;
                         source: in source_type)
    return Boolean;
  end exec;

  package testplan is
    type test_plan_type is new  design_type with null record;
    procedure modify(test_plan: in out test_plan_type;
                     design_doc: in design_type);
  end testplan;

  package testpac is
    type test_unit_type is new design_type with null record;
    procedure modify(test_unit: in out test_unit_type;
                     test_plan: in test_plan_type);
    function testing(exec_unit: in exec_type;
                     test_unit: in test_unit_type)
    return Boolean;
  end test_pac;

end IEEE;

package body IEEE is
relation req_and_design is
  req_doc : req_type;
  design_doc : design_type;
  upon req_modified do
    design.modify(design_doc, req_doc);
  end;
end;

relation design_and_source is
  design_doc : design_type;
  source : source_type;
  upon design_modified do
    source_code.modify(source, design_doc);
  end;
end;

package body requirement is
  procedure modify(req_doc: in out req_type) is
  begin
    1 analyst edit req_doc using editor;
    inform req_and_design to set req_modified;
  end;
end requirement;

package body design is
  procedure modify(design_doc: in out design_type;
                   req_doc: in req_type) is
  current_time : time;
  timeout : exception;
  begin
    current_time := GetCurTime;
    if current_time > review_time
    then raise timeout;
    end if;
    1 designer edit design_doc referring to req_doc using editor;
    inform design_and_source to set design_modified;
  end;

  function reviewing(design_doc: in design_type;
                     req_doc: in req_type) return Boolean is
  result : Boolean;
    begin
    all reviewer review design_doc referring to req_doc
    using review_tool resulted in result;
    return result;
  end;
end design;

package body source_code is
  procedure modify(source: in out source_type;
                   design_doc: in design_type) is
  begin
    1 coder edit source referring to design_doc using editor;
  end;
end source_code;

package body exec is
  function compilation(exec_unit: in out exec_type;
                       source: in source_type) return Boolean is
  result : Boolean;
      begin
    compile source to get exec_unit using compiler resulted in
    result;
    return result;
  end;
end exec;

  package body testplan is
    procedure modify(test_plan: in out test_plan_type;
                     design_doc: in design_type) is
    begin
      1 tester edit test_plan referring to design_doc using editor;
    end;
  end testplan;

package body testpac is
  procedure modify(test_unit: in out test_unit_type;
                   test_plan: in test_plan_type) is
  begin
    1 tester edit test_unit referring to test_plan using editor;
  end;
  function testing(exec_unit: in exec_type;
                   test_unit: in test_unit_type) return Boolean is
  result : Boolean;
      begin
    all tester test exec_unit referring to test_unit
    using test_tool resulted in result;
    return result;
  end;
end testpac;

end IEEE;

with IEEE;
procedure StartTasks is
   req_doc : req_type := "requirement.doc";
  design_doc : design_type := "design.doc";
  source : source_type := "example.c";
  test_plan : test_plan_type := "test.plan";
  test_unit : test_unit_type := "test.unit";
  exec_unit : exec_type := "a.out";
  reviewok : Boolean;
  compileok : Boolean;
  testok : Boolean;
  task ModifyDesign;
  task ReviewDesign is
    entry start(design_doc: in design_type; req_doc: in req_type);
  end;
```

```
task ModifyCode;
task ModifyTestPlan;
task ModifyUnitTestPackage;
task TestUnit;

task body ModifyDesign is
 start : event;
 procedure modify_and_reviewing(design_doc: in out design_type;
                                req_doc: in req_type) is
 begin
  design.modify(design_doc, req_doc);
  ReviewDesign.start(design_doc, req_doc);
 exception
  when timeout =>
    output "Time out, starting review now";
    ReviewDesign.start(design_doc, req_doc);
 end;
begin
 loop
  waitfor start;
  reviewok := False;
  while reviewok = False
  loop
   modify_and_reviewing(design_doc, req_doc);
  end loop;
 end loop;
end;

task body ReviewDesign is
begin
loop
 accept start(design_doc: in design_type; req_doc: in req_type)
 do
      inform ModifyCode to reset design_approved;
  reviewok := design.reviewing(design_doc, req_doc);
  if reviewok = True then
   inform ModifyCode to set design_approved;
  end if;
 end;
end loop;
end;

task body ModifyCode is
design_approved : event;
start : event;
begin
 loop
  waitfor start;
   compileok := False;
  loop
   while compileok = False
   loop
    source_code.modify(source, design_doc);
    compileok := exec.compilation(exec_unit, source);
   end loop;
   exit when design_approved = True;
   compileok := False;
  end loop;
  inform TestUnit to set code_available;
 end loop;
end;

