JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, no. 4, September-October 2002

A Process Modeling Language Consisting of High Level UML-based Diagrams and Low Level Process Language

Shih-Chien Chou, Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan

Abstract

This article presents a process modeling language consisting of high level UML-based diagrams and a low level process language. The high level diagrams facilitate process program development, while the low level process language models processes as process programs. Between them is a mapping to further facilitate program development.

1 INTRODUCTION

Software can be developed by following a process prescribed by a method, such as the Booch method [Booch96]. Recently, many *process-centered software engineering environments* (PSEEs) [Chen97, Sutton95, Belkhatir94, Doppke98, Iida93, Heimann97, Perry91, Peuschel92, Bandinelli93, Jaccheri93] have been developed to facilitate controlling complicated *software processes* (software development processes). Generally, a PSEE provides a modeling language to model processes as *process programs* for enactment (execution). A process modeling language is thus essential in a PSEE.

Generally, a process modeling language should model necessary *process components*, including products, developers, activities, activity sequence and synchronization, exceptions and their handlers, tools, schedules, budgets, and relationships among process components. Moreover, the language should facilitate process program development. The rationale is that complicated software processes may result in large-sized process programs. If process program development is not facilitated, developing the process program of a complicated process tends to be difficult and worse, the process program tends to be unstructured and hence difficult to verify and maintain.

Cite this column as follows: Shih-Chien Chou: *A Process Modeling Language Consisting of High Level UML-based Diagrams and Low Level Process Language*, in Journal of Object Technology, vol. 1, no. 4, September-October 2002, pages 137-163. http://www.jot.fm/issues/issue_2002_09/article3

We have designed a modeling language to model processes using high level UML-based diagrams and a low-level process language (note that UML is the abbreviation for "unified modeling language" [Fowler97]). The high level diagrams facilitate process program development. The low level process language models processes as process programs. Between those two levels is a mapping, which facilitates transforming the high level diagrams into process programs. This article presents our process modeling language. The following text respectively describes the high level UML-based diagrams, the low level process language, the mapping, and an example.

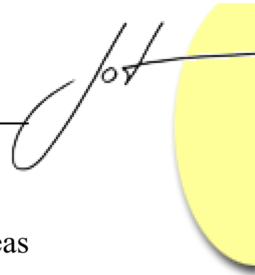
2 HIGH LEVEL UML-BASED DIAGRAMS

The high level UML-based diagrams facilitate process program development. In developing a process program, the process should first be analyzed and designed, during which a model is needed. Since a software process is composed of partially ordered activities [Garg96, Feiler93] and the UML activity diagram is powerful in modeling activities and their order, we designed a diagram based on the UML activity diagram to model activities, activity sequence, and activity synchronization. The diagram is called the *P-activity diagram* [Chou00]. In addition to the P-activity diagram, we also designed a *P-class diagram* [Chou00] to model products, roles, tools, schedules, budgets, and their relationships. The P-activity diagram and P-class diagram are respectively described below.

P-activity diagram

The P-activity diagram models activity-related components. It is designed to improve the understandability of activities in a process. Process components modeled in this diagram include activities, activity sequence, concurrent activities and activity synchronization, activity communication, and exceptions and their handlers. They are respectively described below:

1. Activities and activity sequence. Activities are the most important components to model. To improve understandability, a process's activities can be structured in a layered fashion. That is, activities can be decomposed. With decomposition, a process can be first depicted as a top level P-activity diagram, which is composed of coarse-grained activities. Activities in the top level diagram can be decomposed to form more detailed P-activity diagrams if necessary. The decomposition proceeds until all activities are fine-grained enough. For example, a waterfall model can be first modeled as coarse-grained activities including "Analysis", "Design", "Implementation", and "Testing". The activities can then be decomposed as needed. For example, the activity "Testing" can be decomposed into finer-grained activities such as "Unit test", "Integration test", and "System



test”. Activities that are decomposed are called *non-primitive activities*, whereas those that are not decomposed are called *primitive activities*.

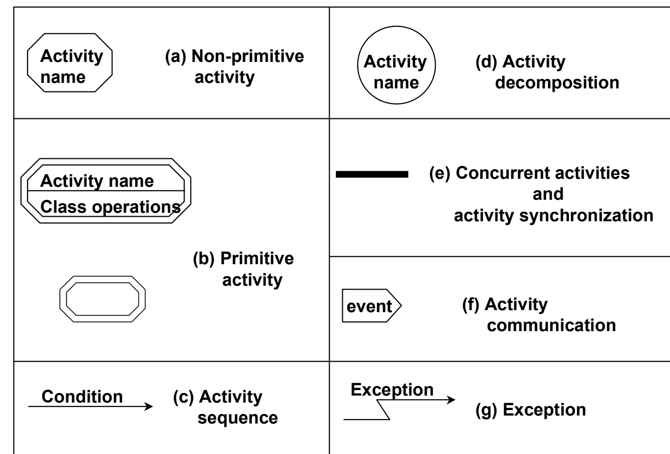


Figure 1. P-activity diagram notations

To model activity and their decomposition, the notations in Figures 1(a) through 1(d) are used. Figure 1(a) models non-primitive activities. Figure 1(b) models primitive activities. Some primitive activities can be accomplished by invoking class operations. In this case, the upper notation in Figure 1(b) is used, in which the first field shows the activity name and the second shows class operations. Placing class operations in primitive activities maintains traceability between P-activity diagrams and classes, with which changing a P-activity diagram can trace back to the affected classes, and vice versa. If a primitive activity is not accomplished by invoking class operations, the lower notation in Figure 1(b) is used.

Figure 1(c) models activity sequence. That is, for the activities connected by arrows, the successors can be started only when the predecessors finish. Figure 1(d) models decomposition relationships among activities.

We use an example (see Figure 2) to depict the usage of Figures 1(a) through 1(d). Figure 2(a) shows a process containing three non-primitive activities. Figure 2(b) shows the decomposition result of the activity “Testing”. Naming the starting circle in Figure 2(b) as “Testing” means that the activity “Testing” is decomposed into a P-activity diagram shown in the figure. Figure 2(a) also shows that conditions can be associated with activity sequence lines. For example, after the design activity, if the design verification passes, the implementation task starts. Otherwise, the design activity is redone.

