Modeling Software Development Process and Generating Development Support Environment

by

Hajimu Iida

September 1993

Dissertation submitted to the Faculty of the Engineering Science of Osaka University in partial fulfillment of the requirements for the degree of Doctor of Engineering

Abstract

In this thesis, methods for describing the software development process and for constructing software development environments based on this process description have been studied. We call such development environments "process-centered environments." The process can be described by extracting limited properties of the process from a certain view point. A view point, simply called a view, is represented as a model which specifies the notation and its semantics.

There are various objectives for describing the process, such as documentation, analysis, simulation, and enaction; however, no simple and universal process model exists that is appropriate to all these objectives. Existing various software process models and process-centered environments support only a few specific views. Therefore, depending on the particular objective, we need to select or newly develop an appropriate model for the process.

This thesis proposes a new method for constructing various process models. Using this method, a specific view of the software process is first selected and a simple abstract model is defined. Using this model, the actual process is then described. The obtained description is formalized and refined systematically in order to be enacted as a supporting system. A functional process description language called *PDL* (Process Description Language) is used to realize the supporting system.

Three models were created to investigate the proposed method: a behavioral model, a concurrent task model, and a product relational model. Several prototype systems were derived from these model descriptions, including an activity navigation system, a cooperative development monitor (Hakoniwa), and a product manipulation system.

In Chapter 1, related topics in software engineering are briefly summarized to provide a background for my research approach.

Chapter 2 describes my conceptual model of software development processes and process models. Some related research in the area of process models is also discussed.

Chapter 3 introduces a new method for modeling the process. This method is described as a sequence of five steps to be performed. The process model, which is defined and described in the language *PDL* by this method, can be enacted by the *PDL* language interpreter. A common process modeling problem initially proposed by Kellner, is introduced to be used as the sample description target in Chapters 4 and 5.

Chapter 4 proposes a model which focuses on the sequence of performed activities. Pro-

cess behavior is described using a formal grammar. This formal description is transformed into the base structure of an activity navigation system.

Chapter 5 proposes a model which focuses on concurrent activities where communication involves multiple people. The concurrent activities are represented as communicating sequential tasks. The description was transformed into a team development supporting system named Hakoniwa.

Chapter 6 proposes another model which focuses on the objects produced through the process and the relations among the objects. Objects (called products) and their relations are described as classes and instances in an object-oriented manner.

Chapter 7 discusses the capabilities of the proposed modeling approach, the defined models, and the generated systems.

Chapter 8 presents a summary of the ideas represented in this thesis. Future research goals are discussed, and key points for designing future development environments are described.

List of Major Publications

1. H. Iida, Y. Nishimura, K. Inoue and K. Torii: Generating Software Development Environment from the Descriptions of Product Relations, in *Proceedings of Computer Software and Applications Conference*, Tokyo, Japan, pp.487-492, September 1991.

2. H. Iida, T. Ogihara, K. Inoue and K. Torii: Generating a Menu-oriented Navigation System from Formal Descriptions of Software Development Activity Sequence, in *Proceedings of 1st International Conference on the Software Process*, Redondo Beach, CA, pp.45-57, October 1991.

3. H. Iida, Y. Okada, K. Inoue and K. Torii: Experience of Solving Example Problem for Software Process Modeling, *Transactions of IEICE*, Vol.E76-D, No.2, pp.302-306, February 1993.

4. H. Iida, K. Mimura, K. Inoue and K. Torii: Hakoniwa: Monitor and Navigation System for Cooperative Development Based on Activity Sequence Model, in *Proceedings of 2nd International Conference on the Software Process*, Berlin, Germany, pp.64-74, February 1993.

5. H. Iida, T. Ogihara, K. Inoue and K. Torii: Formalizing Software Development Activities and Generating Navigation System, *Transactions of IPSJ*, Vol.34, No.3, pp.523-531, March 1993 (in Japanese).

6. Y. Matsunaga, H. Iida, T. Ogihara, K. Inoue and K. Torii: Process Description and Enaction System Based on Graphical Representation, *Transactions of IEICE*, Vol.J76-D-I, No.6, pp.324-326, June 1993 (in Japanese).

7. H. Iida, K. Mimura, K. Inoue and K. Torii: Modeling Cooperative Development Process and Prototyping a Monitor/Navigation System, *Transactions of IPSJ*, Vol.33, No.11, November 1993 (to appear) (in Japanese).

Acknowledgments

During the course of this work, I have received help from many individuals. I would especially like to thank my supervisor Professor Koji Torii for his continuous support, encouragement and guidance for this work.

I am also very grateful to the members of my thesis review committee: Professor Tadao Kasami, Professor Ken-ichi Taniguchi and Professor Tohru Kikuno for their invaluable comments and helpful suggestions concerning this thesis.

I also wish to thank Associate Professor Katsuro Inoue and Assistant Professor Takeshi Ogihara for their valuable suggestions and discussions.

I also wish to thank Associate Professor Ken-ichi Matsumoto of the Advanced Institute of Science and Technology, Nara for his valuable suggestions and discussions.

Many of the courses that I have taken during my graduate career have been helpful in preparing this thesis. I would like to acknowledge the guidance of Professors Nobuki Tokura, Toshinobu Kashiwabara, Hideo Miyahara, Masaru Sudo, Akihiro Hashimoto, Mamoru Fujii, and Hirokazu Nishitani of AIST, Nara.

I would like to express my thanks to Professor David Notkin of the University of Washington and Dr. Margaret Thompson for their insightful comments and valuable discussions on the papers which formed the basis of Chapters 4 and 5 of this thesis.

I wish to express my gratitude to Dr. Kumiyo Nakakouji and Miss Tammy Sumner of the University of Colorado at Boulder for their careful reading of a draft of this thesis. Their suggestions were very helpful.

I would like to acknowledge also the valuable comments of Mr. Minoru Nitta of Software Research Associates, Inc. and Mr. Michio Tsuda of Hitachi, Ltd. on the work presented in this thesis.

Finally, my special thanks are expressed to Mr. Yoshihiro Nishimura, Mr. Yoshihiko Okada and Mr. Yasuaki Matsunaga for their help and cooperation. Also, I would like to thank Mr. Kei-ichi Mimura for his assistance in developing the Hakoniwa system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the size of software systems increase, the overall development process becomes very complex and hard to understand. The desire to describe such complex software processes has inspired much research under the rubric of "software process engineering." By describing software processes in some manner, the hope is that the previously tacit process may now be explicitly studied and understood. Once the process description is created, characteristics of the process can be ascertained. The notion of the process can be transferred to other people through the description more easily and with less ambiguity. Furthermore, the process description can be used to define the structure of a software development environment which supports the development process[45].

Developing software process descriptions is not simple or easy. In general, the writer of the software process description has to be familiar with all aspects of the development process, e. g., process steps, computer resources, developer assignments, product structures, time schedule requirements, tool and availability. However, knowing all of these aspects before the project starts is very difficult; some of the aspects may only be determined as the project proceeds. Even if they can be determined a priori, they may change during the project. As research in this area progresses, the difficulties involved in creating appropriate software process descriptions are being recognized.

A model-based approach to software processes seems to be one of the most promising answers to this issue[49, 57]. In this approach, the target software development process is first depicted with a *model* or *process model*, which is an abstract representation of process architecture, process design, or process definition[12].

In this thesis, the terms *model definition* and *model description* are explicitly distinguished. Model definition means the general definition of the components of the model, The Model description (or sometimes simply called the model) refers to an instance of a

description representing particular development process using the components of a model definition. The model may be expressed in some formal manner using Petri nets or regular expressions, or semi-formally using graphs annotated with natural language.

In general, a process model targets only limited views of software development. For example, in the Petri net model, only the concurrency and synchronization of activities in the process are described. The writer of the process description captures and understands the structure of the software process through thee models. Other process information not involved in the models may be added after the model is defined, creating a more complicated description.

A model-based approach works best when the model and the nature of the development process match well. Also, repeatedly developing models for similar kinds of processes helps the writer to understand these types of processes. On the other hand, when a new process is encountered, a single and fixed model approach may not be sufficient. Various models should be tried in order to analyze and characterize the unknown process.

This thesis presents an approach for describing various processes and for enacting (executing) the associated model descriptions. The approach proposed here especially focuses on process construction activities, and provides a specific method for such activities. This approach is analogous to the program derivation method: in order to get executable program code, a model is first built, specifications are written based on the models, and finally the specifications are transformed into executable programs.

An important feature for the model definition is flexibility. Many approaches for process definition and enaction assume their specific kinds of process model or specific process language paradigms[10, 34]. The kinds of process models proposed here, however, make no such assumptions; in particular, you are not limited to a single process model definition. model definitions can be chosen from existing ones or created as appropriate depending upon the objectives for the process description. Some of model definitions may have formal semantic definitions without any ambiguity, others may have semi-formal definitions.

Another important feature of the proposed method is that a single process language, *PDL* (Process Description Language)[26, 27], is used as the implementation language for various models providing a uniform and formal basis. After the model description has been constructed from the model definition, this model description is transformed into a formal description written in *PDL*.

PDL has been designed to describe and enact various software processes mainly in a

process-centered way using a formal semantic definition. *PDL* descriptions can be written at various levels of abstraction, each of which preserves its formality. With this abstraction facility, the model description is transformed into a highly abstracted description in *PDL* with a limited view of the model. The description is then refined into a concrete one involving various information about the process, similar to the refinement procedure for software specifications. The descriptions in *PDL* are enactable by the execution system we have developed. Through this enactment, a software development environment which supports the activities in the described process can be obtained, or simulation of various factors in the process can be performed.

Chapter 2

Software Process Modeling Concepts

2.1 Software development process

Various approaches and methods for software development such as Jackson System Development (JSD)[28], Structured Analysis (SA)[14], and object-oriented design[7] have been proposed. A ambiguity remains in these definitions, and there is no unique interpretation and understanding of them. People have their own interpretation and understanding; therefore, demand for a clearer definition of software development has emerged. To solve this problem, attempts to describe software development processes in formal ways have been made.

Since software development itself has many factors and characteristics, there are many ways to describe it. One typical way is called *process programming*[45]. In process programming, a sequence of development activities is described mainly in a procedural way. There are also attempts to describe software processes from many other perspectives (such as object-oriented or rule-based approaches).

Many software developers, especially Japanese mainframe computer manufacturers, have developed and applied their own standardized development processes. Most of these process embody a management perspective and are based on traditional waterfall models[11]. Since the process is standardized, it is assumed that the current development status can be estimated[52]. However, actual development activities involve many interruptions, much reworking, backtracking, and many other exceptions, which are not usually described by in the standardized process.

2.2 Software process modeling and related works

There are various approaches to modeling and enacting processes. Some of these approaches are summarized and compared in Mi and Schacci [43]. In this section, we focus on a few of these approaches and evaluate how they deal with representing process sequences.

Process programming using Appl/A:

Osterweil has proposed a concept for describing the software development process as software itself[45]. For the process description, he and his group have developed a language called Appl/A, which is an extension of Ada[54]. In addition to many rich features of Ada, Appl/A provides the features for describing relations between products and. Software development processes can be described using Appl/A in a procedural way. However, one the process is described as a program, the structure and behavior of the process are not clear. Moreover, it is impossible to perform an activity that is not already described in the program's control sequence. Thus, process descriptions need to take into account all exceptional activities.

Marvel:

Kaiser et al. have developed the software environment Marvel, which activates development tools automatically and manipulates software products based on pre-defined rules[29, 30]. Marvel is composed of (1) *Objects* which control products and tools, and (2) extendable *rule* sets which specify the order of activities. A rule is composed of a precondition, a postcondition, and an activity. If the precondition of a rule is satisfied, the tool specified in the activity part of the rule is activated. As compared to procedural descriptions, Marvel's description offers more flexible activity sequences and makes it easier to express exceptional activities. On the other hand, it is difficult to predict from the Marvel description how activities actually proceed. The overall structure of the development process still remains unclear.

HFSP:

Katayama proposed a hierarchical process model named Hierarchical and Functional Software Process (HFSP)[33] that uses attribute grammars to describe processes. Software objects produced by processes are represented as *attributes*, and their synthesizing rules are

specified using a functional notation. Some useful mechanisms such as providing persistent objects for product representations and meta-operations for dynamic process modification are employed by this model.

Chapter 3

View-specific Software Process Modeling and Enactment

3.1 A View-specific modeling approach

Although various software process models have been proposed, actual development processes are generally more complex. They are composed of various elements including activities, products, resource assignment, scheduling, and many others. It is difficult to represent such a complicated process with a simple monolithic model, and thus many of the proposed models are fairly complicated. From the perspectives of documentation and formalization, complex models have disadvantages; they are difficult to understand, evaluate and validate.

We advocate using a simple process model tailored for every specific purpose which is regulated by a certain view of the software process. Various possible views are: a behavioral view, a product-oriented view, a communication-oriented view, and an organizational view. In this thesis, we study some of these views, and use these views as criteria for constructing process models using the process modeling steps described below. These models were consequently formalized, refined and enacted to create development support systems.

3.2 Overall procedure

Each model is defined and formalized by following a procedure consisting of five modeling steps. Some of these steps may be performed implicitly and thus might not be clearly distinguishable.

1 Model-type definition step

Depending upon the requirements of the specific target process, a model representing the structure of the process from a specific view is defined. The result is called the *model definition.* Williams' behavioral "Software Process Model" (SPM)[57], the Petri-net model[13], and the context-free model[23] are all examples of such model definitions. In many cases, only one model definition is selected for the actual description. Sometimes, more than one type of model are chosen and subsequently these are unified into one model description in later steps[26]. Previously defined and well-established model definitions can be chosen. Alternatively, new model definitions may be obtained by modifying existing ones, or they can be created from scratch. This step may be repeated until an appropriate model is obtained.

2 Model description step

In this step, the target process is depicted using the selected model definition. The obtained descriptions are called the *model descriptions.* Only the issues of concern for the target process are sketched from the view of the selected model definition; details are not included at this step. This step is one of the most intellectually challenging, in the sense that people have to understand the target process and to determine the basic policy of the description. Parts or all of the description may need to be repeatedly rewritten; sometimes the model definition may need to be modified. Completeness and consistency are very important properties for the model description produced during this step. However, no mathematically rigorous attempt is made to verify those properties here since the description may involve informal components.

