Effective Testing and Debugging Methods and Its Supporting System with Program Deltas

Makoto Matsushita[†]
matusita@ics.es.osaka-u.ac.jp

Masayoshi Teraguchi[‡]
teraguti@jp.ibm.com

Katsuro Inoue[†*]
inoue@ics.es.osaka-u.ac.jp

[†]Dept. of Information and Computer Sciences
Faculty of Engineering Science
Osaka University
1-3 Machikaneyama, Toyonaka
Osaka 560 JAPAN

[‡]Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa, 242-8502 Japan

[*] Information Technology Center
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma
Nara 630-01 JAPAN

Abstract

In the maintenance phase of software development, it should be checked that all features are performed correctly after some changes are applied to existing software. However, it is not easy to debug the software when a defect is found to the features which is not changed during the changes, although using a regression test. Existing approaches employs the program deltas to specify defects; they have hard limitation of enacting them, don't support an actual debugging activities. Moreover, its system is hard to introduce an actual environment.

In this paper, we propose a new debugging method DMET to solve such problems. DMET supports debugging activities when a defect is found by regression tests through detection, indication, and reflection procedures. We also implement a debugging supporting system DSUS based on DMET. DSUS executes DMET procedures automatically, and easy to configure for existing environment.

Through the experimentation of DMET and DSUS, we confirm that DMET/DSUS reduce debugging time of software significantly. As a result, DMET/DSUS help evolving the software for the aspect of software maintenance.

1 Introduction

In the maintenance phase of software development, many changes are applied to existing software. However, Hetzel shows that it is 50 to 80 percent of probability of adding some defects when some changes are applied to the software[4]; testing should be needed to not only changed features of software but also the unchanged features. Since recent software development environment tends to be more complicated, software maintenancability is important issue for maintaining software effectively and future evolution of the software.

Regression test[2, 8, 9, 13] is commonly used in order to check that the features of software is implemented as specification. Actual software development and maintenance activity employ version control system that recognizes, organizes, and manages software products (source codes and accompanying documents), in order to raise the maintainability of software. Under these development environments, regression test checks the latest version of software. If a defect is found during the regression test, the error that causes a defect will be specified and it will be corrected. However, when the defect that is not changed by the code changes causes a defect, it is not easy to specify the error used as the cause.

There is a research on debugging approach that uses the deltas of code changes between the base version (a version before the code changes are applied) and a latest version[7, 14]. However, these approaches does not detect the delta correctly, and require some environmental setup manually to enact a test tool[5] which requires the inputs of test data and the outputs of test data which is compared with the outputs of program. Since these approaches do not support a code modifi-

cation activity that will fix the error found by a test tool, it is difficult to apply these approaches to actual software maintenance activities.

In this paper, we propose a new debugging method DMET (Debugging METhod) to solve these problems, and a debugging supporting system DSUS (Debugging SUpport System) based on DMET approach. DMET supports debugging activities when a defect is found by regression tests, and finds the defects to the features that are not changed during the code changes. DSUS assumes the existence of a version management system and a regression test tool, and provides auto configuration/execution of test tools and auto detection a delta of code that will contain the defects. Since DSUS also performs as a wrapper of version management system, engineers can concentrate on the error correction work.

We also evaluate our DMET/DSUS with a experimentation. The experimentation shows that total debugging time is reduced by DMET/DSUS when defect detection is succeeded. We also found through experiments that DSUS supports a series of software debugging activities. As a result, DMET/DSUS help maintaining and evolving existing software.

In section 2, we explain related researches about software debugging with a delta of code. In section 3, we propose our DMET and show how to detect code delta that may contain a defect. In section 4, DSUS debugging supporting system are illustrated. We evaluate DMET/DSUS in section 5, and in section 6 we present our conclusion and further research topics.

2 Debugging with Code Deltas

When some modifications are applied to software, the features that were implemented before may be broken. Although the reason is obvious (some errors are incorporated with this modification), it is difficult to detect what is an error and fix it. In this section, we consider about a method of debugging using a code delta, a part for the difference between the base version and the current version.

Generally, debugging activities is usually composed of testing and error collection. In the debugging method using code delta, it is assumed that there are no errors in base version and current version has some errors. In the testing phase, a code delta is detected which contains which causes errors. The code delta is used in the error collection phase.

In this section, we explain previous approaches about debugging with code deltas about preconditions, testing procedure, error collection procedure, and its problems.

2.1 Regression Containment

Ness and Ngo at Gray Research have proposed the method called Regression Containment[7]. In Regression Containment, it is mentioned that a software configuration system (version controlling system)[1, 12], tool-chain for testing are used. This method assumes that the version that carries out normal operation and the version that contains some errors do not appear by turns. Under such premise conditions, Regression Containment identifies the causes of errors by testing automatically.