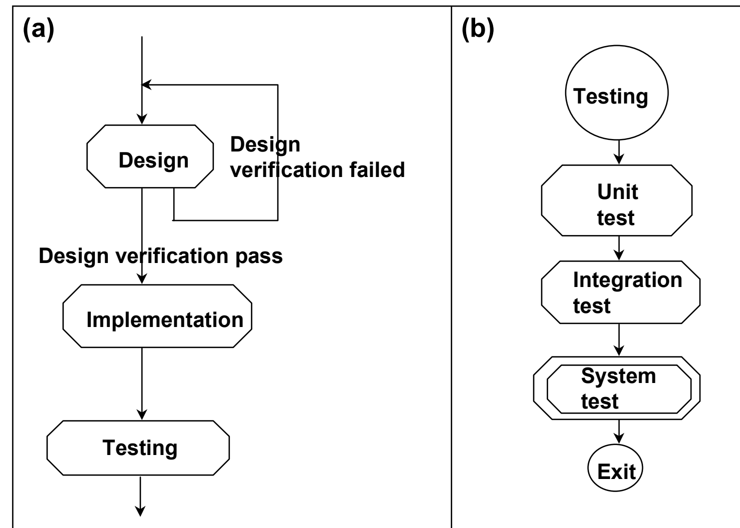


Figure 2. Activity modeling

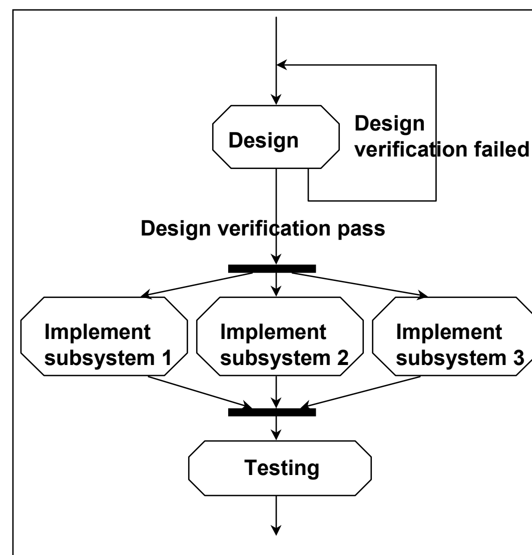
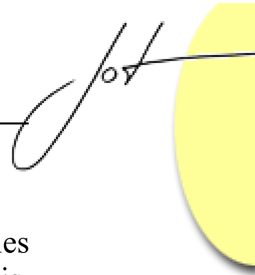


Figure 3. Concurrent activities and activity synchronization

2. Concurrent activities and activity synchronization. Sometimes, activities can be concurrently enacted. Moreover, concurrent activities may need to synchronize. We use the notation in Figure 1(e) to model concurrent activities and activity synchronization. Figure 3 depicts an example of using the notation. The figure



shows that after the design verification passes, three implementation activities start concurrently. After the implementations, the activity “Testing” starts. That is, the concurrent activities are synchronized before the enactment of the activity “Testing”.

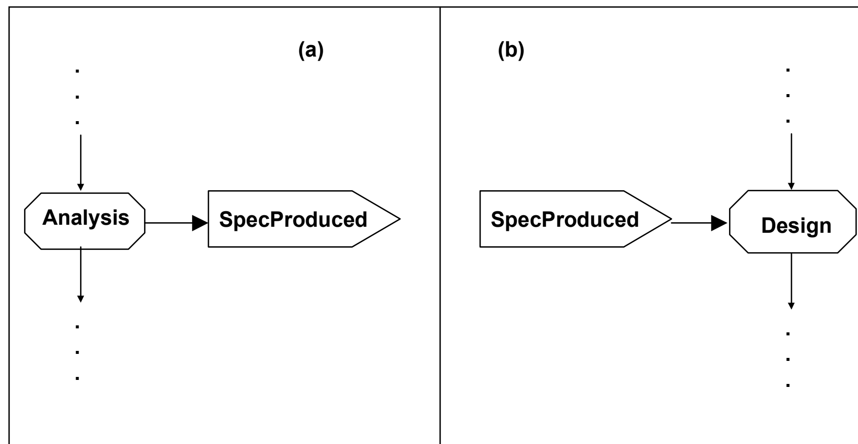


Figure 4. Communication between the activities “Analysis” and “Design”

3. Activity communication. An enacting activity may communicate with others. We use Figure 1(f) to model activity communication, which is accomplished through signaling and waiting for events. For example, Figure 4(a) shows that when the analysis activity finishes, it signals the event “SpecProduced”. Having detected that event, the design activity starts (see Figure 4(b)).
4. Exceptions. Exceptions refer to events that cannot be regularly controlled. For example, customer may change requirements any time during software development. Requirement change should thus be regarded as an exception. When an exception occur, the corresponding handler should be enacted. We use Figure 1(g) to model exceptions. Exception names are associated with the notation. Moreover, the arrow points to the handler of the exception. For example, in Figure 5, when the exception “Schedule overrun” occurs, the exception handler “Timeout handling” is executed.

A P-activity diagram can be constructed using the notations shown in Figure 1. Figure 6 shows an example P-activity diagram, which depicts a waterfall process for software development.

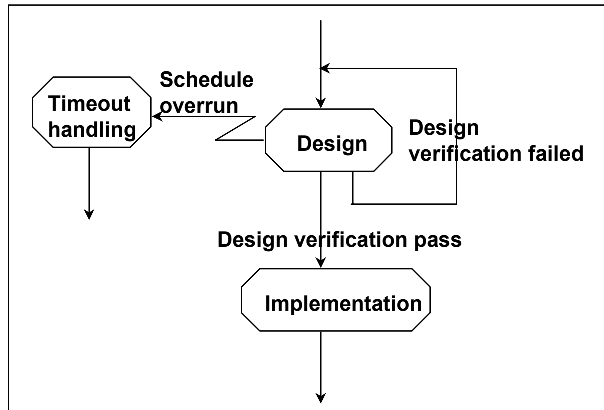


Figure 5. Exception

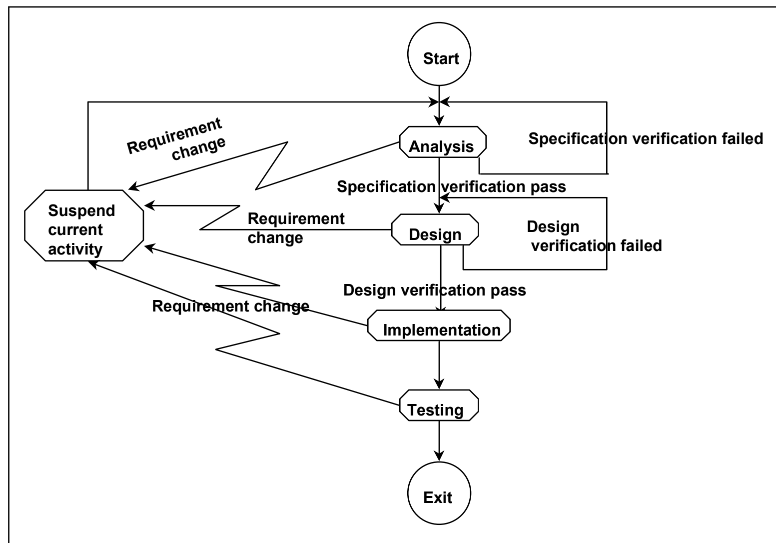
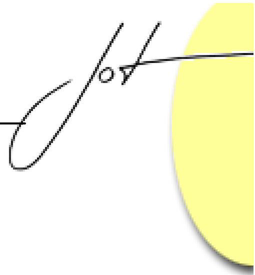