3 Formalization step

The model description is transformed into a formal notation in a process implementation language. In this case, *PDL* is used. The obtained description is called the *abstract formal description.* This transformation step is necessary since the model description produced in step two is ambiguous and subject to multiple interpretations. For instance, sometimes the model descriptions include graphical representations or natural language notations whose meanings are not rigorously defined. After formalization into *PDL*, ambiguity in the interpretation of the model is eliminated; thus the basis for later refinement is established. In order to do this formalization step, we first have to determine correspondences between

the model definition and fragments of *PDL* descriptions, and then the overall model description is transformed into the *PDL* description. The exact transformation method varies depending on the model definition in use.

4 Refinement step

The formalized abstract description obtained in the previous step contains only limited information about the process with respect to the view of the model definition. Depending upon the final objectives for the process description, other information is repeatedly added at this step until the description becomes rich enough. For example, if we create the description in order to get a software development environment which supports the process, we may add information concerning specific tools to use at each process step. The obtained description is called the *concrete formal description.*

5 Enaction step

Finally, the concrete formal description is enacted(executed). Since the formal description is written in *PDL*, the *PDL* interpreter is used to enact the descriptions. There are several objectives of enacting software process descriptions, such as automation, simulation, demonstration, etc. Our objective here is to assist the developer. Thus, enacted descriptions should act as development support systems.

3.3 The Software process description language: PDL

As mentioned above, the functional language *PDL* provides an uniform basis for our process description formalism. *PDL* is a functional programming language based on an algebraic specification language. Development processes in *PDL* may be defined at various levels of abstraction.

PDL includes an abstract data type, type <*state*>. Functions for tool invocation that possibly change the system status take a value of type <*state*> as one of their arguments; they return a value of the type <*state*> which represents an updated system status resulting from the operations. There is only one state value for the system at any point in the execution.

Processes are described as *PDL* functions which take a state value S_1 and return a modified state value S_2. Therefore, *PDL* scripts (programs) describe definitions of state

transition functions such as tool invocations or window operations. For example, a simple implementation of an "Edit, Compile, and Link" process can be written as follows:

```
Main(S)     ==  if error(Trans_1(S)) then Main(Trans_1(S))
                else Trans_2(Trans_1(S));
Trans_1(S)  ==  Compile(Edit(S));
Trans_2(S)  ==  if error(Link(S)) then Main(Link(S))
                else S;
```

The *PDL* interpreter provides various built-in functions for window operations, debugging functions, and tool invocations, which are useful in constructing a development support environment. Thus, a *PDL* program acts as a development support system.

3.4 Example problem illustrating process modeling

In this thesis, an example problem called the "Software Process Modeling Example Problem " initially proposed by Marc Kellner[35], is used as a common basis for discussion on modeling techniques and expressive powers. This problem defines various efforts, caused by a requirements change, with more than 10 pages of English text. Additionally it describes various aspects of the software development, such as processes, products, human resources, and management. Solving this problem means that we must read first and understand the text and represent the described aspects with our modeling and description techniques.

The problem is organized into two parts: the core problem and extensions. The core problem defines activities in an overall process step, named the *Develop Change and Test Unit*. This step is composed of 8 sub-processes (we simply call these *steps*).

In this problem, the effects of the modification are assumed to be limited to a single module only; i.e., and it does not affect other modules. Therefore, there is no need to modify other modules or to check the consistency of related modules. This process starts when the project manager has created the schedule and assigned the tasks. The entire process terminates when the new code for the module passes unit testing.

Figure 3.1 shows an example of an informal view of this process. Steps, products, and communications are specified by bubbles, icons, and arrows respectively.

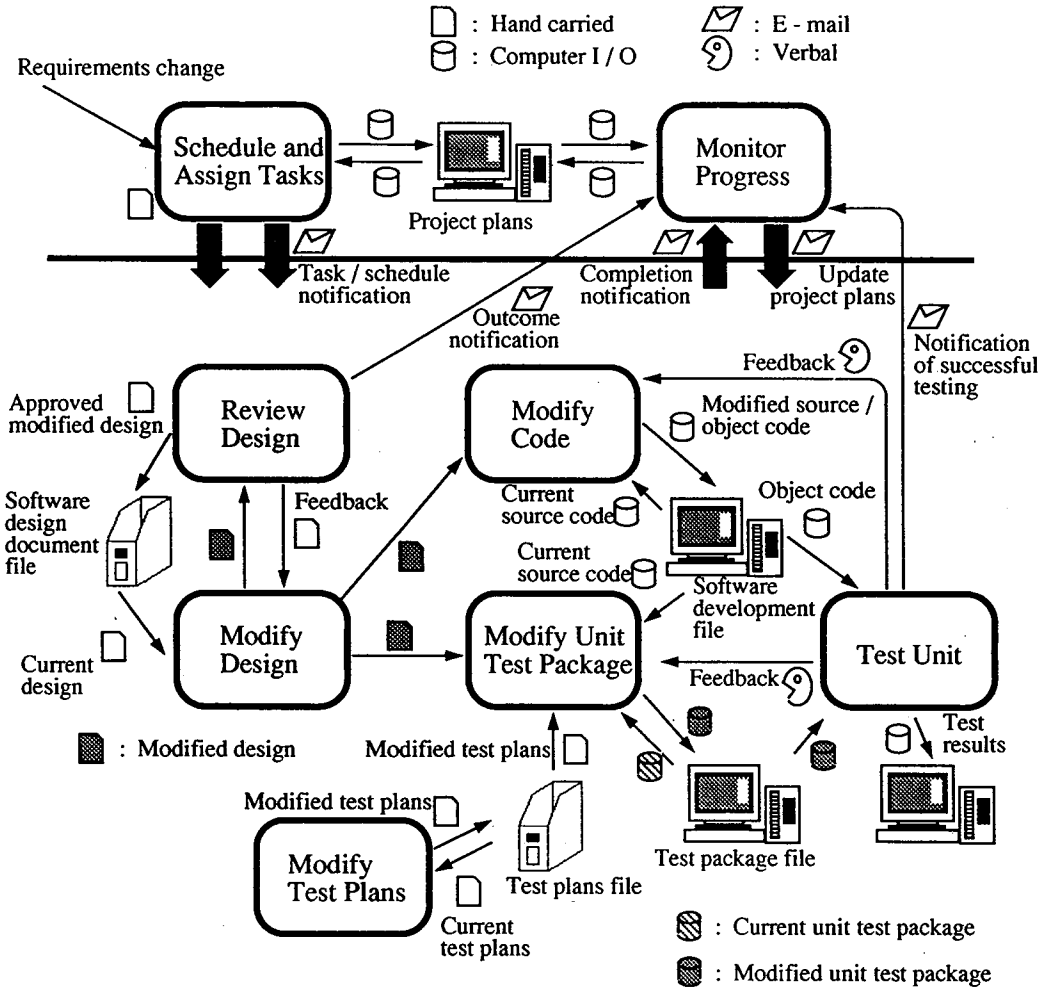


Figure 3.1: Process modeling example problem

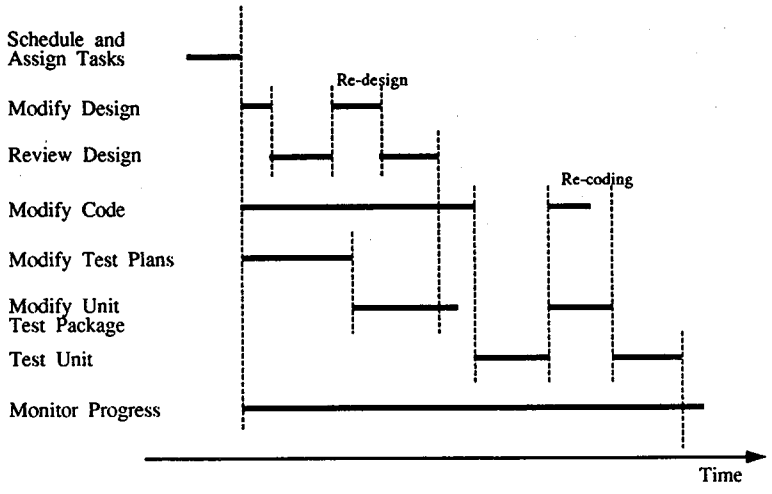


Figure 3.2: Time chart description of modeling problem

Many attempts have been made to extract certain views from this process. Figure 3.2 is an example of a simple time chart view. This time chart shows some characteristics of each step clearly such as initiation and termination; however, there are several difficulties, as follows:

- This description does not clearly capture dependencies between steps. for instance, both the *Modify Code* step and *Modify Unit Test Package* step must terminate before the *Test Unit* step can begin. However, the time chart can't describe that the order of termination for those two steps does not matter.

- The description does not express which step to execute next when there are several possible candidates. For example, if the first *Test Unit* terminates unsuccessfully, then we may proceed to either *Modify Code*, *Modify Unit Test Package*, or both.

Figure 3.3 shows another description of the problem using a Petri net representation. This description can appropriately express the properties of synchronization among the steps mentioned above.

We can construct various kinds of models for this example problem. In the following chapters, this example problem has been modeled in various views.

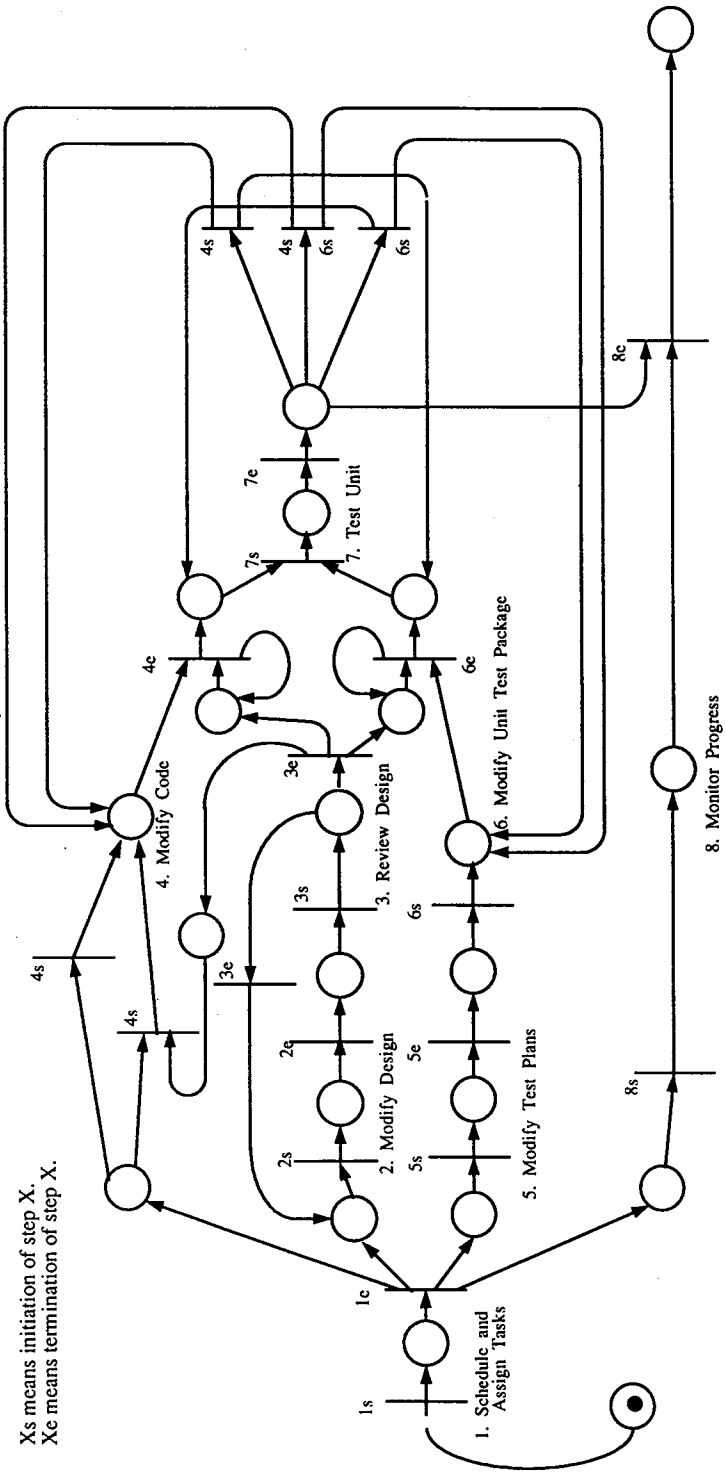


Figure 3.3: Description of modeling problem using a Petri net model

Chapter 4

Behavioral Model

4.1 Behavioral view

In this chapter, the modeling focus is on the representation of the behavior of the software process. Behavior is expressed as the definition of a sequence of development activities. Other factors in the software process, such as the product model and its representation, are not discussed here.

Describing complex sequences of activities corresponding to real work is not a simple task. If the activities in the development sequence are fixed and predetermined, the flexibility and performance of developers' work will be greatly reduced. It is important to support the developers by providing flexible activity sequences with simple descriptions. Mechanisms are preferred to control the sequence of activities interactively with minimal restrictions.

A new method for development assistance that controls activities using a set of relatively large possible sequences is introduced here. To define such a set, a formal context-free grammar (CFG) is employed. Using this formal grammar, we can describe the behavior of activity sequences simply and clearly.

Many previous works have used regular expressions or their extensions to represent processes formally. These representations provide us with clear and straightforward views of the processes; however, they lack the necessary expressive power. Some development processes may form inherently recursive structures — that is, the target problem is solved in a certain way (or method, or strategy), and subproblems occurring in that process are repeatedly solved in the same way. These kinds of recursive structures cannot be represented with regular expressions but can be easily expressed using CFGs[1].

[1]We may choose a restricted CFG or an extended regular set which exist between a CFG and a regular set. However, since no difficulty arises even with non-restricted CFG's in the following discussions, we use general CFGs.

A way to generate development assistant system from the description of the grammars is also proposed here. As an example, a menu-based assistance system which navigates the developer to the appropriate activities is constructed. It acts as a scheduler in a development support environment. This system has been written in the functional language *PDL* [26, 27] and is obtained by translating the grammar into *PDL* functions.