In testing phase, base version of source code are extracted from version management system first, then some deltas to the codes and compiled to executables. Next, compiled executables are performed with test data and testing tools compares the results. If it performs normally, another delta is applied repeatedly. The code modification fragments (deltas) can be detected if there is something wrong is found in the results of test execution; it enables to specify the cause of this defect without modifying existing source code. In error collection phase, detected deltas are deleted from (current version of) source code; it aims at not delaying software development by this defects.

However, before testing, it is needed to setup some environments for running testing tools manually; tools input (source codes, test data, etc) and correct outputs to compare should be clearly specified. Moreover, this method requires applying all deltas to a base version for each test execution, and does not analyze the difference of results. This method simply removes the deltas, which introduces some defects; actually error *correction* is not performed.

2.2 Delta Debugging

In a particular situation, Regression Containment performs effectively for debugging. However, when applying two or more deltas cause the defect, or executables cause core dump and no output are collected, it does not operate well. Zeller has proposed the debugging methods called Delta Debugging[14], which can correspond also to these situations that these problems provide. Although there is no requirement to use software configuration system, it assumes some the existence of test tools[5] that compares the test results.

In order not to take into consideration use of software configuration system, Delta Debugging employs a function which acts like "diff" command in UNIX environment; the code delta between base version and a version which has some defects are extracted by this function. Obtained delta is examined and classified

as a set of "code insertion" and "code deletion". Using these styles of deltas, it can correspond also to the error that cannot be finding out by Regression Containment.

Delta Debugging uses the algorithm to find out the smallest number of set of code composition that can be applied in testing phase. In this phase, this method considers yet another type of error output which corresponds to compilation error or program core dump.

Since an order of code deltas in actual software development is not investigated, a set of program deltas should be examined by combining each various changes which is classified before which can be applied to base version. Therefore, if the number of changes classified finely is n, the number of candidates which will be a set of program deltas detected by this method is 2^n; combination explosion potentially exists. Since the application order of these changes is not take into consideration, error detection may fail in some cases. Manual setup of testing tools is also required, and there is no error correction procedure in this method; it is difficult to apply this method to actual software maintenance work.

3 DMET

In this section, we propose a new debugging method named DMET[11] that solves the problem in previous approaches described in section 2. DMET uses source code deltas to specify defects. In order to support series of debugging procedures, DMET reduces the premise conditions in previous approaches in testing phase, shows detected deltas with current source code, and reflects the modification which fixes the defects to other versions for further debugging.

DMET supports debugging activities, which detecting and fixing defects in unchanged features of software that works with base version of software. Limiting the target software to some extent, DMET supports autoconfiguration of testing tools that is normally by manually. Moreover, not only source codes but also compiled objects are managed by existing version management system, total testing time can be reduced. DMET supports the debugging activities from testing to error correction; it is applicable approach to actual software maintenance activities.

3.1 Premise conditions

There is some premise conditions that are needed for software, software maintenance activity, and software engineers to apply DMET to debugging activity.

- Version controlling system is used.
  We use version controlling system to record the changes of not only source code but also program executables, to reduce the testing time. In case of registering an executable to version controlling system, the relationship between source code and compiled object are also recorded. Figure 1 shows an example of this relationship. Src1 and Src2 are source codes, and Exe is an executable that comes from Src1 and Src2. Vi (i = 1,2,..,5) expresses each version. When registering Exe version V4, the related information that Exe V4 contains of V5 of Src1 and V3 of Src2. These relationships are used later.

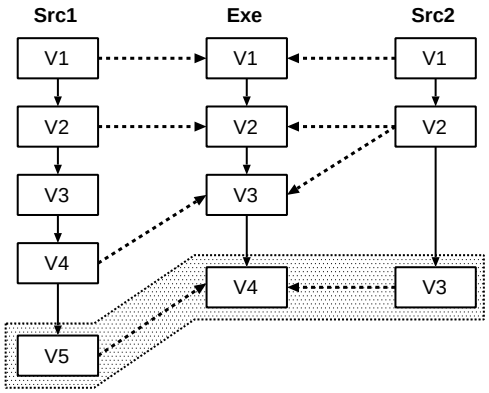

Figure 1. Relationship between source codes and objects

- Base version is existed.
  In a typical software life cycle, in order to check ordinal operations of the functions required in specifications, various test cases are taken into consideration. These test cases are used to check in the testing phase, and if the software correctly passes to all the test case, the software is released and it will shift to the software maintenance phase. Therefore it is thought that the base version that is maybe the first release of the software exists. It is guaranteed that there is two versions that the deltas between versions which contains defects.

- Test results using each version cannot be predicted.
  It cannot predict beforehand how the version which works correctly, and the version which has some defects. Therefore, it is also considered that a correct version and a defect version appear by turns in the sequence of versions. Moreover, the test output result of versions which produce defects always are not the same result of the latest version.

Under these premise condition, DMET is consisted of three phases, named "detection", "indication", and "reflection".

3.2 Detection

In detection phase, auto-detection of source code deltas is established using auto-execution test tools. This phase consists of regression test and localization test. After adding some modification to the base version of software, regression test with test data are performed. If something wrong with regression test, localization test are also performed with a test case that is picked up by regression test. Localization test detects the versions, which causes error in this regression test.

