1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

A HIGH LEVEL PROCESS MODELING TECHNIQUE BASED ON UML AND ACTION CASES

Shih-Chien Chou
Department of Information Management
Minghsin Institute of Technology
Hsinfong, Taiwan
E-mail: lvscchou@ms15.hinet.net

Jen-Yen Jason Chen
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan
E-mail: jychen@csie.nctu.edu.tw

ABSTRACT

This paper describes the high level process modeling technique, which is composed of a high level process model and a meta-process to represent processes in that model. The high level model is designed based on the class diagram and activity diagram of the unified modeling language (UML). The meta-process is designed based on action cases, which are similar to use cases.

1. INTRODUCTION

Software processes (software development processes) are becoming complicated. To facilitate their control, *process-centered software engineering environments* (PSEE) have been developed [1-16]. A PSEE provides a *process language* to write *process programs* which can be *enacted* (executed) in the PSEE.

To control software processes by a PSEE, process programs should be developed first. Since most current PSEEs do not well facilitate process development, process programs are generally implemented with few or no design work. This, however, is not feasible for complicated processes, because large-sized process programs are difficult to implement without analysis and design work. To remedy that, process programs, like software, can be developed by following a software engineering procedure, because software processes are also software [17]. That is, processes can be analyzed and designed before implementation [18]. To support that, a PSEE should facilitate both high level and low level process modeling. With *high level modeling*, a process is analyzed, designed, and represented in a *high level process model*. The high level process model is then used in *low level modeling*, in which process programs are implemented in a *low level process model*.

We have developed a PSEE called CSPL (concurrent software process language) environment [4, 19-20] which provides an Ada-like CSPL process language. The CSPL language is good at low level process modeling. However, it is week at high level modeling. To remedy that, we have designed a high level modeling technique which includes: 1) a high level process model and 2) a meta-process for representing processes in that model. The model is designed based on the class diagram and activity diagram of the unified modeling language (UML) [21]. The meta-process is designed based on *action cases*, which are similar to use cases [22]. See section 3.1 for the action case definition. The CSPL high level modeling technique offers the following features:

1) It provides constructs to model necessary process components

   Necessary processes components should be modeled so that they can be controlled during process enactment. Generally, a software process contains the following necessary components: a) activities, b) activity sequence and synchronization, c) exceptions and their handlers, d) software products, e) developers, f) tools, and g) relationships among software products, developers, and tools. All those components can be modeled in the CSPL high level process model.

2) It is expected to be easy to follow.

   The meta-process of the technique is designed based on action cases. Since action cases are similar to the well known use cases, the technique is expected to be easy to follow.

3) It facilitates process program maintenance.

   Process programs, like software, need to maintain. An easy-to-maintain process program should be modular and easy to understand. The CSPL high level process model is based on the UML, which is object-oriented (O-O). As agreed, O-O software is modular and easy to understand. Therefore, the CSPL high level process model facilitates process program maintenance.

This paper describes high level process modeling in CSPL environment. Section 2 describes the CSPL high level process model. Section 3 describes the meta-process. Finally, section 4 gives the conclusions.

2. CSPL HIGH LEVEL PROCESS MODEL

The high level process model is used to model process components including: 1) activities, 2) activity sequence and synchronization, 3) exceptions and their handlers, 4) software products, 5) developers, 6) tools, and 7) the relationships among software products, developers, and tools. The model is designed based on UML. Among the UML notations, the activity diagram is slightly modified to model the first three process components mentioned above, and the class diagram is extended to model the other process components. The modified class diagram is called the *CSPL class diagram* and the extended activity diagram is called the *CSPL activity diagram*. They are

described in the following subsections.

