PAPER *Special Section on Concurrent Systems Technology*

Experimental Evaluation of Two-Phase Project Control for Software Development Process

Osamu MIZUNO[†], *Student Member*, **Shinji KUSUMOTO**[†], **Tohru KIKUNO**[†], *Members*, **Yasunari TAKAGI**[††], *and* **Keishi SAKAMOTO**[††], *Nonmembers*

SUMMARY

In this paper, we consider a simple development process consisting of design and debug phases, which is derived from actual concurrent development process for embedded software at a certain company. Then we propose two-phase project control that examines the initial development plan at the end of design phase, updates it to the current status of the development process and executes the debug phase under the new plan.

In order to show the usefulness, we define three imaginary projects based on actually executed projects in a certain company: the project that executes debug phase under initial plan, the project that applies the proposed approach, and the project that follows a uniform plan.

Moreover, to execute these projects, we use the project simulator, which has already been developed based on GSPN model. Judging from the number of residual faults in all products, we found that project B is the best among them.

key words: software development, project management, software fault, Petri-net model, software test and debug

1. Introduction

Generally, the actual software development process is a concurrent process in the sense that many activities are executed in parallel by team members[1][2]. For example, the module design activities for subsystems are concurrently executed, and the coding and the coding review are concurrently executed. The former is for the efficiency, but the latter may be for the delay in the schedule.

Therefore, it is not easy to manage the progress of development for such a concurrent process[7]. Especially, using the development plan initially constructed based on insufficient data on the target project, it is very difficult to control the process of development. Furthermore, it is also difficult to estimate the development cost. Many failed examples are shown in so-called death march projects[14].

Especially, for the test phase, the following problems or difficulties are pointed out with respect to controlling the progress of test phase[6][11].

- to determine the amount of efforts for the test phase
- to estimate the quality of delivered code

These problems are tightly coupled with each other. If the efforts are too small, the quality will become very poor. Even if the efforts are too large, the quality may not be improved so much.

Based on these observations, we propose a new method for controlling the progress of software development. The key idea is to update or modify the initial plan at an intermediate stage of the development, and to apply the updated plan to the succeeding stage. Then we propose to control the progress of design and coding phases using the initial plan, and then control the progress of test and debug phases using the updated plan. We call this method two-phase project control.

The updating of the plan should reflect the results of development using the initial plan. For instance, the number of faults introduced into the product and the number of residual faults in design and coding phases are the data to be considered.

In this paper, we confirm the usefulness of the two-phase project control using the project simulator developed based on the Generalized Stochastic Petri-net(GSPN) model[10]. Additionally, in the evaluation we apply the real data, collected from the actual software development projects in a certain company, to the simulator.

This paper is organized as follows: Section 2 shows the actual software development process in the certain company, and then proposes a simple process model. Section 3 proposes the two-phase project control method, and Section 4 explains only the outline of the project simulator. Section 5 shows the case study and the simulation results. Finally, Section 6 summarizes the main results and the future research work.

2. Software Process Model

2.1 Actual Software Process

An example of the actual software process is shown in Fig. 1. This process is currently used for a certain company, which develops the computer control systems with embedded software. The main products include ATMs(Automated Teller Machine), POS(Point Of Sales) terminals and ticket vending machines.

The following concurrent executions are observed for this actual process in Fig. 1.

[†]The authors are with the Graduate School of Engineering Science, Osaka University.

[††]The authors are with the OMRON Corporation.

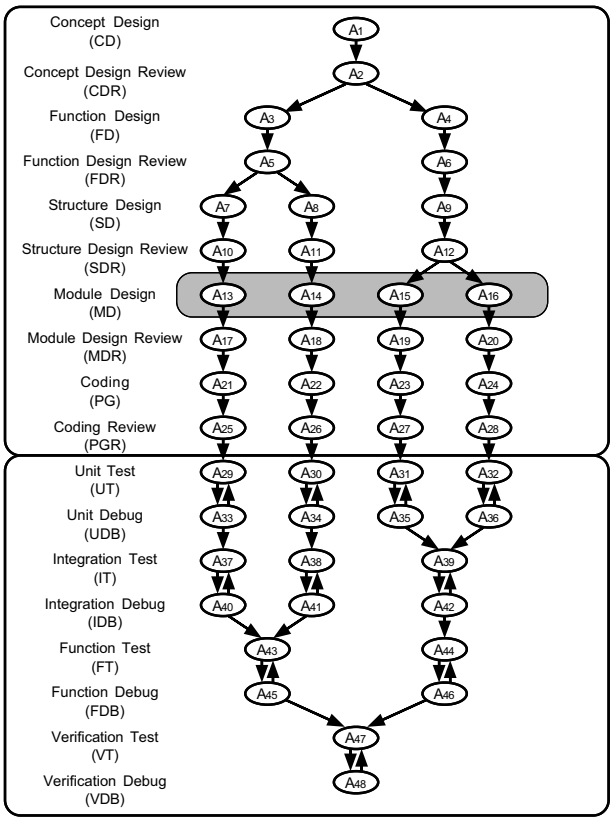

Fig. 1 Actual development process

- Several activities are executed concurrently for certain functions.