The key points of our method are as follows:

- By focusing on the sequences of activities, descriptions become simple and unambiguous.
- By using CFGs instead of regular expressions, any recursive process can be defined naturally.
- By transforming the grammar systematically, menu-based assistance systems can be constructed easily.

4.2 Definition of Behavioral model

In this view, we are not concerned with the details of each activity and instead focus on the sequence of activities. Concrete information about the contents of activities are strongly related to the structures, relations, and operating mechanisms of software products such as documents or source code.

Since the software development processes is assumed to consist of a sequence of development activities performed by a developer on a workstation, an "activity" can be defined as follows:

- The entire development process is an activity.
- Each activity may itself be decomposed and expressed as a sequence of activities.
- An activity which cannot be further decomposed corresponds to activation of a tool or to a developer's action.

With the assumption that development processes can be expressed as serial sequences of activities, two models – a regular expression model and a CFG model – are assessed.

In both models, the following correspondences between grammar and process exist:

- The start symbol designates the activity sequence for the entire development process.

- Nonterminal symbols designate partial activity sequences during the development.
- Terminal symbols designate atomic activities which should be carried out by the developer or the development assistant system.

4.2.1 Regular expression model

Some activity sequences can be represented as regular expressions. For a simple example, assume a traditional waterfall model, where all stages are performed sequentially:

$$Analysis \rightarrow Design \rightarrow Implement \rightarrow Testing \rightarrow Integration.$$

However, in actual development, a stage often is repeatedly performed and sometimes development must backtrack to an appropriate previous stage. Therefore, the activity sequence of the waterfall process model is expressed as:

$$((((Analysis^{+}Design^{+})^{+}Implement^{+})^{+}Testing^{+})^{+}Integration^{+})^{+}$$

where "+" implies the sequence is performed one or more times. For another example, consider the $Edit - Compile - Link$ process. In this case, there are no iterations consisting only of *Compile* or *Link*. Thus, it is expressed as:

$$((Edit^{+}Compile)^{+}Link)^{+}$$

Therefore, sequences such as $Edit - Compile - Link$ and $Edit - Edit - Compile - Edit - Compile - Link$ are *valid* activity sequences. Sequences such as $Compile - Edit - Link$ and $Edit - Compile - Compile - Link$ are *invalid* (underivable) activity sequences.

As shown above, some development processes can be expressed using regular expressions, but some classes of development processes (e.g., processes including recursion) cannot be expressed using regular expressions[2]. Therefore, context-free languages might be preferred when defining the activity sequences.

Because regular expressions can be expressed with context-free languages, the rest of this Chapter treats context-free grammars only.

4.2.2 CFG model

A context-free grammar G is defined as a quadruplet $G = (V_N, V_T, P, S)$, where, V_N and V_T are sets of nonterminal symbols and terminal symbols, respectively. P is a set of production

[2]Of course, a more complex language may be needed for some of the development process, but we think that a CFG is capable of expressing most of the interesting properties of activity sequences.

rule, and $S \in V_N$ is the start symbol. Each element of P is formed as $X_0 \longrightarrow X_1X_2...X_n$, where $X_0 \in V_N and X_i \in V_N \cup V_T (1 \leq i \leq n)$.

Though the notation above is convenient for formal analysis, it is not easy to construct the sentential forms. An extended notation of CFG, Backus-Naur Form (BNF) alleviate this problem. We use an extension of BNF which uses the meta symbols shown in Table 4.1.

Table 4.1: Meta symbols for grammar description

Symbols	Meanings
=	Definition
X | Y	Selection (X or Y)
;	End of rule
X^*	Repeat X 0 or more times
X^+	Repeat X 1 or more times
(...)	Association of symbols

For example, the grammar notation :

S	→	N_1 S
	→	ε
N_1	→	N_2 *Link*
N_2	→	N_3 N_2
	→	ε
N_3	→	N_4 *Compile*
N_4	→	*Edit* N_4
	→	ε

is expressed in our extended notation as:

$$S = ((Edit^*Compile)^*Link)^*;$$

Terminal symbols are indicated in italics.

4.3 The example problem description

4.3.1 A topdown software development process

Activity sequences containing recursion can be expressed using CFG grammars. A description of a topdown development process is shown here as an example. *Topdown development* is the development method in which a program is decomposed into modules, and then each

module is further decomposed into its submodules recursively. Topdown development may be simply expressed by the following steps:

Step 1. Functionally decompose into submodules.

Step 2. Define interfaces between each submodule.

Step 3. Implement each submodule.

Step 4. Write code for the module body.

Implementing each module (step 3) is done by performing steps 1 through to 4 recursively. In the case of the 'smallest' modules, requiring no further decomposition, only step 4 is performed. Each step may be repeatedly performed, or feedback from step 2 to 1 or step 4 to 3 may occur.

The activity sequence in this example can be expressed as shown in Figure 4.1.

TopDown	=	MakeModule;
MakeModule	=	BreakDown | *MakeSimpleModule*;
BreakDown	=	MakeCombinedModule , Implement;
MakeCombinedModule	=	(*FunctionDecomposition* | *InterfaceDesign*)$^+$;
Implement	=	(*BodyCoding* | MakeSubModule)$^+$;
MakeSubModule	=	MakeModule$^+$;

Figure 4.1: Grammar of activities in top-down development

4.3.2 Kellner's example problem

In this example problem, the core problem and some extensions such as tool execution are described. The resulting description is shown in Figure4.2. In this description, version control tools (such as SCCS `delta/get` or RCS `ci/co`) are executed on every execution of *ModifyDesign* and *ModifyCode* so that the latest version is registered. At the end of these steps, progress notification mail is sent and updated products are printed.

The original problem specifies that steps are distributed to several people and performed concurrently. However, our behavioral model can't express this situation properly because this model expresses single sequential behavior only. Therefore, the order of some concurrent steps has been predetermined.

```
Process ::= Schedule_Assign_Tasks, Main;

Main ::=
 (
    (ModifyDesign, ReviewDesign)+,
    (ModifyCode, (ModifyTestPlan,ModifyUnitTestPackage,
    TestUnit)+)+
 )+;

ModifyDesign ::=
    CheckOutDesign, (EditDesign,CheckInDesign)+, Release;

CheckOutDesign ::= execute_sccs_get;

CheckInDesign ::= execute_sccs_put

ReleaseDesign ::=
    execute_mail, execute_lpr, send_hardcopy_by_hand;

ModifyCode ::=
    CheckOutText,
    ((EditText,CheckInText,Compile)+,(CodeCheck|),)+,Release;

Schedule_Assign_Tasks ::= execute_scheduling_tool;

CheckOutText ::= execute_sccs_get;

CheckInText ::= execute_sccs_delta;

Edit ::= execute_emacs;

CodeCheck ::= execute_browser;

Release ::= execute_mail, execute_lpr, send_hardcopy_by_hand;
```

Figure 4.2: Activity grammar for Kellner's example problem

4.4 Generation of an activity navigation system

4.4.1 Supporting issues

There are two uses of the model (activity sequence description) in constructing a development assistance system. One approach is to generate a system which leads the developer to follow the grammar by showing the range of possible activities at each step in the development process. Another approach is to use the grammar for activity monitoring. The developer's activities are monitored while the process proceeds. If the developer attempts to perform an activity deemed invalid according to the grammar, the assistance system generates a warning and shows the developer possible legal activities upon request.

In this section, we show an application example based on the first approach. This system displays menus and guides the developer to appropriate activities based on the grammar. The following are the major features of the system:

Activity selection by menus: If several production rules are applicable at a given time, the system provides the developer with a menu showing the possible choices. Using the menu, the developer can select an appropriate rule. Menus restrict the range of possible activities so inappropriate actions are prevented. Moreover, menu-based user interface reduces the workload of the developer in terms of command inputs required.

Automatic enaction of simple sequential activities: Activities expressed by rules containing only linearly-ordered sequences (i.e.; having no branching) are automatically enacted by the system.

Automatic invocation of tools: Primitive activities, which correspond to terminal symbols in the grammar, are automatically performed if they are editor invocation or compiler execution. This also reduces the developers' workload as they do not need to remember the names of tools, command options, and so on.

4.4.2 Generation of a menu-based assistance system

The menus show multiple selection candidates and prompt the developer to select one of them. Figure 4.3 shows an example of the menus discussed in this thesis. These menus are implemented in a window system with a "mouse" as a pointing device.

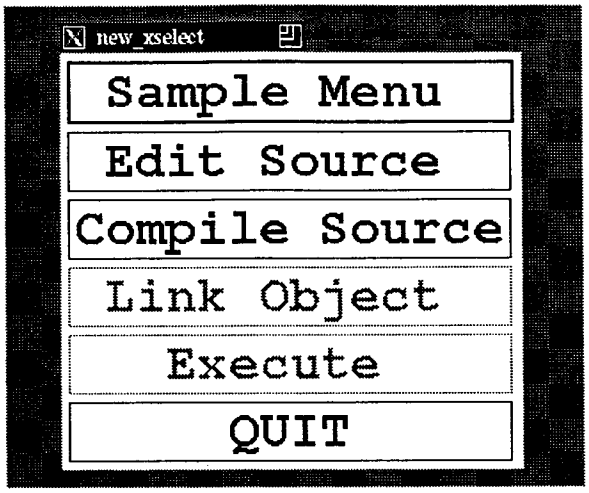


Figure 4.3: An example of an activity selection menu

Development work proceeds by selecting from menus that are generated by the system from the grammar. We constructed this menu system as follows:

Step 1. Determine the set of rules underlying the menu.

Step 2. If semantic sequence control is required, then append *restrictions* to the rules

Step 3. Define the action associated with each activity corresponding to a terminal symbol.

The actions in in Step 3 depend on the environment in which the system is built. Thus, details about Step 3 are not mentioned here. In the following sections, the generation of the logical structure of the menus and also the *restrictions* that are added to each condition are shown.

Generating menus from the grammar

A set of rules for the same non-terminal symbol corresponds to one menu component with multiple entries (items).

For example, a production rule (this is actually a set of n rules):

$$\boxed{X_0 = X_1|X_2|...|X_n;}$$

corresponds to a menu that is composed of selection items X_1 .. X_n. An iterative expression (e.g., +,*) is also a kind of a selection expression and corresponds to a menu also. For

example, an expression X^+ can be rewritten into a selection expression by introducing a new non-terminal symbol, X', and appending the following rules:

$$X'=X(redo\_X|quit\_X);$$
$$redo\_X=X';$$
$$quit\_X=\varepsilon;$$

As shown above, we can determine the items to be included each menu. The label of each item to be displayed in the menu should be specified.

4.4.3 Adding restrictions to menu items

Role of restrictions

In actual development, all parts of an activity sequence might not be expressible using only context-free grammars. Some parts of a sequence should be controlled semantically.

For example, in the development process described as:

$$S = ((Edit^+ Compile)^+ Link)^+;$$

the sequence $Edit - Compile - Link$ is *valid* only if *Compile* has succeeded. It is *invalid* if actual compilation has failed. For the case of the top-down development shown in Figure 4.1, the restriction, "The number of submodules specified in step 1 should be the same as those specified in step 3," can't be represented using only the grammar.

To achieve this, restriction conditions to be satisfied for each menu item can be specified. The restriction condition must be satisfied before an item can be selected. Visually, an item whose restriction condition is not satisfied is displayed grayed as shown in Figure 4.3. The developer can select any item whose restriction conditions are satisfied. Some menu items may have no restriction conditions, thus, they are always selectable.

Semantics of restriction conditions

Actually, restriction conditions are defined as conditions on software products to be satisfied. Here, "software products" means all information produced during the development process. This includes not only program files but also intermediate documents such as specifications or design documents. Design decisions made by the developer are also considered to be products.

Assume a set of products $\{D_1, D_2, ..., D_n\}$ to be an *abstract state* S. Thus, the development process is a state transition function T on the (abstract) state, and each product is

a function D_i which extracts a value from the state. Each restriction condition is defined as a predicate function P_i on the state. Each P_i is defined using the attributes of products (for example, its existence, modification time, whether it satisfies the document syntax or not, etc.).

For example, in the case of the *Edit*, *Compile*, and *Link* sequence, assume the products to be {source, object, executable}. Now, the restriction condition of *Link*, "Current object file has compiled successfully and it is newer than the current source file," is described as follows:

$$valid(object(S)) \ \& \ (update\_time(object(S)) >= update\_time(source(S)))$$

The menu system explained above can be seen as a state transition machine, which can be described easily with various programming languages. In our work, we have used the functional language *PDL*. In the rest of this section, a method for transforming the grammar into a *PDL* program is shown.

4.4.4 Transformation of a CFG description into PDL

Menu functions in PDL

A primitive function, *menubranch*, has been implemented so that selections from menus with restrictions may be simply described. *Menubranch* takes a list of tuples "[a label of type string, a restriction value of type Boolean, a current state of type state]" as an argument, and returns a modified status value. It is used in the following manner:

```
P(S) == menubranch({["s1",pre_s1(S),s1(S)],
                    ["s2",pre_s2(S),s2(S)],
                    ["s3",pre_s3(S),s3(S)]},S);
```

{..} and [..] indicate a list and a tuple respectively. When the function P is evaluated, a menu including three selection items "s1," "s2," and "s3" is displayed. If a restriction condition, `pre_si(S)`, of an item returns false, its label is grayed and unselectable. After the selection has been made by the developer, the state `s_i(S)` corresponding to the selected item i, is evaluated and returned.

Transformation rules

Transformation of a CFG description into *PDL* proceeds as follows:

1. The rule $X_0 = X_1X_2...X_n$ is transformed into the definition of the state transition function X_0 as:

```
X_0(S) == X_n(...X_2(X_1(S))...);
```

2. The rule $X_0 = X_1|X_2|...|X_n$ is transformed into the definition of the state transition function X_0 as:

```
X_0(S) == menubranch({["X_1",R_1(S),X_1(S)],
                      ["X_2",R_2(S),X_2(S)],
                      ...
                      ["X_n",R_n(S),X_n(S)]
        },S);
```

   where R_1, .. ,R_n are the names of functions for the restriction conditions.

