Systems and Computers in Japan, Vol. 37, No. 11, 2006
Translated from Denshi Joho Tsushin Gakkai Ronbunshi, Vol. J87-D-I, No. 8, August 2004, pp. 815–823

Program-Delta Oriented Debugging Supporting Method DMET

Makoto Matsushita,[1] Masayoshi Teraguti,[2] and Katsuro Inoue[1]

[1]Graduate School of Engineering Science, Osaka University, Toyonaka, 560-8531 Japan

[2]Tokyo Research Laboratory, IBM Japan, Ltd., Yamoto, 242-8502 Japan

SUMMARY

Research has been conducted on methods of automatically performing tests that focus on the delta (difference) between a revision that is known to function normally in advance and a revision that contains defects to identify the causes of the defects. However, since software had to be created from source programs for each test, an extremely long test time had been required. Also, since the emphasis had only been on the testing tasks and no consideration was given to debugging tasks, these methods were not practical. Therefore, in the current research, the authors propose the DMET debugging technique, which can be used for actual software maintenance. DMET, which reduces the test execution time compared with conventional techniques, supports the entire sequential processing flow from testing to debugging. Also, to verify the effectiveness of the proposed technique, the authors implemented the DSUS development support environment, which uses DMET, and performed comparison experiments. From the results, it was apparent that using DMET enabled the total debugging time to be shortened. © 2006 Wiley Periodicals, Inc. Syst Comp Jpn, 37(11): 35–43, 2006; Published online in Wiley InterScience (www.interscience.wiley.com). DOI 10.1002/scj.20351

Key words: revision management; delta; debugging.

1. Introduction

Software maintenance is processing that is performed to enable active software to continue operating or to improve that software [3]. The proportion of the total cost of software devoted to maintenance has risen to approximately 80% [4], and approximately 65% of the total time spent by people involved with software is devoted to maintenance or tasks related to it [2].

To understand the essence of software maintenance, Swanson classified the main causes for which software maintenance is required into three basic types [15]:

- Defects that cause errors within the software
- Changes to the environment surrounding the software
- Requests by users or personnel responsible for maintenance

In addition, Swanson defined the maintenance activities that are executed corresponding to these basic causes as follows:

- Corrective maintenance: Identifies and corrects errors
- Adaptive maintenance: Makes corrections according to changes in the environment
- Perfective maintenance: Improves performance and alters or adds functions

Generally, among these software maintenance activities, the ones that tend to be considered to occur most often are activities for corrective maintenance. However, when

Lientz and Swanson conducted surveys in the 1980s for the first time, they reported that corrective maintenance comprises no more than 20% of maintenance tasks, and perfective maintenance accounts for 55% [11].

During these maintenance activities, many changes are made to existing software. However, research by Hetzel showed that the probability that errors will be introduced when software is modified was from 50% to 80% [7]. Therefore, during corrective maintenance and perfective maintenance, which comprise 75% of maintenance activities, operation must be verified not only for functions in the parts that were modified, but also for functions in parts that were not modified.

Regression testing [6, 10, 13] has been widely used for these operation verification activities that occur during development. Also, the software that is developed is often managed by using a software management system such as a revision management system [1, 16]. Attention given to these two points has led to previous research on means of supporting perfective maintenance of software that was being managed by a software management system [12, 17]. However, the test tools [9] that were to be used when executing tests had to be established and testing had to be performed manually in certain circumstances. Also, support for the debugging tasks that are performed after testing was insufficient.

Therefore, the current research targets perfective maintenance tasks that are performed to modify or extend functions. During these kinds of maintenance tasks, parts other than the functions for which the tasks were performed may be affected, and defects may be detected by the tests. In this paper, we propose the DMET debugging technique, which uses regression testing, to efficiently eliminate these kinds of defects.

