PAPER

CooPs: A Cooperative Process Planning System to Negotiate Process Change Requests

Kagetomo GENJI[†] ***and*** **Katsuro INOUE**[††]**,** ***Members***

SUMMARY In order to lead an ongoing software project to success, it is important to flexibly control its dynamically-changing software process. However, it is generally impossible not only to exactly pre-define the production process but also to prescribe the process change process (*meta-process*). To solve the problem, we have focused on communication between the project staff through which process change requests presented by individuals can be immediately shared, designed, verified, validated and implemented. This paper proposes a communication model which can represent a wide variety of communication states between the project manager and developers discussing how to implement process change requests. The communication model has been derived by investigating the sort of process change requests and, based on the model, we have implemented a cooperative process planning system (called *CooPs*). *CooPs* is a communication environment designed for software projects and supports information sharing for discussing the process change requests. By using *CooPs*, the software project can flexibly deal with not only expected change requests but also unexpected ones. To evaluate the applicability of the communication model and the capabilities of *CooPs*, we have conducted an experiment which is an application of *CooPs* to the ISPW6 example problem. This paper describes the concepts of *CooPs*, the system implementation, and the experiment.

key words: *communication model, computer-supported cooperative work, process change discussion, process change process, process planning, product access interface, software process*

1. Introduction

In order to ensure software quality and productivity, it has been recommended to formally describe project policies and strategies, and to systematically control a software project with a formalized software process [1], [12], [13]. The process research community has provided a lot of effort in varied aspects of software process technologies. The research of process modeling has listed process elements (software products, development and management activities, etc.) which should be formally described, resources (humans, tools, duration, etc.) which these elements need, and their structure based on which these elements can be reconstructed to represent the software process. The variety of process modeling methods has lead to the development of various process description languages, each of which has a different formality [10]. These process languages enable a software project to visualize its process, and provide a vehicle to analyze the process for risk management, resource management, cost control and so on. The approach has also lead to the creation of numerous process-centered software engineering environments (PSEE) [11]. Most of the PSEEs have a function which interprets software process descriptions written in a formal process language [11]. In addition, the PSEEs provide to ongoing software projects a wide variety of facilities such as process automation, guidance and monitoring which can support the process control.

One of the main issues of PSEE research is how a PSEE can deal with the dynamic nature of changing software processes. Assume that a staff member makes a process change request for some reason. The software project has to pick up the request, modify the whole process description to meet the request, validate them, and implement the software process changes on individual developers by changing the current state of their ongoing activities. The support functions of existing PSEEs can be categorized into two dimensions: the type of the process change request and the supported activity for the process change as shown in Table 1. For example, Suzuki et al. have proposed *meta-operations* adapted to HFSP [2] which can formally describe dynamic process creation and automatically generate process changes such as process re-execution to support the process change design activity for the expected process change [7], [8]. Peuschel et al have proposed a declarative, rule-based process definition method easy to judge whether a process change conflicts with the other activities [3], which can support the process change verification/validation activity. Bandinelli et al has proposed a Petri net-based process language (called *SLANG*) which can describe both *software development processes* and change policies called *software meta-processes* [9], [20], [21]. Based on a meta-process written in *SLANG*, the interpreter (called *SPADE*) supports a sequence of the process change activities from modifying the process model to changing the process state during the process enactment. Thus, these PSEEs have implemented a lot of functions to mitigate the difficulties of dealing with the dynamic nature of changing software processes.

In the meantime, a software project is generally composed of a number of the project staff. The software

Manuscript received July 27, 1998.

[†]The author is with Gakken School Management, Osaka-shi, 530–0001 Japan.

[††]The author is with the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University, Toyonaka-shi, 560–8531 Japan.

Table 1 The sort of process change support.

change activity / change request type	expected	unexpected
change design	- interpret process descriptions where alternative processes have been embedded - generate changes by interpreting how to change process descriptions	- provide a process editor
change verification/validation	—	- detect deadlock/infinite loop by simulating process descriptions - detect conflicts by parsing process descriptions
change implementation	- lock/unlock products being referred by process descriptions - suspend/resume ongoing production activities to deliver changes to individuals	
a sequence of the change activities	- interpret meta-process descriptions which describe the process from the change generation to the implementation	- interpret meta-process descriptions which describe the process from the editor invocation to the change implementation

process of the project is also composed of a number of process fragments performed by individuals. In such a situation, an unexpected change of a process fragment often causes changes in other process fragments. Also, the propagated changes would cause changes successively. If change propagation is not allowed in the project, the verification/validation functions shown in the second row of Table 1 can work effectively and easily indicate the invalidity of the original change. If it is allowed however, the project must first decide whether or not to employ the propagated changes in addition to the original. Next, for the decision, the project manager and developers in charge of the impacted processes need to exchange the change requests, the current state of ongoing activities, the loss caused by the changes, and so on. That is to say, it is expected that the PSEE has a function which supports communications and communication protocols for staff to discuss successively-arising process change requests. Unfortunately, most of the existing PSEEs lack communication support functions. Even the meta-process interpreter shown in the forth row of Table 1 would not work because the existing meta-process languages have not been developed to trace a huge number of states in the process change discussion where the number of the participants dynamically increases or decreases.

To solve the problem, we have focused on communications between a project manager and a developer who discuss the validity of a process change request and how to change the software process so as to meet the request. Then, we have developed a communication model between the two parties which is described with a state-transition diagram. A huge number of states in the process change discussion can be represented by applying the communication model into every two parties in the project staff. The communication model has been derived by investigating the types of process change requests and, based on the model, we have created a cooperative process planning system (*CooPs*). *CooPs* is a communication environment for project and supports information sharing for process change discussions. By sharing information using *CooPs*, the software project can flexibly deal with not only expected change requests but also unexpected ones. In addition, *CooPs* monitors the process change discussion and notifies the project staff of the current state to promote discussion. To evaluate the applicability of the communication model and the capabilities of *CooPs*, we have conducted an experiment which is an application of *CooPs* to the ISPW6 example problem. This paper describes the concepts of *CooPs*, the system implementation, and the experiment.

Section 2 first shows a variety of process change requests categorized by focusing on their origins and change motivations, and then illustrates the communication model that represents communication protocols for the project in order to share the process change requests and discuss their implementations. Section 3 outlines a cooperative process planning system *CooPs* which has been implemented based on this communication model. The system facilities to support the process change discussion are illustrated in Sect. 4. The system implementation is introduced with some user interfaces of *CooPs* in Sect. 5. Section 6 demonstrates the system capabilities with the extended ISPW6 example problem. Section 7 addresses the advantages of *CooPs* in comparison to other systems.

2. Communication Model for Process Change Discussion

This section first shows several types of process change requests categorized by focusing on their origins and motivations. Next, the communication model is illustrated to represent the exchange between a project manager and developer of these process change requests. It will also be shown that the communication model can represent a wide variety of communication states with more than three project staff discussing how to implement both the original process change request and the propagated ones.

2.1 Process Change Requests

A lot of process change requests can be divided into two types by focusing on their origins. One is the top-down change request which is presented from the project manager to a developer, the other is the bottom-up change request presented from a developer to the manager.

Top-down change request can be divided into three types by focusing on their motivations as follows.

- **Process model application:** The process model is a set of high-abstracted, software development activities which is prescribed in a software-related organization. The process model is sometimes revised by the organization with such an event as *management review of a quality system* [13]. The model change typically falls on ongoing software projects and causes a large impact on the project policies and strategies. The project manager has the right to decide whether or not to apply the process model. To distribute the process model to the project, the manager needs to instantiate the model and divide the process into process fragments for individual developers.
- **Process detail correction:** The developer has to redefine the delegated process in detail enough for the process execution. The detail (called the process detail) is composed of executable tasks and their sequence, and it corresponds to a strategy of the developer to attain his/her local goal. On the other hand, the manager has to attain a global goal of the project by controlling a number of the process details. If the process detail has some inconsistencies with the global goal, the manager has to request the developer to modify it.
- **Process detail propagation:** The process delegated to a developer is generally linked to other processes by the process inputs/outputs. Because of these relationships, the process detail designed by the developer defines not only itself but also other products refereed to by subsequent processes; that is, the design is propagated to the other processes. To ensure the consistency of the whole software process of the project, the manager has to always catch up the dynamically-designed process detail and deliver the propagations to corresponding developers.

Bottom-up change request can be divided into two types by focusing on their motivations as follows.

- **Process detail design:** As mentioned above, the process model divided into process fragments is distributed to individual developers. The delegated process is redefined in detail for the developer to perform it effectively. Even if the process detail is out of some constraints or rules defined in the original, the developer might judge that the process detail is more effective for him/her. Then, the developer has to request the manager to accept the process detail and the changes of the original.
- **Process detail change:** Often the developer finds a fault of the process detail during the process execution. The developer has to correct the process detail, however, the process detail has been accepted by the project and coordinated with the other processes via the manager. The change of the process detail necessarily has a lot of impact on the ongoing project. The developer has to request the manager to accept the change and to coordinate the change with the other processes.