Figure 6. P-activity diagram example



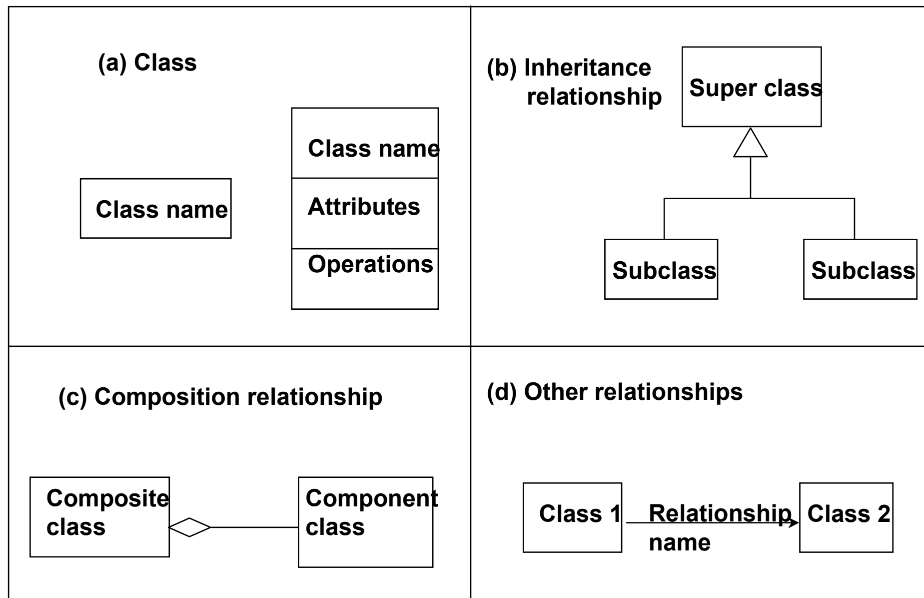


Figure 7. P-class diagram notations

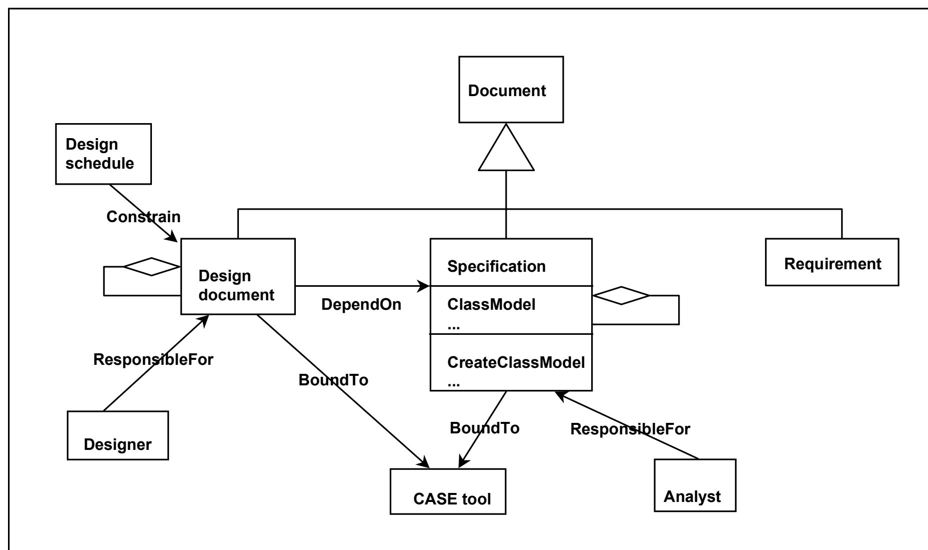


Figure 8. P-class diagram example

P-class diagram

The P-class diagram uses *classes* to model process components other than activities. It also models class relationships such as dependency, inheritance, and composition relationships among products.

Notations used in the P-class diagram are depicted in Figure 7. Figure 7(a) sketches the class notation. The left notation shows a class name only. It is used when class attributes and operations need not show. Otherwise, the right notation is used. Figure 7(b) sketches an inheritance relationship, where the super class is drawn on top of its subclasses. Figure 7(c) depicts a composition relationship, where the composite class appears next to the diamond shape. Figure 7(d) depicts the relationships other than the inheritance and composition relationships. The relationship name is marked along the arrow.

Figure 8 shows a P-class diagram that uses the notations in Figure 7. The figure depicts that the products "Specification", "Requirement", and "Design document" inherit the product "Document". The development of "Design document" is constrained by "Design schedule". And, the reflexive composition relationship associated with "Design document" and that with "Specification" means that those products can be decomposed into sub-products.

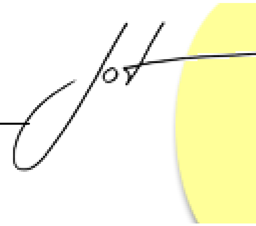
3 LOW LEVEL PROCESS LANGUAGE

The object-oriented low level process language models a process program as a set of classes. Figure 9 presents a subset of BNF grammars for the language. The grammars depict that a process program represented in the language is composed of one `Process` class and one or more other classes (grammar 1). The `Process` class defines *tasks* and exception handlers (grammar 3), in which a task is a collection of related activities. Note that the `start` task (grammar 9) is the entry point of a process program.

In addition to the `Process` class, the following classes can be used in a process program: product class, role class, schedule class, budget class, tool class, and relationship class. They can be instantiated to define corresponding instances. For example, a role class can be instantiated to define developers. In the language, only product classes and role classes can be defined as customized classes (grammar 2). The other classes are all built-in ones. The following subsections respectively describe process component modeling in the low level process language.

Product modeling

A product is defined by instantiating a product class. A product class defines its attributes, constructor, and operations other than the constructor (grammar 4). Example 1 depicts a product class `Specification`, which inherits the built-in class `Document` by using the keyword `extends`.



```
1. ProcessProgram ::= {Class}   ProcessClass
     /* A process program is composed of one "Process" class and one or more other classes. */
2. Class ::= ProductClass | RoleClass
     /* Only product classes and role classes can be defined as customized classes. */
     /* Other classes such as tools, schedules, and budgets are built-in classes. */
3. ProcessClass ::= "class Process"  "{"  (Data)  StartTask  {Task}  (ExceptionBlock)  "}"
     /* "Task" defines a task, which is a collection of related activities */
     /* "StartTask" is the entry point of a process program. "ExceptionBlock" defines an exception handler. */
4. ProductClass ::= ClassDef  "{"  (Attribute)  Constructor  (Operation)  "}"
     /* A class is composed of  attributes, a constructor, and operations. */
5. RoleClass ::= ClassDef  "{"  (Attribute)  Constructor  (Operation)  "}"
     /* An operation of a role class corresponds to an activity of a developer */
6. ClassDef ::= "class"   ClassName   ["extends"   ClassName]
     /* "extends" defines inheritance relationships*/
7. Constructor ::= ClassName  "("  (Parameter)  ")"  "{"  {Statement}  "}"
8. Operation ::= [DataType]  OperationName  "("  (Parameter)  ")"  "{"  {Statement}  "}"
9. StartTask ::= "start()"  "{"  {Statement}  "}"
     /* The entry point of a process program. */
10. Task ::= [DataType]   TaskName   "("  (Parameter)  ")"  "{"  {Statement}  "}"
11. Statement ::= Data | Relationship | ClassInstance| GeneralStatement| SyncStat
     /* "ClassInstance"  instantiates an instance from a class. */
     /* "Relationship" define a relationship among process components, such as products and tools */
12. GeneralStatement ::= ObjectOperationInvocation | WorkAssignment | Branch |Loop
     /* "SyncStat" is for synchronizing activities, including synchronous and asynchronous communication */
13. SyncStat ::= ConcurrencyBlock | EventStat
14. ExceptionBlock ::= "exception" ExceptionName "{" {Statement} "}"
```

(a)

Symbol	Meaning
::=	is defined as
|	alternative
[X]	zero or one instance of X
(X)	zero or more instance of X
{X}	one or more instance of X
/* . . . */	comments
un-quoted symbols	non-terminals
quoted symbols	terminals

(b)

Figure 9. A subset of BNF grammars for the low level process language. (a) The grammars. (b) Definition of symbols used.