2.1 CSPL class diagram

The CSPL class diagram models the following process components: software products, developers, tools, and roles. They are modeled as classes. Moreover, relationships among the classes are also modeled. Important relationships modeled are described below:

1) Responsibility relationships between developers and products.
   Developers are responsible for the products they developed. If a product needs be modified, the responsible developers should be consulted. Maintaining responsibility relationships allows better control over developers.
2) Binding relationships between products and tools.
   A product developed by a specific tool should be operated on by that tool, because different software tools use different formats.
3) Decomposition relationships between software products and their sub-products.
   Software systems are often decomposed into subsystems during development. The decomposition relationships should thus be modeled.
4) Dependency relationships among software products.
   Software products may depend on others. For example, the design document of a system depends on the system's specification. Managing dependency relationships facilitate keeping consistency among software products. For example, when a specification is changed, its corresponding design document and program code, which should also be changed, can be identified by tracing the dependency relationships.
5) Inheritance relationships among software products.
   Inheritance relationships are important in an O-O model. They should thus be modeled.

Notations used in the CSPL class diagram are depicted in Figure 1. Figure 1(a) sketches the notations for classes. The left one displays only a class name. It can be used when class attributes and operations need not be shown. The right notation is used when class attributes and operations should be shown. Class operations will be invoked by activities. Figure 1(b) sketches an inheritance relationship, where the super class is drawn on top of its subclasses. Figure 1(c) depicts a composition relationship, where the composite class is next to the diamond shape. Figure 1(d) depicts relationships other than the inheritance and composition relationships. The relationship name is marked on the line. The arrow head indicates source and sink classes of the relationship. For example, Figure 2 sketches a dependency relationship between the classes "Specification" and "Design document". The arrow head indicates that "Design document" depends on "Specification". Figure 3 is an example CSPL class diagram.

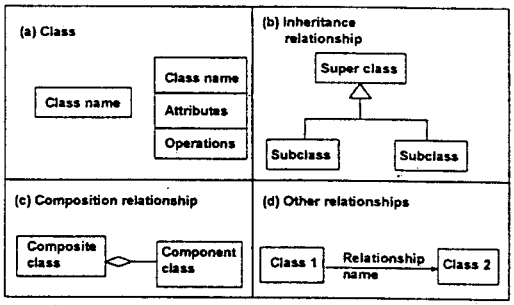

Figure 1. CSPL class diagram notations

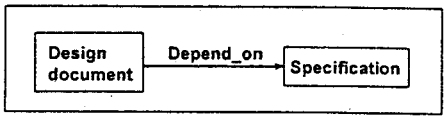

Figure 2. Dependency relationship

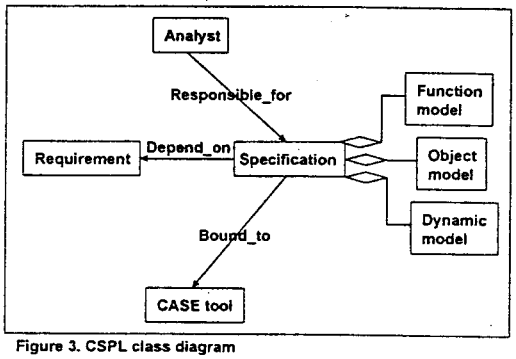

Figure 3. CSPL class diagram

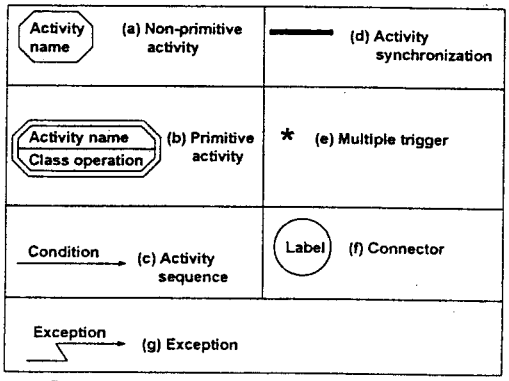

Figure 4. CSPL activity diagram notations

2.2 CSPL activity diagram

The CSPL activity diagram models the following process components: 1) activities, 2) activity sequence and synchronization, and 3) exceptions and their handlers. It also shows the invoking relationships between activities and class operations. Notations used in the CSPL activity diagram are shown in Figure 4. The notations are described below:

1) The notation in Figure 4(a) models non-primitive activities. A non-primitive activity is a complicated activity that can be decomposed into more detailed ones.
2) The notation in Figure 4(b) models primitive activities, which are normally easy to control and need not be decomposed. Some primitive

activities can be accomplished by invoking class operations. Note that this notation is not included in the UML activity diagram.