  For example, module designs with A_{13}, A_{14}, A_{15} and A_{16} are included in Fig. 1. This implies that the target system consists of four subsystems and that module designs are executed by four groups in the team.

- Certain activity is unwillingly executed with previous activity concurrently.

  This concurrency is not explicitly shown in Fig. 1, but it is recognized by the interviews with project manager in the certain company. Additionally, the analysis of effects of this concurrency on the quality and productivity has already been reported[8].

However, this actual development process is too complicated for further discussions in this paper. Then, we propose a simple process model, shown in Fig. 2, in which the fundamental flow structure in Fig. 1 is remained. By this simplification, for example four module design activities A_{13}, A_{14}, A_{15} and A_{16} in Fig. 1 are merged into one module design(MD) in Fig. 2. (But, the simple process in Fig. 2 is used to define the proposed methods clearly. The actual simulation in Sect. 5 is performed using the actual process in Fig. 1.)

Additionally, we divide logically the whole process into two phases from the viewpoint of faults: design phase and debug phase. The reason is that the activities in the design phase are essentially for increasing the size of the product and thus introducing faults, but the activities in the debug phase are for detecting and removing faults from the product(thus decreasing the size of the product).

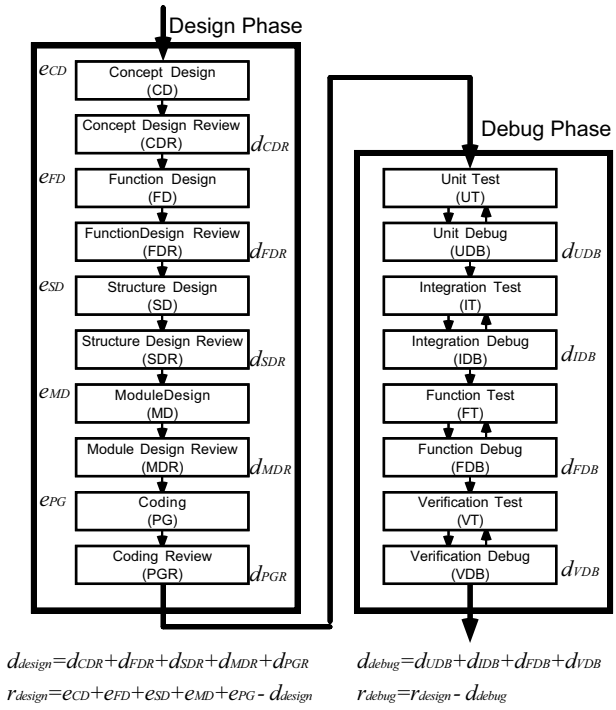

Fig. 2 Simple development process

2.2 Simple Software Process Model

Then, we assume that software development process consists of two successive phases: design and debug phases. Design phase includes ten activities (Concept Design (CD), Concept Design Review (CDR), Function Design (FD), Function Design Review (FDR) Structure Design (SD), Structure Design Review (SDR), Module Design (MD), Module Design Review (MDR), Coding (PG) and Code Review (PGR)). Debug phase includes eight activities (Unit Test (UT), Unit Debug (UDB), Integration Test (IT), Integration Debug (IDB), Function Test (FT), Function Debug (FDB), Verification Test (VT) and Verification Debug (VDB)).

As shown in Fig. 2, we introduce several parameters to make the discussions clear. Let e_i be the number of faults introduced into the product developed in design/coding activity i. Let d_j be the number of detected and removed faults in reviews/debug activity j. Let d_{design} be the total of d_{CDR}, d_{FDR}, d_{SDR}, d_{MDR} and d_{PGR}. Let r_{design} be the number of residual faults in design phase and is calculated by $\Sigma e_i - d_{design}$. Also, let d_{debug} be the total of d_{UDB}, d_{IDB}, d_{FDB}, and d_{VDB}. Let r_{debug} be the number of residual faults in debug

phase and is calculated by $r_{design} - d_{debug}$.

Strictly speaking, acceptance test and debug are executed after VDB. Since these are not activities of development team, we omit them from Fig. 1 and Fig. 2. Let r_{last} be the number of residual faults in the acceptance test and debug. Thus we can consider that r_{last} is the number of residual faults in the last product(usually program codes).

However, as a matter of fact, we cannot collect all values of the above parameters. Thus, in this paper, we aim to estimate the value of d_{design}, r_{design}, d_{debug}, r_{debug} and r_{last}. In Sect. 5, we evaluate the usefulness of the proposed project control method by comparing the estimated values of d_{design}, r_{design}, d_{debug}, r_{debug} and r_{last} with the actual ones.

2.3 Project Plan

The initial project plan consists of the assignments of both developers and the time to each activity. The assignments are determined, at the beginning of project, based on the documents such as WBS(Work Breakdown Structure) charts, Organization charts of project, PERT charts and a list of software products to be developed. Simultaneously, the initial plan also reflects experience and knowledge of the project manager.