An operation of a product class starts with its name, followed by the parameters it uses and then the statements implementing the operation (grammar 8). If necessary, an operation can return value by using the `return` statement. In this case, the type of return value is put in front of the operation name (see the `verify` operation in Example 1). The operation with the same name as the class is the constructor of that class, which normally assigns attribute values and establishes tool binding relationships.

A product class can be instantiated to define products using the following syntax:

```
instanceName is a className(paramenets);
```

For example, the following statement defines a product `systemSpec` which belongs to the product class `Specification`.

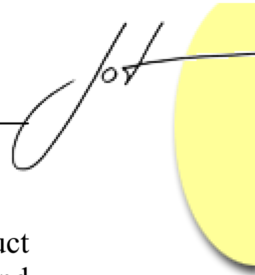
```
    systemSpec is a Specification("System specification",
        "Supermarket system", "systemSpec.doc",
        "systemSpec.ROSE", "WORD97", "ROSE98");
```

Example 1. Product class

```
class Specification extends Document {
    TextFile: TextSpec;
    NonTextFile: NonTextSpec;
    Tool: EditTool, DrawTool;

    Specification(String dName, String pName, String tFile,
        String nFile, String eTool, String dTool) {
        DocName = dName;
        ProjectName = pName;
        TextSpec is a TextFile(tFile);
        NonTextSpec is a NonTextFile(nFile);
        EditTool is a Tool(eTool);
        DrawTool is a Tool(dTool);
        TextSpec BoundTo EditTool;
        NonTextSpec BoundTo DrawTool;
    }

    edit(Analyst analyst1, Requirement req) {
        analyst1 develops thisProduct referring to req;
    }

    int verify(Analyst analyst1, Requirement reference_docu) {
        int: VerificationPass;
        analyst1 develops thisProduct referring to
            reference_docu;
        input "Verification Pass? (1: pass, 0: failed)",
            VerificationPass;
        return VerificationPass;
    }
}
```

The most important statement used in class operations is the work assignment statement, which assigns work to developers, requires developers to develop products, associates limits to the development, and indicates products for reference. It has the following syntax:

```
    developer_name develops product_name referring to reference_list
        with limits schedule_name, budget_name;
```



The statement requires the developer developer_name to develop the product product_name. The schedule and budget limits are respectively schedule_name and budget_name. Moreover, the products to be referred to are listed in reference_list. The following example depicts a work assignment statement:

```
    analyst1 develops thisProduct referring to req;
```

This statement requires analyst1 to develop thisProduct, and indicates that the product for reference is req. The keyword thisProduct used in the statement, which resembles the keyword this in JAVA, indicates that the product to be developed is the very instance instantiated from the product class.

Developer modeling

A developer is defined by instantiating a *role class*. A role class defines its attributes, constructor, and operations other than the constructor (grammar 5). Example 2 depicts a role class Analyst, which inherits the built-in class Role.

Example 2. Role class

```
class Analyst extends Role {
    Analyst(String ipAdd, String email, String dName,String
        rName){
        IpAddress = ipAdd;
        EmailAddress = email;
        DeveloperName = dName;
        RoleName = rName;
    }

    EditSpec(Requirement req, Specification spec) {
        spec.edit(thisDeveloper,req);
    }
}
```

The operations of a role class can be defined similar to those of a product class (grammar 8), in which the one with the same name as the role class is the class's constructor. Each operation other than the constructor models an activity assigned to the role. For example, the "analyze requirements and develop a specification" activity of an analyst is modeled as the EditSpec operation in Example 2. Since an activity normally requires a developer to develop a product, statements in the operations of a role class generally invoke operations of product classes. For example, the operation EditSpec of the role class "Analyst" is accomplished by invoking the statement spec.edit(thisDeveloper,req);. Note that the keyword thisDeveloper indicates that the developer involved in the statement is the very developer instantiated from the role class.

Tool modeling

Tools are modeled using the built-in class `Tool`, which possesses the attribute `ToolName`. The following statement defines a tool `EditTool` with the name `WORD97`.

```
EditTool is a Tool("WORD97");
```

When the tool `EditTool` is used, `WORD97` will be invoked for developers to use.

Schedule and budget modeling

Schedules and budgets are modeled using the built-in classes `Schedule` and `Budget`, which are generally used to limit activities. A schedule is defined using the following syntax:

```
schedule_name is a Schedule(deadline);
```

For example, the statement `Analysis_schedule is a Schedule("2001/12/31");` defines a schedule `Analysis_schedule` with the deadline December 31, 2001.

To define a budget limit, the following syntax is used:

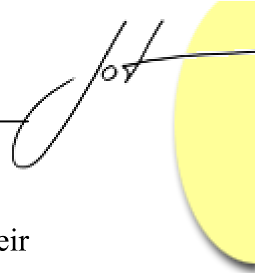
```
budget_name is a Budget(maximal available budget);
```

For example, the statement `Analysis_budget is a Budget(200);` defines a budget `Analysis_budget` with a maximal available budget of US$ 200.

Activity modeling

Activities assigned to a role are specified in a role class (see section 3.2). Nevertheless, role classes cannot model activity sequence and synchronization. The proposed language models that sequence and synchronization in the `Process` class, inside which related activities and their sequence and synchronization are modeled as a task (i.e., an operation of the `Process` class). As shown in grammar 3 of Figure 9, there is one `start` task as the entry point of the program. There are also other tasks, in which each task models one P-activity diagram of the high level process model.

Various statements are provided for tasks (grammars 11 and 12). Among them, those for describing concurrent activities, activity synchronization, and activity communication are the most important. The proposed language provides concurrency block to describe concurrent activities and activity synchronization, and provides event statements for activity communication (grammar 13). The concurrency block and event statements are described below.