Process change requests arising in the ongoing project can be categorized into any of the five kinds of process change requests mentioned above. The five kinds of change requests can represent communication between the project manager and the developer who join in the process change discussion.

2.2 Communication Model

We have modeled communication protocols of exchanging the five kinds of process requests shown in Sect. 2.1. The communication model has been developed by extending *Conversation Theory* proposed by Winograd et al [18]. The essential idea of the theory is that speaking is doing, and that the application of the theory can trace/support a variety of communicative activities between two parties A and B. For example, if A makes a *Request* to B, then either a reply of *Reject*, *Accept* or *Counter Offer* is made by B. A lot of application systems which support cooperative work, so-called CSCW (Computer-Supported Cooperative Work), have employed this theory and proposed enhancements. We have also extended the theory to model the process change discussion. The discussions related to the enhancement is stated in Sect. 7.

The communication model is represented in the state transition diagram. Figure 1 shows the model between two parties of the manager M and of the developer D. Every state shows a global state between M and D.

Deliverable is the initial state where no communication has been started between the manager M and the developer D.

Delivered is a state where the developer D has received the event *Request* from the manager M un-

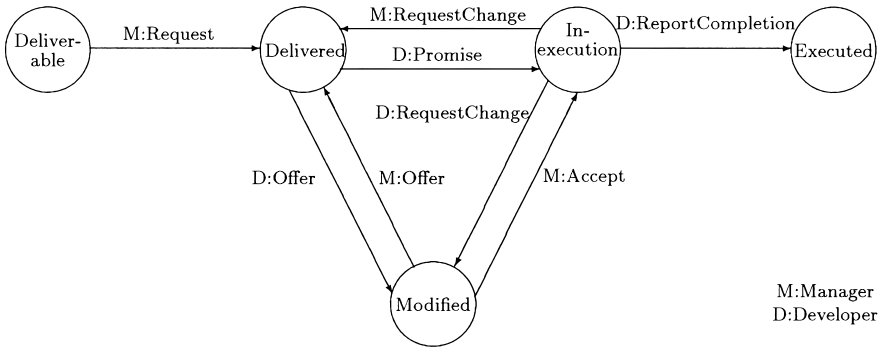


Fig. 1 A communication model for process change process.

der Deliverable. The **process model application** of the top-down change request from *M* to *D* is the event. Then, *D* can start to design the process detail to attain the delegated local goal. In turn, *M* is waiting for the response from *D*. *M* may cancel his/her original request at any state. In the diagram, the event *Cancel* from *M* is omitted.

Modified is a state where the manager *M* has received an event *Offer* from the developer *D* under Delivered. The **process detail design** of the bottom-up change request from *D* is identified with the event. Then, *M* can start to evaluate whether or not the process detail is suitable to the global goal of the project. After that, *M* may transmit an event *Offer* if the process detail needs to be modified. The event *Offer* corresponds to the **process detail correction** of the top-down change request.

In-execution is a state where the developer *D* has received an event *Accept* from the manager *M*, or where *D* has transmitted an event *Promise* under Delivered. The former event implies that the process detail has been approved by *M*. The latter is a declaration that *D* employs the process detail suggested by *M*. Then, *D* can start to execute the process based on the process detail.

Executed is a terminal state where the process change discussion has ended with an event *ReportCompletion* from the developer *D*. After that, no change request related to the corresponding process arises.

The above state transitions are equal to a part of *Conversation Theory*. The communication model includes two extended events in addition to the above.

M: RequestChange is an event from the manager *M* under In-execution where the developer *D* is executing the process. The event corresponds to the **process model application** of the top-down change request or the **process detail propagation**.

D: RequestChange is an event from the developer *D* under In-execution. The event is the **process detail change** of the bottom-up change request.

The state transition diagram shows the process change discussion where only two parties of the manager and the developer participate. By applying the communication model to all parties in the project staff, multiple process change discussions can be represented with tuples of states. For example, assume that the current state between the project manager *M* and the developer *D1* is **In-execution** and that the current state between *M* and the other *D2* is **Delivered**. That is, *D1* is executing *D1*'s process and *D2* is designing *D2*'s process detail. When *D2* transmits *D2: Offer* of the **process detail** to *M*, the state between *M* and *D2* transits to **Modified**. The state transition can be represented as follows: from (*In-execution, Delivered*) to (*In-execution, Modified*). Moreover, assume that *M* transmits to *D1 M: RequestChange* of the **process detail propagation** caused by the **process detail** from *D2*. The state (*In-execution, Modified*) transits to (*Delivered, Modified*). Thus, the multiple application of the communication model can represent a huge number of states in the process change discussion where more than three staff participate. Also, the state transition represented with the communication model can notify every participant of the next action in the process change discussion.

3. Cooperative Process Planning System Overview

Based on the communication model shown in Sect. 2, a cooperative process planning system (called *CooPs*) has been developed. To support both the production process and the process change discussion, several facilities to be provided at each state in the model have been designed and implemented. Figure 2 shows the overview of *CooPs* which is composed of a process planning subsystem for the manager and process execution subsystems for the developers. Referring to the abovementioned model and the figure, the outline is illustrated.

Under Deliverable: The process planning system provides a graphical process editor for the manager to design the software process specific to the project

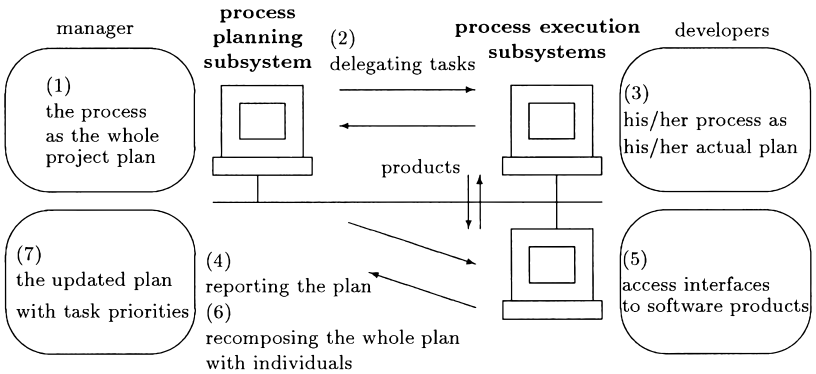


Fig. 2 An overview of a cooperative process planning system.

(function (1) shown in Fig. 2). To support the **process model application** by transmitting the event *M: Request*, the system decomposes the whole process of the project into the process fragments to be performed by the developers (function (2)). The decomposed processes are automatically delivered by *CooPs*. The delivery is identified as *M: Request* in the communication model.

Under Delivered: The process execution system notifies the developer of the arrival of the delegated process. Then, the system provides a graphical process editor to support the design of the process detail (function (3)). The process detail is returned to the manager as the event *D: Offer* by *CooPs* (function (4)). During the **process detail design**, *CooPs* provides to the developer the access interface to the input products which the process needs (function (5)).

Under Modified: The process planning system recomposes the whole software process of the project with the process detail returned from the developer (function (6)). The maintenance of the software process makes the manager validate the process detail against the global goal. Moreover, the maintenance also indicates the impact to the other processes which are caused by the returned process detail. This facility supports the **process detail correction** and the **process detail propagation**.

If the propagations have problems, the manager can correct the whole software process including the returned process detail by using function (1), again. If modified by the manager *M*, the process is decomposed and delivered again as the event *M: Offer* by using function (2). If accepted by *M*, the event *M: Accept* is transmitted to the developer *D* by *CooPs*. *CooPs* also simulates the execution of the returned process detail and suggests the priority of the tasks in the process detail which can minimize the whole loss of the project (function (7)).

Under In-execution: The process execution system provides the same supports as under *Delivered* (function (5)). In this case, the access interface to software products is provided to support the process execution. When the developer *D* modifies the process detail for some reason, the modified process detail is transmitted as *D: RequestChange* by *CooPs*. On the other hand, when the manager *M* modifies the whole process of the project, the same supports as under *Deliverable* are provided by the process planning system and the modification is delivered as the event *M: RequestChange* by *CooPs*.

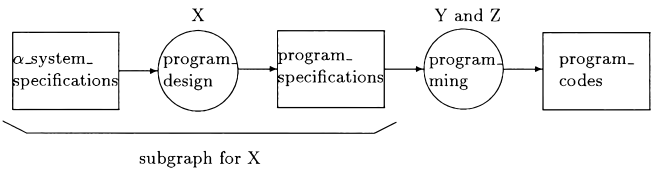
Thus, the fundamental support facility of *CooPs* is to exchange the dynamically-changing software process among the manager and the developers, and is to monitor the process change discussion. In addition, *CooPs* has the graphical process editor, the process decomposer and recomposer, the process simulator and the product access interface. These facilities are needed for the project staff to handle the software process at each state in the communication model and the details are described in Sect. 4. The user interfaces of the facilities are shown in Sect. 5 and the section also illustrates how *CooPs* monitors the process-centered software development including the process change discussion.