First, we perform an estimation using the initial plan. However, some difference usually occurs between the initial plan and the actual progress of the project. If the developers follow the initial plan regardless of the existence of the difference, then the fatal confusion may be caused. On the contrary, if we can take the progress into consideration and reconstruct a new plan, then we might get more accurate estimation. In Section 3, we will propose such a new project control method.

3. Project Control Methods

In this Section, we explain about proposed project control methods and project plan. The detail of project plan will be explained in subsection 4.3.

3.1 Single-Phase Control

In single-phase control of software development process, we construct the initial project plan p at the beginning of project, and execute whole project under the initial project plan p(See Fig. 3).

In the following, p_{design} and p_{debug} denote the project plan of design and debug phases, respectively. Then we often use $p = (p_{design}, p_{debug})$ to denote a whole plan of project.

Let the p^{init}_{design} and p^{init}_{debug} denote the initial plans of design and debug phases, respectively. We assume that both p^{init}_{design} and p^{init}_{debug} are constructed at the beginning of the project. Then, the project plan of the single-phase controlled project is described as follows:

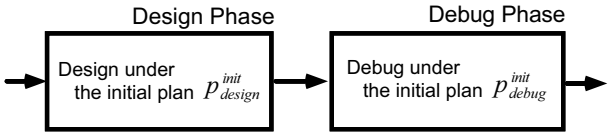

Fig. 3 Single-phase control

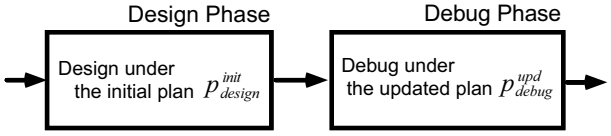

Fig. 4 Two-phase control

$$p = (p^{init}_{design}, p^{init}_{debug})$$

3.2 Two-Phase Control

Next, in two-phase control of software development process, we construct two versions of project plans $p = (p^{init}_{design}, p^{init}_{debug})$ and $p' = (p^{init}_{design}, p^{upd}_{debug})$. At the beginning of project, we construct the project plan $p = (p^{init}_{design}, p^{init}_{debug})$ and execute the design phase under p^{init}_{design}. Then we stop the execution of the project at the end of design phase, and update the plan of debug phase p^{init}_{debug} into a new plan p^{upd}_{debug} taking the development situation into account. After that we continue the project under the updated new project plan p^{upd}_{debug}(See Fig. 4).

For simplicity, we represent the project plan of the two-phase controlled project as follows:

$$p = (p^{init}_{design}, p^{upd}_{debug})$$

As already mentioned, in the project plan p for two-phase control, p^{init}_{design} is the initial project plan for design phase and p^{upd}_{debug} is the updated plan of debug phase, which is constructed at the end of design phase.

4. Simulator based on GSPN Model[8]

4.1 Outline of the GSPN Model

The proposed model for project simulator consists of Project model and Process model. Figure 5 shows the outline of the proposed model.

Project model includes three key components: activities, products and developers. Some attributes are attached to each of them, to be shown in Table 1.

Process model includes a set of Activity models which include specifications of design, coding, review, test, debug activities, and so on. Activity model is described by an extended GSPN, to be shown in Fig. 6.

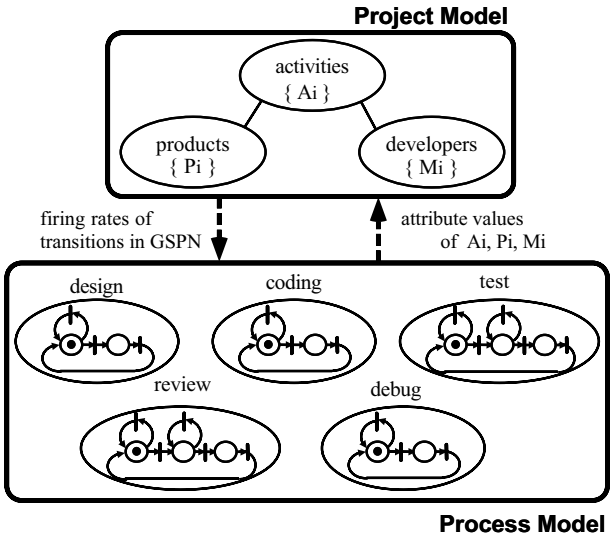

Fig. 5 Outline of the proposed model

4.2 Workload

In [10], we define the term "workload" of an activity as the total time needed for a developer who has the standard capability to complete the activity. Additionally, an efficiency of the activity under such a condition is quantified as 1. The value of efficiency depends on the environment, such as the number of the developers, the necessity of communication and performance of CASE tools. Then, the development time is calculated as the result of dividing the workload by the efficiency of the activity.

If we get the workload of an activity, then we can estimate the development time appropriate for specific several activity conditions dependent on a given environment.

In the proposed model, the workload is assigned to each activity depending on the input products for the activity. That is, for example, workload of design activity (W_{design}) is defined as the following formula:

$$W_{design} = s_{design} \times K_{design}.$$

Here, s_{design} denotes the size of input product of design activity and K_{design} denotes the workload parameter for design activity. Before simulation, workload parameter must be given to each activity of the target project.