1. Concurrency block. Concurrency block models concurrent activities and their synchronization. The block has the following syntax:

```
concurrent {
    statement block 1;
    statement block 2;
    . . .
}
next_statements;
```

   The above block depicts that the statement blocks `statement block 1`, `statement block 2`, and so on are concurrently enacted. Note that statements inside `statement block 1` and `statement block 2` are executed sequentially. The above block also depicts that the statements `next_statements` can be enacted only when the statement blocks `statement block 1`, `statement block 2`, and so on have been finished. That is, the concurrent activities should be synchronized before the enactment of `next_statements`.
   Concurrency block can also be used to model multiple developers in developing the same product concurrently. For example, the following concurrency block depicts that the developers `analyst1` and `analyst2` concurrently develop the product `subSpec` by performing the activity `EditSpec`. In this case, the analysts should cooperate on the development.

```
concurrent {
     analyst1.EditSpec(req, subSpec);
     analyst2.EditSpec(req, subSpec);
}
```

2. Event statements. Event statements model activity communication. We offer two event statements, namely signaling and waiting for events. Example 3 depicts the communication between the analysis and design activities. Statements in the example depict that when a specification is produced by the `Analysis` activity, the `SpecProduced` event is signaled (i.e., the flag `SpecProduced` is set TRUE). With this, the `waitfor` statement in the `Design` activity will detect the event and triggers the design activity to develop a design document.

   Example 3. Activity communication

```
class Process {
 start() {
  Event SpecProduced = FALSE;
  . . . .
  concurrent {
     Analysis(. . . .);
     Design(. . . .);
  }
```

```
    . . . .
  }

  Analysis(. . . .) {
    . . . .
    // Develop a specification here
    signal SpecProduced;
    . . . .
  } // end of Analysis

  Design(. . . .) {
    . . . .
    waitfor SpecProduced;
    // Develop a design document for the specification here
    . . . .
  } // end of Design
  . . . .
} // end of Process class
```

Relationship modeling

Relationships are defined by instantiating *relationship classes*. The proposed language provides the following built-in relationship classes:

1. The class `PartOf` establishes decomposition relationships between a product and its sub-products. It can be instantiated using the following syntax:

   ```
   sub-product PartOf product;
   ```

2. The class `ResponsibleFor` establishes responsibility relationships between developers and products. It can be instantiated using the following syntax:

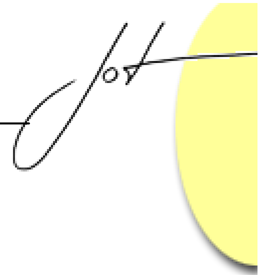
   ```
   developer ResponsibleFor product;
   ```

3. The class `BoundTo` establishes binding relationships between tools and products. It can be instantiated using the following syntax:

   ```
   product BoundTo tool;
   ```

Exceptions and their handlers

Exceptions and their handlers are specified inside the `start` operation of the `Process` class (see the "ExceptionBlock" in grammars 3 and 14). For example, the following statements define an exception `RequirementChange` and its handler:



```
exception RequirementChange {
    // suspend all the current work
    allDevelopers halt;
    // change the requirement document
    analyst1.ChangeReq();
    // restart the process
    . . . .
} // end of exception "RequirementChange"\
```

Mapping

The mapping between the high level UML-based diagrams and the low level process language is tabulated in Table 1. From the table, one can see that the mapping is clear. Therefore, transforming high level diagrams into a process program is straightforward. This facilitates process program development.

Table 1. Mapping between high level diagrams and low level language

High level UML-based diagram constructs		Low level process language constructs
P-class diagram	role class	role class
	tool class	tool class
	budget class	budget class
	schedule class	schedule class
	software product class	product class
	relationship	relationship classes
P-activity diagram	activity	Operations of the " Process" class
	concurrent activities and activity synchronization	concurrency block
	activity communication	Waiting for and signaling events
	exception	exception block

4 EXAMPLE

A process to analyze and design a system is used as an example. Suppose that the system is decomposed into two sub-systems. Activities of the process are shown in the upper portion of Figure 10. Flow of the activities is sketched in the lower portion of the figure,

where arrows dictate activity sequence. Activities that are not linked by arrows can be enacted in parallel. For example, “a” and “b” can be enacted in parallel. Moreover, conditions are marked on the lines.

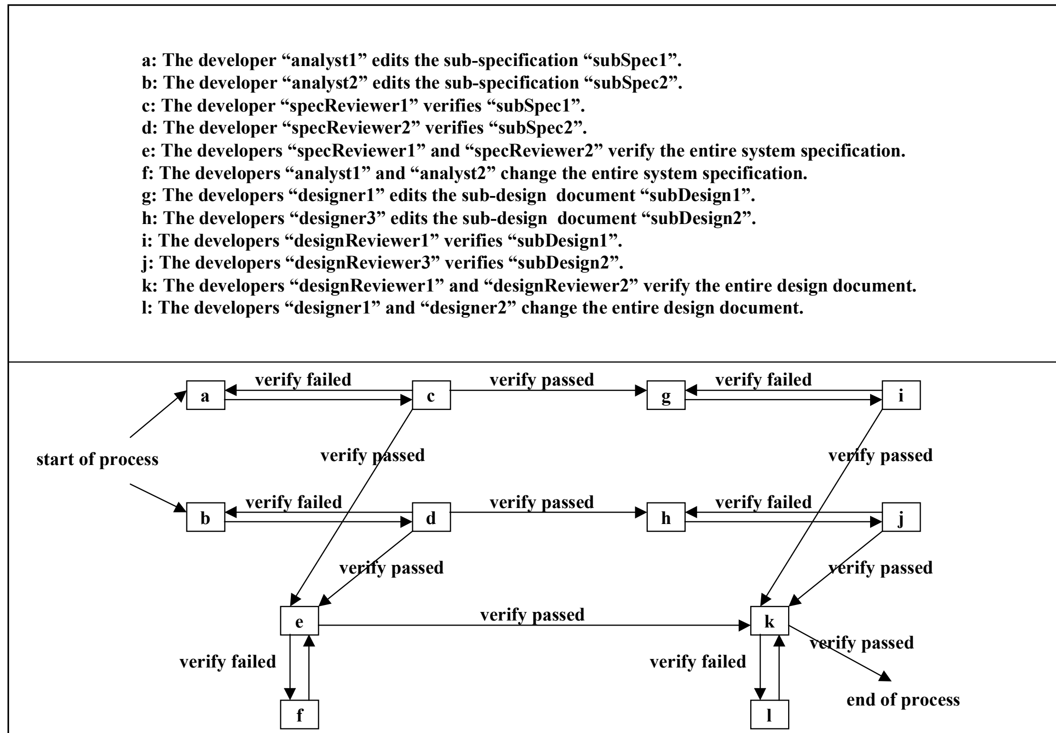
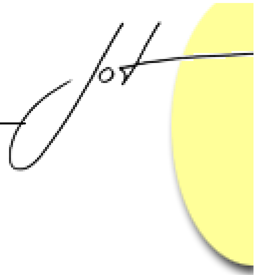


Figure 10. Process used in the example

After analyzing and designing the process, the P-class diagram (Figure 11) and the P-activity diagrams (Figures 12 through 14) are obtained. Figure 12 shows the top level P-activity diagram, in which the activity “Analyze and design subsystem 1” and “Verification” are further decomposed into the P-activity diagrams as shown in Figures 13 and 14, respectively. Moreover, the activity “Analyze and design subsystem 2” are decomposed into a P-activity diagram similar to that in Figure 13. Note that the exception “Requirement change” is also modeled in Figure 12.