3. For each terminal symbol X_i, a corresponding state transition function X_i(S) is defined in *PDL*. For example, the terminal symbol *Edit* , which means editing the source file using some editor, is described in *PDL* as:

```
Edit(S) == exec("vi " + SourceFile(S),S);
```

4. For every item in the menus, a Boolean function R_i(S) is defined which corresponds to the restriction condition of an item. For example, a restriction condition for *Compile* stating that the source file should be newer than the object file is described as:

```
Res(S) == time_stamp(Sourcefile(S)) >= time_stamp(Objectfile(S));
```

The *PDL* program obtained by applying these transformations to the grammar in Figure 4.1 is shown in Figure 4.4. This program is executed by the *PDL* interpreter and acts as the skeleton of the assistance system supporting top-down development.

4.4.5 Enacting example problem description

The *PDL* scripts for both topdown development process and for Kellner's example have the following features:

```
TopDownDevelop(S) == Makemodule(S);
Makemodule(S) ==
    menu_branch("Make Module",{
    [ "Break Down",Breakdown(S),true ],
    [ "Implement" ,Construct(S),pre_Construct(S) ]
    }, S );
Breakdown(S) ==
    menu_branch("Break Down",{
    [ "Complete",S,true ],
    [ "Proceed", Breakdown_main(S),true ]
    }, S);
Breakdown_main(S) == interface(divide(S));
divide(S) ==
    menu_branch("Division",{
    [ "Redo", divide(divide_main(S):S1), true ],
    [ "Quit", S1, true ]
    }, S1);
divide_main(S) == exec( "emacs " + ModuleStructureDocument(S), S);
interface(S) ==
    menu_branch("Division",{
    [ "Redo", interface(interface_main(S):S1), true ],
    [ "Quit", S1, true ]
    }, S1);
interface_main(S) == exec( "emacs " + ModuleInterfaceDocument(S), S);
Construct(S) ==
    menu_branch("Construction",{
    [ "Redo", Construct(Construct_main(S):S1), true ],
    [ "Quit", S1, post_Construct(S) ]
    }, S1);
Construct_main(S) ==
    menu_branch("Construction",{
    [ "Push Down",Makemodule(S),pre_Makemodule(S) ],
    [ "Body Coding" ,makebody(S),true ]
    }, S );
makebody(S) == exec( "emacs " + CurrentModuleSource(S), S);
// restrictions
pre_Construct(S) == NoErrorOfModuleStructureDocument(S) &
        NoErrorOfModuleInterfaceDocument(S);
pre_Makemodule(S)== CurrentNumberOfSubmoduleToMake(S)>=0;
post_Construct(S)== CurrentNumberOfSubmoduleToMake(S)>=0 &
        MadeCurrentModuleBody(S)
```

Figure 4.4: PDL script obtained from top-down development grammar description

- They navigate activities by menus.
- They automatically execute tools such as editors and compilers.
- They store documents and source files in appropriate locations (e.g., directories).

Moreover, top-down development system manages the number and structures of modules to be built.

Chapter 5

Concurrent Task Model

In this chapter, we will discuss process models focusing on managing concurrent software processes among multiple developers. The issues described below are very important and indispensable when monitoring and controlling large projects.

Actual software development is performed through the cooperation of many developers. However, many standardized processes do not explicitly deal with the various issues relating to multiple developers. Such issues include coordination issues concerning the synchronization of developers' activities and data collection issues concerning data necessary for estimating developer status.

5.1 Concurrent task view

The model has to cleary define activities performed by multiple developers and the relations of these activities. Given such a model, navigation through these activities can be easily provided and each developer's progress can be easily monitored.

In this chapter, a process model concerning coordination issues in a multiple developers' software process is proposed. The model proposed here is an extension of the one in Chapter 4, which modeled development processes performed by a single developer using a CFG to express activity sequences. The activities performed by a single developer are assumed to be basically sequential, and formal grammars are considered to be a very good vehicle for representing these activities. However, such a model can't handle parallelism which is essential for representing multiple developers' work. The new model uses the single developer model to define the activities of individual developers and uses additional communication primitives to coordinate these activities.

Communication primitives control the progress of activities and organize the overall tasks

in the model. Each developer is responsible for accomplishing several tasks. Using this model, reworking and backtracking are easily represented by repetitions of activities in the regular expressions. Suspending and resuming work is denoted by communication of task. Using this model, the current status of each task and each developer is easily tracked, and the progress of the overall project can be determined by the project manager.

5.2 Definition of concurrent task model

The concurrent task model proposed here is based on a concurrent process model such as Communicating Sequential Processes (CSP)[16]. This model is composed of several concurrent *tasks*. There are many kinds of activities in actual development, and these activities are performed by several developers concurrently. Each developer communicates to other developers, and this communication coordinates and controls the progress of activities (Figure 5.3).

If we model this process from a human-centered view point, the model becomes complicated, since one developer might be responsible for performing several independent activities which have to be described in the model. Therefore, an activity-centered model is proposed here, where related activities are treated as a single task, and each developer performs several tasks. The developers are assumed to be *processors* for the tasks.

5.2.1 Task identification

An activity can be a small unit of work such as "*editing a file*", or a relatively large unit of work such as "*changing the system specification*," each of which is performed by several people. An atomic unit of work such as tool execution or decision making is defined as a *primitive activity.* A *task* is defined as a sequence of primitive activities[1] which can be represented without concurrent actions. Although a CFG can be used to specify activity sequences, a simple regular grammar (RG) is employed here. A task is defined as a regular expression of primitive activities. Some operators representing iteration and selection useful for specifying activity sequences are described in Table 5.1. Meta symbols ('[ ]') are added to the ordinary regular expression operators. With these operators, we can easily and simply describe the expressions generated by a RG.

For example, a task to repeatedly edit and compile a file until the compilation succeeds can be specified as follows:

[1]From now on, we may refer to primitive activity simply as *activity.*

Table 5.1: Regular Expression operators used in task descriptions

A*	Zero or more repetitions of symbol A
A^+	One or more repetitions of symbol A
[A]	Zero or one appearances of symbol A
( )	Grouping
|	Selection

task = (<Edit file>$^+$ <Compile file>)$^+$;

An overall development process is represented as a set of tasks which may be performed concurrently. Inter-task synchronization and cooperation are represented by simple communications.

5.2.2 Communication primitive insertion

Communications are represented by asynchronous messages (strings of characters). Synchronization of tasks and control of other tasks are performed by these communications. Communication operators are categorized into two types: data transfer and task control.

The following are features of the data transfer operations:

- Data type of the transfer is "string."
- One task may have several input ports for incoming data.
- To send data, a destination task and its input port should be specified.
- Input ports have operations for reading data and for checking the presence of data.

Specifically, there are three primitive operations shown in Table 5.2.

Table 5.2: Message transfer primitives

Operator	Argument	Operation	Return value
send	task,port	send a string to another task	——
recv	port	receive a string from a port	string
peek	port	inspect existence of message in a port	boolean

A send operation asynchronously sends a string to another task and then terminates. There is a default port for every task, and if the port name is omitted, the default is used. Recv and peek operations return values. A recv operation reads the first string in a specified port, and returns it. This operation is blocked if there are no messages in the port.

A peek operation is non-blocking. It checks for the existence of a message in a specified port. If some messages are found, it returns `true`. Otherwise, `false` is returned immediately. The returned values are used in the restriction conditions of selection expressions as mentioned Chapter 4, and their values do not appear in the expressions explicitly.

These communication operations can also be used as events for synchronizing and controlling tasks. For example, peek can be used to represent a process executing a job and waiting for a message repeatedly (Figure 5.1).

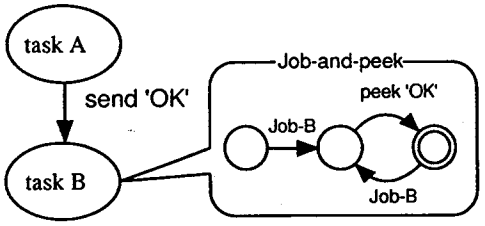


Figure 5.1: Task synchronization example

In Figure 5.1, task$A$ and task$B$ proceed concurrently, and task$B$ waits for an 'OK' message from task$A$ during the execution loop of its own job. This can be specified as follows by using the peek operator:

```
task A = <Job-A> <send taskB, OKport, 'OK'>;
task B = (<Job-B> <peek OKport>)+;
```

Features of the second type of communication operators, task control operators, are:

- Initiation (activation) of other tasks
- Synchronization with another task's termination
- Termination of other tasks

These features are represented with constructs shown in Table 5.3 by using message transfer primitives procedurally. The `Start` operator is represented as a `send` message in one task and a `recv` message at the top of another task. Thus, by default, every task has a `recv` operation implicitly at the top of its sequence. The `Wait` operator is a special case of the `recv` operation.

These communication primitives are also treated as activities, and together with other activities, they constitute activity sequences. The activity sequences are represented as

Table 5.3: Task control primitives

Operator	Argument	Operation
`start`	task	send a request to initiate (activate) another task
`wait`	task	wait for termination of another task
`exit`	task	inform other tasks of termination

regular expressions. Sequences which can't be represented with regular expressions may be decomposed into smaller tasks[2].

For example, assume task$A$ activates task$B$ and task$C$, and task$D$ waits for their termination (Figure 5.2).

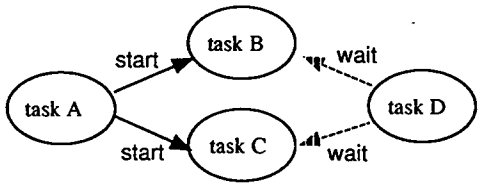


Figure 5.2: Example of concurrent tasks

This example can be specified in the concurrent task model as follows:

```
task A = <Job-A> <start B> <start C>;
task B = <Job-B> <exit D>;
task C = <Job-C> <exit D>;
task D = <wait B> <wait C> <Job-D>;
```

Finally, each developer is assigned several tasks by the project manager, and performs specified activities according to their regular expressions (Figure 5.3).

5.2.3 Task classification

Several tasks may have the same activity sequence but different properties such as input/output module names. For example, assume task1 and task2 have the same activity sequences but task1 modifies module$A$, and task2, module$B$. The common activity sequences are defined as **task class**, and tasks are defined as instances of the task class. For this example, we may define task class Task i, which modifies Module$X$. Properties depending on the particular instance, such as module name, are defined as instance variables.

[2]The class of representation language will be discussed in Section 6.

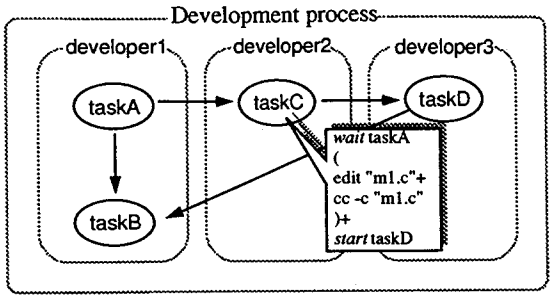


Figure 5.3: Overview of concurrent task model

5.2.4 Other alternative models

There are some alternative models for capturing concurrent tasks. For example, Kellner uses a state transition model in STATEMATE system[36]. Although it provides a simulation-based enaction mechanism with concurrent constructs, sophisticated communications can't be established. Also, its state chart representation cannot express abstractions such as task classification and parameterization. In [51], Saeki uses LOTOS (Language of Temporal Ordering Specification) which is based on the CCS model. It has so many constructs that the descriptions are difficult to describe, understand, and execute. To express specific perspectives of a process, this model would be overly complex.

Williams's SPM[57] is similar to ours and uses regular expressions extended with the "shuffle operator" to express concurrent activities. SPM also includes a message-passing definition. Although the shuffle operator is for simply describing process concurrency, SPM does not have sufficient functionality for the communications required for our purpose. In SPM, all concurrency synchronization are expected to be handled by the shuffle operator. Message-passing is subsidiary and does not provide additional synchronization mechanisms. However, some activity sequences, such as how the previous example process waited on a job, can't be specified using the shuffle operators.

5.3 The example problem description

We consider the 8 substeps in the example problem such as *ModifyDesign* and *ModifyCode* as tasks. The overview of the example process in the concurrent task model is shown in Figure 5.4.

There are some constraints on the initiation and termination of the tasks, e.g., "*Modi-*

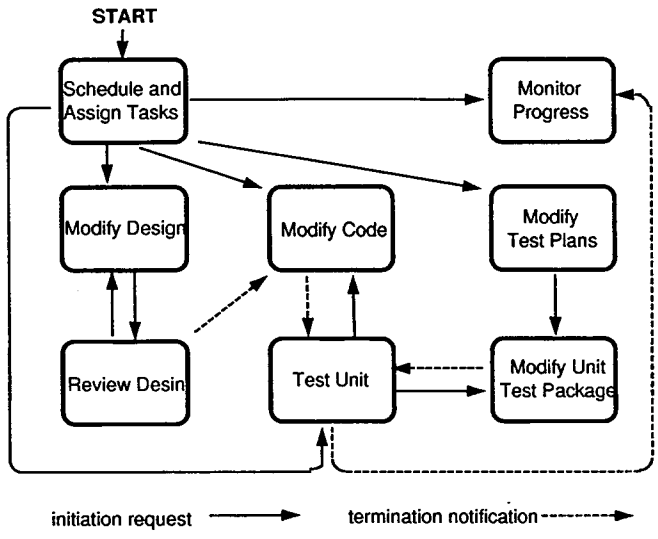


Figure 5.4: Concurrent task view of the example problem

fyCode can't terminate until ReviewDesign terminates," or "*TestUnit can't start until both ModifyCode and ModifyTestPackage terminate.*" These constraints are represented by task communications in the concurrent task model.

Figure 5.5 shows the example task definitions. For example, task *ScheduleAndAssign-Tasks* sends requests to 5 other tasks which are performed concurrently. *ModifyDesign* first repeats (represented by '+') actual modification work such as invoking an editor (<modify design>), and sends an initiation request to *ReviewDesign* (<start ReviewDesign>). These definitions are parameterized by the module names. By giving the actual module name to the definitions, we get an instance of the task definition.

5.4 Generation of the concurrent task monitor/ navigator

5.4.1 Supporting issues

In this section, we discuss ways of supporting and managing development using our concurrent task model.