Consuming of the workload assigned to an activity corresponds to the progress of the activity in the development. Growth of product can be modeled by changing values of the size or the number of faults in the output product.

Table 1 Project template

Attributes of activity A_i
type
entry condition
exit condition
input products
output products
workforce
deadline
workload

Attributes of product P_i
size
number of faults
completion rate

Attribute of developer M_i
experience level

4.3 Project Model and Project Plan

Project model focuses on three key components: activities, products and developers, and attaches several attributes to each of them (See Table 1). The project template with values assigned for each attribute is generally called the project plan.

An activity has eight kinds of attributes, which are *type*, *entry/exit conditions*, *input/output products*, *workforce*, *deadline* and *workload*.

(1) *Type* shows which the activity corresponds to and describes currently one of design, coding, review, test and debug.

(2) *Entry condition* and (3) *exit condition* specify conditions for beginning and ending the activity, respectively.

(4) *Input products* describes the products given to the activity as the input products and the workload parameter, that is the degree of contribution of each input products to determine workload of the activity.

(5) *Output products* describes the output products that are developed in the activity and the weight assigned to each product. The variation of the product size and that of the number of faults are distributed to the products according to weights. Thus, the sum of each weight must be one.

(6) *Workforce* specifies tuples of the developers who engage in the activity and the ratio of time in which each developer can engage in the activity to his or her business hours.

(7) *Deadline* represents the appointed date for the completion of the activity, which is fixed on the development plan.

(8) *Workload* represents a tuple of the workload assigned to the activity and consumed amount of it.

Next, a product has three kinds of attributes(See Table 1), which are size, the number of faults and completion rate. (1) *Size* represents the product size in document pages or the lines of source code. (2) *Number of faults* counts faults in the product. (3) *Completion rate* represents the ratio of the consumed workload to the assigned workload.

Then, a developer has an attribute *experience level*(See Table 1) which is determined according to his/her length of service. We classify developers' experience levels into the following three levels: novice,

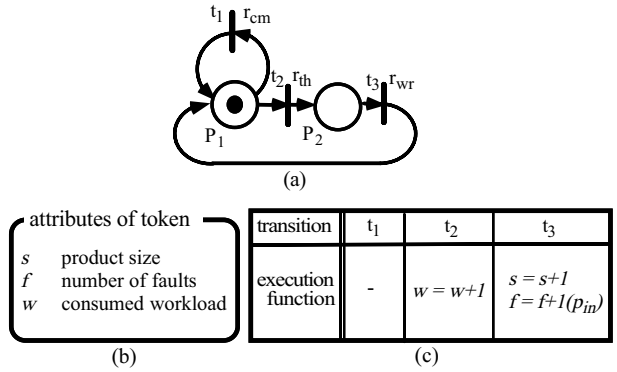

Fig. 6 Activity model

standard and expert levels. They are quantified as discrete values 1, 2 and 3, respectively.

4.4 Activity Model for Process

Activity model is prepared for each type of activities such as design, coding, review, test, debug and so on. The descriptions of Activity models are given using an extended GSPN. Figure 6(a) shows an example of the description of the design activity. In the extended GSPN, a token has three attributes: product size s, number of faults f and consumed workload w, as shown in Fig. 6(b). These attributes are used to represent the current status of development that varies over the execution of each Activity model.

Transitions used here are timed transitions. The firing delay of each transition is exponentially distributed and the average firing delay of a transition is specified by a firing rate assigned to it. In Fig. 6(a), the firing rate r_{cm} of transition t_1 means that the average firing delay of transition t_1 is $1/r_{cm}$.

In addition, each transition has a function (called execution function) to be evaluated on its firing, as shown in Fig. 6(c). The execution of the function updates the attribute values of the token. Intuitively speaking, each transition corresponds to the developers' behavior such as thinking, writing and communicating or an event which occurs during execution of activity. Places correspond to waiting states for occurrences of behaviors or events.

[Example] Consider transitions t_1 and t_2 in the design Activity model depicted in Fig. 6. These two transitions represent communicating and thinking behavior, respectively, and are enabled to fire when a token exists in the place P_1. If the communicating transition t_1 fires, it has no effect on the attributes values, and the token returns to the place P_1 and only time elapses by the firing delay. On the other hand, if the transition t_2 fires by evaluating its execution function, then consumed workload w is increased by one, and the token moves to the place P_2. When the token exists in the place P_2, only the transition t_3 which represents writing behavior is enabled. If the transition t_3 fires, then product size s is increased by one, and the number of faults f could be increased according to the fault injection rate p_{in}(to be explained later). After the firing of t_3, the token moves back to the place P_1.

The firing rates of the transitions are formulated by the following ten functions f_{cm}, f_{th}, f_{wr}, f_{pr}, f_{rd}, f_{dt}, f_{md}, f_{ps}, f_{lc} and f_{in}. These functions should be concretely specified based on the property of the target project.