The notation in Figure 4(b) is partitioned into two fields, where the first field shows the activity name. If the activity is accomplished by invoking a class operation, the operation name is placed in the second field. Placing class operations in primitive activities shows the invoking relationships between activities and class operations.

3) The notation in Figure 4(c) models activity sequence. That is, for the activities connected by arrow-headed lines, the successors can be started only when the predecessors are finished. Conditions can be associated with the line. For example (see Figure 5), after the analysis activity, if the specification verification passed, the design activity can be started. Otherwise, the analysis activity should be redone.

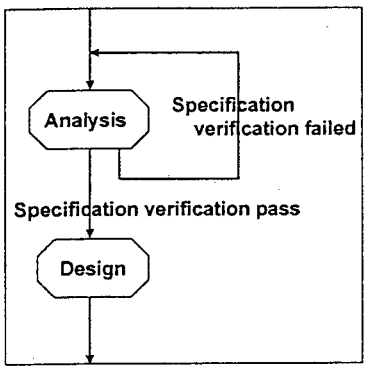

Figure 5. Conditions

4) The notation in Figure 4 (d) models activity synchronization. For example, in Figure 6, after the specification verification passes, three design activities, namely "Desigh subsystem 1", "Design subsystem 2", and "Design subsystem 3", are started concurrently. After the three activities are all finished, the activity "Verify design" can be started.
5) The notation in Figure 4(e) models multiple triggers. For example, in Figure 7, after the specification verification passed, multiple design activities are started concurrently, where each activity designs a subsystem. After the design activities are all finished, the activity "Verify design" can be started.
6) The notation in Figure 4(f), which denotes a *connector*, is used to connect CSPL activity diagrams located in different sheets. It is also used to show the decomposition relationships among activities. For example, Figure 8 shows the activity diagram obtained by decomposing the activity "Analysis" in Figure 7. Naming the starting connector as "Analysis" shows the activity decomposition relationships. Note that this notation is not included in the UML activity diagram.
7) The notation in Figure 4(g) models exceptions. Exception names are associated with the notation. In addition, the arrow head points to the handler of the exception, which can be a sequence of activities or a connector connecting to a CSPL activity diagram. For example, in Figure 9, when the exception "Requirement change" occurs, the activity "Suspend design" is executed, then the analysis activity is restarted. When the exception "Schedule overrun" occurs, the exception handler "Timeout handling" is executed. Note that the notation for exceptions is not included in the UML activity diagram.