4. System Facilities of *CooPs*

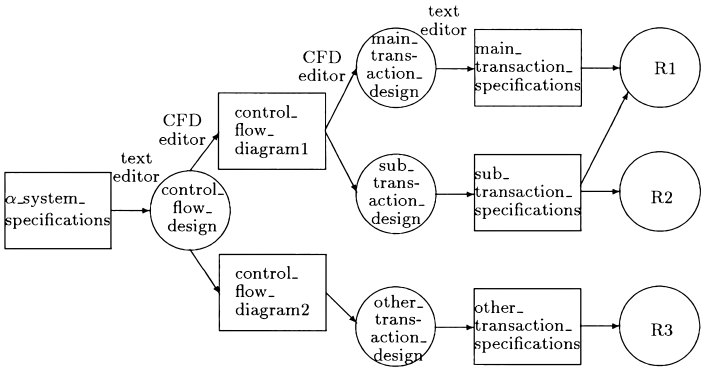
As mentioned above, *CooPs* has been developed on the basis of the communication model to support the process-centered software development including the process change discussion. This section details the system facilities provided at each state on the model. The facilities are (1) the process representation which the graphical process editor employs [14], [15], (2) the process decomposer from the whole process of the project to individual process fragments, (3) the process recomposer to maintain the dynamically-changing process of the project, (4) the access interface to products, and (5) the process simulator [16], [17].

4.1 Process Representation

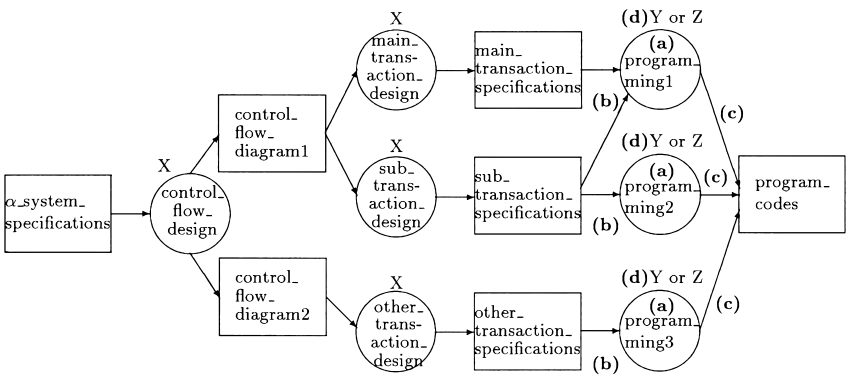
CooPs employs a directed graph to graphically represent the software process. For convenience' sake, the graph which shows the whole process of the project is



(1) *process graph*



(2) *partial process graph* **for program_design**



(3) *process graph* **recomposed with the process detail**

Fig. 3 Examples of software process descriptions.

called the *Process Graph* (*PG*) and the graph which shows either the process fragment or the process detail is called the *Partial Process Graph* (*PPG*). That is, the PG is handled by the project manager and the PPGs are handled by the developers.

Figure 3 (1) shows a simple example of the PG. The PG implies that the developer X is in charge of the task *program_design* and that the developers Y and Z are assigned to the task *programming*. The task sequence is represented as *program_design* followed by *programming*. The local goal of *program_design* is to create *program_specifications* from *α_system_specifications*. The subsequent task *programming* inputs *program_specifications* and outputs *program_codes*.

The PG is defined as a six-tuple (A, T, I, O, S, G) where

A **:** is a set of software products,
e.g., $\{\alpha\_system\_specifications, program\_specifications, program\_codes\}$,

T **:** is a set of software development tasks,
e.g., $\{program\_design, programming\}$,

I **:** is a set of relationships between tasks and input products; $I : T \rightarrow A^n$, provided that n is the number of elements in A,
e.g., $I(program\_design) = \{\alpha\_system\_specifications\}$,
$I(\emptyset) = \{program\_codes\}$,

O **:** is a set of relationships between tasks and their output products; $O : T \rightarrow A^n$, provided that n is the number of elements in A,
e.g., $O(\emptyset) = \{\alpha\_system\_specifications\}$,
$O(program\_design) = \{program\_specifications\}$,

S **:** is a set of project staff, e.g., $\{X, Y, Z\}$, and

G **:** is a set of relationships between tasks and staff; $G : T \rightarrow S^m$, provided that m is the number of

elements in S,
e.g., $G(program\_design) = \{X\}$,
$G(programming) = \{Y, Z\}$.

Figure 3 (2) shows an example of the PPG. The PPG shows the process detail of *program_design* designed in detail by the developer X. The PPG implies that X performs *control_flow_design* followed by *main_*, *sub_* and *other_transaction_design*. The goal *program_specifications* is redefined by X with *main_*, *sub_* and *other_transaction specifications*. *R1*, *R2* and *R3* are hypothesis nodes to show the product dependencies among the final output products. We call such special nodes *product dependency nodes*. The product dependency nodes are used for the process recomposer to generate candidates of the **process detail propagations** as discussed in Sect. 4.3. For example, *R1* shows that *main_transaction_specifications* refer to *sub_transaction_specifications*. And *R3* shows that *other_transaction_specifications* are independent of the others. In the meantime, each arc which is an element of the set I or O, can have an additional information which defines a software development tool, e.g., *text editor*. Each element of the set A can also have a product address, and each element of the set T can have the time required for the process execution.

The graphical process editor based on the process representation method is provided to support the process design for the **top-down change** and **bottom-up change**.

4.2 Process Decomposition

The PG is composed of multiple PPGs each of which is delegated to its corresponding developer. For the project manager to support the distribution of the PPGs, *CooPs* performs the process decomposition. To be short, the process decomposer divides the PG into multiple subgraphs each of which is composed of the process nodes accompanied with the same staff. The detail of the process decomposition procedure is shown in Appendix A. The subgraph indicated in Fig. 3 (1) is an example of the PPG derived by the process decomposer.

The PPG is $(A_x, T_x, I_x, O_x, S_x, G_x)$ where

$A_x = \{\alpha\_system\_specifications,$
$\qquad program\_specifications\}$,
$T_x = \{program\_design\}$,
$I_x(program\_design) = \{\alpha\_system\_specifications\}$,
$O_x(program\_design) = \{program\_specifications\}$,
$S_x = \{X\}$, and
$G_x(program\_design) = \{X\}$.

The process decomposer works to support the process distribution for the **process model application**, **process detail correction** and **process detail propagation** as mentioned in Sect. 2.

4.3 Process Recomposition

The PPG derived from the PG is delivered to the developer and is redefined by the developer. Then, *CooPs* recomposes the PG including the detailed PPG to present the manager the current PG. Basically, the process recomposer replaces the original PPG with the detailed PPG. By placing the detailed PPG in the PG, the process recomposer allows the manager to determine whether or not the PPG meets the global goal of the project. Figure 3 (3) shows an example of the PG updated by the process recomposer. The subgraph indicated in Fig. 3 (1) which corresponds to the PG for X is replaced with the PPG shown in Fig. 3 (2). At the same time, the subsequent task *programming* is replaced with the product dependency nodes *R1*, *R2* and *R3*. The replacements implies that the process detail of *programming* needs to be composed of three tasks, *programming1*, *programming2* and *programming3*. That is to say, the process recomposer generates the candidates of the **process detail propagations** to help the project manager verify/validate the impact caused by the updated PPG.

Thus, *CooPs* has the process recomposer notify the project manager of the necessity of the **process detail correction** and the possibility of the **process detail propagation**. The detail procedure of the process recomposition is shown in Appendix B.

4.4 Process Simulation

The process simulator parses the PG when the PG is updated, and detects the inconsistency of the PG such as deadlock and infinite loop. Furthermore, the process simulator suggests the task sequence of the detail PPG which minimizes the loss of the project. This section focuses on the latter.

The developer can achieve his/her own goal by interpreting the PPG accepted by the project manager. The PPG involves some temporal relations among the tasks to be executed by the developer. However, it never indicates a unique sequence of all of the tasks. For example, the PPG of Fig. 3 (2) shows that *control_flow_design* should be followed by any of *main_*, *sub_*, and *other_transaction_design*. But the priority of *main_*, *sub_*, and *other_transaction_design* is never clarified by the PPG.

However, the sequence of the task execution determines the start-date of other developers' process execution. In the example shown in Fig. 3 (3), either the developer Y or Z can start *programming3* when *other_transaction_design* is finished. If *other_transaction_design* is firstly executed by X, the waiting time of Y and Z is only the needed time of the design task. If not, the loss of their waiting time increases.

