Hakoniwa: Monitor and Navigation System for Cooperative Development Based on Activity Sequence Model

Hajimu IIDA†, Kei-ichi MIMURA†, Katsuro INOUE†, and Koji TORII†‡

†Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University

‡Graduate School of Information Science, Advanced Institute of Science and Technology, Nara

Abstract

This paper proposes a process model for the software development performed by a group of developers. The model is defined with a set of tasks assigned over the developers and performed by them concurrently. A task is defined with a sequence of primitive activities, and it may involve communication primitives which establish coordination with other tasks. By this model, reworks and backtrackings are easily represented with repetitions in the regular expressions. Interactions among developers are denoted by the tasks communication. The current status of each task and each developer is easily followed, and the progress of the overall project would be effectively known. Based on this model, a prototype system for supporting and monitoring process, named "Hakoniwa", has been implemented. The Hakoniwa system is composed of multiple developers' navigators and a manager's monitor. A navigator leads a single developer by showing possible succeeding activities on menus and automatically activates appropriate development tools. The monitor reports the advances in activity sequences of developers, with various data such as the initiation-termination times and the number of activity repetitions. A commonly used example problem, proposed by Kellner et al., is used to show the applicability of our model and system.

1 Introduction

Including the pioneer research of process programming[20], there have been numerous efforts of modeling software development processes in some formal or semi-formal manners[6, 15]. Software development processes are abstractly defined with respect to activities, decisions, and environments of the developments. By modeling the software process, we may explicitly understand the development process, which might have been unconsciously used by the developers. Once the process model is established, we can figure out the characteristics of the process on the basis of the model. The notion of the process is easily transferred to other people through the model more easily and less ambiguously. Furthermore, with the process model, we are able to define the structure of the software development environment, which supports the development process.

It is also expected that process models would help to achieve good management for various activities in the process, as well as other prospects mentioned above. Many software developers, especially Japanese mainframe computer manufacturers, have developed and applied their own standardized development processes with management point of views, most of which are based on traditional water-fall models[4]. Using the standardization, the current status of the development is expected to be estimated[26]; however, actual development activities involve many interruptions, reworks, backtracking, and many other exceptions, which are not usually well described in the standardization. Therefore, comprehending the current status only through the advance of the standardized process is fairly difficult. In most cases, experienced managers check the various documents and source codes, and they assess the current status without relying on the progress of the standardized process.

Concerning on the issue of knowing the current status of the development, researches of product metrics have been pursued[2, 18]. The product metrics such as lines of code, number of faults, number of control path, and many others are used for estimating completeness of the activities. However, most of these researches only propose the product metrics without any underlying process models. It is naturally observed that the metrics to measure certain products are closely depend on the development process, and if the process changes then the metrics have to be changed also. Introducing the process models into product metrics environment would be essential so that we can determine the metrics and expect better assessment of the current status of the development.

The actual software development is performed with cooperation of many developers in general, although the standardized processes do not concern explicitly about the various issues relating to multiple developers. For example, it is essential to note coordination issue of the developers such as control of concurrency and synchronization of activities for each developer. Also, it seems very important for the standardization to include data collection issue, while currently there is no implication to estimate the current status of each developer under the multiple developers environment.

In this paper, we will discuss process models focusing on managing software processes with multiple developers, which are indispensable for these issues. The models have to define activities and their relations clearly. Based on such models, navigation of activities would be straightforwardly provided, and monitoring each developer's progress would be easily accomplished.

We propose here a process model for the multiple developers' software development, mainly concerning

The authors' address: Osaka university, Toyonaka, Osaka 560, JAPAN. Internet mail address: iida@ics.osaka-u.ac.jp.

on the coordination issue of activities of each developer. In our previous work, we have modeled development processes performed by a single developer with activity sequences specified as CFG (context free grammar)[10]. We assumed that the activities performed by a single developer are basically sequential, and the formal grammars are considered to be a very good vehicle to represent them. However, that model can not handle parallelism which is essential to represent the multiple developers' work. Thus, we decided to design a new model for the multiple developers' activities, importing the previous model to define the activities of the individual developers. The model defines the development process with a set of tasks with communication primitives.

Communication primitives are defined and involved in the tasks, which control the progress of activities and organize the overall tasks in the model, and each developer is responsible for accomplishing several tasks. By this model, reworks and backtrackings are easily represented with repetitions of activities in the regular expressions. Suspending and resuming works are denoted by tasks for a developer with communication. Using this model, also the current status of each task and each developer is easily followed, and the progress of the overall project would be effectively known to the project manager.