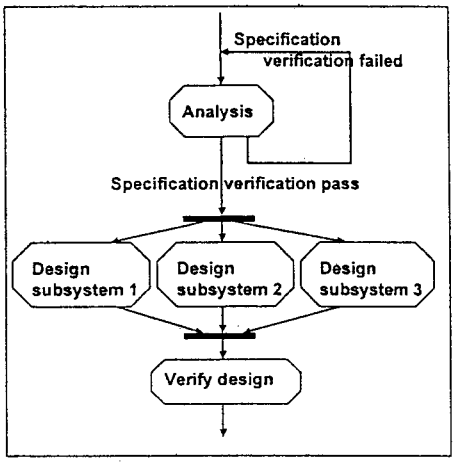

Figure 6. Activity Synchronization

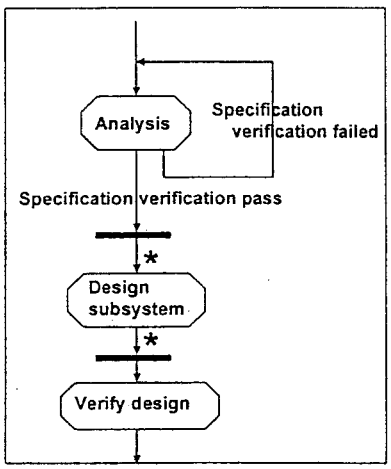

Figure 7. Multiple triggers

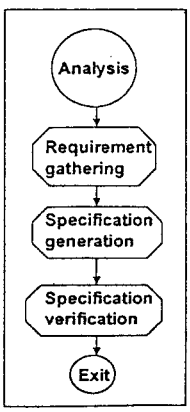

Figure 8. Activity decomposition

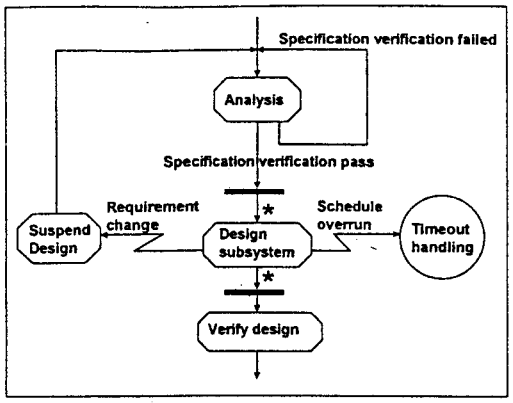

Figure 9. Exceptions

A CSPL activity diagram can be constructed using the notations shown in Figure 4. Figure 10 is an example CSPL activity diagram.

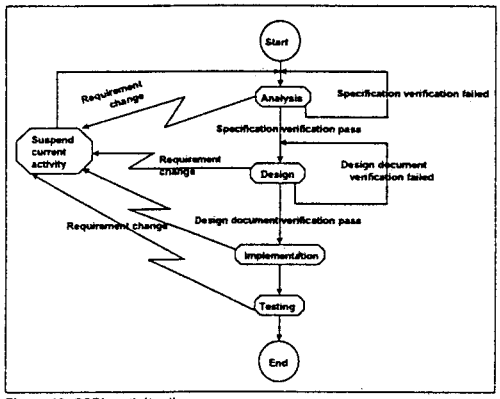

Figure 10. CSPL activity diagram

3. META-PROCESS

To represent processes in a high level model, process components as described in section 2 are identified first. Then, the components are represented in the high level model. For well defined processes, process components can be easily identified from the process descriptions. For example, activities, software products, roles, and so on, can be easily identified from chapter 11 of Rumbaugh's book [23]. Accordingly, well-defined processes are relatively easy to model. On the other hand, modeling a not-well-defined process may be difficult. A meta-process is thus needed.

The meta-process in the CSPL environment is based on action cases, which are similar to the well-known use cases. The action case concept is described in section 3.1. The meta-process for CSPL high level process modeling is described in section 3.2.

3.1 Action case

In CSPL environment, the widely accepted use case concept [22] is adopted for high level process modeling. A use case in a software system is an interaction sequence between an actor and the system. Since use cases carry out functions to satisfy user's requirements, system analysis can be accomplished by analyzing use cases.

For a software process, the actors are developers, which are modeled in various *roles* in a process program. A role in a process accomplishes a piece of work by doing some activities. For example, an analyst accomplishes the analysis work by gathering requirements, generating a specification, verifying the specification, and so on. To apply the use case concept, a role can be taken to be an actor and a piece of work as a use case. However, the followings should be noticed:

1) In a software system, actors are outside the system [22]. On the other hand, roles are included in a process.
2) In a software system, actors are those to be served. That is, a system responses to an actor's request. On the other hand, in a process, roles are those who provide services. For example, in system analysis, analysts carry out the analysis work.

Owing to the above differences, the term *use case* is not used in the meta-process. Instead, the term *action case* is used, which is *a piece of work accomplished by certain roles*. For example, "Requirement gathering" is an action case accomplished by the roles "Analyst", "Customer", and "Domain expert". With the action case concept, a process corresponds to a set of action cases. A process can thus be analyzed by identifying and analyzing action cases.

3.2 Meta-process

The meta-process for high level process modeling is designed based on action cases and this observation: *most software processes are partitioned into phases*. For example, a typical waterfall model process is composed of the following phases: analysis, design, implementation, testing, and maintenance. With that observation, a software process can be decomposed into sub-processes (i.e., phases), then the sub-processes are analyzed. This concept is similar to decomposing software system into subsystems.

According to the above description, the meta-process for high level process modeling is described as follows:

Step1. Decompose the process to be modeled into sub-processes, and draw a top level CSPL activity diagram for the process. Generally, a sub-process corresponds to a phase of the process. To draw a top level CSPL activity diagram, each sub-process is taken as an activity.