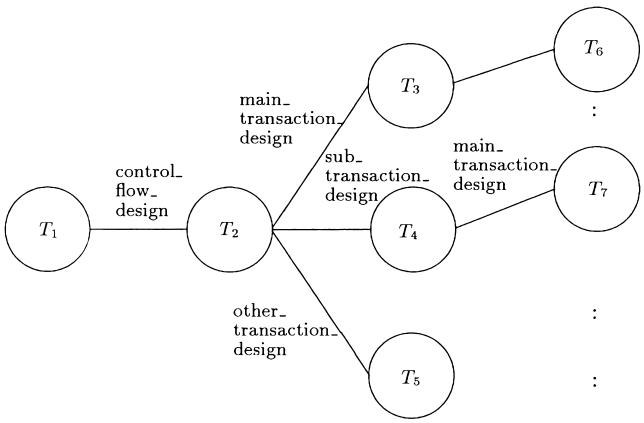


Fig. 4 An example of the search tree.

Therefore, the process simulator generates the task sequence which minimizes the loss of manpower. To be short, the task sequence derivation creates the search tree which consists of all possible task sequences, and then performs the breadth-first search to find the path where the total waiting cost is the smallest. Figure 4 shows an example of the search tree for X's task sequence. The search tree is created from the PG shown in Fig. 3 (3). T_1 is the state where any task in T is not yet finished. It is the initial state of X's task execution. T_2 is the state where X finished the task *control_flow_design*. T_3, T_4 and T_5 are the alternative states subsequent to T_2. The path from T_1 to T_4 represents the task sequence of *control_flow_design* followed by *sub_transaction_design*.

The total waiting cost can be calculated by multiplying the needed time of X's task execution by the number of waiting developers. Assume that the needed time of X's tasks shown in Fig. 3 (3) is as follows: *other_transaction_design* takes three hours and the others respectively take two hours. The task sequence in the path from T_1 to T_4 wastes eight man-hours. And then, either Y or Z can start *programming2*. Accordingly, the path from T_4 to T_7 wastes two man-hours. In this example, the path from T_1 to T_7 is the most effective task sequence. The task sequence derivation is detailed in Appendix C.

The process simulator helps the manager not only design the **process detail correction**, but also improve the process detail.

4.5 Product Access Interface

CooPs provides the product access interface to the PPG editor; that is to say, the project developer can browse/create software products by manipulating the PPG. According to the state of the communication model, the access interface plays different roles. Referring to Fig. 1 and Fig. 3, the roles are explained in this section.

During the design of the process detail under *Delivered* as shown in Fig. 1, the access interface works as a browser of the input products which the delegated process receives from other developers' processes. By browsing the inputs, it becomes easier for the developer to understand the delegated goal and to plan his/her strategy of how to accomplish the process. In the example of Fig. 3 (1), X can open α*_system_specifications* with the read-only access right when the PPG of *program_design* is delivered by the manager. Then, the access interface supports the **process detail design**.

During the process execution under *In-execution*, the access interface corresponds to a command interpreter which invokes software tools, e.g. vi, emacs, make, etc. When the task is indicated by the developer, the adjacent inputs/outputs in the PPG are allocated by *CooPs* and are opened within the tools invoked by *CooPs*. For example, α*_system_specifications* in Fig. 3 (2) is opened within *texteditor* whose access right is read-only. On the other hand, both *control_flow_diagrams* are allocated by *CFDeditor* and are opened with read/write access. However, the developer loses write access when he/she starts modifying the PPG. This is the product access control during the design of the **process detail change**. The product access control is also performed when either of the state transitions from *In-execution* to *Delivered/Modified* occurs, because the state transitions are triggered by the **process detail correction**, **process detail propagation** or **process detail change**.

Thus, the main function of the access interface is to control the access right to products. By the access control, *CooPs* can suspend/resume the process execution of the developer to change both the PPG and the current state.

5. System Implementation

The system *CooPs* presented here has been implemented. The subsystems of the process planning system and the process execution systems are distributed on a computer network composed of UNIX Worksta-

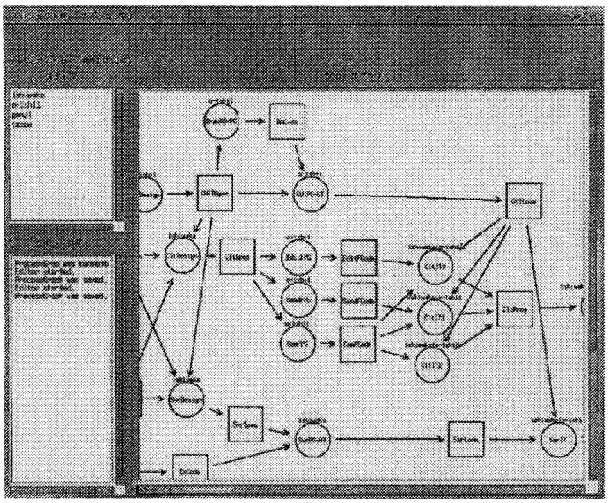


Fig. 5 A user interface of the process decomposer.

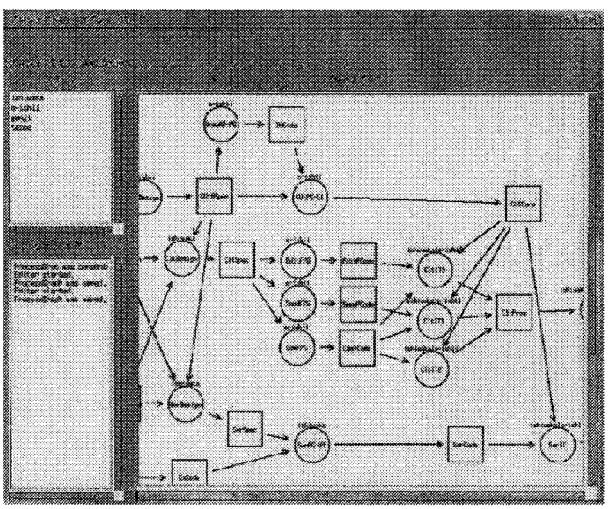


Fig. 6 A user interface of the process monitor.

tions. The system functions have been programmed with C and C++ languages. The user interfaces are built with OSF Motif on X-Window system. This section first shows two examples of the user interfaces, and then explains how the user interfaces can represent the current state of the software process including the process change discussion.

Figure 5 is the user interface of the process decomposer which is presented to the project manager. The right-side window shows a part of the PG of a sample project which consists of three project staff. Existing products are *GUIDSpec*, *CltSpec*, *DataASDesign*, *ProcADesign*, *ExSpec* and the other input of the task *ExPG-UT*. The process decomposer has already presented the PPGs which can be executed by referring to the products. The names of the PPGs are *DrawAD-PG*, *SVRDesign*, *ExPG-UT* and *CltPG-UT* as listed in the left-side windows of Fig. 5. By indicating one of them, the PPG is emphasized within the PG by the process decomposer. The subgraph from *CltSpec* to *CltCode*, i.e., the PPG *CltPG-UT* is emphasized in the figure. The manager then can distribute each of the PPGs to the corresponding developer.

Figure 6 shows the user interface for the manager to monitor the whole process of the project. The left-side window lists the project staff as *ishiwaka*, *o-ishii*, *genji* and *tazoe*. The PG shown in the right-side window involves the PPG redefined in detail by *o-ishii*, whose original is the PPG *CltPG-UT* emphasized in Fig. 5. That is to say, the detailed PPG *CltPG-UT* has been returned to the manager and accepted by the manager, The tasks *CltIT0*, *CltIT1* and *CltIT2* which have been redefined from *CltIT*, are the candidates of the *process detail propagations* generated by the process recomposer.

The possible state of the PPG is *Deliverable*, *Deliv-*

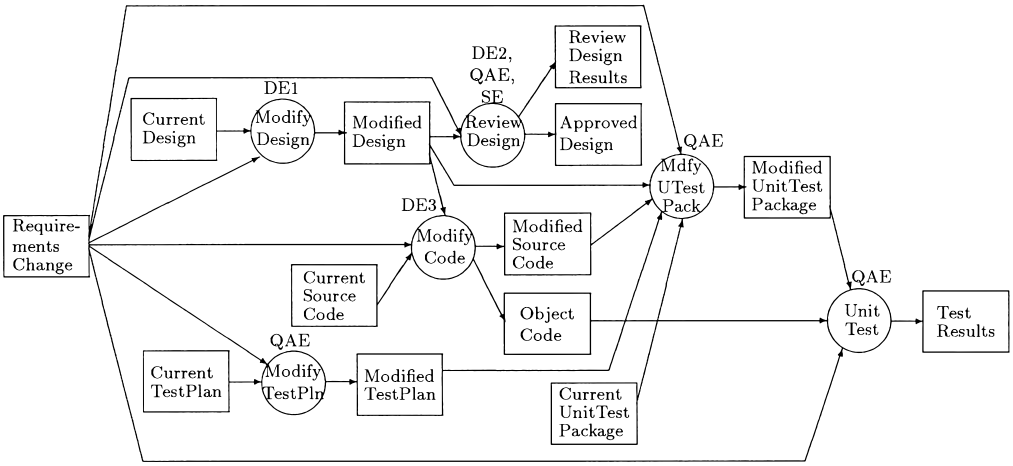


Fig. 7 A process graph of the whole process.

ered, *Modified*, *In-execution* or *Executed* shown in the communication model. They respectively correspond to *deliverable*, *delivered*, *modified*, *in-execution* and *executed*. The five kinds of states are represented in the PG by using five different colors. When the state of the PPG is *in-execution* however, the task being executed is drawn with another color. In addition, the user interface of the process decomposer lists up the PPGs whose state is *deliverable*. The process recomposer also shows the list of the PPGs whose state is *modified*. Thus, the current state of the software process including the process change discussion can be represented by tuples of states mentioned above.