Development support

The following issues are addressed here:

```
ScheduleAndAssignTasks =
   <start MonitorProgress> <start ModifyDesgin>
   <start ModifyCode> <start ModifyTestPlan>
   <start TestUnit>;
ModifyDesign =
   <modify desgin> + <start ReviewDesign>;
ReviewDesign =
   <review design>
   ( <start ModifyDesign> |
     <send "ModifyCode", "Result", "OK"> );
ModifyCode =
   ( ( <edit>+ <compile> )+ <peek "Result"> )+
   <recv "Result">;
ModifyTestPlan =
   <edit>+ <start ModifyUnitTestPackage>;
ModifyUnitTestPackage =
   <edit unitTestPackage>+;
TestUnit=
   ( <wait ModifyCode>
     <wait ModifyUnitTestPackage> <test>
     [ ( <start ModifyCode> |
         <start ModifyUnitTestPackage> |
         <start ModifyCode>
         <start ModifyUnitTestPackage> ) ] )+;
MonitorProgress = <wait TestUnit>;
```

Figure 5.5: Activity model description of the example problem

(1) Navigating activity

When a developer has many complicated assignments (which can be considered as tasks), it is difficult for the developer to know the following:

- How many tasks are there and which of these tasks are assigned?
- What activities follow each other in each task?
- Are there any tasks that are suspended and not yet resumed?

Information for answering these questions is essential when supporting the developer. automatically providing such information and navigating the developers through the activities are very important aspects to be considered.

(2) Communication support

By defining the processes based on the concurrent task model, the following would be clear:

- What kind of communication primitives are needed among the tasks ?
- What is the timing constraints of communication primitives ?

Using the definition, some simple communication primitives would be automatically executed.

Management support

For project managers, the following benefits could be expected when using the concurrent task model:

- The task definitions can be used as a measure of elaboration when assigning work to each developer.
- In the case of distributed development, communication among the development sites can be clarified by task communications in the model allowing communication costs to be assessed and reduced.
- By monitoring each task status, progress and workload can be estimated. Moreover, it is easier to find suspended tasks or frozen tasks (in dead lock) that still need to be completed.
- The concurrent task model can be used as a milestone (measure) of project progress.

We have developed a prototype system (Hakoniwa) for supporting and navigating the developers and the manager based on these issues.

5.4.2 The Hakoniwa system

System overview

Based on the concurrent task model, we have developed a prototype of a cooperative development support system called "**Hakoniwa.**"[3] Figure 5.6 shows an overview of the system architecture.

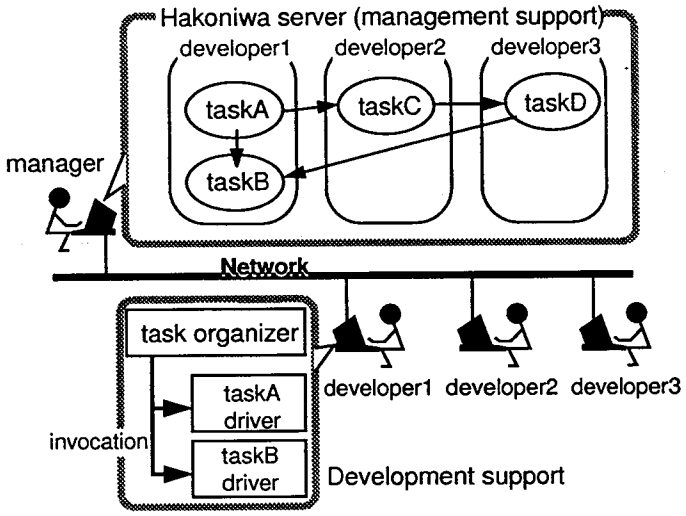


Figure 5.6: "Hakoniwa" system architecture

The concurrent task model description of all tasks in the development is defined in a monitor of tasks and communications named **Hakoniwa server.** Support/navigation managers for each developer are found in **task organizers.** A task organizer invokes and controls several task execution engines (**task driver**) instantiated for each task. Each task driver controls the sequence of activities. Products are manipulated through a product server instantiated from the product relation model in the composite software process model mentioned in Section 2.

Major features of the Hakoniwa system include:

[3] "Hakoniwa" is a Japanese word which means diorama or a small-scaled model or representation of a scene.

(1) Activity navigation

Based on the assignment of tasks to each developer (this may be made by a project manager), task organizers for each developer and task drivers for each task are generated. A task driver navigates the developer by providing menu selections for the next activities. These menus are automatically generated from the definition of the activity sequence. If an activity in the sequence is a primitive one accomplished by a tool invocation, the task driver automatically activates the tool.

(2) Progress monitoring

Each task driver reports to the Hakoniwa server log information concerning the task progress collected from the menu selection history. The project manager can assess the current status of the entire project through the Hakoniwa server. It displays the status of each task, and it also shows the history of activities for each developer.

(3) Communication support

All communications among the tasks are relayed through the Hakoniwa server. Simple communication primitives such as task initiation requests and task termination notifications are automatically executed without any action by the developers.

Implementation

The cooperative development environment is distributed over several distinct workstations connected via networks. The underlying mechanisms are the TCP/IP communication protocol of the *UNIX™* operating system. The current implementation works on a Sparc station running Sun OS 4.1.*.

A functional language for software process description, *PDL*[27], and its interpreter system have been extended in order to be used as the task drivers. The *PDL* interpreter has built-in functions for menu selection and tool invocation, and we have implemented the communication primitives. A task organizer is implemented as an application program in *PDL*. Each task organizer activates the task drivers corresponding to the assigned tasks.

The same approach as described in Chapter 4 is used to generate task drivers. Task drivers are generated from the task descriptions (grammar), adding restriction conditions for the selection expressions. Selection expressions such as '$A|B$' are translated into menu

selections in *PDL*. The sensitivities of selection items depend on their restriction conditions. For example, the edit-compile-link activities expressed as `(edit|compile|link)`$^+$ are activated by a menu which is composed of three items, "`editB`," "`compile`," and "`link`." The restriction condition for selecting "`link`" in this menu is the success of the previous "`compile`." Restriction conditions are determined by using the results of communication operations or the attributes of the concerned products[24, 23]. These restrictions are supplied to the generator as a set of *PDL* functions (Figure 5.7).

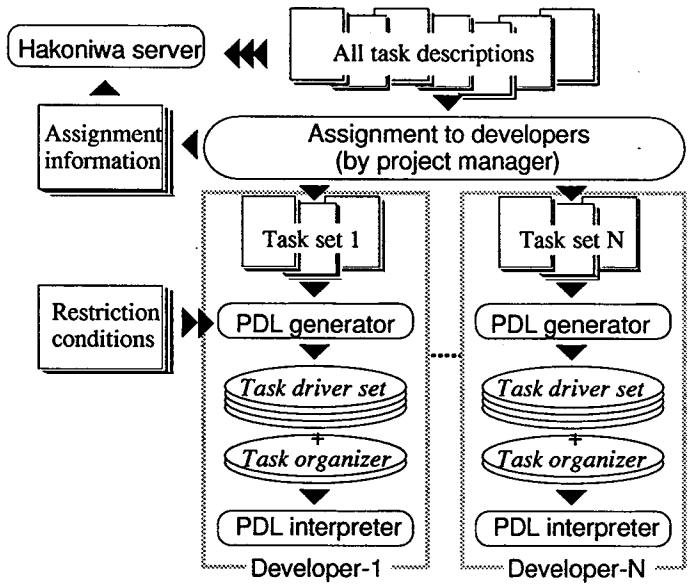


Figure 5.7: Generation flow in the Hakoniwa environment

Task communication

Communication operations described in Table 5.2 and Table 5.3 are implemented as built-in functions of the *PDL* interpreter. They are implemented using a server-client architecture, and all messages are relayed and controlled by the Hakoniwa server.

Task status display

The project manager cannot get an intuitive understanding of the current status and progress of the project by simply examining raw data such as the sequence of time stamps for initiation and termination of each activity. It is not easy to grasp the current status of the overall task given such limited information. The Hakoniwa system displays the activity

history and current activity as a tree form illustrating the regular expression of the activity sequence (Figure 5.8).

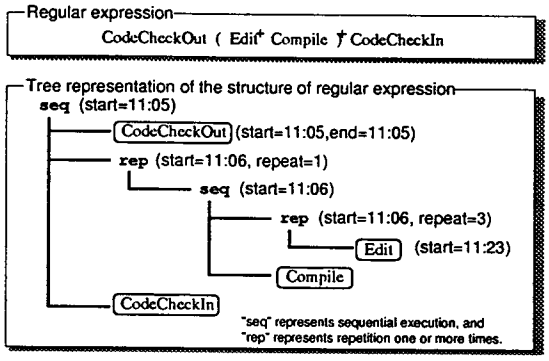


Figure 5.8: Regular expression and tree display of task

Figure 5.8 shows an example of a task defined by a regular expression and its structured status information. In this example, the developer first retrieves a program from the repository (represented by `CodeCheckOut`). After repeatedly editing (`Edit`) and compiling (`Compile`), the code is stored back into the repository (`CodeCheckIn`). The structure of the regular expression is shown with components `seq` and `rep` similar to those in Jackson's tree in the JSD method (`seq` means "sequel" and `rep` means "repeat one or more times"), and the overall tree shows the hierarchical structure with intermediate nodes for each sequel or repetition. The Hakoniwa server records and displays the initiation time and termination time for each node; for iteration nodes, it also displays the number of iterations. In this example, `Edit` started at 11:23 and was repeated 3 times. To display this information, the Hakoniwa server has an internal state transition map for each task. It keeps track of actual state transitions which are advanced by the activity sequence.

5.4.3 Enacting the example problem process

Task organizers and task drivers are obtained by translating the instantiated definitions into *PDL* programs.

By executing the obtained *PDL* programs, task organizers and task drivers are activated and developers are navigated through activities using menus similar to the one shown in Figure 5.9. The Hakoniwa server monitors the progress of each task and displays information shown in Figure 5.10, 5.11, and5.12. Figure 5.10 shows the top view of the concurrent task model description. Active tasks and communications are graphically displayed in real

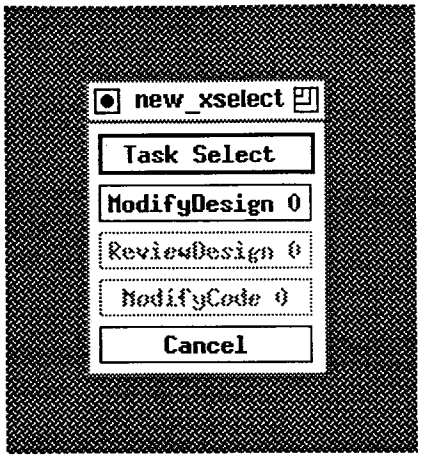


Figure 5.9: Menu displayed by the task organizer

time. Figure 5.11 shows a summary of activities performed by a developer. Figure 5.12 displays the internal status of a task.