Having drawn the top level CSPL activity diagram, the project manager should be consulted to identify exceptions that should be handled in each sub-process. The exceptions and their handlers are then added to the top level CSPL activity diagram. Figure 10 shows an example top level CSPL activity diagram for the waterfall model process, which is decomposed into these sub-processes: "Analysis", "Design", "Implementation", and

"Testing". Note that the exception "Requirement change" should be handled in all the sub-processes.

Step2. For each sub-process and exception handler in the top level CSPL activity diagram, do the following work:

Step2.1 Identify action cases.

Action cases can be identified by analyzing roles' responsibilities, because action cases are carried out by roles. For example, the following roles are in the analysis phase: analysts, customers, and domain experts. The customers provide requirements and verify a specification with the analysts. The experts is consulted to resolve problems. And, the analysts analyze the requirements, generate a specification, and verify the specification with the customers. From the roles' responsibilities described above, the following action cases can be identified: 1) requirement gathering, 2) specification generation, and 3) specification verification.

Step2.2 Draw a CSPL activity diagram for the sub-process.

This CSPL activity diagram is composed of the action cases identified from the sub-process, where each action case is taken as an activity. Note that exceptions and the sequence and synchronization between action cases should also be sketched. Figure 11 shows the CSPL activity diagram for the sub-process "Analysis" which is composed of the three action cases mentioned above. It also shows an exception "Schedule overrun".

Step 2.3 Describe the action cases.

Action cases can be described by informal strategy [24], structure English [25], and so on. For example, the upper part of Figure 12 is the description of the action case "Specification generation", where the specification is represented in Rumbaugh's model [23].

Step 2.4 Identify process components from the action case descriptions; draw a CSPL activity diagram for each action case; and draw a CSPL class diagram for the action cases.

Process components including activities, software products, roles, tools, and so on, can be identified from the action case descriptions. For example, the lower part of Figure 12 shows the process components identified from the description in the upper part.

The activities identified from the description of an action case are then used to draw a CSPL activity diagram for that action case. To draw that diagram, the sequence and synchronization of the activities should be identified. For example, the activities identified from Figure 12 are used to draw a CSPL activity diagram as shown in Figure 13. Those activities can be executed in parallel. Note that the activities in the figure are all non-primitive ones that will be decomposed later.

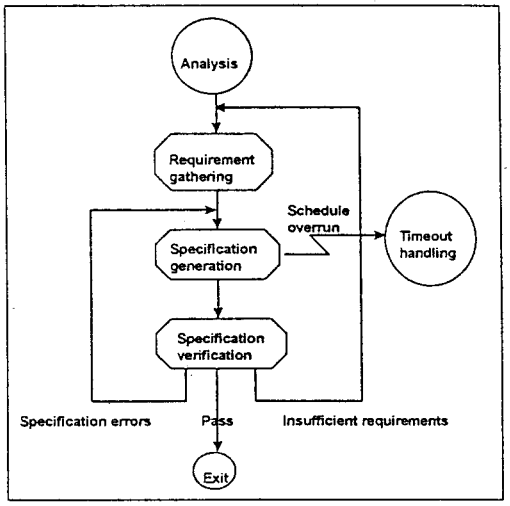

Figure 11. CSPL activity diagram for a sub-process

Action case: Specification generation
Description: 1. The analysts create an object model. 2. The analysts create a function model. 3. The analysts create a dynamic model. 4. The object model, function model, and dynamic model constitute a specification of the system. 5. The specification is create and edited with a CASE tool.
Process components identified: 1. Activities: "Object model creation", "Function model creation", "Dynamic model creation" 2. Roles: "Analyst" 3. Tools: "CASE tool" 4. Software products: "Requirement", "Specification", "Object model", "Function model", "Dynamic model" 5. Relationships: "Responsible_for" relationship between "Analyst" and "Specification", "Depend_on" relationship between "Specification" and "Requirement", "Bound_to" relationship between "Specification" and "CASE tool", "Composition" relationships between "Specification" and the following products: "Object model", "Function model", and "Dynamic model"

Figure 12. An action case description

The classes (including roles, tools, and software products) and their relationships identified from the action cases are then used to draw a CSPL class diagram. Figure 3 shows a class diagram containing the classes and relationships identified from the action case "Specification generation". Note that a high level process model contains only one CSPL class diagram. It is composed of the classes and class relationships identified from all the action cases.