Assume that the PPG *CltPG-UT* has been delivered as shown in Fig. 5. The state is described as follows: {*Deliverable*, *Deliverable*, *Deliverable*, *Delivered*}, provided that the four states are respectively of the four PPGs listed in Fig. 5 and that the PPGs *executed* are omitted. In addition, assume that the detailed PPG *CltPG-UT* has been accepted as shown in Fig. 6 and that the PPG *ExPG-UT* has been delivered. The state is described as follows: {*Deliverable*, *Deliverable*, *Delivered*, $In-execution$}. The state transitions from the former to the latter can be traced by applying the communication model in parallel into parties of the project manager and individual developers. All states in the transitions can be presented with the six colors to the project manager as mentioned above.

6. System Capabilities

Using an example of the software process which describes the ISPW6 example problem [19], this section addresses the system capabilities of *CooPs*. Especially, it is shown that some situations where *CooPs* can work effectively actually exist in the example problem. The example problem which we employ is extended in respect of the design engineer. It is that the project has multiple design engineers *DE1*, *DE2* and *DE3* who are respectively in charge of *ModifyDesign*, *ReviewDesign* and *ModifyCode*. Figure 7 shows the PG which the project manager has planned to attain the goal given by the Configuration Control Board (*CCB*).

6.1 Top-Down Change Request for Re-execution

The original example problem has employed the iteration of the software process to compare the representabilities of existing process languages with each other. The PG employs the directed graph, but it is never allowed to have any arc which represents the iterative process. The concept of *CooPs* is to control the project by sharing the software process detail which is gradually-clarifying and dynamically-changing. Then, the process detail to be executed for the first time is quite different from the one to be re-executed. For example, consider that *ModifyDesign* needs to be re-executed if *ReviewDesign* indicates some faults of *ModifiedDesign* as shown in Fig. 7. Figure 8 shows the detailed PPG of *ModifyDesign* which describes the outputs *Function_xDesign*, *Function_yDesign* and *DataDesign*. The execution of *ModifyDesign* is accomplished by performing the four tasks shown in the figure. But the re-execution may be accomplished by performing only *DetailFunction_x* if *ReviewDesignResults* indicates that some faults exist only in *Function_xDesign*. If some faults are detected in both *DataDesign* and *Function_yDesign*, *DesignData* followed by *DetailFunction_y* needs to be re-executed at least. Besides, *BreakDown* should also be re-executed if the faults in *Function_yDesign* are caused by *Function_yOutline*. Though the modifications of *Function_yOutline* must not conflict with *Function_xDesign*.

Although *CooPs* does not support the definition of the re-execution process, it helps the project dynamically design the detailed process which can precisely represent the re-execution process. In the above

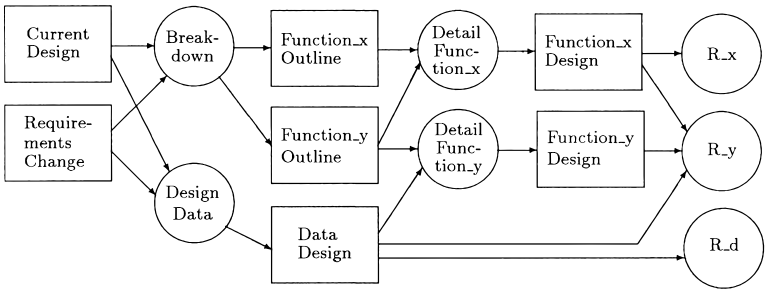


Fig. 8 A partial process graph of ModifyDesign.

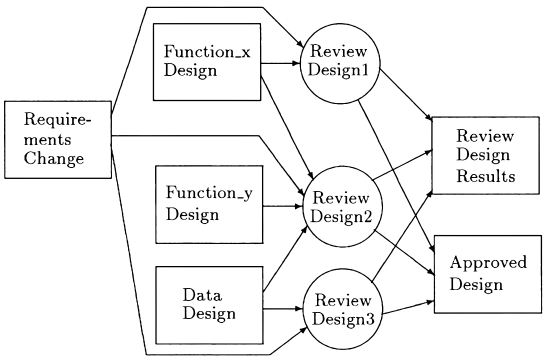


Fig. 9 An example of ReviewDesign modified by process detail propagations.

example, the re-execution request is triggered by the completion of *ReviewDesign*. The trigger which corresponds to *ReportCompletion* shown in Fig. 1 is delivered to the project manager by *CooPs*. By notification, the manager can start to judge whether or not the re-execution of *ModifyDesign* should be done, and then he/she can also start to modify the PG with the graphical process editor. *CooPs* then delivers the PPG for the re-execution after performing the process decomposition of the updated PG. The PPG is identified with *M: Request* in the communication model, and *CooPs* recognizes that the process change discussion has been restarted between the manager and *DE1*. The process change discussion can be supported based on the communication model as mentioned in Sect. 3. That is to say, the process re-execution can be processed by *CooPs* as the **top-down change** from the project manager.

6.2 Process Detail Design and Propagation

The PG shown in Fig. 7 can be decomposed by *CooPs* into the PPGs of *ModifyDesign* and *ModifyTestPln*. There already exist the products of *RequirementsChange*, *CurrentDesign*, *CurrentSourceCode* and *CurrentTestPlan*. The developers *DE1* and *QAE* can start executing both of the PPGs. However, *ReviewDesign* and *ModifyCode* can not be executed since they need the output of *ModifyDesign*. *DE3* and the others in charge of the tasks have to await for the completion of *ModifyDesign*. The waiting cost and duration is a loss to the project. By recomposing the PG with the process detail of *ModifyDesign*, *CooPs* can assist the manager in planning how to minimize the loss.

Assume that the PPG *ModifyDesign* shown in Fig. 8 has been returned to the manager. By the process recomposer, the PPG can be embedded into the PG and can be presented to the manager. Figure 9 is a part of the PG where *ReviewDesign* has been redefined by *CooPs*. The figure shows that *ReviewDesign* can be composed of the three tasks of *ReviewDesign1*, *ReviewDesign2* and *ReviewDesign3*. They can be clarified with the process recomposition triggered by the **process detail design** of *ModifyDesign*, and they correspond to the **process detail propagations**. The manager can learn that *ReviewDesign1* can start if *Function_xDesign* exists and that *ReviewDesign3* can start if *DataDesign* exists. The process simulator is then useful for the manager to determine the task sequence of *ModifyDesign* which minimize the total waiting cost.

6.3 State Representation for Process Change Discussion

This section illustrates the support of communications between the project manager and the developers who discuss the process change request. Table 2 shows an example of the state transitions during changing the PG shown in Fig. 7.

The PG shows that there exist the products of *CurrentDesign*, *RequirementsChange* and *CurrentTestPlan*,

Table 2 An example of state transitions in process change discussion.

	PPG (task name shown in Fig. 7)					
No.	ModifyDesign	ReviewDesign	ModifyCode	ModifyTestPln	ModifyUtestPack	UnitTest
1	delivered	–	modified	delivered	–	–
2	modified	–	in-execution	delivered	–	–
3	in-execution	–	delivered	modified	–	–
4	in-execution	deliverable	modified	in-execution	–	–

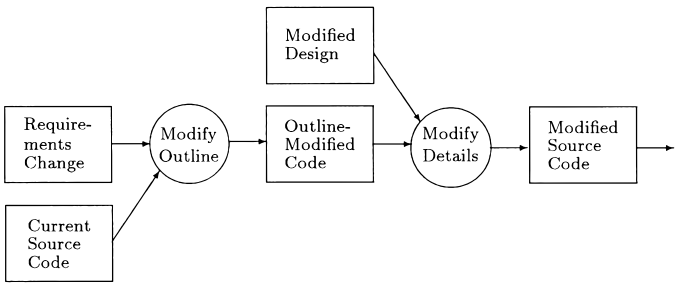


Fig. 10 A partial process graph of ModifyCode.

and that the tasks of *ModifyDesign* and *ModifyTestPln* can be started. *ModifyCode* can not be started now because it needs the output of *ModifyDesign*. Assume that the manager has planned to perform *ModifyCode* in parallel with *ModifyDesign*. Accordingly, the PPGs of *ModifyDesign* and *ModifyTestPln* have been automatically derived by the process decomposer, and the PPG of *ModifyCode* has been manually derived by the manager. As mentioned in Sect. 5, the global state of the project is represented with tuples of states such as *Deliverable*, *Delivered*, ..., *Executed* in the communication model. Each state of the derived PPGs is *Deliverable*. The states can be represented by the colored PG to the manager. The following illustration assumes that the PPGs have been delivered to the corresponding developers and that the current global state is {*Delivered*, *Delivered*, *Delivered*}.