We will also discuss the use of this model for navigation and monitoring of each developer. Based on the proposed model, we have developed a prototype system, named Hakoniwa (which means diorama in Japanese), for the navigation and monitoring under cooperative development environment. This system provide menus for next proper activities to each developer. It also collects and displays data of current status of the developers, which are served to the project manager. The project manager easily comprehends the progress of each task assigned to each developer.

We have used the Hakoniwa system in order to model and construct an environment for the standard example problem[16]. The process of the problem is composed of eight tasks, each of which corresponds to primary substep of the problem. The constraints and communications among substeps are naturally represented by communication primitives introduced to each task.

Our approach of process modeling, with specific views of multi-developers' activities and their synchronization and communication, seems a novel one, while there are related researches on modeling processes through various views[13, 14, 24, 28, 29, 30]. Also, we have proposed a method of building environment from the model, and actually built a prototype system. Applicability of our approach would be partially presented by developing an environment for the example problem.

Representation methods of tasks have been examined; we have compared language classes for the representation of activity sequences between regular expressions and context free grammars. Also, the deadlock detection method which is a important issue of multi-developers' environment has been studied.

In the followings, we will propose the process model in Section 2, and will discuss a method to support development and management with this model in Section 3. The Hakoniwa system will be described in Section 4, and an experience of use of Hakoniwa for Kellner's example problem is presented in Section 5. We will give discussion of the language class and the deadlock detection methods in Section 6, and conclude in Section 7.

2 Modeling software process

2.1 Composite modeling approach

Although various software process models have been proposed, real development processes are so complicated. They are composed of various factors such as 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 view points of documentation and formalization, complicated models have disadvantages, i.e., they are difficult to understand, evaluate and validate. Here, we propose a *composite* software process model consisting of several simple models. Each model which is a component of the composite model represents single view of the software process from a certain point. We assume here two component models: activity model and product relation model (Figure 1).

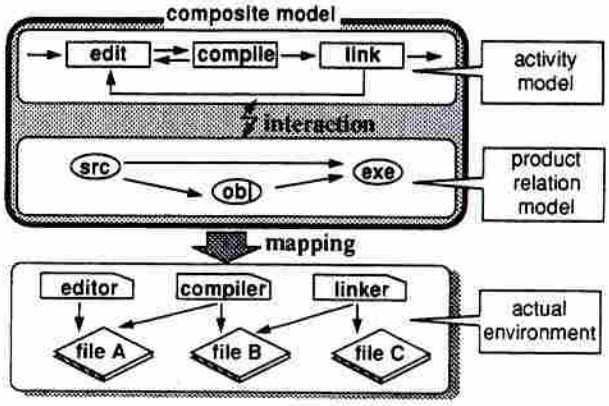


Figure 1: Overview of Composite Software Process Model

In activity model, constraints of the order of activities appearing in the development are defined, and in product relation model, relations among the products appearing during the development are specified. In this paper, we focus on the activity model reflecting the aspects of cooperative development works performed by multiple developers.

2.2 Activity model

The activity model proposed here is based on concurrent process model such as CSP (Communicating Sequential Processes)[8]. Our activity model is composed of several concurrent tasks. There are many

kinds of activities in actual development, and they are performed by several developers concurrently. Each developer communicates to other developers, and coordinates and controls the progress of activities.

If we model this process from a human-centered view point, the model may become complicated, since one developer might be responsible for performing several independent activities which have to be involved in the model. Here, we propose activity-centered model, where related activities are treated as a single task, and each developer performs several tasks. We assume that the developers are *processors* for the tasks.

Task definition

An activity may be considered as a small unit of work such as "*editing a file*", or as a relatively large unit of work such as "*changing the system specification*", each of which is performed by several people. In this paper, we define a **primitive activity** as an atomic unit of work such as tool execution or decision making. We define a **task** as a sequence of primitive activities* which can be represented without concurrent actions. Although it is allowed to use CFG to specify activity sequences of tasks, we simply employ RG (regular grammar) here. A task is defined as a regular expression of primitive activities. In regular expressions, we can also use operators representing iterations and selections which are useful to specify activity sequences. In this paper, we use the operators described in Table 1. Meta symbols ('[ ]') are added to the ordinary regular expression operators. With these operators, we can easily and simply describe the expressions generated by RG.