In the following, M is the number of the developers engage in the activity, L is developer's experience level, ΣL is the sum of each developer's experience, S is the total size of the input products, R is the completion rate of the input products, F is the number of faults of the input products, D is the number of the days from the current date to the deadline of the activity. Activity model parameters (e.g., K_{cm}, K_{th}, K_{wr} and K_{in}) are given to each activity and concerned with the developers' behavior. For example, K_{cm}, K_{th}, K_{wr} and K_{in} correspond to communicating, thinking, writing and fault injection, respectively.

(1) Communicating rate: $r_{cm} = f_{cm}(K_{cm}, M, \Sigma L, R)$

(2) Thinking rate: $r_{th} = f_{th}(K_{th}, M, \Sigma L)$

(3) Writing rate: $r_{wr} = f_{wr}(K_{wr}, M, \Sigma L)$

(4) Preparing rate: $r_{pr} = f_{pr}(K_{pr}, M, \Sigma L, S)$

(5) Reading rate: $r_{rd} = f_{rd}(K_{rd}, M, \Sigma L)$

(6) Fault detecting rate: $r_{dt} = f_{dt}(K_{dt}, M, \Sigma L, S, F)$

(7) Fault modifying rate: $r_{md} = f_{md}(K_{md}, M, \Sigma L)$

(8) Test-case passing rate: $r_{ps} = f_{ps}(K_{ps}, M)$

(9) Fault localizing rate: $r_{lc} = f_{lc}(K_{lc}, M, \Sigma L, S, F)$

These make it possible to dynamically determine the frequency of communications or the difficulty in thinking and writing according to the number of developers, experience levels of developers and/or completion rates of input products.

Moreover, the increase of product size s at every firing of writing transition t_3 and the consumption of workload at every firing of thinking transition t_2 are described by the corresponding execution functions. At each firing of the transition, the values of token's attributes can be changed by evaluating its execution function.

Activity model handles fault injections in the design activity as the stochastic events whose occurrences depend on the fault injection rate p_{in}. In general, p_{in} is formulated by the following function:

(10) Fault injection rate: $p_{in} = f_{in}(K_{in}, M, \Sigma L, D, R)$

By using this function, it is possible to take account of dynamic influence on the fault injection rate caused by the stress from deadline of the activity or developers' experience levels.

Though the functions (1)–(3) are used in Fig. 6, the rests (4)–(9) are not included in Fig. 6. Figure 6 shows an example of design and coding activities and thus the improvement of model should be conducted through case studies. Besides we also modeled other activities(review, test and debug), in which the functions (4)–(9) are used. The detail of activity models is shown in [9].

4.5 Simulator

We have designed and implemented a simulator which supports description of the target process, executes the process described by Activity model and analyses the simulation results statistically.

The main simulation results include d_{design}, r_{design}, d_{debug} and r_{debug}. As already explained, acceptance test and debug are executed just after verification debug in Fig. 2. From the simulation for these activities, we also get r_{last}.

The system consists of five functional units: project control unit, activity simulator, user interface unit, display unit and editor.

Simulations proceed at the intervals of unit time†. At first, project control unit determines activities to be executed, based on the current status of the progress and *entry/exit conditions* of each activities. Next, for each executable activity, project control unit delivers the parameters to activity simulator and directs it to execute activities for a day. Then, activity simulator executes all of the activities, which are directed to execute by project control unit, using given parameters and extended GSPN. The execution of an activity is expressed by the consumption of its workload. When an activity consumes all of assigned workload, the activity is regarded to be completed.

The execution of simulation is able to be suspended or restarted at any time. Moreover, at every unit time of simulation, intermediate simulation results can be stored in the simulation database. The intermediate simulation results are stored in the same format of the original project description. Thus, it is possible to restart the simulation using the intermediate simulation results as the input. Also it is possible to change the values of parameters(attributes of project) at any time of the simulation. For example, we can modify the number of developers at any time of the simulation and simulate it immediately.

†Currently, one unit time is a day (8 hours).

Table 2 Target projects

	Type of Project	Project Plan
Project 0	Actual	$p_1 = (P^{actual}_{design}, P^{actual}_{debug})$
Project 1	Imaginary	$p_0 = (P^{init}_{design}, P^{init}_{debug})$
Project 2	Imaginary	$p_2 = (P^{actual}_{design}, P^{init}_{debug})$
Project 3	Imaginary	$p_1 = (P^{actual}_{design}, P^{actual}_{debug})$
Project 4	Imaginary	$p_3 = (P^{actual}_{design}, P^{uniform}_{debug})$

5. Case Study

In order to evaluate the usefulness of the proposed project control method, we apply the simulator to similar software development projects, which are constructed based on the actual project data in an organization. In the following, we call all of software development projects thus constructed as imaginary software development projects.

5.1 Characteristics of Target Project

The main characteristics of the project are summarized as follows[13]:

(1) Development effort of the project was 62 (man-days).

(2) The size of the system of the project was about 6.9 (Ksteps).

(3) The project uses a standard waterfall model.