task body ModifyTestPlan is
start : event;
begin
 loop
  waitfor start;
  testplan.modify(test_plan, design_doc);
  inform ModifyUnitTestPackage to set start;
 end loop;
end;

task body ModifyUnitTestPackage is
start : event;
begin
 loop
  waitfor start;
  testpac.modify(test_unit, test_plan);
  inform TestUnit to set test_available;
 end loop;
end;

task body TestUnit is
code_available : event;
test_available : event;
begin
 loop
  waitfor code_available and test_available;
  testok := testpac.testing(exec_unit, test_unit);
  if testok = False then
   inform ModifyCode to set start;
   inform ModifyUnitTestPackage to set start;
    else
     exit;
   end if;
  end loop;
 end;

begin
 requirement.modify(req_doc);
 review_time := SetTime(19,3,15,95);
 inform ModifyDesign to set start;
inform ModifyCode to set start;
inform ModifyTestPlan to set start;
 waitfor testok;
 req_and_design.insert(req_doc, design_doc);
 design_and_source.insert(design_doc, source);
end;
```

ACKNOWLEDGMENTS

The author wishes to thank Dewayne E. Perry, Stanley Sutton, Wuu Yang, and the anonymous reviewers for their valuable comments. This research cannot be completed without dedicated work by the following members of the Software Engineering Environment laboratory: B.K. Tzeng, C.M. Tu, Y.M. Chen, C.Y. Wang, B.J. Hsu, Y.L. Liu, and C.P. Lai. Special thanks to Pei Hsia for his encouragement over the years. This research was sponsored by the National Science Council, Taiwan, ROC, under Grant No. NSC 85-2213-E-009-058.

REFERENCES

[1] G. Anderson and P. Anderson, *The UNIX C Shell Field Guide*. Englewood Cliffs, N.J.: Prentice Hall, 1986.
[2] S. Bandinelli, A. Fuggetta, and S. Grigolli, "Process Modeling In-The-Large with SLANG," *Proc. Second Int'l Conf. Software Process*, Berlin, pp. 75-83, Feb. 1993.
[3] J. Barnes, *Introducing Ada 9X*. *Private Communication*. June 1994.
[4] N. Belkhatir and W. L. Melo, "Supporting Software Development Process in Adele 2," *The Computer J.*, vol. 37, no. 2, pp. 621-628, 1994.
[5] G. Booch and D. Bryan, *Software Engineering with Ada*, 2nd edition. Benjamin Cummings, 1994.
[6] G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd edition. Benjamin Cummings, pp. 77-79, 1994.
[7] M. Bourdon, *Process Weaver: Process Modeling Experience Report*. France: Cap, Gemini Innovation, 1992.

[8] J.Y. Chen and P. Hsia, "MDL (Methodology Definition Language): A Language for Defining and Automating Software Development Process," *J. Computing Language*, vol. 17, no. 3, pp. 199-211, July 1992.
[9] J.Y. Chen and C.M. Tu, "An Ada-Like Software Process Language," *J. Systems Software*, vol. 27, no. 1, pp. 17-25, Oct. 1994.
[10] J.Y. Chen and C.M. Tu, "CSPL: A Process-Centred Environment," *Information and Software Technology*, vol. 36, no. 1, pp. 3-10, Jan. 1994.
[11] G. Chroust, H. Goldmann, and O. Gschwandtner, "The Role of Work Management in Application Development," *IBM Systems J*, vol. 29, no. 2, pp. 189-207, 1990.
[12] R. Conradi, M. Hagaseth, J-O Larsen, M. Nguyen, B. Munch, P. Westby, W. Zhu, M. Letizia, and C. Liu, "Object-Oriented and Cooperative Process Modeling in EPOS," *Software Process Modeling and Technology*, A. Finkelstein, J. Kramer, and B.A. Nuseibeh, eds., *Advanced Software Development Series*. Research Studies Press Ltd. (John Wiley), pp. 33-70, 1994.
[13] W. Deiters, "Support for Interworking in Kernel/2r," *ESF Seminar*, Berlin, Nov. 1992.
[14] J.C. Derniame, C. Godart, V. Gruhn, and J. Lonchamp, "Process Centered IPSEs in ALF," *Proc. 15th Int'l Workshop Computer-Aided Software Eng.*, Montreal, pp. 179-190, July 1992.
[15] D.W. Drew, "Developing Formal Software Process Definitions," *Proc. Conf. Software Maintenance*, Montreal, pp. 12-20, 1993.
[16] D. Heimbigner, "Experience with an Object Manager for a Process-Centered Environment," *Proc. 18th VLDB Conf.*, Vancouver, pp. 585-595, Aug. 1992.
[17] B. Holtkamp and H. Weber, "Kernel/2r-A Software Infrastructure for Building Distributed Applications," *Proc. Fourth Int'l Conf. Future Trends in Distributed Computing Systems*, Lisboa, Sept. 1993.