Table 1: 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 appearance of symbol A
( )	Grouping
|	Selection

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

```
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.

Communication primitives

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

*In the followings, we may refer to primitive activity as **activity** simply.

The followings are features of the data transfer operations:

- Data type of the transfer is string.
- One task may have several input ports for data receive.
- To send a data, a destination task and its input port should be specified.
- For an input port, there are the operations to read data and to check its emptiness.

Specifically, there are three primitive operations shown in Table 2.

A `send` operation asynchronously sends a strings to another task, and then it terminates immediately after that. There is a default port in every task, and if the port name is omitted, this port 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 message in the port. A `peek` operation is a non-blocking one, and it inspects the existence of message in a specified port. If some messages are found, it returns `true`. Otherwise, `false` is returned immediately. The returned values are used in restriction conditions of selective expressions[10], and their values do not appear in the expressions explicitly.

These communication operations can be also used as events for synchronization and control of tasks. For example, `peek` can be used to represent a process executing a job and waiting for a message repeatedly (Figure 2).

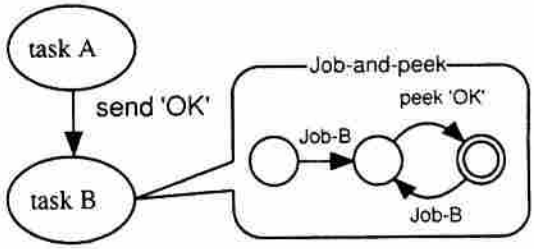


Figure 2: Task Synchronization Example

In Figure 2, task$A$ and task$B$ proceed concurrently, and task$B$ waits for 'OK' message sent from task$A$ on executing its own job. This can be specified as follows by using `peek` operator:

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

Task controlling issues are as follows:

- Initiation (activation) of other task
- Termination of other task

These issues are represented as constructs shown in Table 3, by using message transfer primitives procedurally: `Start` operator is represented as a pair of

Table 2: Message Transfer Primitives

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

Table 3: Task Control Primitives

Operator	Argument	Operation
start	task	send a request of initiation (activation) to other task
wait	task	wait for a termination of other task
exit	task	inform its termination to other task

send in a task and recv at the top of another task. Thus, by default, every task has a recv operation implicitly at the top of the sequence. Wait operator is a special case of 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 regular expressions. Sequences which can not be represented with regular expressions may be decomposed into smaller tasks[†].

For example, assume task$A$ activates task$B$ and task$C$, and task$D$ waits for termination of them (Figure 3).

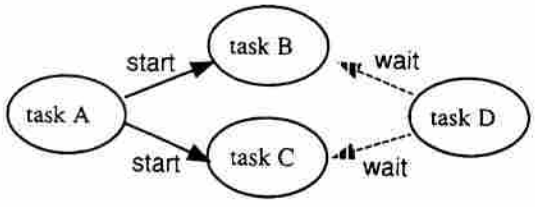


Figure 3: Example of Concurrent Tasks

This example can be specified in activity 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 project manager, and performs specified activities according to their regular expressions (Figure 4).

Task classification

Several tasks may have the same activity sequence but different properties such as input/output module names. For example, 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 the example, we may define task class Task i, which modifies Module$X$. The properties depending on the instances such as module names are defined as instance variables.

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

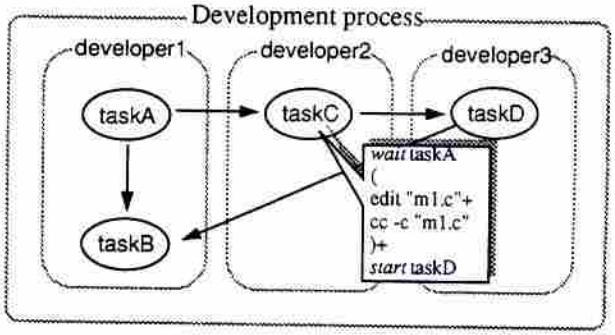


Figure 4: Overview of Activity Model

Other alternative models

There are some alternatives to model concurrent development processes. For example, Kellner uses a state transition model of STATEMATE[17]. Although it provides simulation-based enaction mechanism with concurrent constructs, sophisticated communications would not be established. Also, state chart representation is not enough for the abstraction such as task classification and parameterization. Saeki uses LOTOS (Language of Temporal Ordering Specification) which is based on CCS model[25]. It has so many constructs that the descriptions would be difficult to be described, understood, and executed. To express specific perspectives of a process, this model would be considered more than enough.