This technique first chooses product revisions sequentially and uses a test tool to automatically perform testing. When a revision that outputs defects and a revision that outputs normally are detected as a result of testing, the corrections that were performed between these revisions are assumed to contain an error. Next, a check is performed to determine which parts in the current revision correspond to these correction contents (hereafter, referred to simply as the delta). The user uses these results to perform debugging and correct the defects. By also reflecting those changes in the old revision after the defects were corrected, they can be used for subsequent tests.

To verify the effectiveness of DMET, we created a trial debugging support system named DSUS, which uses DMET, and performed comparison experiments using DSUS. From the experimental results, it was apparent that DMET was able to support the sequential processing flow from testing to debugging and that it could shorten the time required for debugging tasks during actual maintenance.

2. Software Maintenance

2.1. Maintenance tasks

The tasks that are performed to modify software during maintenance activities basically can be classified into the following three types.

(1) Understanding the software and the modifications that must be made
(2) Modifying the software
(3) Verifying the operation of the modified software

Of these, if the type (1) tasks are performed correctly, they are useful for investigating the causes of defects that are discovered. Conversely, if the software is poorly understood, a great deal of time will be required to investigate the causes of defects and correct them. As a result, one technique for improving software understandability and maintainability is to perform software configuration management or revision management.

Software configuration management refers to tasks for identifying, controlling, and understanding the states of various products that are created during the software development and maintenance processes [5]. Also, revision management consists of tasks for correctly recognizing, organizing, and managing the various corrections made to the products (software or accompanying documents) that were created by the development team. Models of various management techniques have been proposed. In addition, revision management systems based on those models have been implemented [1, 16]. Normally, products are revised multiple times, and the product that is created for each alteration is called a revision.

When performing type (3) tasks, one technique for testing whether or not the software operates with the performance required by the specifications after the software was modified is to perform regression testing [6, 10, 13]. Regression testing, which has been put to practical use during actual software maintenance, is used for examining whether corrections were performed properly.

2.2. Problems with existing techniques

During the maintenance stage for software that was developed by using a revision management system, there exists a revision (hereafter, referred to as the base revision) that operates with the performance required by the specifications. With conventional software maintenance, the program of the base revision is changed and the operation of the software is verified by using regression testing during the types (2) and (3) maintenance tasks described above.

If a defect is discovered during testing, the types (1) and (2) maintenance tasks are performed repeatedly to eliminate the error that is the cause of that defect (to perform debugging). However, if the defect is found in a function that has not been changed, it is generally difficult to find the error that caused it.

Therefore, research has been conducted [12, 17] to identify the error that is the cause of the defect by automatically performing testing using the corrections that were made from the base revision until the current revision. However, a great deal of time was required for testing because each time a test was performed, software for applying the test was created by modifying and compiling the base revision. Education and training related to the test tool [9] that was used for testing were also required since each time a defect was discovered, test cases had to be sequentially created corresponding to that defect. In addition, since only the testing task for identifying the cause of the defect was emphasized and no consideration was given to the subsequent tasks up to debugging, the technique could not be directly applied to actual maintenance tasks.

3. DMET Debugging Technique

This section describes our proposed DMET debugging technique, which uses regression testing and program delta information that is maintained by a revision management system.

3.1. Overview

DMET, which derives its name from Debugging METhod, is a technique for improving the efficiency of debugging tasks when all functions work normally in the base revision but defects are created and incorporated in some functions during maintenance tasks. DMET eliminates defects by repeatedly applying a testing technique, display technique, and reflection technique.

The testing technique automatically performs tests by using a test tool to identify the causes of defects between revisions. DMET uses regression testing to verify whether or not each revision contains defects. If a defect is detected during regression testing, a test (hereafter, referred to as a localization test) is performed to verify which revision that defect was created in and incorporated.

The display technique displays the delta with the revision of the executable program that was identified by the testing technique on the latest revision of the source program. Since source program correction tasks are generally performed on the latest revision, this technique automatically obtains the modifications that were made from the identified revision until the latest revision.