5.2 Experimental Projects

Now, we consider five development projects: Project 0, Project 1, Project 2, Project 3 and Project 4, based on the development data of actual project. Table 2 summarizes the characteristic attributes of these projects. Project 1 and Project 2 correspond to the project that executes debug phase under initial plan. Clearly, Project 3 is the project that applies the proposed approach and Project 4 is the project that follows a uniform plan. The details will be explained in subsections 5.2.1, 5.2.2 and 5.2.3.

Project 0 is the only actual project and the rests Project 1 through Project 4 are imaginary projects. All imaginary projects are executed by the project simulator mentioned in Sect. 4. Since we evaluate all imaginary projects by comparing with Project 0(for which we have data collected from actual project in the company), we assume that the design phase of any imaginary projects(except for Project 1) is executed under the actual plan P^{actual}_{design}. From this assumption, we can get the simulation results with high accuracy for the debug phase, and can compare in detail the resultant data among five development projects.

5.2.1 Actual development

This is the actual development executed in a certain company.

Let P^{init}_{design} and P^{init}_{debug} denote initial project plans for the design and debug phases, respectively, which were actually constructed at the beginning of the project in the company. On the other hand, let P^{actual}_{design} and P^{actual}_{debug} denote specific project plans for the design and debug phases, which we constructed based on the records and the data collected at the completed target project.

In summary, the project was originally planned to be executed under the following initial plan p_0:

$$p_0 = (P^{init}_{design}, P^{init}_{debug})$$

From the actual resultant data of the project, we can interpret that the project was executed under the following project plan p_1:

$$p_1 = (P^{actual}_{design}, P^{actual}_{debug})$$

Therefore, we call this actual project (which was executed by the developers under the project plan p_1) as Project 0.

5.2.2 Imaginary projects(single-phase control)

Here, we define two imaginary development projects which are executed under single-phase controlled plan.

At first, we define an imaginary project, Project 1. In Project 1, design phase and debug phase are executed under the initial plan P^{init}_{design} and P^{init}_{debug}, respectively. That is, the whole project is executed by project simulator under the initial plan p_0.

Next, we consider an imaginary project, Project 2, in which design phase is executed under the actual plan P^{actual}_{design} and debug phase is executed under the initial plan P^{init}_{debug}. By this, we want to make a case that the plan for design phase in Project 2 was not updated(which is the difference from Project 0).

Thus, Project 2 is executed under the following plan p_2 of single-phase control:

$$p_2 = (P^{actual}_{design}, P^{init}_{debug})$$

Note that two projects Project 1 and Project 2 have the same project plan P^{init}_{debug} for debug phase, and that they have the different plans for design phase.

5.2.3 Imaginary projects(two-phase control)

Now, we define two imaginary development projects Project 3 and Project 4 under two-phase project control method. As mentioned before, we assume that the design phase of Project 3 and Project 4 is executed under the actual plan P^{actual}_{design}.

At first, we consider an imaginary project Project 3 in which design phase is executed under the actual plan P^{actual}_{design}, and its debug phase is also executed under the actual plan P^{actual}_{debug}. That is, Project 3 is executed under the project plan p_1.

Note that Project 3 is very close to Project 0, since Project 3 is entirely simulated by using actual data of Project 0. The difference is that Project 0 is executed by developers, while Project 3 is executed by project simulator.

Finally, we consider another imaginary project Project 4, in which design phase is executed under the actual plan P^{actual}_{design}, and its debug phase is executed under the virtual uniform plan $P^{uniform}_{debug}$. We construct the uniform plan $P^{uniform}_{debug}$ as follows: First, we obtain the total time to debug from the debug plan data. Next, each debug activity in this project is assigned uniformly the same amount of workload to be consumed. With respect to other attributes, the value in P^{init}_{debug} is also used in $P^{uniform}_{debug}$. Thus, Project 4 is executed under the following uniform project plan p_3.

$$p_3 = (P^{actual}_{design}, P^{uniform}_{debug})$$

5.3 Assumptions on Simulation

Here, we formulate each firing rate and fault injection rate in the Activity models. For example, we show the formulas used in the design activity. In the following formulas, M is the number of the developers engaged in the activity, ΣL is the sum of each developer's experience level, R is the completion rate of the input products, D is the number of the days from the current date to the deadline of the activity. K_{cm}, K_{th}, K_{wr} and K_{in} are parameters given to each activity and concerned with communicating, thinking, writing and fault injection rate, respectively.

$$r_{cm} = K_{cm} \times \frac{M^2}{\Sigma L \times R}$$

$$r_{th} = K_{th} \times \frac{\Sigma L}{M} \times M = K_{th} \times \Sigma L$$

$$r_{wr} = K_{wr} \times \frac{\Sigma L}{M} \times M = K_{wr} \times \Sigma L$$

$$p_{in} = K_{in} \times \frac{M}{\Sigma L \times R \times D} \times M$$

Here, we explain the sources of these formulas. The relationship between the number of developers and the frequency of communication is discussed in [3]. Similarly, it is said that incomplete input products induce frequent inquiries about the omission of the description[4]. On the other hand, from the reference [12], the individual capability of a developer has an effect on both productivity and quality of software. The influence of mental stress caused by the deadline is discussed

Table 3 Assignment of workload

	UT & UDB	IT & IDB	FT & FDB	VT & VDB
P_{debug}^{init}	44	44	80	56
P_{debug}^{actual}	45	27	96	64
$P_{debug}^{uniform}$	52	52	52	52

in [5]. Based on the knowledge from these sources and the interviews with developers in a certain company, we determined these formulas for convenience. Although, these formulas are not theoretically proved, we can simulated the projects appropriately using these formulas and values of parameters(to be explained in subsection 5.4).