1. Row No. 1 shows the state where the detailed PPG of *ModifyCode* has been returned to the manager. The process recomposer of *CooPs* can notify the manager of the arrival and show the PPG within the PG to allow the manager to validate the PPG. The PPG is shown in Fig. 10. The figure shows that *DE3* first modifies the outline of *CurrentSourceCode* based on *RequirementsChange* and that he/she next modifies the details by referring to *ModifiedDesign* which is the output of *ModifyDesign*. Since the PPG can be started in parallel with *ModifyDesign* as the manager desires, the manager can deliver the acceptance of the PPG through *CooPs*.
2. Row No. 2 shows that the PPG of *ModifyDesign* shown in Fig. 8 has been returned and that *ModifyCode* has already been started based on Fig. 10. The process recomposer can update the PG with the PPG of *ModifyDesign* and support the manager to validate the PPG. Also, the process recomposer can redefine *ModifyDetails* of Fig. 10 which inputs the products detailed in the process detail of *ModifyDesign*. Then, the process decomposer can inform the manager that *ModifyDetails* should be delivered again to *DE3*. This is an example of that the process change discussion between the manager and *DE1* activates another discussion between the manager and *DE3*.
3. Row No. 3 shows that *ModifyDesign* has been started with the acceptance of the manager and that *ModifyDetails* which is a sub-task of *ModifyCode* has been delivered again. These states can be represented by the process monitor with the colored PG. Also, the process recomposer can represent another state where *ModifyTestPln* has been returned to get the acceptance of the manager.
4. Row No. 4 shows that *ReviewDesign* has been able to be started with the progress of *ModifyDesign* execution. The progress of the process execution can be also represented by the process monitor. Through the monitoring, the manager can learn whether to deliver *ReviewDesign* or not.

By means of applying the communication model, *CooPs* can identify the current state of the project with tuples of states between all parties who discuss the dynamically changing software process. At each state, the process decomposer, recomposer and monitor can present the current state to the manager.

7. Related Works

This section emphasizes characteristics of *CooPs* with comparison to related works.

The CSCW research community has provided a lot of effort concerning how to model cooperative work in organizations; one of which is based on the theory of

communicative activity like *Conversation Theory* [18]. Medina-Mora et al have proposed the importance of focusing on atomic loops (called action workflow loop) in which a performer completes an action to satisfy a request of a customer [29]. Bowers et al. have indicated that *Conversation Theory* misses the locality of conversations and that local and global structure coexist in conversations, and they have proposed a structure definition language (SDL) [30]. Michelis et al. have addressed that it is often difficult for two communicating parties with different experiences to share the same context and viewpoint in conversations and that it is impossible to assume that the parties interpret a speech act theory in the same way, and they have proposed the Milan Conversation Model where commitment negotiations are embedded within conversations [31]. The above debate discusses the communicative activity theory in general conversations where two parties communicate in a natural language. *CooPs* employs the communication model to structure the communication activity specific to the process sharing between the project manager and the developer in the hierarchically-organized software project. The communication model involves the communication tasks from process delegation to process completion, and also includes process detail/modification exchange as mentioned in Sect. 2. In the specific situation, the communication tasks described in the communication model can be regarded as the atomic action workflow loop through which the distributed software project can share the software process. The communications within the project can be accomplished by exchanging the formally-described process, i.e., PG/PPG. Besides, the process detail/modification exchange in the communication model can correspond to the negotiation tasks for the manager and the developer to accommodate/accept the dynamically-changing PG/PPG. We believe that it is easy for the project staff to share the communication context in the communication model and that the above debate supports our approach to modeling the atomic action workflow loop in the hierarchically-organized software project.

PROCESS WEAVER proposed by Fernström et al. [6] is a PSEE which is provided to distributed, heterogeneous environments in an organized software project. In addition, Avrilionis et al have proposed a view mechanism called OPSIS [27] which can extract role-based processes (called view) from the entire process model of the project. The main purpose of the extraction is the modularization of complex process models and the support of the process reuse based on this view. The extraction mechanism of OPSIS is similar to the process decomposer and recomposer of *CooPs* presented in this paper. The purpose of our approach is, however, to support the ongoing project to negotiate the dynamically-changing, gradually-clarifying software process by means of sharing the process detail between the manager and the developer.

The Articulator [4] provides to the developers the process-driven Softman environments [23] and to the manager the process monitoring mechanism which can represent every process state with a set of product states such as *Active*, *Done*, etc. Hakoniwa [5] implemented by Iida et al can monitor the state of the process execution by recording the history of the task check-in/check-out date. To represent the state of the entire software process of the project, *CooPs* can deal with the state of the process negotiation in addition to the state of the process execution. The process negotiation is called the process change discussion in this paper and the communication model is employed to represent the state and the state transition. The purpose of the state representation is to monitor/guide the process change discussion in the hierarchically-organized software project.

By dividing the process change into the planned change and the on-the-fly deviation, Bandinelli et al [28] discussed the characteristics of the PSEE facilities as follows. They said that there exist two approaches to the PSEE to deal with the planned change. One is that the PSEE contains a process change policy embedded into its specific functions to support the process change. The other is that the PSEE interprets a reflective process language which can describe the meta-process in addition to the production process in the same formalism. The latter approach includes EPOS [22] with a centralized object-oriented database EPOSDB where the software process can be modeled, and SPADE-1 [32] of an implementation of the SPADE [9], [20], [21] concepts which also supports changes in a management policy for a cooperative process. The PG/PPG which *CooPs* employs is not to describe the meta-process but to describe only the production process. We have considered that it is difficult or impossible to describe all process change propagations caused by a process change even if the original can be planned in advance and pre-defined. Instead, we have provided *CooPs* with the process editor which can be invoked by any project staff at any time, and the process decomposer and recomposer which can assist the project staff in exchanging unexpected change requests including arising change propagations. However, *CooPs* is not a PSEE into which only a specific process change policy is embedded. The communication model based on which *CooPs* works contains several change policies since a change policy corresponds to an instance of the communication model which is a communication history between the project staff accompanied with the invocation of *CooPs* functions.

Bandinelli et al [28] defined the on-the-fly deviation as an event which causes the inconsistency between the PSEE and the process model. They addressed that the PSEE must have the following three mechanisms: a *bypass mechanism* to escape from the PSEE to perform

an operation not described in the process model, a *triggering mechanism* which catches the event, and a support mechanism for a *reconciling process* which analyzes the event and reconcile it with the PSEE. The purpose of the above mechanisms is to tolerate the temporary deviation from the pre-defined process. However, the purpose of *CooPs* is to make the project staff report deviations in advance and to make the project manager to accommodate them within the whole process of the project before the execution of the deviations. As mentioned above, the process editor is provided to any project staff at any time. The process decomposer and recomposer support the project to share process changes and change propagations. The product access control is dynamically performed by *CooPs* by tracking the communication history accompanied with the tool invocation. These functions have been provided to reconcile the on-the-fly deviation in advance with *CooPs*.

8. Conclusions

This paper has described a PSEE called the cooperative process planning system, *CooPs* which can cope with the dynamic nature of dynamically-changing software process. *CooPs* distinguishes the software production process and the process change discussion process. The former corresponds to the global goal of the project which consists of the local goals to be delegated to the developers, and they are formally described in the process graphs to reduce the ambiguity within them. The latter is modeled by a set of the negotiation tasks from the process delegation to the process completion involving the process change/modification exchange. We have addressed that PSEEs should support the process discussion process to deal with both expected changes of the production process and unexpected ones.

To implement the system, we have developed a communication model between the two parties of the project manager and the developer, which is described with a state-transition diagram. Even if the process changes propagate in a number of the developers and they have to participate in the process change discussion, the overall discussion states can be tracked by applying the communication model into all parties of the project manager and individual developers. The communication model has been derived by investigating the types of process change requests. First, a wide variety of process changes has been categorized into five change types according to their origins and motivations. Next, the communication by exchanging the process changes has been structured within the extended communicative activity theory based on *Conversation Theory*.