The reflection technique reflects the modifications that were made during the correction tasks in the latest revision while going back through the revisions to the revision that was identified by the testing technique. In this paper, we will refer to the collection of revisions in which defects occurred during tests and for which corrective modifications must be reflected in the latest revision as reflection-required revisions. By applying the corrections to the reflection-required revisions, subsequent testing tasks can continue more efficiently.

In the following sections, we will first describe the software development environment that is predicated for the proposed technique and then explain specific procedures for the testing technique, display technique, and reflection techniques.

3.2. Prerequisite environment for applying DMET

This section describes prerequisite conditions for the software, its development and maintenance, and the personnel responsible for software maintenance when our proposed debugging technique is used during software maintenance. These types of prerequisites are widely used such as in the development of open source software, which has reached a certain degree of maturity. Therefore, we believe that these prerequisite conditions can be applied without problem in the software development environments that have been used in recent years.

- Development is performed by using a revision management system

We assume that development tasks are performed using a revision management system to understand the work history during development in terms of relatively detailed units. The revision management targets are software source programs and compiled files. When compiled files are registered as revisions, relationships between the relevant files and source program revisions are also registered as collateral information.

- A base revision exists

The target of this technique is software maintenance tasks. Therefore, we assume that by letting the software at the time that the maintenance tasks started be the base revision, a revision always exists for which all functions are guaranteed to operate before modifications are made.

- Some output is produced in response to input

Since this technique uses a testing tool to judge whether the output is correct for the entered test data, the target of this technique is software that produces output. We assume here that the output is generally some visible result such as the output of a character string to a terminal, for example.

- The input and usage environment are not changed

Since large-scale specification changes are not generally made during software maintenance activities, we assume that the input to the software does not change. In other words, the input parameters themselves do not increase or decrease. For similar reasons, the proposed technique also is based on the assumption that the software usage environment does not change. In other words, the hardware or operating environment does not change.

In this kind of software development environment, a developer generally registers development results in the revision management system in terms of small work units. When a single developer is carrying out small-scale development, the regression testing can also be executed each time a revision is registered. However, in general, it is often the case that multiple developers perform individual tasks while cooperating or that a single developer simultaneously performs multiple tasks in parallel. Also, development results in which corrections are incomplete are also often registered to record the intermediate progress of a collection of tasks. Since it is meaningless under these circumstances for regression testing to be executed each time the developer registers a revision, regression testing is used to verify whether the corrections were made properly at the time that a certain collection of tasks has ended.

3.3. Testing technique

The testing technique automatically performs regression testing for executing all prepared tests each time any change is made to the software.

Let T_1 to T_n denote the set of tests used for performing perfective maintenance, which were prepared for the software to be tested. These tests are prepared in advance when the maintenance tasks begin. Also, when V_i denotes the i-th revision for a certain file, let V_B represent the base revision and V_L ($B \leq L$) represent the latest revision. Now, consider the case when test T_i ($1 \leq i \leq n$) is applied to the software at a certain time. First, for each file, let the normal output of this software be denoted by output result $O_{B,i}$ when I_i, which is the input of T_i, is assigned to V_B, which is the revision that is known in advance to operate correctly.

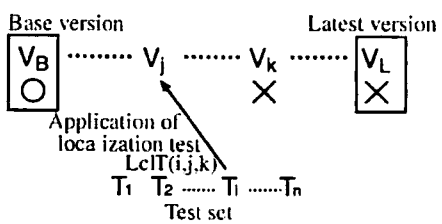

Fig. 1. Localization test.

The test technique first applies all T_i ($1 \leq i \leq n$) to V_L to verify whether the corresponding output $O_{L,i}$ matches $O_{B,i}$. If a defect is detected when a certain test T_i is applied, the localization test $LclT(i, L-1, L)$ is performed.