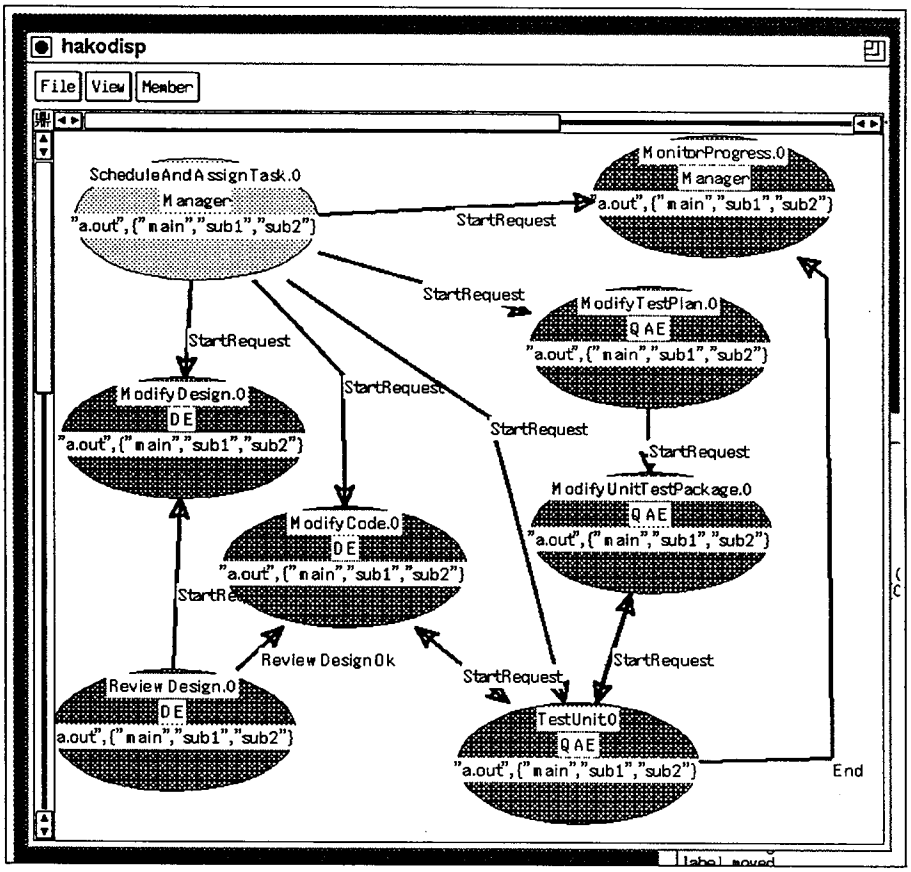


Figure 5.10: Overview display by Hakoniwa server

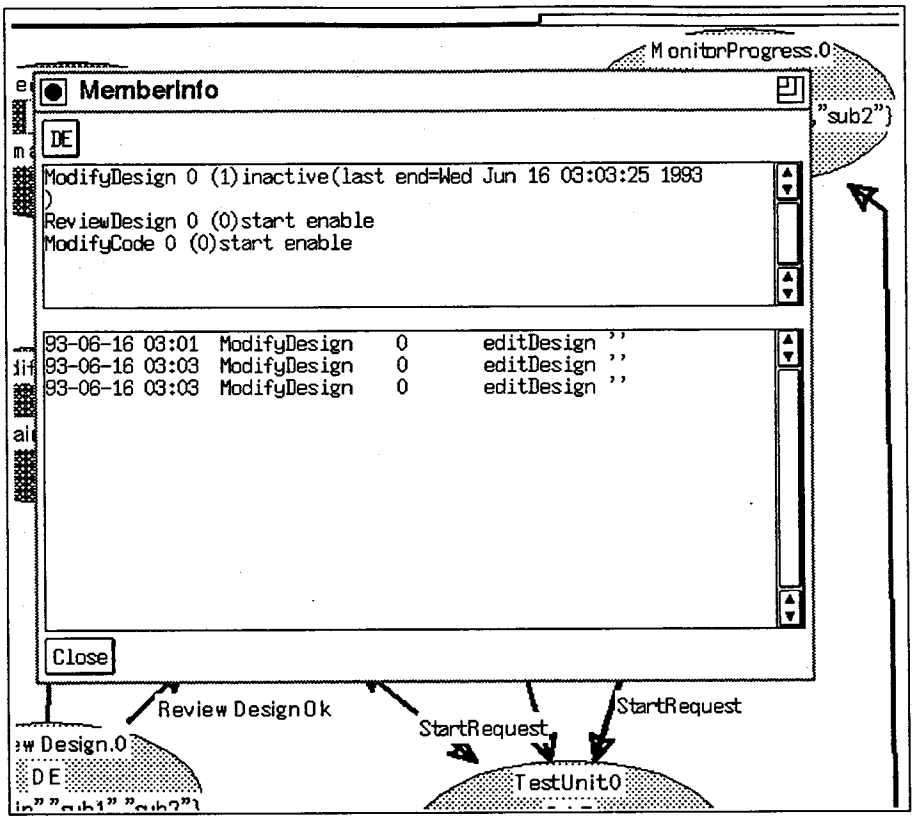


Figure 5.11: Activity summarization display

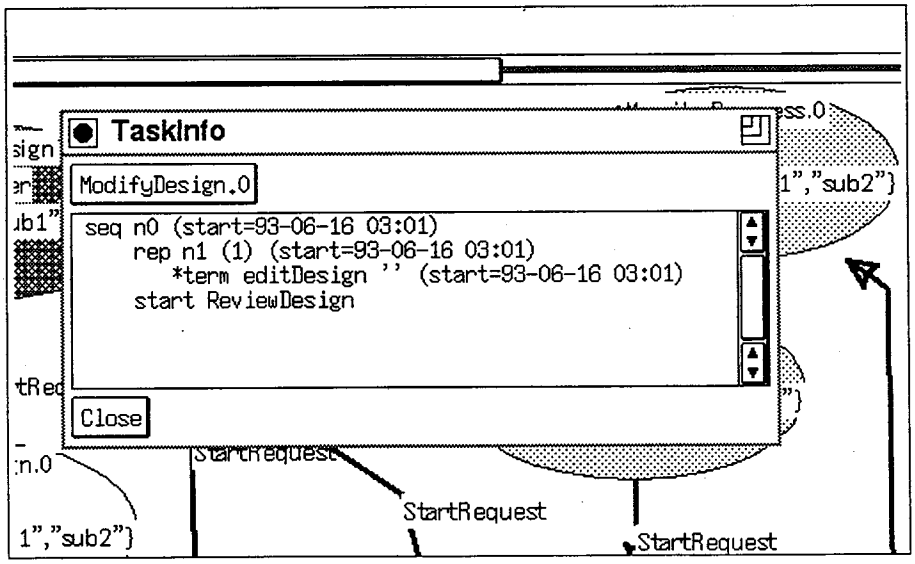


Figure 5.12: Task status display

Chapter 6

Product Relational Model

6.1 Product relational view

Both of previous models focused on the operations performed in the development process. In this chapter, we present the software process from the *products* view point on which operations are performed. We employ the object oriented approach for modeling products and their relations. The key issues proposed here are as follows:

- Software products are organized in hierarchical tree structures. A node in a tree represents a software product, or a set (composite) of software products and other nodes. With this hierarchical grouping mechanism of software products, we can represent various relations on software products very naturally and flexibly. For example, we can define correspondences between source code and object code, included and including files, specification files and source code, etc.

- These tree structures are established using types of products, such as source-file type and object-file type. We do not need to define the tree for each product one by one, but we define skeletons (classes) of actual products through these types, which can be applied to many different products in a project or in various different projects.

- The definition of activities applicable to each node is associated with the definition of the node. Operations in software development such as editing, compiling, and file-inclusion, are examples of these activities. Tools used for the operations, and the conditions determining activation of operations, are defined. Also, the status of a node, (e.g., the validity of products involved in the node) can be defined. Thus, the definition of a node is composed of two parts: the definition of the class (the hierarchy of types), and the definitions of the processes. These definitions provide a

very simple and composite view of process and product relations, which are generally shown in very complex and isolated ways.

- By defining every node, we can build an overall tree structure. Using the defined tree, we can construct a software development environment where software developers perform various operations automatically or manually, according to the definitions of the processes appearing in each node definition in the tree. Constructing the environment from the tree definition is very straightforward. The description of the product relation model (i.e., the definition of the tree structure) is transformed into an executable *PDL* script by a simple translator.

6.2 Definition of product relation model

6.2.1 Software product identification & classification

Product identification

There are various sorts of software objects produced during the development process: specifications, design documents, source files, inclusion files, object files, and executable files. They are categorized based on their properties, roles, and operations performed on them.

In this chapter, these objects (products) and the relations among these products are focused on. An object-oriented approach is used to describe the products. Compared to other object-oriented descriptions, however, the frame work for describing product properties is very simple and straightforward. We apply a grouping structure to the representation of each product and its relation.

Since Kellner's example problem, which is used in previous chapters, does not include the detailed description of its products, a simple development process for C programs (Figure 6.1) is used in this chapter.

Product classification

Software products can be divided into groups (types) according to their properties and their roles. For example, the types of files in C program development are categorized as follows:

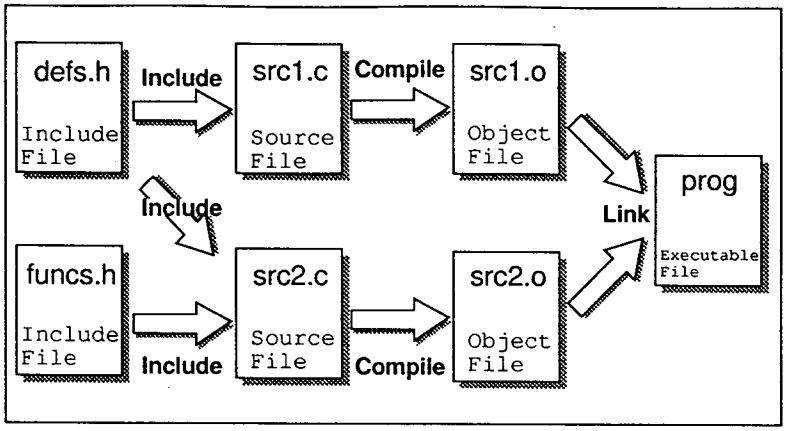


Figure 6.1: Example process of C programming

.h file	def.h, function.h
.c file	src1.c, src2.c
.o file	src1.o, src2.o
Executable file	prog

Furthermore, we can create product types by combining these basic types:

Program	composite of Modules and executable file
Module	composite of source and .o file
Source	composite of .h files and .c file

This classification is expressed in a tree structure as shown in Figure 6.2. In Figure 6.2, all of *real* files are assigned to terminal nodes (leaves) of the tree, and composite products correspond to non-terminal nodes (internal nodes).

This structure applies not only to a specific project but to the general C program development. For the each case of programming, a specific tree as shown in Figure 6.3 is instantiated.

Notation

We introduce a notation for describing the logical structures and the operations of each product. Note that this is not a notation for defining the physical structure of software products, but rather the logical structure which represents systematic operations on software products.

To define a composite type of software product (product "class" in the object-oriented world), we use the following notation:

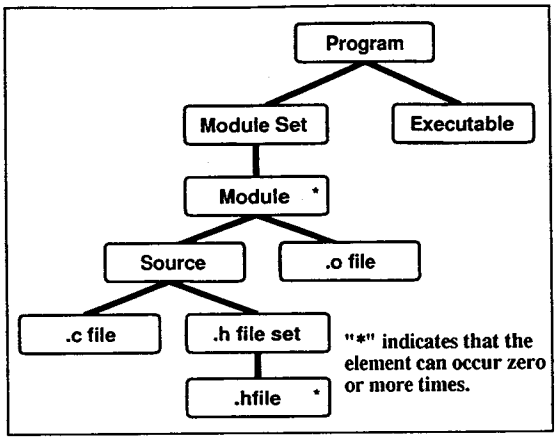


Figure 6.2: Product classification tree

```
#Class-Name( Component, ... )
{
        MemberFunction( Arguments ) = Expression;
        ...
}
```

Here, "#" indicates that this name is a *composite* type. A composite type product consists of *component products* and *member functions*. Each component may be defined as a composite product or a primitive product. Member functions represent operations on the product, and their notation is an extension of *PDL* [27].

For example, the product class *#header* , which is assumed to be a set of inclusion files, is defined as:

```
#header(hdset:#htext*)
{
        procedure() = textedit( oneof(hdset).body() );
}
```

The postfix mark "*" means a composition of any number of #htext. Each component is public and may be referenced by parent products in the form of *product.component*.

6.2.2 Manipulation supplement

Two member functions, *procedure* and *body* , are treated as special ones. *Procedure* is used to specify a sequence of actions to be performed. To define selective works, primitive function *menubranch* is prepared. This function corresponds to a primitive function in *PDL*

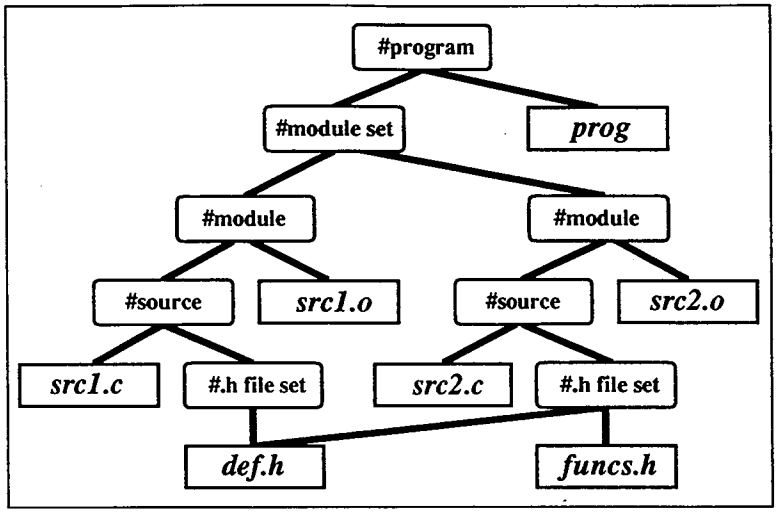


Figure 6.3: Product tree for a specific case

which allows user to select a work interactively from the system menu. Preconditions to be satisfied can be specified in the *procedure* function using an "if" clause. The precondition itself can be defined as a member function in the declarations of other related product. Function *body* is used to define the sub-products required to represent the product.

Procedure and *body* do not always have to be defined in the product definitions. Except for these two functions, there are no restrictions on defining the member functions which represent properties of the product. These member functions are accessible from the parent products.

For example, product class *#source*, which is assumed to be a composite of a C source file and its inclusion files, is defined as:

```
#source(inc:#header, src:#ctext)
{
        procedure () = if someprecondition() then
                menubranch(
                        textedit( src.body () ),
                        textview( src.body () ) );
        valid() = allexist(hdr.body ());
        body () = src.body ();
}
```

There are two activities, 'edit text' and 'view text', for the *source* class products. There is also a conditional function *,valid,* that is evaluated by the other products as part of their

preconditions. Function *body* specifies the subproduct that is used for the construction of products in upper levels.

An example definition for all product types in C programming are shown in Figure 6.4. These definitions are independent from the actual file names in a particular C program development and they can be applied to many projects.

```
#htext {
   body() = text;
}

#header( ht:#htext* ) {
   procedure() = menubranch(
                 edit(oneof(ht).body()),
                 view(oneof(ht).body()));
   valid() = allexist(ht.body());
}

#ctext {
   body() = text;
}

#source( hd:#header, ct:#ctext ){
   procedure() = if hd.valid() then
      menubranch( edit(ct.body()),
                  view(ct.body()));
   valid() = allexist(ct.body());
   body() = ct.body();
}
```

```
#obj{
   body() = text;
}

#module( so:#source, ob:#obj ){
   procedure() = if so.valid() then
                    compile(so.body());
   valid() = allexist(ob.body());
   body() = ob.body();
}

#object( mo:#module* ){
   body() = mo.body();
   valid() = alltrue(mo.valid());
}

#exe {
   body() = binary;
}

#program( obj:#object, ex:#exe ){
   procedure() = if obj.valid() then
          link(obj.body(),ex.body());
}
```

Figure 6.4: Product class definition for C programming

6.2.3 Product instantiation

By using type definitions as mentioned above, we can instantiate an actual product structure. To do this, we declare the unique names of the products used in the current project. For example, the real names of the inclusion files are described as:

```
#header   head1 = {"def.h"},
          head2 = {"def.h", "function.h"};
```

In this example, *head1* and *head2* are instances of #header. All declarations of product instances are shown in Figure 6.5. In Figure 6.5, we have assigned names to all non-terminal nodes of the tree structure. These name assignments can be omitted as shown in Figure 6.6 (i.e. both Figures 6.5 and 6.6 show the same assignments).