Based on the communication model, several facilities of *CooPs* have been designed. The main facilities are the process decomposer and recomposer. The former can divide the whole software process into the process fragments based on the roles to delegate them from the manager to the developers. The latter can reconstruct the whole software process with the process detail which has been redesigned by the developer. These facilities based on the model assists the project in controlling the dynamically-changing, gradually-clarifying software process. Namely, *CooPs* is a communication environment which supports information sharing for the process change discussion. By sharing such information supported by *CooPs*, the software project can flexibly deal with not only expected change requests but also unexpected ones. In addition, *CooPs* monitors the process change discussion and notifies the project staff of the current state to promote discussion.

The study of *CooPs* which assists multiple project staff in cooperatively performing both the process design and the process execution, is also included in the research area of concurrent engineering [24] and groupware [25], [26]. The remaining issue is that such support facilities for synchronous communications as above are designed and embedded into *CooPs*.

Acknowledgement

We would like to thank Prof. Koji Torii of Nara Institute of Science and Technology for his helpful suggestions. We are also grateful to Prof. Takeshi Ogihara of Kobe University, Mr. Michitoshi Ishiwaka of Sakura KCS and Prof. Hajimu Iida of Nara Institute of Science and Technology for their discussions. This work was encouraged and supported by Mr. Masatoshi Morita, director of Sakura KCS. We would also like to thank him. Finally, we would like to thank Mr. Kyle Barrow of X-9 and Mr. Brian Dean von Ahsen of GSM for proofreading this paper.

References

[1] L. Osterweil, "Software processes are software too," Proc. 9th Int. Conf. Software Engineering, 1987.
[2] T. Katayama, "A hierarchical and functional software process description and its enaction," Proc. 11th Int. Conf. Software Engineering, 1989.
[3] B. Peuschel and W. Schäfer, "Concepts and implementation of a rule-based process engine," Proc. 14th Int. Conf. Software Engineering, May 1992.
[4] P. Mi and W. Scacchi, "Process integration in CASE environments," IEEE Software Magazine, vol.9, no.2, March 1992.
[5] H. Iida, K. Mimura, K. Inoue, and K. Torii, "Hakoniwa: Monitor and navigation system for cooperative development based on activity sequence model," Proc. 2nd Int. Conf. Software Process, Feb. 1993.
[6] C. Fernström, "PROCESS WEAVER: Adding process support to UNIX," Proc. 2nd Int. Conf. Software Process, Feb. 1993.
[7] M. Suzuki and T. Katayama, "Meta-operations in the process model HFSP for dynamics and flexibility of software processes," Proc. 1st Int. Conf. Software Process, 1991.
[8] M. Suzuki, A. Iwai, and T. Katayama, "A formal model of re-execution in software process," Proc. 2nd Int. Conf. Software Process, Feb. 1993.

[9] S.C. Bandinelli, A. Fuggetta, and C. Ghezzi, "Software process model evolution in the SPADE environment," IEEE Trans. Software Engineering, vol.19, no.12, Dec. 1993.
[10] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti, "A survey and assessment of software process representation formalisms," Int. Journal Software Engineering and Knowledge Engineering, vol.3, no.3, 1993.
[11] M. Penedo and W. Riddle, "Process-sensitive SEE architecture (PSEEA) workshop summary," ACM Software Engineering Notes, vol.18, July 1993.
[12] M.C. Paulk, C.V. Weber, S.M. Garcia, M. Chrissis, and M. Bush, "Key practices of the capability maturity model, Version 1.1," Software Engineering Inst. Tech. Rep. CMU/SEI-93-TR-25, Feb. 1993.
[13] ISO 9000-3 1991 Quality Management and Quality Assurance Standards– Part 3: Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software, Int. Org. Standardization, 1991.
[14] K. Genji, M. Ishiwaka, T. Ogihara, and K. Inoue, "Project and process management by cooperative software process description," Proc. 8th Int. Software Process Workshop, March 1993.
[15] K. Genji, M. Ishiwaka, T. Ogihara, and K. Inoue, "Process-centered project management system by stepwise particularizing software process," Proc. 17th Int. Computer Software & Applications Conf., Nov. 1993.
[16] K. Genji, M. Ishiwaka, T. Ogihara, and K. Torii, "How to implant software process description processes into actual project management activities," Proc. Joint Conf. Software Engineering, Japan, Nov. 1993.
[17] M. Ishiwaka, K. Genji, T. Ogihara, and K. Inoue, "Project support system using stepwise-particularized software process descriptions," Proc. 3rd Int. Conf. Young Computer Scientists, China, June 1993.
[18] T. Winograd and F. Flores, Understanding computers and cognition, Addison-Weisley Publishing Company, Inc., 1986.
[19] M. Kellner, "Software process modeling example problem," Proc. 6th Int. Software Process Workshop, 1990.
[20] S. Bandinelli, A. Fuggetta, and S. Grigolli, "Process modeling in-the-large with SLANG," Proc. 2nd Int. Conf. Software Process, Feb. 1993.
[21] S. Bandinelli and A. Fuggetta, "Computational reflection in software process modeling," Proc. 15th Int. Conf. Software Engineering, May 1993.
[22] M.L. Jaccheri and R. Conradi, "Techniques for process model evolution in EPOS," IEEE Trans. Software Engineering, vol.19, no.12, Dec. 1993.
[23] S.C. Choi and W. Scacchi, "Softman: An environment for forward and reverse CASE," Information and Software Technology, Nov. 1991.
[24] P. Dewan and J. Riedl, "Toward computer-supported concurrent software engineering," IEEE Computer Magazine, vol.26, no.1, Jan. 1993.
[25] C.A. Ellis, "Groupware — Some issues and experiences," Communications of the ACM, vol.34, no.1, 1991.
[26] L. Brothers, V. Sembugamoorthy, and M. Muller, "ICICLE: Groupware for code inspection," Proc. Int. Conf. Computer-Supported Cooperative Work, Oct. 1990.
[27] D. Avrilionis, P.Y. Cunin, and C. Fernström, "OPSIS: A view mechanism for software processes which supports their evolution and reuse," Proc. 18th Int. Conf. Software Engineering, March 1996.
[28] S. Bandinelli, E.D. Nitto, and A. Fuggetta, "Policies and mechanisms to support process evolution in PSEEs," Proc. 3rd Int. Conf. Software Process, 1994
[29] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores, "The action workflow approach to workflow management technology," Proc. Int. Conf. Computer-Supported Cooperative Work, Nov. 1992.
[30] J. Bowers and J. Churcher, "Local and global structuring of computer mediated communication: Developing linguistic perspectives on CSCW in COSMOS," Proc. Int. Conf. Computer-Supported Cooperative Work, Sept. 1988.
[31] G.D. Michelis and M.A. Grasso, "Situating conversations within the language/action perspective: the Milan Conversation Model," Proc. Int. Conf. Computer-Supported Cooperative Work, 1994.
[32] S. Bandinelli, E.D. Nitto, and A. Fuggetta, "Supporting cooperation in the SPADE-1 environment," IEEE Trans. Software Engineering, vol.22, no.12, Dec. 1996.

Appendix A: Decomposition Procedure

Referring to Fig. 3 (1), the process decomposition procedure is detailed in the following illustration. In the illustration, the PG is defined as (A, T, I, O, S, G). The PPGs for the practitioners s_1, s_2, ..., s_n are defined as $(A_{s_n}, T_{s_n}, I_{s_n}, O_{s_n}, S_{s_n}, G_{s_n})$.

1) set 1 to n,
2) pick up an element s_n from the set S;

if fail **then** go to end
else create the set S_{s_n} with s_n,

3) retrieve every element t_i from the set T where $G(t_i) \ni s_n$ and create the set T_{s_n} with the elements,
4) retrieve every element a_j from the set A where $a_j \in I(t_i) \cup O(t_i)$ and $t_i \in T_{s_n}$ and create the set A_{s_n} with the elements,
5) generate the set $G_{s_n}(t_i)$ where $t_i \in T_{s_n}$ and $G_{s_n}(t_i) = \{s_n\}$,
6) generate the set $I_{s_n}(t_i)$ where $t_i \in T_{s_n}$ and $I_{s_n}(t_i) = I(t_i)$,
7) generate the set $O_{s_n}(t_i)$ where $t_i \in T_{s_n}$ and $O_{s_n}(t_i) = O(t_i)$,
8) add 1 to n, and
9) repeat the routine from 2) to 9).

Appendix B: Recomposition Procedure

Referring to Fig. 3, the following illustration details the process recomposition procedure. We define three graphs as follows:

(A, T, I, O, S, G) **:** the original PG as shown in Fig. 3 (1),
$(A_x, T_x, I_x, O_x, S_x, G_x)$ **:** the original PPG which is the subgraph indicated in Fig. 3 (1), and
$(A'_x, T'_x, I'_x, O'_x, S'_x, G'_x)$ **:** the detailed PPG as shown in Fig. 3 (2).

In addition, we define six sets RA, FT'_x, FA'_x, FA_x, FT and RT. The set RA needs to be given by the developer and the others can be automatically derived by *CooPs*.