The localization test $LclT(i, j, k)$ judges the result when T_i is applied to V_j ($j \leq k$) and returns the two revisions between which the cause of the defect exists (Fig. 1). However, we know here that when T_i is used to perform the test for V_k, its output $O_{k,i}$ is the same as $O_{L,i}$. Figure 2 shows the algorithm for the localization test $LclT(i, j, k)$.

(1) If j matches the base revision B, decide that the "defect exists between V_B and V_k" and end.

(2) Apply test T_i to V_j. Specifically, assign I_i to V_j to obtain output $O_{j,i}$.

(3) If $O_{j,i}$ matches $O_{k,i}$, since a defect already exists in revision j, decide that the test execution result is "×" (the same defect is included). In addition, execute $LclT(i, j-1, j)$ recursively and let that result be the result that is obtained.

(4) If $O_{j,i}$ matches $O_{B,i}$, since no defect had been included at the time of revision j, decide that the test execution result is "○" (no defect is included). Also, decide that the "defect exists between V_j and V_k" and end.

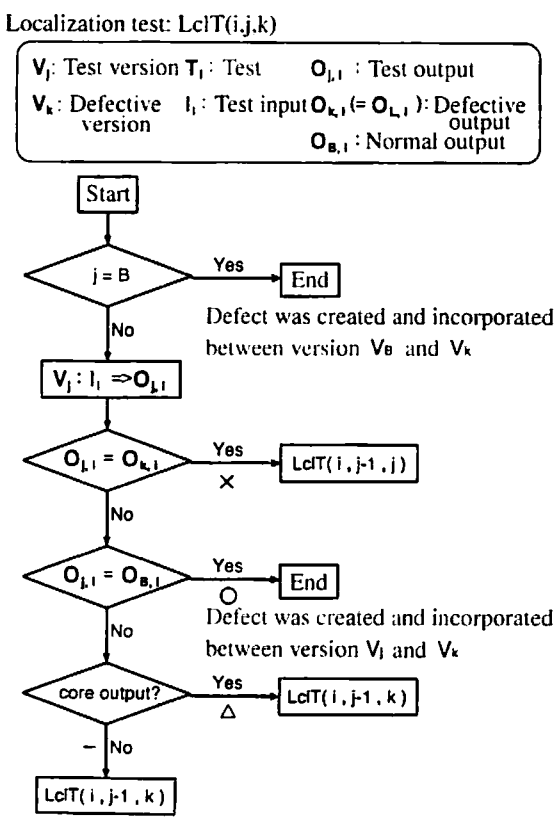

Fig. 2. Localization test algorithm.

(5) If some other $O_{j,t}$ is obtained, decide that the test execution result is "–" (a new defect was detected). Also, if no execution result was obtained because the software was abnormally terminated, set "△" (no result was obtained). In either case, execute $LclT(i, j-1, j)$ recursively and let that result be the result that is obtained.

This technique identifies the location that contains the defect by linearly searching each revision sequentially starting from a new location. If, for example, the results are only "○" and "×" and a transition from "○" to "×" occurs only once, then efficient searching can be done by using a binary search. However, if another output is included as a prerequisite or if multiple transitions from "○" to "×" or vice versa occur for some reason, then a binary search cannot be used here.

3.4. Display technique

To display the delta with the revision that was detected by the testing technique [this is assumed here to be between revision C and revision E ($C < E \leq L$)] in the latest revision L, first the software and source program relationship information is used to check which revisions of which source program correspond to the delta that is highlighted. The highlight is displayed as follows according to whether code was deleted or inserted between the two revisions.

[Deletion] Nothing appears in the source program of the current revision, and the pre-modified source program cannot be highlighted (Fig. 3). As a result, this situation is highlighted in the latest revision by a mark indicating the location and contents of the deleted source code (dotted line in V_L in Fig. 3).

[Insertion] The part that appears in the source program of the current revision is highlighted (see Fig. 4). Although some modifications are also made after the insertion, the part that is the fatal cause of the defect is considered to have not been changed.