Next, we explain an assignment of workload to each activity. As we mentioned before, we use five project plans, P_{design}^{init}, P_{design}^{actual}, P_{debug}^{init}, P_{debug}^{actual} and $P_{debug}^{uniform}$. Among them, Table 3 shows a part of project plans and summarizes the assignment of workload to each activity(UT and UDB, IT and IDB, FT and FDB, and VT and VDB)(see Fig. 2). For P_{debug}^{init}, this assignment is obtained from the plan of the project. For P_{debug}^{actual}, the assignment is obtained from the resultant data of project. For $P_{debug}^{uniform}$, the same amount of workload is assigned uniformly to each activity.

As spaces are limited, we omit the description of Project model, formulas of firing rates, and parameters used by other activity models.

5.4 Simulation Result

By using the project simulator, we iterate simulations for 1000 times per each project and calculate the average value of quality metrics.

Project 0 (actual development) has no information about residual faults at the end of the design/debug phase. As for the residual faults in the last product(r_{last}), we can find the value from detected faults at the acceptance test, but as for the residual faults in other products, we cannot obtain such data from the company.

As for Project 1, r_{debug} became 7.42. It gives the worst value among all projects. One of the reason is that Project 1 was executed using the initial project plan p_0 with no check points. (In some sense, Project 1 is essentially the same as Project 0. However Project 0 can be interpreted that a check point was prepared at the end of design phase.)

In Project 2, the design phase is executed under the actual plan P_{design}^{actual}, and the debug phase is executed under the initial plan P_{debug}^{init}. The result $r_{debug}(= 6.04)$ is better than that of Projects 1 and 4, but worse than that of Project 3. From the definition of Project 2, the project plan P_{debug}^{init} never reflects the situation at the end of design phase, and thus it could not remove faults sufficiently.

The value of $r_{debug}(= 4.69)$ in Project 3 becomes the best in this experiment. Regarding to r_{debug}, we can say that the debug phase works very effectively under the project plan P_{debug}^{actual}. Since the plan of debug phase was updated from the initial one by considering the resultant situation of the design phase, the updated plan might give adequate control for the succeeding debug phase(as shown in Table 3), and improve the value of r_{debug} compared with Project 2.

To tell the truth, we predicted that Project 4 would not show good performance because we constructed the debug plan without considering the situation of design phase. The result, $r_{debug} = 6.97$, is better than that of Project 1, but is worse than that of Projects 2 and 3.

As explained in subsection 5.2, the design phases of Projects 2, 3 and 4 were executed under the same actual project plan. That is, on the Table 4, the values of d_{design} and r_{design} are almost the same among Projects 2, 3 and 4.

From the viewpoint of residual faults in all products(that is, the value of r_{debug}) after debug phase, Project 3 has the least faults, and is the most superior to other Projects 1, 2 and 4. Besides, judging from residual faults in the last product(the value of r_{last}) after acceptance test and debug, there are no essential differences among four projects.

Here, we investigate the reason why differences appear in five experimental projects. From the further investigations of the actual development data of Project 0, the time assignment in the project plan P_{debug}^{actual} for the debug phase was rather reasonable. Especially, the data of Project 0 implies that if concept design takes longer time compared with the estimated time in the initial plan, then developers need to spend longer times on verification test and debug.

Of course, there are many other factors, such as developers and fault density, to affect the fault injection and removal in the project. So we cannot conclude that the time assignment (that is, workload assignment) to each activity in debug phase is the most effective factor to the number of detected/residual faults. However, as far as we concern the simulation results, updating the time assignment in the project plan of debug phase will effect influentially the number of detected/residual faults and quality of the products at the end of debug phase.

6. Conclusion

In this paper, we have proposed the two-phase project control method to realize the high quality of the products. In order to evaluate the proposed method, we have defined four imaginary projects using the actual project data from a certain company. Finally, from the results of project simulation, we have confirmed the effectiveness of the proposed method.