Williams's SPM[30] is similar to ours, which uses regular expressions extended with the shuffle operator to express concurrent activities. SPM also includes a message-passing definition. Although the shuffle operator is convenient to describe the concurrency of the processes simply, SPM does not have suffi-

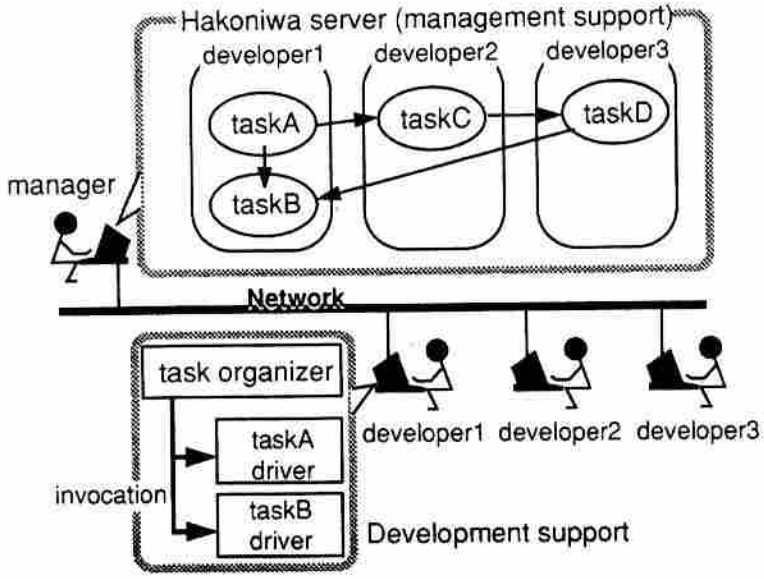


Figure 5: "Hakoniwa" System Architecture

cient functionalities for the communication needed for our purpose; In SPM, concurrency synchronizations are expected to be handled by the shuffle operators. Message-passing is subsidiary and does not provide the mechanism for that[30]. However, the activity sequence such as previous example process waiting on job can not be specified by the shuffle operators.

3 Supporting and managing with activity model

In this section, we discuss ways of supporting and managing the development by using our activity model.

3.1 Development support

First, we consider the ways of supporting each developer.

(1) Navigating activity

When a developer has many complicated works (which can be considered as tasks), it would be difficult for the developer to know the followings:

- How many tasks and which of the tasks are assigned?
- Which activity is next to an activity in each task?
- Are there any tasks suspended and not resumed yet?

Information to answer the questions is essential to support the developer. Giving such information automatically and navigating the developers are very important aspects to be considered.

(2) Communication support

By defining the processes based on the activity model, the followings would be clear:

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

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

3.2 Management support

For the project managers, the following benefits will be expected by the activity model:

- The task definitions are used as a measure of elaboration when assigning works to each developer.
- In the case of distributed development, communications among the development sites could be clarified with task communications in the model, so that communication costs can be assessed and reduced.
- By monitoring each task status, progress and workload are estimated. Moreover, it would be easier to find suspended and not resumed tasks or freezed tasks (in dead lock).
- The activity model is used as a milestone (measure) of project progress.

We have developed a prototype system of supporting and navigating the developers and the manager based on these discussions.

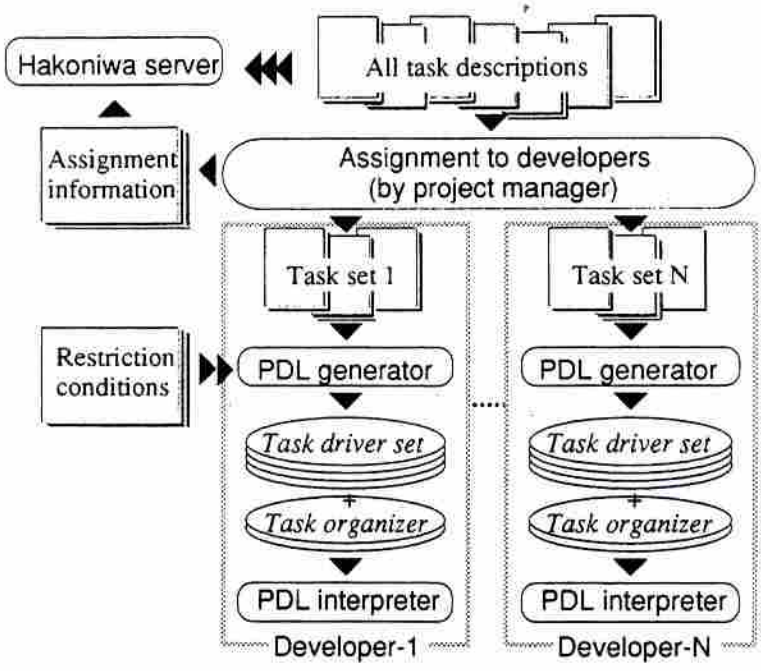


Figure 6: Generation Flow of Hakoniwa Environment

4 Hakoniwa system

4.1 Overview

Based on the activity model, we have developed a prototype of a cooperative development support system"**Hakoniwa**"[‡]. Figure 5 shows the overview of the system architecture.

For an activity model description defining all tasks in the development, there is a task monitor/communication server named **Hakoniwa server**, and there are support/navigation managers for a single developer named **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 as mentioned in Section 2, which is out of the Hakoniwa system.

Major features of the Hakoniwa system are as follows.

(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 primitive one accomplished by a tool invocation, the task driver automatically activates the tool.

[‡]"Hakoniwa" is a Japanese word which means diorama.

(2) Progress monitoring

Each task driver reports to the Hakoniwa server log information of the task progress collected from the menu selection history. The project manager can capture the current status of whole 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 request and task termination notification are automatically executed without any action of the developers.

4.2 Implementation

We assume that the cooperative development environment is under distributed one over several distinct workstations connected through networks. The mechanisms underlying are TCP/IP communication protocol and UNIX operating system. The current implementation works on Sparc station with Sun OS 4.1.1/

A functional language for software process description PDL[12] and its interpreter system have been extended to be used as the task drivers. The PDL interpreter has built-in functions such as menu selection and tool invocation, while we have newly implemented the communication primitives. A task organizer is implemented as an application program of the PDL. Each task organizer activates the task drivers corresponding to the assigned tasks.

We employ the same approach as one proposed in [10] to generate task drivers. Task drivers are gen-