3.5. Reflection technique

If a new defect is detected by the localization test, that defect is first eliminated for the latest revision L (hereafter,

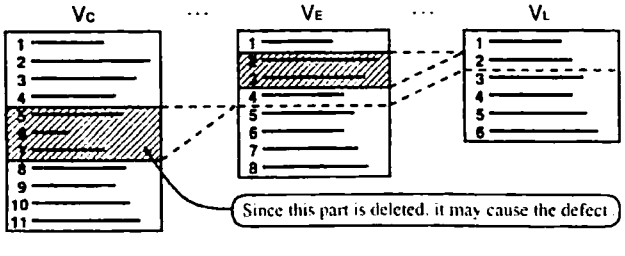

Fig. 3. Code deleted between revisions.

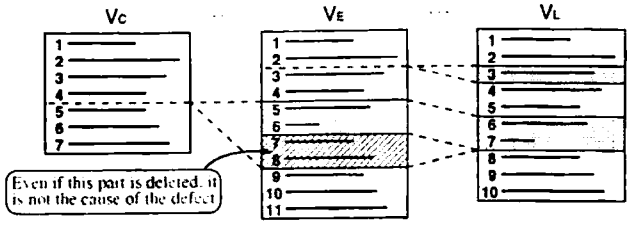

Fig. 4. Code inserted between revisions.

the delta in which this is done will be referred to as the correction ΔL). Next, to continue the localization tests, ΔL is reflected in past revisions. First, a patch program is used to apply ΔL to past revisions, and the new software that should be created is compiled. If compilation is successful, the result is used as a new program to be tested. If the ΔL patch fails or if the ΔL patch succeeds and compilation fails, that revision is excluded from subsequent regression testing.

4. Evaluation Experiments

This section describes DMET evaluation experiments using the DSUS debugging support system that is based on DMET.

4.1. Trial system DSUS

To construct a debugging support system based on DMET, we paid careful attention to the following points when creating the system.

- Independence from a programming language

Generally, constructing a system that depends on a programming language enables more detailed debugging support to be provided. However, since many programming languages are currently used for program development and since DMET itself is a technique that is not dependent on a language, we decided that DSUS should not have functions that are dependent on a specific programming language.

- Support for both testing and error correction

DSUS provides an environment for correcting errors that were detected by tests, not just for automatically executing tests. As a result, development tasks can be completed within DSUS.

- Automatic execution of tools

Sequentially executing each of the tasks that were described for DMET beginning with test execution can impose an even greater load on the developer. With DSUS, we aim to reduce the load incurred by introducing DMET by automatically performing the DMET procedure and the tests that accompany it.

DSUS consists of $DSUS_{main}$, RCS [16], DejaGnu [14], and a GUI for the user (Fig. 5). The GUI is used for editing source code and displaying deltas that are thought to contain errors.

RCS is a revision management system that is used in many development environments. We created DSUS to operate as an interface for user operations in RCS so that mistaken or dangerous operations would not be performed. We also reduced the size of the deltas that are registered in RCS when a developer edits source code by registering the edited contents in RCS at fixed intervals. DejaGnu, which is a test execution framework that is being developed according to open source principles, is used in DSUS for executing regression testing.

$DSUS_{main}$ is the part that forms the core of the entire DSUS system. GUI management, RCS operations, DejaGnu execution environment configuration, and test management are performed by this part. The DSUS GUI (Fig. 6) mostly consists of an editor window (left part of the screen) and status windows (right part of the screen). The developer can edit the source code of any revision by selecting it from the status window. The developer can also specify actions such as executing a test by using the buttons at the top of the screen. If an error is detected as a result of a test, the revision in which the error is considered to be contained is highlighted in a color.