Step 3. Decompose the non-primitive activities.

Some activities in the CSPL activity diagrams are primitive and others not. Non-primitive ones should be decomposed. In decomposing an activity, the activity is first

described in more details. From that description, new activities, class, and class relationships may be identified. The newly identified activities are used to draw a CSPL activity diagram, and the newly identified classes and class relationships are used to update the CSPL class diagram. Figure 14 shows the decomposition of the activity "Object model creation" in Figure 13. The newly identified activities are used to draw a CSPL activity diagram as shown in Figure 15, which is a more detailed representation of the activity "Object model creation".

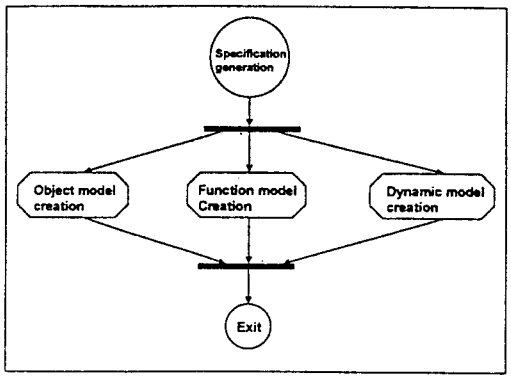

Figure 13. CSPL activity diagram for an action case

The meta-process described above results in a high level process model that contains a CSPL class diagram and multiple CSPL activity diagrams. To trace the process model, the top level CSPL activity diagram is browsed first. The CSPL activity diagrams for the activities in the top level diagram are then browsed. As the top-down browsing process proceeds, more details are revealed and understood. Eventually, primitive activities will be reached. They show the invoking relationships between activities and class operations. To understand the details of the primitive activities, the CSPL class diagram should be traced.

Activity: Object model creation
Description: 1. The analysts identify classes and class relationships from the requirements. 2. The analysts create an object model.
New process components identified: Activities: "Identify classes and class relationships", "Create object model"

Figure 14. Activity decomposition

4. CONCLUSIONS

This paper describes the high level process modeling technique in the CSPL (concurrent software process language) environment, which is composed of a high level process model and a meta-process to represent processes in that model. The high level model is composed of the CSPL class diagram and activity diagram, which are derived from the UML class diagram and activity diagram, respectively. The meta-process is based on action cases, which are similar to the widely accepted use cases. The modeling technique offers the following features:

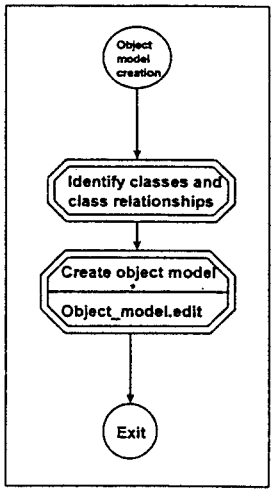

Figure 15. Activity diagram after activity decomposition

1) The high level model provides constructs to model necessary process components.
   Process components to be modeled include: a) activities, b) activity sequence and synchronization, c) exceptions and their handlers, d) software products, e) developers, f) tools, and g) the relationships among software products, developers, and tools. The first three can be modeled in the CSPL activity diagram and the others can be modeled in the CSPL class diagram.
2) The meta-process is expected to be easy to follow.
   The meta-process is based on action cases, which is similar to the well-known use cases. Accordingly, the meta-process is expected to be easy to follow.
3) The technique facilitates process program maintenance.
   As mentioned before, an easy-to-maintain process program should be modular and easy to understand. In CSPL, the high level process model is based on the UML, which is object-oriented (O-O). As agreed, O-O software is modular and easy to understand. Therefore, the CSPL process program development technique facilitates process program maintenance.