In the experimental evaluations, we have assumed

Table 4 Values of quality metrics

	Executor	Plan	Design Phase		Debug Phase		Acceptance Test
			d_{design}	r_{design}	d_{debug}	r_{debug}	r_{last}
Project 0	developmers	$p_1 = (P^{actual}_{design}, P^{actual}_{debug})$	19	—	16	—	1
Project 1	simulator	$p_0 = (P^{init}_{design}, P^{init}_{debug})$	22.80	25.39	17.97	7.42	0.82
Project 2	simulator	$p_2 = (P^{actual}_{design}, P^{init}_{debug})$	19.14	19.83	13.79	6.04	0.94
Project 3	simulator	$p_1 = (P^{actual}_{design}, P^{actual}_{debug})$	19.23	19.71	15.02	4.69	1.16
Project 4	simulator	$p_3 = (P^{actual}_{design}, P^{uniform}_{debug})$	19.58	19.87	12.90	6.97	1.13

that project plan P^{actual}_{debug} was constructed by the experts with project managements. This construction of P^{actual}_{debug} or the updating P^{init}_{debug} into P^{actual}_{debug} is one of the most important and challenging future research works.

We consider that the functions such as f_{cm}, f_{th}, etc. in subsection 4.4 are basically generalized ones. But the formulas in subsection 5.3 are considered to be target specific. Now, we are applying this model and simulator to much more projects. If we can get successful simulation results from the applications also, then it will become a good evidence for generalizing our result of the case study.

Additionally, we have simplified the actual development process shown in Fig. 1 to make our discussions clear. In order to show the practical effectiveness of the two-phase project control method, we are now trying to extend the process model shown in Fig. 2 to distinctly and explicitly include such concurrent activities as described in subsection 2.1.

References

[1] T. K. Abdel-Hamid: "The dynamics of software project staffing: A system dynamics based simulation approach," IEEE Trans. Software Eng., vol.15, no.2, pp.109–110, 1989.

[2] M. Aoyama: "Agile software process model," Proc. 21th Annual International Computer Software and Applications Conference, pp.454–459, 1997.

[3] F. P. Brooks Jr.: "The Mythical Man-Month," Addison Wesley, 1975.

[4] B. Curtis, H. Krasner and N. Iscoe: "A field study of the software design process for large systems," Commun. ACM, vol.31, no.11, pp.1268–1287, 1988.

[5] T. Furuyama, Y. Arai and K. Iio: "Fault generation model and mental stress effect analysis," The Journal of Systems and Software, vol.26, pp.31–42, 1994.

[6] F. J. Heemstra: "Software cost estimation," Information and Software Technology, vol.34, no.10, pp.627–639, 1992.

[7] W. S. Humphrey: "Managing the Software Process," Addison Wesley, MA, 1989.

[8] Y. Hirayama, O. Mizuno, S. Kusumoto, T. Kikuno: "Hierarchical Project Management Model for Quantitative Evaluation of Software Process," Proc. International Symposium on Software Engineering for the Next Generation, pp.40–49, 1996.

[9] Y. Hirayama: "Quantitative evaluation of software process based on hierarchical project management model," Master dissertation, Osaka University, 1996.

[10] S. Kusumoto, O. Mizuno, Y.Hirayama, T. Kikuno, Y. Takagi and K.Sakamoto: "A new software project simulator based on generalized stochastic petri-net," Proc. 19th International Conference on Software Engineering, pp.293–302, 1997.

[11] J. D. Musa, A. Iannino and K. Okumoto: "Software reliability: measurement, prediction, application," McGraw-Hill, 1987.

[12] H. Sackman, W. J. Erickson and E. E. Grant: "Exploratory experimental studies comparing online and offline programming performance," Commun. ACM, vol.11, no.1, pp.3–11, 1968.

[13] Y. Takagi, T. Tanaka, N. Niihara, K. Sakamoto, S. Kusumoto and T. Kikuno: "Analysis of review's effectiveness based on software metrics," Proc. of the 6th International Symposium on Software Reliability Engineering, pp.34–39, 1995.

[14] E. Yourdon: "Death March : The Complete Software Developer's Guide to Surviving 'Mission Impossible' Projects," Prentice Hall Computer Books, 1997.

Osamu Mizuno received the B.E. degree in information and computer sciences from Osaka University, in 1996. He is currently Master course student in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. His research interests are software process, software quality assurance technique. He is a student member of the IEEE.

Shinji Kusumoto received the B.E., M.E. and Dr. Eng. degrees in information and computer sciences from Osaka University, in 1988, 1990 and 1993, respectively. He is currently Assistant Professor in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. His research interests are software metrics, software quality assurance technique. He is a member of the IEEE and IPSJ.

Tohru Kikuno received the B.E., M.E. and Ph.D. degrees in electrical engineering from Osaka University, in 1970,

1972, and 1975, respectively. He joined Hiroshima University from 1975 to 1987, from 1988 he has been working for Osaka University. He is currently Professor in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. His research interests include analysis and design of fault-tolerant systems, quantitative evaluation of software development process and design of testing procedure of computer protocols. He was a general co-chairman of the Second International Workshop on Responsive Computer Systems in 1992. He is a member of the IEEE, ACM and IPSJ.

Yasunari Takagi received the B.E. degree in information and computer science, from Nagoya Institute of Technology, Nagoya, Japan, in 1985. He has been working for OMRON Corporation.

Keishi Sakamoto received the B.E. degree in electrical engineering, from Kobe University, Kobe, Hyogo, Japan, in 1969. He has been working for OMRON Corporation.