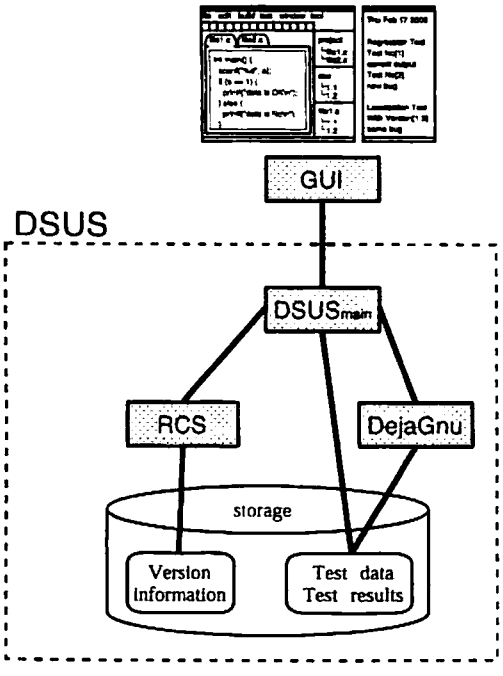

Fig. 5. DSUS configuration.

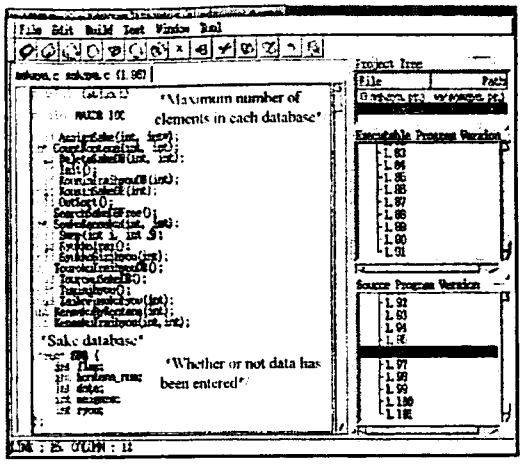

Fig. 6. Screenshot of DSUS GUI.

DSUS has been implemented using the C language and contains approximately 20,000 lines of code. Also, GTK+ was used for the GUI implementation.

4.2. Overview of the experiment

To show the practicality of the proposed technique, we used DSUS to examine the difference in the amount of time that was required when debugging tasks were performed according to DMET and when they were performed according to the conventional technique.

The test subjects in this experiment were 10 college seniors and graduate students who all had a certain amount of C programming experience. The subjects were divided into two groups (G_1 and G_2). G_1 performed debugging tasks by using DSUS, and G_2 performed them by using DSUS which had been configured so that only specific revision functions based on DMET could not be used. The software problem that was used as the debugging target was the sake dealer problem [8].

The experiment was divided into steps 1 and 2. In step 1, the debugging-target software that is to be used in step 2 is developed and collected. This experiment performs debugging by using the software that was collected.

4.3. Step 1

Software modules that properly implemented the parts other than the "mark empty container" part of the sake dealer problem have already been prepared, and a function for marking empty containers is being added to them. The relevant software modules, which have been written in the C language, contain approximately 500 lines of code. Two of these software modules (hereafter, referred to as software A and software B) in which defects occurred in other

Table 1. Development history of software used in this experiment

Software name	Number of revisions	Number of times tests were performed
A	28	25
B	91	10

functions due to this extension are the software modules that are to be debugged in step 2. Table 1 shows a list of the software modules that were collected and their development history.

For example, software A has a total of 28 revisions where the software that was provided first was the first version. Localization tests were executed 25 times according to DMET for software A, and as a result, it was apparent that the deltas between revisions that were detected by DMET contained defects.

4.4. Step 2

Groups G_1 and G_2 were each given three software modules that contain defects, which were obtained in step 1, and asked to use DSUS to perform debugging tasks. Test data that had been prepared in advance was used as the test data used for debugging, and debugging ended at the time that the test subject verified that the processing could be performed correctly for the assigned test data. The time taken by each test subject from when debugging started until it ended was totaled. Tables 2 and 3 show the debugging times (in minutes) obtained by this experiment for the test subjects (T1 to T10).