erated from the task descriptions (grammar), adding restriction conditions of the selective expression. Selective 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, '`edit`','`compile`', and '`link`'. The restriction condition of 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 concerning products[10]. These restrictions are supplied to the generator as a set of PDL functions (Figure 6).

Task communication

Communication operations described in Table 2 and Table 3 are implemented as built-in functions of PDL interpreter. They are implemented under a server-client architecture, and all messages are relayed and controlled by the Hakoniwa server.

Task status display

In order that the project manager gets intuitive understanding of the current status and progress of the project, it is insufficient to simply display raw data such as sequence of time stamps for initiation and termination of each activities. This is because it is not easy to grasp the current position in the overall task only with such information. Since an activity sequence of a task is represented by a regular expression in the activity model, the Hakoniwa system displays the activity history and current activity as a tree form of the regular expression (Figure 7).

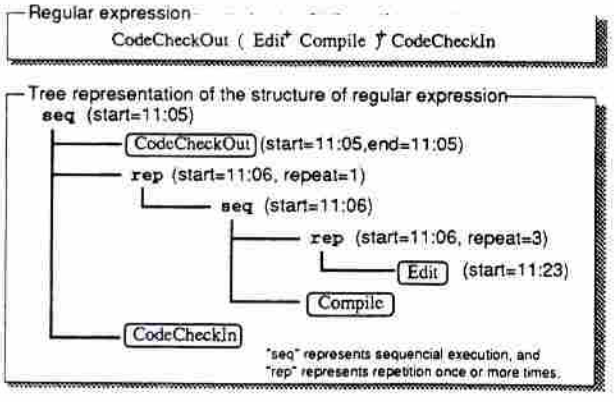


Figure 7: Regular Expression and Tree Display of Task

Figure 7 shows an example of a task defined by a regular expression and its structured status information. In this example, the developer at first retrieves a program code from the repository (represented by `CodeCheckOut`). After repeatedly editing (`Edit`) and compiling (`Compile`), the code is stored into the repository (`CodeCheckIn`). The structure of regular expression is shown with components `seq` and `rep` similar to those in Jackson's tree of JSD method (`seq` means the sequel and `rep` means the repetition one or more times), and the overall shows the hierarchical structure with intermediate nodes for such as sequel or iteration. The Hakoniwa server records and displays the initiation time and termination time for each node, and especially for iteration nodes, it displays also the number of iterations. In this example, `Edit` actually started at 11:23 and it was repeated 3 times. To display this information, the Hakoniwa server has an internal state transition map for each task, and it keeps track of actual state transition which is advanced by the activity sequence.

5 Example of model description and execution

In this section, we show an example of the description of the activity model, and its execution result. As a target process, we use "Software Process Modeling Example Problem" proposed by Kellner et al.[16], which offers the basis of comparison and evaluation of various software process modeling approaches. Some solutions have been already reported[17, 25].

This problem specifies a process of modifying one module in a system, and the process is composed of 8 substeps. Each substep is defined with some pre-conditions, post-conditions or other constraints. In this problem, the effects of the modification are assumed to be limited to the module only, and it does not affect to 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 made the schedule and assigned the tasks. The whole of the process terminates when the new code for the module passes the unit testing. The view of this process by the activity model is shown in Figure 8.

In this example, 8 substeps such as **ModifyDesign** and **ModifyCode** are considered as tasks. There are some constraints on the initiation and termination of the tasks such that "*ModifyCode can not terminate until ReviewDesign terminates*", or "*TestUnit can not start until both ModifyCode and ModifyTestPackage terminate*". These constraints are represented by task communications in the activity model.

Figure 9 shows the example of task definitions. For example, task **ScheduleAndAssignTasks** sends requests to 5 other tasks which are performed in concurrent. **ModifyDesign** first repeats (represented by '+') actual modification work such as invoking an editor (`<modify design>`), and send the 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. Task organizers and task drivers are obtained by translating the instantiated definitions into the PDL programs.

By executing obtained PDL programs, task organizers and task are established and the developers are navigated with the menus shown as in Figure 10. The Hakoniwa server monitors the progress of each task and displays the information such as in Figure 11.

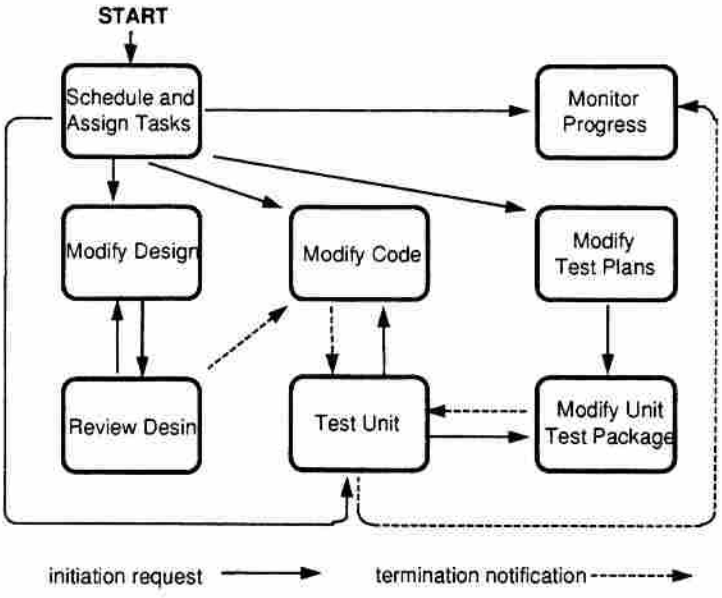


Figure 8: Activity Model for The Example Problem