The diagrams are then used to implement the process program as shown in APPENDIX I, in which classes are defined first, followed by activities specified in the `Process` class. Each operation in the `Process` class models a P-activity diagram. The P-activity diagram in Figure 12 is modeled as the “start” operation, which is the entry point of the process program. That operation concurrently starts the following three activities (which are further decomposed): `AnalyzeAndDesignSubsystem1`, `AnalyzeAndDesignSubsystem2`, and `Verification`. Each of those activities is then modeled as an operation of the class `Process`. To improve the readability of the process program, unimportant details have been replaced by “`. . .`”.



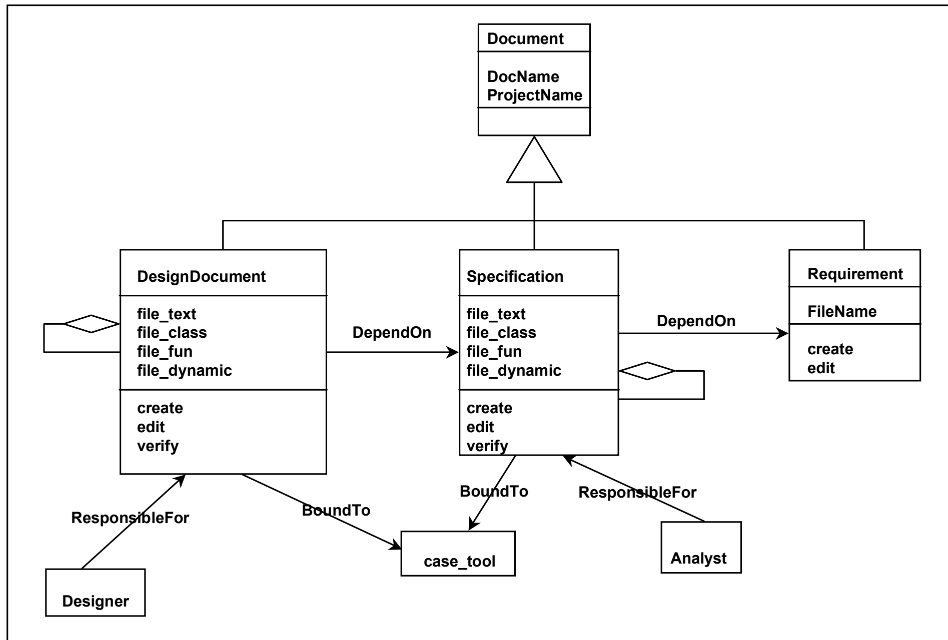


Figure 11. P-class diagram

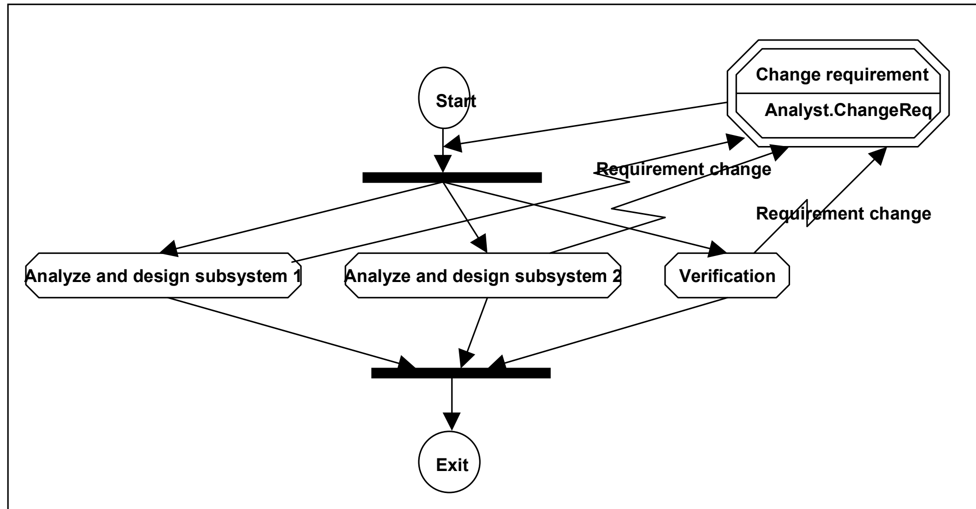


Figure 12. Top level P-activity diagram

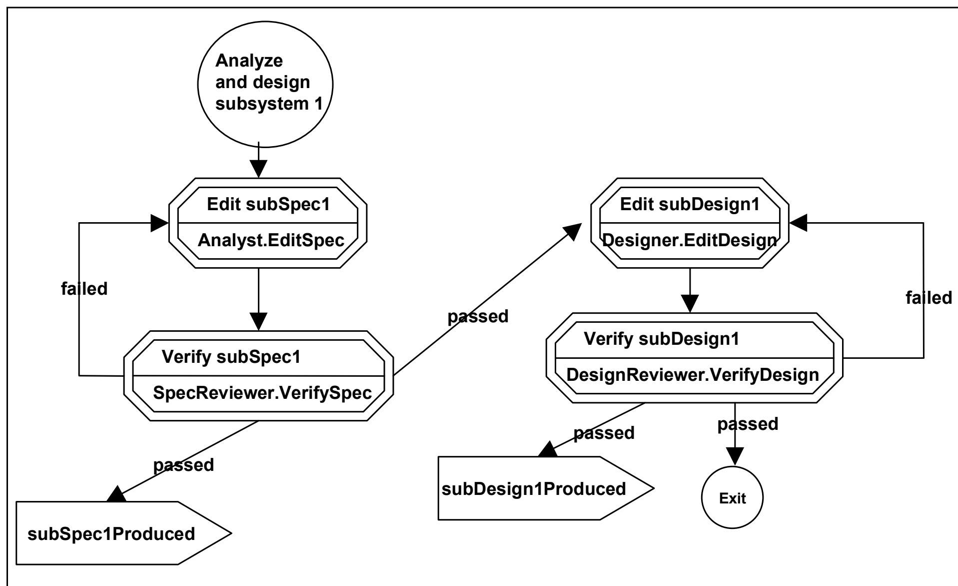


Figure 13. P-activity diagram for “Analyze and design subsystem 1”