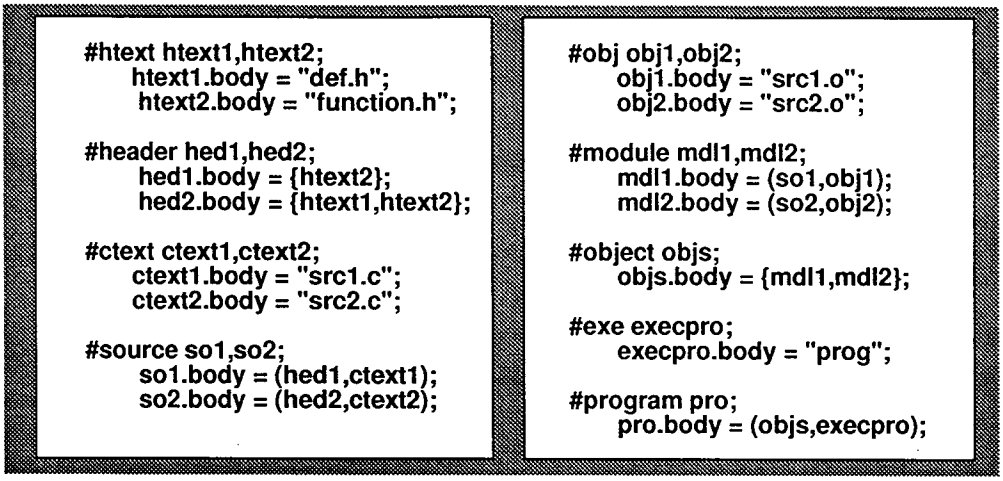


Figure 6.5: Product assignment declaration

```
#program prog = ( { ( ( {"def.h"},                    "src1.c" ), "src1.o" ),
                    ( ( {"def.h", "function.h"}, "src2.c" ), "src2.o" )
                  }, "prog" );
```

Figure 6.6: Simplified form of the product assignment declaration

6.3 Generation of a product manipulation system

6.3.1 Transformation of the description into PDL script

Each member function in the definition of the product structure can be transformed easily into a *PDL* script by renaming. For example, assume the declaration is:

```
#source(h:#header,c:#text)
{
        condition() = allexist(ct.body());
        procedure() = if h.valid() then
                        menubranch( edit(ct.body()),
                                    view(ct.body()));
}
```

and the name assignment is :

```
#source src1;
src1.body = ( hdr1, ctext1 );
```

Member functions *condition* and *procedure* are translated into *PDL* as follows:

```
src1_condition(S) == allexist(ctext1_body(S));
src1_procedure(S) == if hdr1.valid(S) then
                menubranch("src1-procedure",{
                        ["edit",edit(ct_body(S))],
                        ["view",view(ct_body(S))]
                },S) else S;
```

Here, argument S is a variable of type <*state*>, and functions such as *src1_procedure, edit,* and *view* are defined as state transition functions.

The obtained *PDL* script consists of function definitions. The definition of a product type is translated into a set of functions. To execute (evaluate) these functions systematically, the translator also outputs a scheduling function as an entry point for the execution.

6.3.2 Executing the script

Using the *PDL* interpreter, the obtained *PDL* script acts as a menu-based C programming support system. It produces an operation menu for each product. The project proceeds by selecting an operation from a menu associated with a product (Figure 6.7). As shown in the figure, a product which does not satisfy its pre-conditions is grayed in the menu. This means that there are currently no possible or needed activities to perform.

The facilities provided by this system involve those provided by the *make* program in *UNIX™* . According to the tree structure of the products, this system produces the target products assigned to the non-terminal nodes of the product tree. Moreover, our system shows the status of the products visually, and leads the developer through appropriate actions.

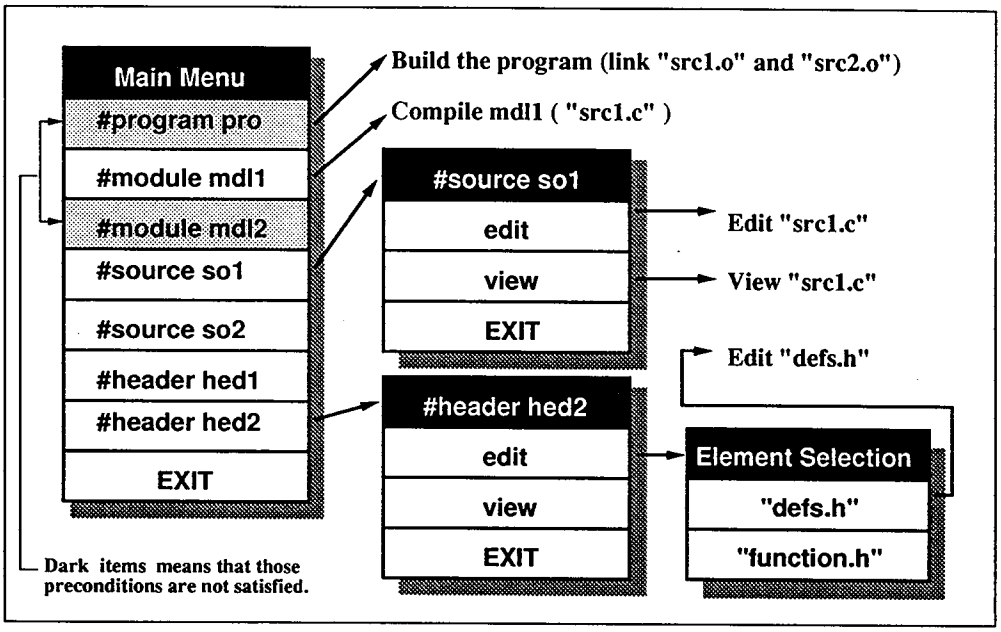


Figure 6.7: Menu images during execution

Chapter 7

Discussion

7.1 Criteria for model selection

A model description is an abstract description of the target process with a specific view, and it is often employed when use of the complete description is undesirable or impractical[12]. Although three specific models has been discussed, the factors presented here are applicable to other models with different views such as resource assignment and project management.

It is reported that various goals and objectives of software process modeling can be clustered into five objectives: Automated execution/control, Human interaction, Management, Understanding, and Analysis[34]. Our primary concerns at model definition time are "Understanding" the overall structure of the target process and "Automated execution/control" of process-centered development environments. Representative requirements for understanding the process have been listed in Kellner[34]. These requirements are: (a) easy to understand, (b) unambiguous, (c) compendious, (d) representing human procedural level of information, (e) supporting multiple levels of abstraction/refinement, (f) view mechanism for various roles, and (g) representing multiple perspectives.

The criteria and the factors have been chosen based on the requirements for the model-type definition step. These partly cover the above requirements (i.e., (a), (b), (c), (d), (f), (g)), and the remaining requirement (i.e., (e)) is also reflected. Criteria and factors guiding model selection, which may overlap or closely relate to one another, are described as follows:

(1) Pictorial and Textual Representation of Models

Syntactic representations of models may be roughly classified into two categories: pictorial and textual. In a pictorial representation, simple graphs often composed of boxes, circles,

and edges are used. In textual forms, both formal languages (subset/superset of various computer languages or newly designed ones) and natural language annotations are widely used. Selecting an appropriate representation is very important since the representation heavily affects the comprehensibility of the structure of the processes and the expressiveness of the model descriptions as discussed later. We have used simple pictorial representations in most cases (as shown in Figure 1, for example), since the target processes we have described were unknown to us and we had to understand them by repeatedly drawing pictorial sketches.

(2) Formality of Semantic Definition

One major concern with the semantics of process models is formality. Some models have formal semantic definitions based on their background mathematical definitions. For example, pure Petri net and state transition diagram are mathematical models having formal meanings. Since these pure mathematical models represent limited views of the processes, we sometimes want to extend such models to involve informal expressions, such as natural language annotations, or undefined pictorial structures.

Informality in the model is justified and sometimes essential, since when we first construct a model for an unfamiliar development process, we may try various formal and semi-formal models concentrating our attentions only on the points which interest us. Even using model definitions with rigorous formal semantics, the consistency and completeness of the model description cannot be guaranteed automatically. On the other hand, we would be able to write less ambiguous descriptions of the models if enough attention is paid.

Formal models are, for example, based on the Petri net graph and context free grammar, and semi-formal ones contain pictorial structures which leave room for interpretation during later refinement steps.

Criteria for choosing the level of formality are affected by the objectives of the process descriptions. If we need the description only for understanding the overall structure of the development and no detailed analysis or automatic enaction of the process description is necessary, informal constructs may be used in the model. If enaction for a development support environment and/or for machine simulation is expected, we should pursue high formality at the initial steps.

(3) Clarity of Description

This factor is very important since one of the purposes of process modeling is to represent the target development process in easily understandable ways. The clarity of the model is a great contribution towards understanding the process. It seems that the clarity of a model is directly based on the clarity of the syntactic and semantic definitions. We have no established metrics for measuring these factors, although techniques for measuring graph and text complexity would be used for measuring the syntactic complexity[32]. It seems intuitive that clarity increases as complexity decreases.

(4) Compactness of Description

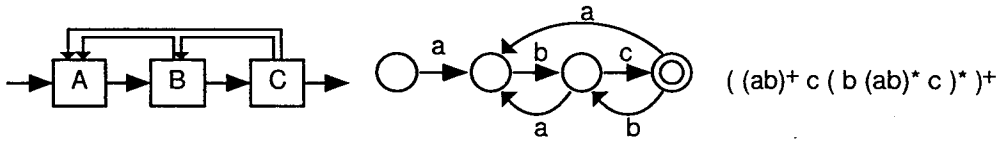
This factor is directly based on the complexity of the semantic definition. If a small model describes abundant meaning, it generally seems that the semantic definition must be rather complicated. Simple semantic definitions tend to require complex model descriptions in order to depict a certain development process. In our framework, we generally put higher priority on clarity, as apposed to compactness. This is because we wanted to concentrate on simplistic views of the processes so that we could easily grasp their overall structures. The information included in the model definitions was limited and the model descriptions became more verbose.

7.2 Language class of behavior specification

7.2.1 Language class of activity sequence

In the concurrent task model, the activity sequences of tasks are defined with regular expressions. However, there exist complex processes which can't be specified well by regular expressions. In some cases, such as interleaving sequences, one may split them into communicating parallel subsequences each of which can be expressed by a regular expression. In other cases, as shown in Chapter 4, more powerful language classes such as context free grammars are needed for more complex sequences containing recursion. Another problem with regular expressions[24, 23] is that some constraints can't be simply expressed even if they were represented with a finite state machine. For example, a simple waterfall activity sequence A-B-C which may contain backtracking loops (Figure 7.1a) is easily represented by deterministic finite automaton (DFA) as shown in Figure 7.1b. However, the regular expression representation does not efficiently reflect the nature of this process (Figure 7.1c).

On the other hand, regular expressions are suitable for static analyses such as dead-lock detection, since many operations on the regular expressions are decodable while those on context free grammars are not in many cases. Another merit of regular expressions is the simplicity of the history display. The Hakoniwa system displays the progress of tasks with tree structures of regular expressions. If we used context free languages, the tree easily grows huge in size since it allows recursion. Thus, it is difficult to display and we can't easily determine actual progress through the tree.



(a) Activity Flow (b) DFA Representation (c) Regular Expression

Figure 7.1: Representations for simple A-B-C sequence

7.2.2 Process enaction

As mentioned in Chapter 3, process descriptions are enacted to provide software development environments and to simulate process steps and check their validity.

In Chapters 4 and 5, Kellner's Example Problem has been enacted and the behavior of each activity step has been checked through the actions of windows corresponding to each step. It is possible to get statistical information concerning the development, if we introduce into the description statistical operators having various probability distributions, such as an operator whose active time period is given by a normal distribution.

A similar idea is presented in Kellner[36], in which a state transition model is assumed and simulation is the main purpose. The approach proposed in this thesis has no specific presumed models and uses descriptions for other purposes in addition to simulation.

By invoking various tools in the descriptions, different kinds of software development environments may be established. These software development environments adapt a workstation-based collection of tools to a particular organization, project, or individual and integrate existing tools as proposed by Software Designers' Associate (SDA)[37]. The conceptual issues concerning development of SDA affected the design and implementation of *PDL*.

The *Arcadia* Project [54] has objectives similar to those of the SDA project. The Arcadia project is attempting to create a development environment for very large and complex projects by automating aspects of lifecycle activities and building up collections of tools. *Tame* [5] has related goals, but focuses mainly on collecting information on a developer's activities and reflecting this information back to the developer so that the feedback can positively affect productivity and quality may be obtained.

7.3 Process telemetering

7.3.1 Hakoniwa system as a monitoring environment

Research efforts into product metrics for determining development status have been pursued[5, 39]. Product metrics such as lines of code, number of faults, number of control paths, and many others are used for estimating completeness of the activities. However, most of this research only proposes the product metrics without any underlying process models. However, the metrics for measuring products closely depend on the development process, and if the process changes then the metrics have to be changed also. Introducing process models into product metrics environment is essential so that we can determine the appropriate metrics and more accurately assess the current status of the development.

As shown in Chapter 5, the Hakoniwa system provides data which helps to assess the project's progress, such as:

- Number of iterations of each activity
- Time duration of each activity (initiation and termination time)
- Current activities

An experienced manager may easily comprehend the project progress from these data; however, it is insufficient to only provide the data; it is also desirable to have goal values for these data[5]. It is difficult to set such goals and to estimate progress only from single project data. For example, even if we have the data on the number of iterations at a given moment, we can't predict the total number of iterations by the end. However, if we had data for similar projects, we could assess the current status using this analogous data.

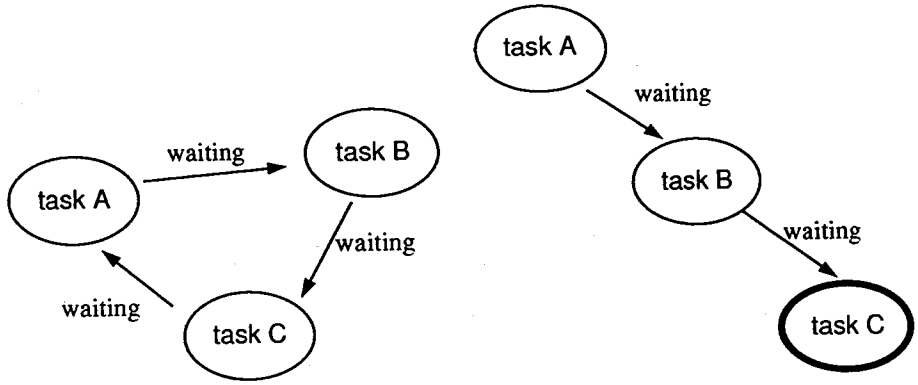
In order to perform this kind of statistical prediction effectively, it is essential to store large numbers of project profiles. The Hakoniwa system collects such data automatically, without incurring data collection costs. The collected data are more reliable than data

collected by hand such as reports from developers. Furthermore, the collected data are directly used for evaluation of the current status and for project profiling.