$RA : A_x \rightarrow A'^n_x$ **:** is a set of relationships among the original products in A_x and the redefined products in A'_x, provided that n is the number of elements in A'_x,
e.g., $RA(program\_specifications)$
$= \{main\_transaction\_specifications,$
$sub\_transaction\_specifications,$
$other\_transaction\_specifications\}$,
$RA(\alpha\_system\_specifications)$
$= \{\alpha\_system\_specifications\}$.

$FT'_x = \{t_i \mid t_i \in T'_x, O'_x(t_i) = \emptyset\}$ **:** is a set of the product dependency nodes on the detailed PPG,
e.g., $\{R1, R2, R3\}$.

$FA'_x = \{a_j \mid a_j \in I'_x(t_i), t_i \in FT'_x\}$ **:** is a set of the final outputs on the detailed PPG,
e.g., $\{a_j \mid a_j \in RA(program\_specifications)\}$.

$FA_x = \{a_j \mid a_j \in A_x, RA(a_j) \subseteq FA'_x\}$ **:** is a set of the final outputs on the original PPG,
e.g., $\{program\_specification\}$.

$FT = \{t_i \mid t_i \in T \cap \overline{T_x}, I(t_i) \cap FA_x \neq \emptyset\}$ **:** is a set of tasks on the original PG which refer to one or more final outputs in FA_x, e.g., $\{programming\}$.

$RT : FT \rightarrow FT'^n_x$ **:** is a set of relationships between tasks in FT and the product dependency nodes in FT'_x, provided that n is the number of elements of FT'_x;
$RT(t_i) = \{t_j \mid t_i \in FT, t_j \in FT'_x,$
$I'_x(t_j) \subseteq \bigcup RA(a_k), a_k \in I(t_i) \cap FA_x\}$,
e.g., $RT(programming) = \{R1, R2, R3\}$.

The following procedure generates the current PG (CA, CT, CI, CO, CS, CG) as shown in Fig. 3 (3):

1) create the sets CA, CT, CI, CO, CS and CG where
$CA = (A \cup A'_x) \cap \overline{A_x}$,
$CT = (T \cup T'_x) \cap (\overline{T_x \cup FT \cup FT'_x})$,
$CI = (I \cup I'_x) \cap \overline{I_x}$,
$CO = (O \cup O'_x) \cap \overline{O_x}$,
$CS = S$, and
$CG = (G \cup G'_x) \cap \overline{G_x}$; that is, the original PPG is replaced with the detailed PPG except the elements concerning the process detail propagation,
2) remove from CI every relationship $(t_i, CI(t_i))$ where $t_i \in FT'_x$,
3) remove from CO every relationship $(t_i, CO(t_i))$ where $t_i \in FT$,
4) remove from CG every relationship $(t_i, CG(t_i))$ where $t_i \in FT$,
5) set 1 to n,
6) pick up an element t_n from the set FT; **if** fail **then** goes to end,
7) add into CT every element t_{ni} which is a copy of $t_i \in RT(t_n)$, provided that the task name and additional information are replaced with of t_n; that is, the tasks broken down by the process detail are generated as shown in Fig. 3 (3) (a),
e.g., *programming2*,
8) add into CT every relationship $(t_{ni}, CI(t_{ni}))$ where $t_{ni} \in CT, CI(t_{ni}) = I'_x(t_i)$ and $t_i \in RT(t_n)$; that is, the relationships between the broken-down tasks and the inputs are defined as shown in Fig. 3 (3) (b),
e.g., $CI(programming2)$
$= \{sub\_transaction\_specifications\}$,
9) create every relationship $(t_{ni}, CO(t_{ni}))$ where $t_{ni} \in CT$ and $CO(t_{ni}) = O(t_n)$; that is, the relationships between the broken-down tasks and the outputs are defined as shown in Fig. 3 (3) (c),
e.g., $CO(programming2) = \{program\_codes\}$,
10) create every relationship $(t_{ni}, CG(t_{ni}))$ where $t_{ni} \in CT$ and $CG(t_{ni}) = G(t_n)$; that is, the relationships between the broken-down tasks and the developers are defined as shown in Fig. 3 (3) (d),
e.g., $CG(programming2) = \{Y, Z\}$,
11) add 1 to n, and
12) repeat the routine from 6) to 12).

Appendix C: Derivation Procedure

Referring to Fig. 3, the task sequence derivation procedure is detailed in the following illustration. In the illustration, the PG is defined as (A, T, I, O, S, G) and the search tree is defined as a three-tuple (V, E, R) where

V**:** is a set of states each of which is represented by a subset of T,
e.g.. $T_1 = \emptyset \in V$,
$T_3 = \{control\_flow\_design,$
$main\_transaction\_design\} \in V$,

E **:** is a subset of T,

R **:** is a set of relationships between tasks and states; $R : E, V \rightarrow V$,
e.g., $R(main\_transaction\_design, T_2) = T_3$.

Another graph (AV, AE, AR) is also defined same as the above to save the result of the most effective task sequence. The set S_w is defined as a subset of S and is composed of the waiting developers, i.e., $\{Y, Z\}$. The procedure of the task sequence derivation is as follows:

1) retrieve $t_i = t_1, t_2, \ldots, t_k \in T \cap \overline{T_1}$, provided that $G(t_i) \ni X, I(t_i) \subseteq \bigcup O(t_j), t_j = t_1, t_2, \ldots, t_k \in T_1$; that is, all candidates which X can firstly execute are picked up, e.g., *control_flow_design*,

if succeed **then**

- **a)** set 2 to m,
- **b)** repeat the following routine from c) to f) for every t_i,
- **c)** add $T_m = \{t_i\} \cup T_1$ into V,
  e.g., $T_2 = \{control\_flow\_design\}$,
- **d)** add t_i into E,
- **e)** generate $R(t_i, T_1) = T_m$,
  e.g., $R(control\_flow\_design, T_1) = T_2$, and
- **f)** add 1 to m,

2) set 1 to n,
3) add 1 to n,
4) pick up $T_n \in V$,

if fail **then** go to end **else**

if $S_w \subseteq \bigcup G(t_i)$, provided that $t_i = t_1, t_2, \ldots, t_k \in T_n \cap \overline{T_1}$; that is, every waiting developer can start the process execution,
then if the waiting cost in the path from T_1 to T_n is equal to in a path saved in (AV, AE, AR)

then add the path from T_1 to T_n into (AV, AE, AR)
else if smaller

then initialize (AV, AE, AR) and save the path from T_1 to T_n in (AV, AE, AR); that is, multiple task sequences saved in (AV, AE, AR) can make the waiting developers active at the smallest waiting cost,

else if T_n is not equal to any element $T_k \in V$, provided that $n \neq k$,

then perform the subroutine 6)
else if the waiting cost in the path from T_1 to T_n is the smallest in the paths from T_1 to T_k where $T_k \in V$ and $T_n = T_k$, provided that $n \neq k$,

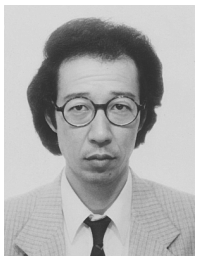
then perform the subroutine 6); that is, if there exist multiple task sequences which reach the same state, one or more sequences which need the smallest waiting cost are selected,

5) repeat the above routine from 3) to 5).
6) is a subroutine which retrieves $t_i = t_1, t_2, \ldots, t_k \in T \cap \overline{T_n}$, provided that $G(t_i) \ni X$, $I(t_i) \subseteq \bigcup O(t_j)$, $t_j = t_1, t_2, \ldots, t_k \in T_n$; that is, all candidates which X can execute at the state T_n are picked up, e.g., *main_*, *sub_* and *other_transaction_design* at T_2,

if succeed **then**

a) repeat the following routine from b) to e) for every t_i,
b) add $T_m = T_n \cup \{t_i\}$ into V, e.g., $T_3 = \{control\_flow\_design, main\_transaction\_design\}$,
c) add t_i into E
d) generate $R(t_i, T_n) = T_m$, e.g., $R(main\_transaction\_design, T_2) = T_3$, and
e) adds 1 to m;

that is, one or more reachable states of the developer X from the current state T_n are saved in the search tree, e.g., T_3, T_4 and T_5 triggered by the three kinds of transaction design processes as shown in Fig. 4.



Kagetomo Genji joined Sakura KCS at Kobe in 1985. From 1989 to 1991, he worked for ATR Communication Systems Research Laboratories. Since 1995, he has been working as a research engineer for Gakken School Management. His interest is in networking technologies and formal specification methods for communication and collaboration.

Katsuro Inoue received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was assistant Prof. of University of Hawaii at Manoa in 1984-1986. He was research associate of Osaka University in 1984-1989, and assistant professor in 1989–1995, and professor from 1995. His interests are various topics on software engineering such as software process modeling, program analysis, and software development environment.