ACKNOWLEDGMENT

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

REFERENCES

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

2. Maryse Bourdon, *Process Weaver: Process Modeling Experience Report*, Cap Gemini Innovation, 1992.
3. J.Y. Chen and P. Hsia, "MDL (Methodology Definition Language): A Language for Defining and Automating Software Development Process," *J. Comput. Lang*, vol. 17, no. 3 , pp. 199-211, Jul. 1992.
4. Jen-Yen Jason Chen, "CSPL: An Ada95-like, Unix-based Process Environment," the *IEEE Transactions on Software Engineering*, vol. 23, no. 3, pp. 171 - 184, March 1997.
5. Reidar Conradi et al, "Design, use and implementation of SPELL, a language for software process modeling and evolution," *Proc. Second European Workshop on Software Process Technology*, pp. 167-177, 1992.
6. John C. Doppke, Dennis Heimbigner, and Alexander L. Wolf, "Software Process Modeling and Execution within Virtual Environments", *ACM Trans. on Software Engineering and Methodology*, vol. 7, no. 1 pp1-40, Jan. 1998.
7. Christer Fernstrom,"Process Weaver: Adding Process Support to Unix," *Proceedings of the 2nd international conference on the software process*, IEEE Computer Society, pp.12-26, 1993.
8. Peter Heimann, Carl-Arndt Krapp, and Bernhard Westfechtel, "Graph-Based Software Process Management", *International J. Software Eng. Knowledge Eng.*, Vol 7, No. 4, pp431-455, 1997.
9. B. Holtkamp and H. Weber, "Kernel/2r-A Software Infrastructure for Building Distributed Applications," *Proc. 4th Int'l Conf. on Future Trends in Distributed Computing Systems*, Lisboa, Sept. 1993.
10. K.E. Huff, "Probing Limits to Automation: Towards Deeper Process Models," *Proc. 4th Int'l Software Process Workshop*, New York, NY, pp. 79-81, 1988.
11. Hajimu Iida, Kei-ichi Mimura, Katsuro Inoue and Koji Torii, "Hakoniwa: Monitor and Navigation System for Cooperative Development Based on Activity Sequence Model," *Proceedings of the 2nd international conference on the software process*, IEEE Computer Society, pp. 64-74, 1993.
12. T. Katayama, "A Hierarchical and Functional Approach to Software Process Description," *Proc. 4th Int'l Software Process Workshop*, NewYork, NY, pp. 87-92, 1989.
13. D. E. Perry, "Policy-Directed Coordination and Cooperation," *Proc. 7th Software Process Workshop*, Yountville, CA, pp. 111-113, Oct. 1991.
14. D. E. Perry, "Enactment Control in Interact/Intermediate," *Proc. 3rd European Workshop on Software Process, EWSPT 94*, Villard de Lans, France, Feb. 1994. Ed. Brian C. Warboys, *Lecture Notes in Computer Science*, 772, Springer Verlag, 1994, pp. 107-113.
15. B. Peuschel and W. Schafer, "Concepts and Implementation of Rule-based Process Engine," *Proc. 14th Int'l Conf. on Software Engineering*, pp. 262-279, 1992.
16. S.M. Sutton Jr., D. Heimbigner and L.J. Osterweil , "APPL/A: A Language for Software Process Programming," *ACM Trans. on Software Engineering and Methodology*, vol. 4, no. 3, pp. 221-286, 1995.
17. Leon Osterweil, "Software Processes Are Software Too", in *Proc. 9th Int'l Conf. Softawre Eng.*, pp2-13, New York, 1987
18. Vincenzo Ambriola, Reidar Conadi, and Alfonso Fuggetta, "Assessing Process-Centered Software Engineering Environments", *ACM Trans. on Software Engineering and Methodology*, Vol. 6, No. 3, pp 283-328, July 1997
19. Jen-Yen Chen and Chia-Ming Tu, "An Ada-Like Software Process Language," *J. System and Software*, vol. 27, no. 1, pp. 17-25, Oct. 1994.
20. Jen-Yen Chen and Chia-Ming Tu, "CSPL: a process-centered environment," *Information and Software Technology*, vol. 36, no. 1, pp. 3-10, 1994.
21. Martin Fowler and Kendall Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.
22. Ivar Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
23. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
24. Grady Booch, *Object-Oriented Design with Applications*, The Bejamin/Cummings, 1991.
25. T. DeMarco, *Structured Analysis and System Specification*, Prentice-Hall, 1979.