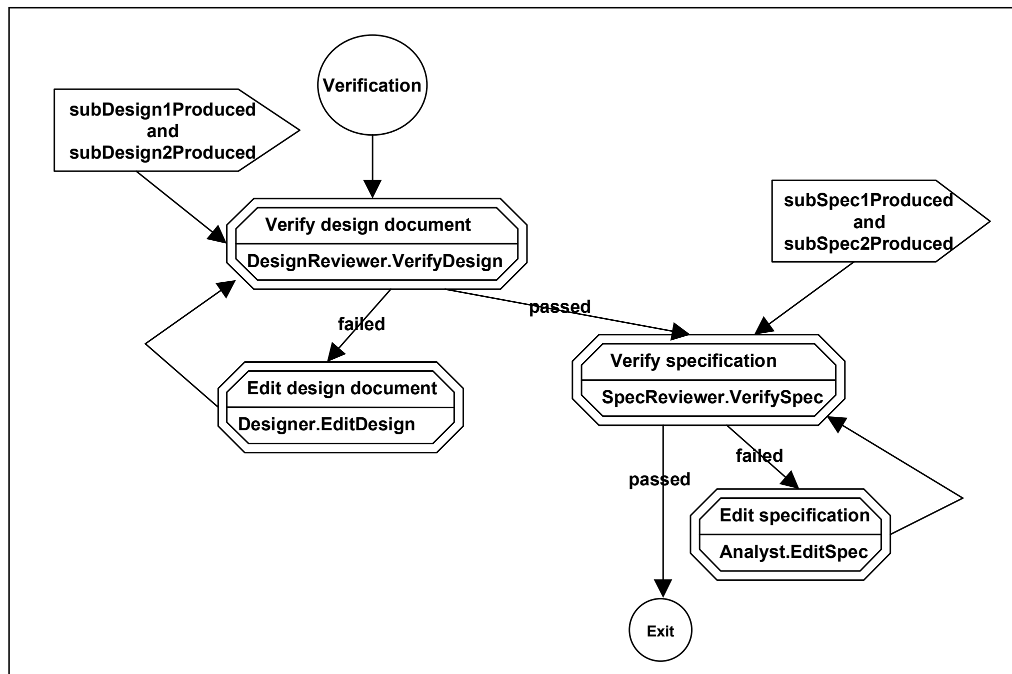


Figure 14. P-activity diagram for "Verification"

5 CONCLUSIONS

This article proposes a process modeling language, which is composed of high level UML-based diagrams and a low level process language. The high level diagrams can be used to analyze and design processes. This facilitates process program development. The diagrams are composed of P-activity diagrams and a P-class diagram. P-activity diagrams model activities, activity sequence, activity synchronization, and exceptions. The P-class diagram models products, roles, tools, schedules, budgets, and their relationships. The objet-oriented low level process language models processes as process programs. Between the two levels is a mapping, which facilitates transforming the high level diagrams into a process program. This further facilitates process program development.

To develop the process program of a process using the proposed modeling language, the process is first analyzed and designed. The results obtained are represented in P-activity diagrams and a P-class diagram. The diagrams are then transformed into a process program by referring to the mapping mentioned above. The proposed process modeling language offers the following features:

1. It facilitates process program development by providing high level UML-based diagrams and a clear mapping between the high level diagrams and the low level process language.
2. It models all necessary process components including products, developers, activities, activity sequence and synchronization, exceptions and their handlers, tools, schedules, budgets, and relationships among process components.

6 ACKNOWLEDGMENT

This research is sponsored by the National Science Council in Taiwan under Grant Number NSC89-2213-E-259-012. Special thanks are given to Professor Jen-Yen Chen in Department of Computer Science and Information Engineering, National Central University, Taiwan for his valuable comments.

REFERENCES

[Bandinelli93] Sergio C. Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi, "Software Process Model Evolution in the SPADE Environment", *IEEE Transactions on Software Engineering*, vol. 19, no. 12, pp.1128-1144, Dec. 1993.

[Belkhatir94] N. Belkhatir and W. L. Melo, "Supporting Software Development Process in Adele 2", *The Computer Journal*, vol. 37, no. 2, pp. 621-628, 1994.

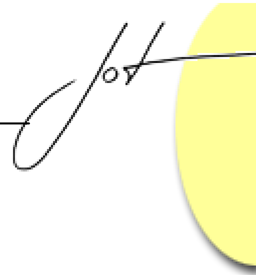
[Booch96] Grady Booch, *Object-oriented Analysis and Design with Applications, 2nd ed.*, The Benjamin/Cummings Publishing Company, 1996.

[Chen97] Jen-Yen Jason Chen, "CSPL: An Ada95-like, Unix-based Process Environment", *IEEE Transactions on Software Engineering*, vol. 23, no. 3, pp. 171 - 184, March 1997.

[Chou00] Shih-Chien Chou and Jen-Yen Jason Chen, "Process Program Development Based on UML and Action Cases, Part 1: the Model", *Journal of Object-Oriented Programming*, vol. 13, no. 2, pp. 21-27, 2000.

[Doppke98] John C. Doppke, Dennis Heimbigner, and Alexander L. Wolf, "Software Process Modeling and Execution within Virtual Environments", *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 1 pp.1-40, Jan. 1998.

[Feiler93] Peter H. Feiler and Watts S. Humphrey, "Softawre Process Development and Enactment: Concepts and Definitions", in *Proceedings of the 2nd*



International Conference on Software Process, pp.28-40, Los Alamitos, Calif., 1993.

[Fowler97] Martin Fowler and Kendall Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.

[Garg96] Pankaj K. Garg and Mehdi Jazayeri, *Process-Centered Software Engineering Environments, p. 17*, IEEE Computer Society Press, 1996.

[Heimann97] Peter Heimann, Carl-Arndt Krapp, and Bernhard Westfechtel, "Graph-Based Software Process Management", *International Journal on Software Engineering and Knowledge Engineering*, vol. 7, no. 4, pp.431-455, 1997.

[Iida93] Hajimu Iida, Kei-ichi Mimura, Katsuro Inoue and Koji Torii, "Hakoniwa: Monitor and Navigation System for Cooperative Development Based on Activity Sequence Model", in *Proceedings of the 2nd International Conference on the Software Process*, IEEE Computer Society, pp. 64-74, 1993.

[Jaccheri93] Maria Letizia Jaccheri and Reidar Conradi, "Technoques for Process Model Evolution in EPOS", *IEEE Transactions on Software Engineering*, vol. 19, no. 12, pp.1145-1156, Dec. 1993.

[Perry91] D. E. Perry, "Policy-Directed Coordination and Cooperation", in *Proceedings of the 7th Software Process Workshop*, Yountville, CA, pp. 111-113, Oct. 1991.

[Peuschel92] B. Peuschel and W. Schafer, "Concepts and Implementation of Rule-based Process Engine", in *Proceedings of the 14th International Conference on Software Engineering*, pp. 262-279, 1992.

[Sutton95] S.M. Sutton Jr., D. Heimbigner and L.J. Osterweil, "APPL/A: A Language for Software Process Programming", *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 3, pp. 221-286, 1995.

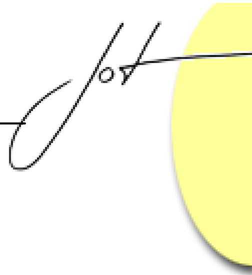
About the author

Shih-Chien Chou received a Ph. D. degree from the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan. He is currently an associated professor in the Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. His research interests include software engineering, process environment, software reuse, and information flow control. He can be contacted through the e-mail address scchou@mail.ndhu.edu.tw.