4.5. Discussion

When we used a Welch test with a significance level of 5% for the debugging time totals for software A and B, we verified that a significant difference was seen between G_1 and G_2. It is apparent from this experiment that using DMET can shorten the debugging time.

Table 2. Debugging times for G_1 (when DMET was used)

Subject	Average (Minutes)
T1	75
T2	62
T3	65
T4	79
T5	54
All of G_1	67.0

Table 3. Debugging times for G_2 (when DMET was not used)

Subject	Average (Minutes)
T6	237
T7	107
T8	237
T9	69
T10	165
All of G_2	163.0

5. Related Research

This section considers research related to techniques that had been used previously for identifying errors that cause defects and describes problems involved in those existing techniques.

5.1. Research of Ness and Ngo

Ness and Ngo used a technique called *regression containment* for compiler development at Cray Research [12]. The technique of Ness and Ngo first performs regression testing automatically. If regression testing fails, then using the fact that the base revision operates correctly, testing is repeatedly performed while executing the corrections that were obtained from configuration management in the order that the corrections were actually applied. At the stage when the test failed, the correction that was applied at that time is identified as the error that caused a bug.

However, although the technique of Ness and Ngo operates well in certain situations, it does not operate well when the test first fails because multiple deltas are applied instead of just a single delta or when an inconsistency occurs so that the program cannot be compiled when a change is applied.

5.2. Research of Zeller

Zeller proposed a technique [17] that can also deal with defects due to multiple errors or inconsistencies due to the application of changes, which are problems with the technique of Ness and Ngo. Zeller's technique considers a correction as one element of a set without taking into consideration the order in which corrections are applied. Therefore, if the number of corrections that were made is n, then the number of sets that are considered is 2^n. Zeller uses an algorithm that finds the set having the minimum

number of elements for which a defect occurs from among the sets that are considered. By using sets of corrections, this algorithm succeeds in identifying the causes of defects that occur due to multiple errors. The algorithm, which can also deal with inconsistencies due to the application of changes, can detect errors that could not be found by using the technique of Ness and Ngo.

However, although making the algorithm more complex increases the precision of identifying the delta that caused the bug, the number of sets that must be tested increases exponentially relative to the number of corrections. Therefore, the number of times tests are performed increases, and an enormous amount of time is required for testing.

5.3. Problems

The techniques of both Ness and Ngo and Zeller consider the source program of the base revision as a baseline, and corrections must be applied to that source code each time and the corrected source code must be compiled to perform testing. Also, since compilation must be performed while accurately managing which corrections were made to which source programs, the task is complex even if only simple compilation is being performed.

To find the cause of a defect, you must create test cases and perform testing yourself, and you must study about or undergo training for the test tool that is being used for each technique. In addition, since these techniques emphasize only testing tasks for identifying the causes of defects and do not take into consideration debugging tasks, it is difficult to apply them to actual maintenance tasks.

DMET enables compilation tasks to be easily performed by using a support environment such as DSUS since information indicating which source programs are to be compiled at which times are managed within the framework of DMET itself. Also, since test cases are first created in DMET and used for debugging, you need not create test cases corresponding to defects yourself, and the time required for creating test cases is reduced. Since DMET is intended not only for identifying defects but also for all tasks that are included up to debugging, it can easily be applied even in actual maintenance tasks, which we showed by the current experiments.

6. Conclusions

In the current research, we proposed the DMET debugging technique, which is intended for maintenance tasks. We also showed the effectiveness of DMET through experiments using the DSUS trial system. Using the proposed technique can be expected to enable debugging tasks to be performed more easily during actual maintenance.