[18] R.N. Horspool, *The Berkeley UNIX Environment*. Prentice Hall, 1992.
[19] K.E. Huff, "Probing Limits to Automation: Towards Deeper Process Models," *Proc. Fourth Int'l Software Process Workshop*, New York, pp. 79-81, 1988.
[20] H. Iida, K. Mimura, K. Inoue, and K. Torii, "Hakoniwa: Monitor and Navigation System for Cooperative Development Based on Activity Sequence Model," *Proc. Second Int'l Conf. Software Process*, pp. 64-74, Feb. 1993.
[21] G.E. Kaiser and N.S. Barghouti, "Database Support for Knowledge-Based Engineering Environments," *IEEE Expert*, vol. 3, no. 2, pp. 18-32, 1988.
[22] T. Katayama, "A Hierarchical and *Functional* Approach to Software Process Description," *Proc. Fourth Int'l Software Process Workshop*, NewYork, pp. 87-92, 1989.
[23] M. Kellner et al., "ISPW-6 Software Process Example," *Proc. First Int'l Conf. Software Process*, Redondo Beach, Calif., pp. 176-186, 1991.
[24] J. Lonchamp, "A Structured Conceptual and Terminological Framework for Software Process Engineering," *Proc. Second Int'l Conf. Software Process*, Berlin, pp. 41-53, 1993.
[25] L.J. Osterweil, "Software Processes are Software Too," *Proc. Ninth Int'l Conf. Software Eng.*, pp. 2-13, 1987.
[26] D.E. Perry, "Policy-Directed Coordination and Cooperation," *Proc. Seventh Software Process Workshop*, Yountville, Calif., pp. 111-113, Oct. 1991.
[27] D.E. Perry, "Enactment Control in Interact/Intermediate," *Proc. Third European Workshop on Software Process, EWSPT 94*, Villard de Lans, France, Feb. 1994. Brian C. Warboys, ed., *Lecture Notes in Computer Science, 772*, pp. 107-113. Springer-Verlag, 1994.
[28] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, N.J.:, Prentice Hall, 1981.
[29] B. Peuschel and W. Schafer, "Concepts and Implementation of Rule-Based Process Engine," *Proc. 14th Int'l Conf. Software Eng.*, pp. 262-279, 1992.
[30] R.W. Sebesta, *Concepts of Programming Languages*, 2nd edition. Benjamin Cummings, pp. 491-493, 1993.
[31] B. Stroustrup, *The C++ Programming Language*, 2nd edition, pp. 143-179. Addison-Wesley, 1993.
[32] S.M. Sutton Jr., D. Heimbigner, and L.J. Osterweil, "APPL/A: A Language for Software Process Programming," *ACM Trans. Software Eng. and Methodology*, vol. 4, no. 3, pp. 221-286, 1995.
[33] S.M. Sutton, private communication, May 1995.
[34] S.M. Sutton and L.J. Osterweil, "The Design of a Next-Generation Process Language," Technical Report 96-030, Dept. of Computer Science, Univ. of Massachusetts at Amherst, 1996.
[35] S.T. Taft, "Ada 9X: From Abstraction-Oriented to Object-Oriented," *Proc. OOPSLA '93*, pp. 127-136, 1993.
[36] P. Tarr and L.A. Clarke, "PLEIADES: An Object Management System for Software Engineering Environments," *Proc. ACM SIGSOFT '93 Symp.*, Dec. 1993.

Jen-Yen Jason Chen received a BS degree in industrial engineering from Tung Hai University, Taiwan; an MS degree in industrial engineering, an MS degree in computer science, and a PhD degree in computer science and engineering from the University of Texas at Arlington. He was with Gearhart Industry Rresearch Laboratory, Texas, in 1981 and 1982. From 1983 to 1986 he participated in the research on software engineering environment at the University of Texas. He has been on the faculty at National Chiao Tung University, Taiwan, since 1987 and is now a professor in the Department of Computer Science and Information Engineering. Dr. Chen served as an advisor to the Institute for Information Industry during 1994–1996. He is currently a visiting professor at the University of New South Wales, Australia.

His research interests center on software process modeling, process-centered environment (process environment), and object-oriented analysis and design method. In particular, he is interested in software process improvements through process environment supports. Dr. Chen won *Top Scholar* in an assessment of system and software engineering scholars in 1995. The assessment was based on cumulative publication frequency in six leading journals of that field in the world. He is a member of the IEEE.