APPENDIX:

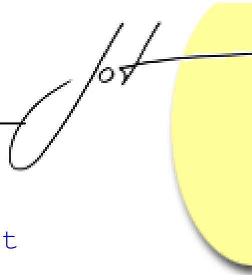
An example process program for the analysis and design process described in section 4 is shown here.

```
class Requirement extends Document {
    // attributes below
    TextFile: ReqFile;
    Tool: EditTool;
    // constructor
    Requirement( String dName, String pName, String fName, String
    eTool) {
        DocName = dName;
        // initialize files and tools used by the product
        ProjectName = pName;
        ReqFile is a TextFile(fName);
        EditTool is a Tool(eTool);
        ReqFile BoundTo EditTool;
    }

    change(Analyst analyst1) {
        analyst1 develops thisProduct;
    }
}

class Specification extends Document {
    // attributes
    . . . .
    Specification(String dName, String pName, String tFile,
        String nFile, String eTool, String dTool) {
        // initialize files and tools used by the product
        . . . .
    }

    edit(Analyst analyst1, Requirement req) {
        analyst1 develops thisProduct referring to req;
    }

    int verify(SpecReviewer specReviewer1, Requirement
        reference_docu) {
        int VerificationPass;
        specReviewer1 reviews thisProduct referring to
            reference_docu;
```



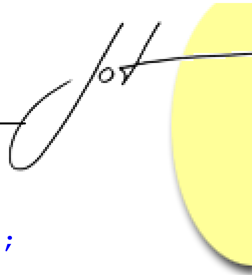
```
        input "Verification Pass? (1: pass, 0: failed)",
            VerificationPass;
        return VerificationPass;
    }
}

class DesignDocument extends Document {
    // attributes
    . . . .
    DesignDocument(String dName, String pName, String tFile,
        String nFile, String eTool, String dTool) {
        // initialize files and tools used by the product
        . . . .
    }

    edit(Designer designer1, Specification spec) {
        designer1 develops thisProduct referring to spec;
    }

    int verify(DesignReviewer designReviewer1, Specification
        spec) {
        int VerificationPass;
        designReviewer1 reviews thisProduct referring to spec;
        input "Verification Pass? (1: pass, 0: failed)",
            VerificationPass;
        return VerificationPass;
    }
}

class Analyst extends Role {
    Analyst(String ipAdd, String email, String dName,String
        rName){
        // initialize the attribute of an analyst
        IpAddress = ipAdd;
        EmailAddress = email;
        DeveloperName = dName;
        RoleName = rName;
    }

    ChangeReq(Requirement req) {
        req.change(thisDeveloper);
    }

    EditSpec(Requirement req, Specification spec) {
        spec.edit(thisDeveloper,req);
    }
} // end of class Analyst
```

```
class SpecReviewer extends Role {
    SpecReviewer(String ipAdd, String email, String dName,String
        rName){
        // initialize the attribute of a specification reviewer
        . . . .
    }

    int VerifySpec(Requirement req, Specification spec){
        return spec.verify(thisDeveloper,req);
    }
}
class Designer extend Role {
    Designer(String ipAdd, String email, String dName,String
        rName){
        // initialize the attribute of a designer
        . . . .
    }
    EditDesign(Specification spec, DesignDocument designDoc) {
        design.edit(thisDeveloper, spec, design);
    }
}

class DesignReviewer extend Role {
    DesignReviewer(String ipAdd, String email, String
        dName,String rName){
        // initialize the attribute of a design reviewer
        . . . .
    }

    int VerifyDesign(Specification spec, DesignDocument
        designDoc) {
        return designDoc.verify(thisDeveloper, spec);
    }
}

// Activities are specified as follows:
// Each P-activity diagram is modeled as an operation of the
class "Process"
class Process {
    // developers
    analyst1 is a Analyst("203.64.100.125",
        "yychuang@csie.ndhu.edu.tw", "Y. Y.Y.Chuang", "analyst");
    // other developers declared below, including analyst,
        designer1, and so on
    . . . .

    // products
```



```
systemReq is a Requirement("System requirement", "Supermarket
    system", "systemReq.doc", "WORD97");
// other products declared below, including systemSpec,
    subSpec1,
//    systemDesign, and so on
. . . .

// part-of relationships
subspec1 PartOf systemSpec;
subSpec2 PartOf systemSpec;
subDesign1 PartOf systemDesign;
subDesign2 PartOf systemDesign;

// events, for asynchronous communication
event subSpec1Produced, subSpec2Produced, subDesign1Produced,
    subDesign2Produced;

// "start" is the starting task
start() {
    concurrent {
        AnalyzeAndDesignSubSystem1();
        AnalyzeAndDesignSubSystem2();
        Verification();
    }

    // Exception handlers
    exception RequirementChange {
        // suspend all the current work
        allDevelopers halt;
        // change the requirement document
        analyst1.ChangeReq(systemReq);
        // restart the process
        concurrent {
            AnalyzeAndDesignSubSystem1();
            AnalyzeAndDesignSubSystem2();
            Verification();
        }
    } // end of exception
} // end of "start" task

AnalyzeAndDesignSubsystem1() {
    int: verificationPass = 0;
    while (verificationPass == 0) {
      analyst1.EditSpec(systemReq, subSpec1);
      verificationPass = specReviewer1.VerifySpec(systemReq,
        subSpec1);
    }
    signal subSpec1Produced;
```

```
    verificationPass=0;
    while (verificationPass == 0) {
      designer1.EditDesign(subSpec1, sunDesign1);
      verificationPass = designReviewer1.VerifySpec(subSpec1,
        subDesign1);
    }
    signal subDesign1Produced;
} // end of AnalyzeAndDesignSubSystem1

AnalyzeAndDesignSubsystem2() {
    // similar to AnalyzeAndDesignSubSystem1
}

Verification(){
  int verificationPass;
  waitfor subSpec1Produced and subSpec2Produced;
  concurrent { // two reviewers cooperate to verify the
    specification

    verificationPass=specReviewer1.VerifySpec(systemReq,syste
    mSpec);

    verificationPass=specReviewer2.VerifySpec(systemReq,syste
    mSpec);
  }
  while verificationPass == 0 {// verify and edit until
    verification passed
    concurrent {// if verification failed, edit the
    specification
      analyst1.EditSpec(systemReq,systemSpec);
      analyst2.EditSpec(systemReq,systemSpec);
    }
    concurrent {// re-verify

    verificationPass=specReviewer1.VerifySpec(systemReq,syste
    mSpec);

    verificationPass=specReviewer2.VerifySpec(systemReq,syste
    mSpec);
    }
  } // end of while
  waitfor subDesign1Produced and subDesign2Produced;
  verificationPass=0;
  concurrent { // two reviewers cooperate to verify the
    design document
    verificationPass =
       designReviewer1.VerifyDesign(systemSpec,systemDesign);
    verificationPass =
```



```
             designReviewer2.VerifyDesign(systemSpec,systemDesign);
        }
        while verificationPass == 0 {// verify and edit until
          verification passed
          concurrent {// if verification failed, edit the design
          document
            designer1.EditDesign(systemSpec,systemDesign);
            designer2.EditDesign(systemSpec,systemDesign);
          }
          concurrent {// re-verify
            verificationPass =

          designReviewer1.VerifyDesign(systemSpec,systemDesign);
            verificationPass =

          designReviewer2.VerifyDesign(systemSpec,systemDesign);
          }
        } // end of while
} // end of class "Process"
```