Some future topics of research are as follows. First, we plan to improve the delta detection algorithm so that deltas that contain defects are detected more accurately. When the deltas that are detected by DMET become relatively large, even if the results that are obtained by using DMET are used, they will not contribute to a reduction of the working time. For such cases, we plan to investigate methods of having DSUS judge this situation in advance and alert the developer instead of just simply exhibiting the results.

REFERENCES

1. Babich WA. Software configuration management. Addison–Wesley; 1986.
2. CASE 1988-89, Sentry Market Research, Westborough, MA, p 13–14, 1989.
3. McClure C (author). Best CASE Research Group (translators). The three Rs of software automation. Kyoritsu Shuppan; 1993.
4. Cashman PM, Holt AW. A communication-oriented approach to structuring the software maintenance environment. Software Engineering Notes 1980:5:4–17.
5. Conradi R, Westfechtel B. Version models for software configuration management. ACM Computing Surveys 1998;30:232–280.
6. Dogsa T, Rozman I. CAMOTE—Computer aided module testing and design environment. Proc Conference on Software Maintenance-88, p 404–408, Phoenix, AZ.
7. Hetzel W. The complete guide to software testing. QED Information Sciences; 1984.
8. Kudo H, Sugiyama Y, Fujii M, Torii K. Quantifying a design process based on experiments. Proc 21st International Conference on System Sciences, p 285–292, Hawaii, 1988.
9. IEEE. Test methods for measuring conformance to POSIX. ANSI/IEEE Standard 1003.3-1991, ISO/IEC Standard 13210-1994.
10. Leung HKN, White L. Insights into regression testing. Proc Conference on Software Maintenance-89, p 60–69, Miami, FL.
11. Lientz B, Swanson E. Software maintenance management: A study of the maintenance of computer application software in 487 data processing organizations. Addison–Wesley; 1980. p 151–157.
12. Ness B, Ngo V. Regression containment through source code isolation. Proc 21st Annual International Computer & Applications Conference (COMPSAC '97), p 616–621, IEEE Computer Society Press, 1997.

13. Raither B, Osterweil I. TRICS: A testing tool for C. Proc First European Software Engineering Conference, p 254–262, Strasbourg, France, 1987.
14. Savoye R. Test DejaGnu testing framework for DejaGnu version 1.3. Free Software Foundation; 1996.
15. Swanson E. The dimensions of maintenance. Second International Conference on Software Engineering Proceedings. San Francisco, p 492–497, 1976.
16. Tichy WF. RCS—A system for version control. Software–Practice and Experience 1985;15:637–654.
17. Zeller A. Yesterday, my program worked. Today, it does not. Why? Proc 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE '99), p 253–267, Toulouse, France.

AUTHORS (from left to right)

Makoto Matsushita graduated in 1993 with a specialty in Information Science from the Department of Engineering Science at Osaka University, took a leave from the second half of his doctoral course in 1998, and became an assistant in the Department of Engineering Science. He has been an assistant in the Graduate School of Engineering Science since 2002 and an associate professor since 2005. He is engaged in research on software development environments, software development processes, and open source development. He holds a D.Eng. degree.

Masayoshi Teraguti graduated in 1993 with a specialty in Information Science from the Department of Engineering Science at Osaka University, completed the first half of his doctoral course in 2000, and joined the Tokyo Research Laboratory, IBM Japan, Ltd. While at Osaka University, he was engaged in research on debugging support environments. He holds an M.E. degree.

Katsuro Inoue (member) graduated in 1979 with a specialty in Information Science from the Department of Engineering Science at Osaka University, completed his doctoral course in 1984, and became an assistant in the Information Division of the Department of Engineering Science there. From 1984 to 1986, he was an assistant professor in the Computer Science Course at the University of Hawaii Manoa Campus. In 1988, he became a lecturer in the Information Science Division of the Department of Engineering Science at Osaka University, an assistant professor in 1991, and a professor in 1995. Since 2002, he has been a professor in the Graduate School of Engineering Science. He holds a D.Eng. degree. He is engaged in research on software engineering.