```
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 9: Activity Model Description of The Example Problem

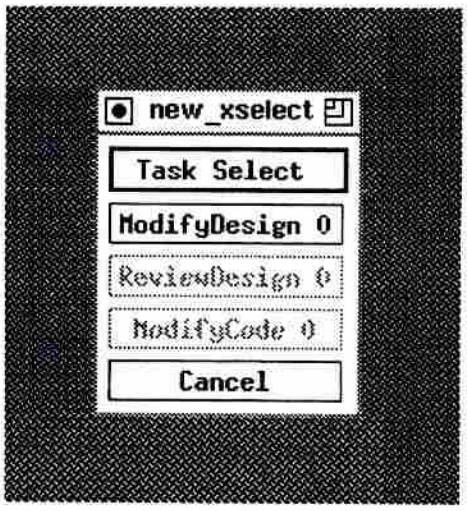


Figure 10: Menu Displayed by the Task Organizer

Figure 11: Output Displayed by the Hakoniwa Server

6 Discussion

6.1 Language class of activity sequence

As discussed in Section 2, the activity sequences of tasks are defined with regular expressions in the activity model. However, there would exist complex processes which can not be specified well by the linear regular expressions. In some cases, such as interleaving sequences, one may split them into parallel subsequences with communication each of which fits to be expressed by a regular expression, as we have shown in this paper. In other cases, such as recursive sequences, one may use more powerful language class such as context free grammars. For instance, the process of an incremental development such as prototype development can be defined simply and naturally with recursions of context free grammars. Another difficulty of regular expression is, as we have shown in [10], that some constraints can not be expressed simply even if they were simply represented with a finite state machine; for example, a simple water-fall activity sequence A-B-C which may contain back-tracking loops (Figure 12(a)) is well represented by deterministic finite automaton as shown in Figure 12(b). However, the representation with a regular expression (not with the regular *grammar*) does not efficiently reflect the nature of this process (Figure 12(c)).