7.3.2 Dead-lock detection in the concurrent task model description

The concurrent task model contains dead-lock possibilities as do many other concurrent models. Dead-lock is defined here as: "Infinitely waiting for a message, causing task execution to freeze"[1]. According to the shape of the dependency graph depicting tasks waiting for messages, we consider the following two kinds of dead-lock:

- Linear waiting – A non-circular path whose start node does not send a message.
- Circular waiting – A circular path.



(a) Circular waiting (b) Linear waiting

Figure 7.2: Task deadlocking patterns

To statically detect the existence of dead-lock in the description, we focus on the types of messages, especially on initiation requests and termination notification messages. For these message types, we look for the following conditions:

- For linear waiting:
  In a directed graph showing only the relation of initiation requests, there is a task unreachable from the task initially activated.

[1]Here we do not consider task freezes caused by exclusive data access, since the product model is not in the concurrent task model.

- For circular waiting:

  In a directed graph showing only the relation of termination notifications, there is a cycle.

In the case of dynamic detection during the execution, linear wait dead-locks have to be detected by human analysis, while circular wait dead-locks can be detected automatically by checking for cycles in the message waiting paths.

Chapter 8

Conclusion

8.1 Summary

This thesis has proposed a new method of modeling software processes and enacting the modeled process descriptions. The modeling steps (the model definition step and the model description step) do not rely on a specific model definition but allow appropriate ones to be chosen. The formalization step is performed using the language *PDL*, and refinement and enaction steps are made on the basis of *PDL* and its system. The overall structure of this method is simple and straightforward but very effective, and actual applications of this method have been demonstrated in this thesis.

In Chapter 4, a model based on a behavioral view was proposed, and an activity navigation system was generated from the description of the model. A method for generating a development assistance system with flexible process control from the grammar description was shown. The obtained system allows for flexible activity selection, valid sequence generation, and ease-of-use via a restricted menu mechanism.

In Chapter 5, a model focusing on concurrent development and communications performed by several developers was constructed. The Hakoniwa process monitoring system was constructed based on this model. Hakoniwa controls tasks performed by multiple developers and displays their progress status. It also provides menus that guide the developers towards appropriate next steps. At the same time, data for analyzing the progress status of the project are automatically collected and provided to the project manager. These data are useful for estimation and prediction of the project as progress.

In Chapter 6, a simple product oriented model was defined and a product manipulation system was obtained from the model description. Using this model, relations among products are easily and naturally expressed for many cases of software development. The

hierarchical tree structure employed here is a very good representation for organizing product relations.

Through these experiences, we find that similar models were repeatedly used. Thus, model reuse, i.e.; retrieving existing models and adapting them to the target process are very important issue. Mechanisms for supporting the reuse and evolution of process models is an emerging research topic.

The method discussed here mainly targets modeling, describing, and enacting software processes. Other activities in the software process lifecycle, such as requirement analysis, process data analysis, and evaluation, are not studied here. These activities would need to be included when creating a support environment for the entire process lifecycle.

8.2 Future work

Several types of view-specific process models have been proposed in this thesis. Composing these models into one complex process model is a challenging theme. Since the proposed approach uses PDL as a basis for formalisms and semantics of model definitions, it may be possible to define the interactions among these model definitions by describing the correspondence rules of every element in the models in PDL. There is an on-going attempt to extend Hakoniwa system prototype into an integrated development support environment which supports multiple process views such as the concurrent task model, the product relation model, and an organization model. This version of the Hakoniwa system will partially support dynamic modifications of the software process also. Moreover, the merger of some existing product management server; such as Portable Common Tool Environment (PCTE)[8], into this approach should be investigated.

Another important future topic is applying these process models to practical development processes to collect data on real processes. Such data would be useful for examining the models as well as for examining the processes.

Bibliography

[1] Ambriola, V., Ciancarini, P. and Montangero, C., "Software process enactment in Oikos," in *Proceedings of 4th Symposium on Software Development Environments.((ACM SigSoft Software Engineering Notes, 15, 6),* pp.183-192, December 1990.

[2] Avrunin, G.S., Dillon, L.K., Wileden, J.C. and Riddle, W.E., "Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems," *IEEE Transactions on Software Engineering,* vol.SE-12, no.2, pp.278-292, 1986.

[3] Balzer, R., "Tolerating inconsistency," in *Proceedings of 13th International Conference on Software Engineering,* pp.158-165, May 1991.

[4] Basili, V.R., Caldiera, G. and Cantone, G., "A reference architecture for the component factory," *ACM Transactions on Software Engineering and Methodology,* vol.1, no.1, pp.53-80, January 1992.

[5] Basili, V.R. and Rombach, D.H., "The TAME project: Towards improvement-oriented software environments," *IEEE Transactions on Software Engineering,* vol.14, no.6, pp.758-773, June 1988.

[6] Boehm, B.W., "A spiral model of software development and enhancement," *IEEE Computer,* vol.21, no.5, pp.61-72, May 1988.

[7] Booch, G., *Object-Oriented Design with Applications* , Benjamin/Cummings, 1990.

[8] Boudier, G., Gallo, F., Minot, R. and Thomas, I., "An overview of PCTE and PCTE+," in *Proceedings of 3rd Software Development Environments Symposium(ACM SigSoft Software Engineering Notes, 13, 5),* pp.248-257, November 1988.

[9] Cameron, J.R., "An overview of JSD," *IEEE Transactions on Software Engineering,* vol.12, no.2, pp.222-240, February 1986.

[10] Curtis, B., Kellner, M. and Over, J., "Process modeling," *Communications of the ACM,* vol.35, no.9, pp.75-90, September 1992.

[11] Cusumano, M.A., *Japan's Software Factories* , Oxford University Press, 1991.

[12] Dart, S.A., Ellison, R.J., Feiler, P.H. and Habermann, N., "Software development environments," *IEEE Computer,* vol.20, no.11, pp.18-28, November 1987.

[13] Deiters, W. and Gruhn, V., "Managing software processes in the environment MELMAC," in *Proceedings of 4th Software Development Environments,* pp.193-205, December 1990.

[14] DeMarco, T., *Structured Analysis and System Specification* , Prentice-Hall, 1978.

[15] Harel, D., Lachover, H. and Naamad, A., "STATEMATE: a working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering,* vol.16, no.4, pp.403-414, April 1990.

[16] Hoare, C.A.R., *Communicating Sequential Processes* , Prentice-Hall, 1985.

[17] Hoare, C.A.R., "An axiomatic basis for computer programming," *Communications of the ACM,* vol.12, no.10, pp.576-583, October 1969.

[18] Hopcroft, J.E., Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation* , Addison Wesley, 1979.

[19] Iida, H., Mimura, K., Inoue, K. and Torii, K., "Hakoniwa: monitor and navigation system for cooperative development based on activity sequence model," in *Proceedings of 2nd International Conference on the Software Process,* pp.64-74, February 1993.

[20] Iida, H., Mimura, K., Inoue, K. and Torii, K., "Modeling cooperative development process and prototyping a monitor/navigation system," *Transactions of IPSJ,* (conditionally acceptted and resubmitted).

[21] Iida, H., Mimura, K., Inoue, K. and Torii, K., "Hakoniwa – a process monitoring system for cooperative development management," IEICE Japan Technical Report, SS91-38, pp.31-38, 1992,(in Japanese).

[22] Iida, H., Nishimura, Y., Inoue, K. and Torii, K., "Generating software development environments from the description of product relations," in *Proceedings of 15th Computer Software and Applications Conference,* pp.487-492, September 1991.

[23] Iida, H., Ogihara, T., Inoue, K. and Torii, K., "Formalizing software development activities and generating navigation system," *Transactions of IPSJ,* vol.34, no.3, pp.523-531, March 1993.

[24] Iida, H., Ogihara, T., Inoue, K. and Torii, K., "Generating a menu-oriented navigation system from formal descriptions of software development activity sequence," in *Proceedings of 1st International Conference on the Software Process,* pp.45-57, October 1991.

[25] Iida, H., Okada, Y., Inoue, K. and Torii, K., "Experience of solving example problem for software process," *Transaction IEICE,* vol.E76-D, no.2, pp.302-306, March 1993.

[26] Inoue, K., Ogihara, T., Kikuno, T. and Torii, K., "A formal adaptation method for process descriptions," in *Proceedings of 11th International Conference on Software Engineering,* pp.145-153, May 1989.

[27] Inoue, K., Ogihara, T., Iida, H. and Nitta, M., "A functional language for enacting software processes," in *Proceedings of 15th Computer Software and Applications Conference,* pp.219-224, September 1991.

[28] Jackson, M.A., *System Development* , Prentice-Hall, 1982.

[29] Kaiser, G.E. and Feiler, P.H., "An architecture for intelligent assistance in software development," in *Proceedings of 9th International Conference on Software Engineering,* pp.180-188, 1987.

[30] Kaiser, G.E., Feiler, P.H. and Popovich, S.S., "Intelligent assistance for software development and maintenance," *IEEE Software,* vol.5, no.3, pp.40-49, May 1988.

[31] Kajitani, K., Ito, M., Matsuura, T., Taniguchi, T. and Kasami, T., "An algebraic specification of an office work and its verification – a specification of a book purchasing work –," IEICE Japan Technical Report, AL84-38, pp.51-62, 1984,(in Japanese).

[32] Kataoka, Y., Matsumoto, K., Kumamoto, A. and Torii, K., "A new metric for impartiality of hierarchical data flow diagram in structured analysis method," IEICE Japan Technical Report, KBSE92-18, pp.33-40, July 1992,(in Japanese).

[33] Katayama, T., "A hierarchical and functional software process description and its enaction," in *Proceedings of 11th International Conference on Software Engineering,* pp.343-352, May 1989.

[34] Kellner, M., "Multiple-paradigm approaches for software process modeling," in *Proceedings of 7th International software Process Workshop,* pp.82-85, October 1991.

[35] Kellner, M., "Software process modeling example problem," in *Proceedings of 1st International Conference on the Software Process,* pp.176-186, October 1991.

[36] Kellner, M., "Software process modeling support for management planning and control," in *Proceedings of 1st International Conference on the Software Process,* pp.8-28, October 1991.

[37] Kishida, K., Katayama, T., Matsuo, M., Miyamoto, I., Ochimizu, K., Saito, N., Sayler, J.H., Torii, K. and Williams, L.G., "SDA: a novel approach to software environment design and construction," in *Proceedings of 10th International Conference on Software Engineering,* pp.69-79, April 1988.

[38] Kudo, H., Sugiyama, Y., Fujii, M. and Torii, K., "Quantifying a design process based on experiments," in *Proceedings of 21st Hawaii International Conference on System Sciences,* pp.285-292, January 1988.

[39] Kusumoto, S., Matsumoto, K., Kikuno, T. and Torii, K., "On a measurement environment for controlling software development activities," *Transaction IEICE,* vol.E73, no.5, pp.1051-1054, 1991.

[40] Madhavji, N., "Environment evolution: The prism model of changes," *IEEE Transactions on Software Engineering,* vol.18, no.5, pp.380-392, May 1992.

[41] Madhavji, N. and Schäefer,W., "Prism – methodology and process-oriented environment," *IEEE Transactions on Software Engineering,* vol.17, no.12, pp.1270-1283, December 1991.

[42] Matsunaga, Y., Iida, H., Ogihara, T., Inoue, K. and Torii, K., "Process description and enaction system based on graphical representation," *Transaction of IEICE,* vol.J76-D-I, no.6, pp.324-326, June 1993.

[43] Mi, P. and Scacchi, W., "Modeling articulation work in software engineering processes," in *Proceedings of 1st International Conference on the Software Process,* pp.188-201, October 1991.

[44] Ogihara, T., Inoue, K. and Torii, K., "Process description and enaction model based on objects," IPSJ Technical Report, SE70-4, pp.1-5, December 1989.

[45] Osterweil, L.J., "Software processes are software too," in *Proceedings of 9th International Conference on Software Engineering,* pp.2-13, April 1987.

[46] Penedo, M.H. and Riddle, W.E., "Software engineering environment architectures," *IEEE Transactions on Software Engineering,* vol.14, no.6, pp.689-696, June 1988.

[47] Perry, D. and Kaiser, G., "Models of software development environments," *IEEE Transactions on Software Engineering,* vol.17, no.3, pp.283-295, March 1991.

[48] Peuschel, B. and Schäfer,W., "Concepts and implementation of a rule-based process engine," in *Proceedings of 14th International Conference on Software Engineering,* pp.262-279, May 1992.

[49] Riddle, W.E., "Software designer's associates: A preliminary description," in *Proceedings of 20th Annual Hawaii International Conference on System Sciences,* pp.371-381, 1987.

[50] Royce, W., "TRW's ada process model for incremental development of large software systems," in *Proceedings of 12th International Conference on Software Engineering,* pp.2-11, March 1990.

[51] Saeki, M., Kaneko, T., Sakamoto, M., "A method for software process modeling and description using LOTOS," in *Proceedings of 1st International Conference on the Software Process,* pp.90-104, October 1991.

[52] Sommerville, I., *Software Engineering* , Addison Wesley, 1989.

[53] Sutton,S.M.,Jr.,, Heimbinger, D. and Osterweil, L.J., "Language constructs for managing change in process-centered environments," in *Proceedings of ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT Software Engineering Notes, 15-6),* pp.206-217, December 1990.

[54] Taylor, R.N., Belz, F.C., Clarke, L.A., Osterweil, L., Selby, R.W., Wileden, J.C., Wolf, A.L. and Young, M., "Foundations for the arcadia environment architecture," in *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments, (SIGSOFT Software Engineering Notes, 13-5),* pp.1-13, November 1988.

[55] Torii, K., Morisawa, Y., Sugiyama, Y. and Kasami, T., "Functional programming and logic programming for the telegram analysis problem," in *Proceedings of 7th International Conference on Software Engineering,* pp.57-64, March 1984.

[56] Weber, H., *Towards a Software Factory Reference Model* , Tutorial Text of COMPSAC'91, 1991.

[57] Williams, L.G., "Software process modeling: A behavioral approach," in *Proceedings of 10th International Conference on Software Engineering,* pp.174-186, April 1988.

[58] *Proceedings of 1st International Conference on the Software Process* (Redondo Beach, CA, 1991) .

[59] *Proceedings of 6th International software Process Workshop: Support for the Software Process* (Hakodate, Japan, October 1990) .

[60] *Proceedings of 7th International software Process Workshop* (Yountville, CA, October 1991) .