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

6.2 Hakoniwa system as a monitoring environment

As we have discussed, the Hakoniwa system provides various data which help to assess the progress of project, such as:

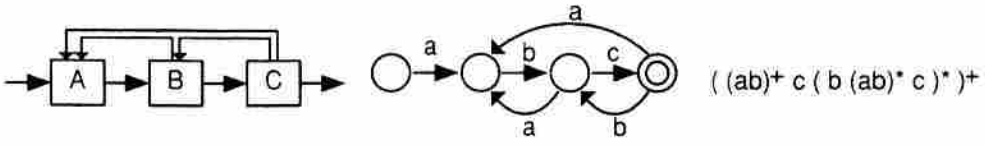
- Number of iterations of each activity
- Time duration of each activity (initiation and termination time)
- Current activities

An experienced manager may easily comprehend the project progress from these data; however, it is not all the cases. Not only to get those data, it is also desirable to have goal values for those data[2]. It is difficult to set such goals and to estimate the progress only from single project data. For example, even if we have the data of the number of the iterations at this moment, we can not predict the total numbers of the iterations at the end. However, if we knew the data of similar projects, we would assess the current status from the data.

In order to perform this kind of statistical prediction effectively, it is essential to store large number of project profiles. The Hakoniwa system collects such data automatically, without paying data collection costs. The collected data are more reliable than the data collected by hand such as reports from the developers. Furthermore, the collected data are directly used for evaluation of current status and assertion for project profiling.

6.3 Dead-lock detection in activity model

The activity model contains dead-lock possibilities as many other concurrent models. In this paper, we



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

Figure 12: Representations for Simple Sequence

define a dead-lock as: "Infinite wait of a message, causing task execution freeze"[§]. According to the shape of dependency graph of the message waiting of tasks, we consider following two kinds of dead-locks:

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

To statically detect the existence of dead-lock in the description, we focus on the types of messages, especially on initiation request and termination notification messages. Concerning with these message types, we have the following sufficient conditions:

- For linear waiting:
  In the directed graph which only shows the relation of the initiation requests, there is a task unreachable from the task initially activated.
- For circular waiting:
  In the directed graph which depict only the relation of termination notification, there is a circle.

On the other hand, in the case of dynamic detection on the execution, linear waiting dead-locks have to be detected by human analysis, while the circular waiting dead-locks can be detected automatically by checking the circularities of the message waiting paths.

6.4 Associating with groupware

Researches on cooperative development model and their systems have been emerged as a basis of CSCW (Computer Supported Cooperative Works)[7]. The model we have proposed here targets the cooperation among the relatively discrete and independent works, while systems so called groupwares in CSCW targets more tight and complex cooperative works which need much wider band width of communication.

For example, we have assumed that one task is performed by one developer in our model; however, tasks such as reviewing may be well performed by several developers, and it is difficult to decompose the reviewing task into sub tasks for each developer. We would think that this kind of tasks is to be associated with groupware systems, by sharing a task and defining the interface for it.

[§]Here we do not think task freezes caused by exclusive data access, since the product model is out of the activity model.

7 Conclusion

We have discussed the advantages of managing the development projects based on the software process models, and we have proposed a cooperative development process model for supporting and monitoring the developers.

We have also implemented a prototype system, Hakoniwa, which controls the tasks of the developers and displays the progress status of each task. It provides the menus and navigates the developers to appropriate next steps. At the same time, the data to grasp the progress status of the project are automatically collected and provided to the project manager. These data are useful for estimation and prediction.

We have developed the prototype system. Collecting more experiences of practical use of the system has been undertaken. The Hakoniwa system will be extended to handle dynamic evolution of the tasks and asynchronous control of the task drivers.

The proposed activity model does not handle the products, and we have proposed a product relation model, which is another important component of the composite software process model[9]. We will also extend our system to have the interface between the activity model and the product relation model.

Acknowledgments

The authors gratefully acknowledge the comments for improvement of Dr. Karen Huff on earlier versions of this paper, as well as other reviewers' ones.

References

[1] Balzer,R.: Tolerating Inconsistency, *Proc. 13th Int. Conf. Software Engineering,* Austin,Texas, pp.158-165 (1991).

[2] Basili,V.R., and Rombach, H.D.: The TAME project: Towards Improvement-oriented Software Environments, *IEEE Trans. Softw. Eng.,* SE-14, 6, pp.758-773 (1988).

[3] Boehm, B.W.: A Spiral Model of Software Development and Enhancement, *Computer* May 1988, pp.61-72 (1988).

[4] Cusumano,M.A.: *Japan's Software Factories,* Oxford University Press, (1991).

[5] Dowson,M.: Software Process Themes and Issues, *Software Process Symposium,* Washington, DC, (1990).

[6] Dowson,M. (Ed.): *Proceedings of the 1st Int. Conf. on the Software Process,* Redondo Beach, CA,(1991).

[7] Ellis,C.A., Gibbs,S.J., and Rein,G.L.: Groupware, Some Issues and Experiences, *Communications of the ACM,* Vol.34 No.1 (1991).

[8] Hoare,C.A.R.: *Communicating Sequential Processes,* Prentice-Hall, (1985).

[9] Iida,H., Nishimura,Y., Inoue,K., and Torii,K.: Generating Software Development Environment from The Descriptions of Product Relations, *Proc. COMPSAC'91,* Tokyo,JAPAN, pp.487-492 (1991).

[10] Iida,H., Ogihara,T., Inoue,K., and Torii,K.: Generating a Menu-oriented Navigation System from Formal Descriptions of Software Development Activity Sequence, *Proc. 1st Int. Conf. Software Process,* Redondo Beach, CA, pp45-57, (1991).

[11] Inoue,K., Ogihara,T., Kikuno,T., and Torii,K.: A Formal Adaptation Method for Process Descriptions, *Proc. 11th Int. Conf. on Software Engineering,* Pittsburgh, PA, pp.145-153(1989).

[12] Inoue,K., Ogihara,T., Iida,H., and Nitta,M.: Functional Language for Enacting Software Process, *Proc. COMPSAC'91,* Tokyo,JAPAN, pp.487-492 (1991).

[13] Kaiser,G.E. and Feiler,P.H.: An Architecture for Intelligent Assistance in Software Development, *Proc. 9th Int. Conf. on Software Engineering,* Monterey, CA, pp.180-188 (1987).

[14] Katayama,T.: A Hierarchical and Functional Software Process Description and Its Enaction, *Proc. 11th Int. Conf. on Software Engineering,* Pittsburgh, PA, pp.343-352 (1989).

[15] Katayama,T. (ed.): *Proc. 6th Int. Software Process Workshop: Support for the Software Process,* Hakodate, Japan, (1990).

[16] Kellner,M. et al.: ISPW-6 Software Process Example, *Proc. 1st Int. Conf. on Software Process,* Redondo Beach, CA, pp.176-186, (1991).

[17] Kellner,M.: Software Process Modeling Support for Management Planning and Control, *Proc. 1st Int. Conf. on Software Process,* Redondo Beach, CA, pp.8-28, (1991).

[18] Kusumoto,S., Matsumoto,K., Kikuno,T., and Torii,K.: On a Measurement Environment for Controlling Software Development Activities, *Trans. IEICE* Vol.E73, No.5 pp.1051-1054, (1991).

[19] Madhavji,N.H.: Environment Evolution: The Prism Model of Changes, *IEEE Trans. Softw. Eng.,* SE-18, 5, pp.380-392 (1992).

[20] Osterweil,L. : Software Processes Are Software Too, *Proc. 9th Int. Conf. on Software Engineering,* Monterey, CA, pp.2-13 (1987).

[21] Penedo,M.H. and Shu,C.: Acquiring Experiences with the Modeling and Implementation of the Project Life-Cycle Process: the PMDB Work, *Software Engineering Journal,* Sep. 1991, pp. 259-274 (1991).

[22] Perry,D.E. and Kaiser,G.E.: Models of Software Development Environments, *IEEE Trans. Softw. Eng.,* SE-17, 3, pp. 283-295 (1991).

[23] Peuschel,B. and Schäfer,W.: Concepts and Implementation of a Rule-based Process Engine, *Proc. 14th Int. Conf. on Software Engineering,* Melbourne, Australia, pp.262-279 (1992).

[24] Riddle,W.E.: Software Designer's Associates: A Preliminary Description *Proc. 20th Annual Hawaii Int. Conf. on System Sciences,* Hawaii, pp.371-381 (1987).

[25] Saeki,M., Kaneko,T., Sakamoto,M. : A Method for Software Process Modeling and Description using LOTOS, *Proc. 1st Int. Conf. on Software Process,* Redondo Beach, CA, pp.90-104 (1991).

[26] Sommerville,I.: *Software Engineering,* 3rd Ed., Addison Wesley, (1989).

[27] Sutton,S.M., Heimbigner,D. and Osterweil,L.: Language Constructs for Managing Change in Process Centered Environments, *Proc. 4th ACM SIGSOFT Symposium on Software Development Environments,* Irvine, CA, pp 206-216 (1990).

[28] Taylor,R.N.,et al.:Foundations for the Arcadia environment architecture, *Proc. ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (SIGSOFT Software Engineering Notes, 13-5), Boston, MA, pp.1-13(1988).

[29] Weber, H.: Towards a Software Factory Reference Model *Tutorial Text in COMPSAC'91,* Tokyo,Japan, (1991).

[30] Williams,L.G. : Software Process Modeling: A Behavioral Approach, *Proc. 10th Int. Conf. on Software Engineering,* Singapore, pp.174-186 (1988).