In this section, we use following notations: V_j shows the software which version is j, and V_B/V_L shows the base version and latest version of software respectively. Regression test has N test, and each test is shown as $T_i(1 \leq i \leq N)$. I_i is input data of test T_i, and $O_{j,i}$ is an output of version j. Each regression test is shown as $RegT(i,\ N)$ where T_i is used in regression test. If $O_{L,i}$ is a wrong output, each localization test to V_j is shown as $LclT(i,\ j,\ k)$ where output $O_{k,i}$ $(j \leq k)$ is also wrong.

3.2.1 Regression test

The regression test is automatically done whenever certain changes are applied to software. In $T(i)(1 \leq i \leq N)$, $O_{B,i}$ is used as correct output. If something wrong with T_i, localization test $LctT(i, L-1, L)$ is performed. Figure 2 shows an algorithm of $RegT(1, N)$.

3.2.2 Localization test

Localization test checks whether there is a version which output is the same of $O_{B,i}$. This checking starts from V_L to V_B by linear search, then finds out two versions. DMET assumes there is a base version which output $O_{B,i}$ is correct, this checking should be stopped at least all versions are checked. That is, there is at least one version, which produces correct output between V_L and V_B. Figure 3 shows an algorithm of localization test.

Under localization tests $LctT(i, j, k)$, $O_{j,i}$ are produced and examined, then the results are categorized as four types.

1. Same wrong result of $O_{k,i}$ $(\times)$
   We assume that same wrong output comes from the same defects. Therefore, changes between V_j and V_k does not affects the defects. In Figure 4 (a), there is no errors affected the latest version's output between V_6/V_9 and V_9/V_{10}.

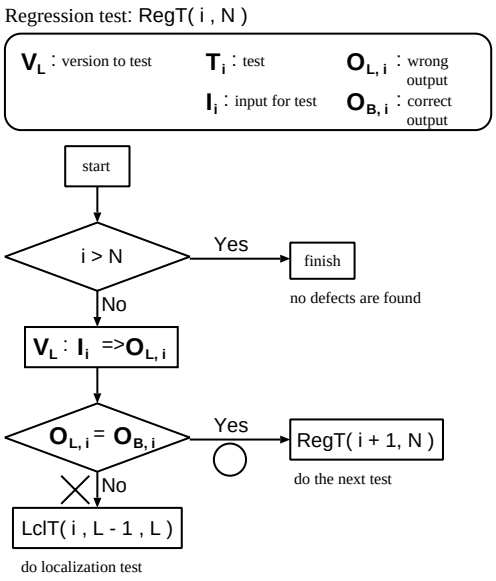

Figure 2. Regression test

2. Same correct result of $O_{B,i}$ $(\bigcirc)$
   If the output of V_j is correct, the localization test is finished and specifies the code deltas between V_j and V_k has a defect. In Figure 4 (a), deltas between V_5 and V_6 has the defects which causes an error at V_{10}.

3. No outputs $(\triangle)$
   There is nothing special that the test execution cannot be done because of the segmentation faults of the program. Under these circumstances, it is difficult to check the output of test results. Therefore, DMET ignores this version to detect a defect. However, in the special case, if this is found in the regression test (Figure 4 (b)), localization test is also performed with this test data. In this case, detected delta should be between version V_c (where B ¡ c ¡ L) and V_L

4. Other outputs (—)
   It is assumes that there is yet another defect is found. However, it is also ignored if the defect is not what to be detected in this localization test. In Figure 4, V_7 of (a) and V_9 of (b) are not considered in localization test.

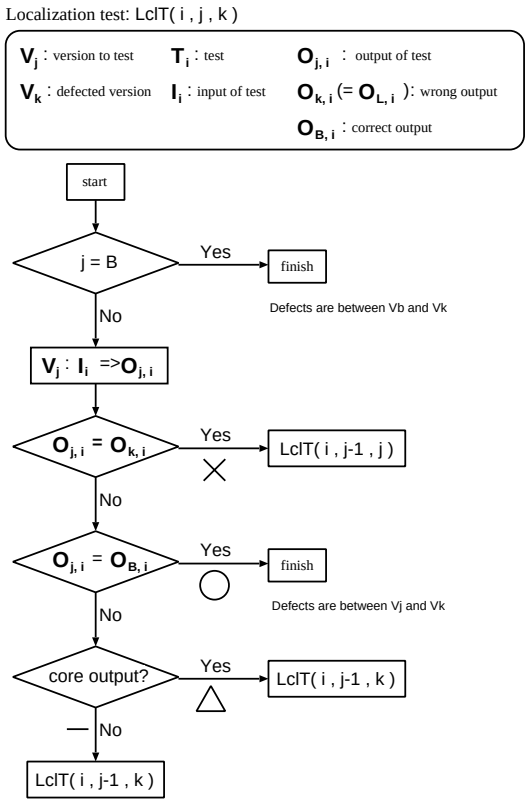

Figure 3. Localization test

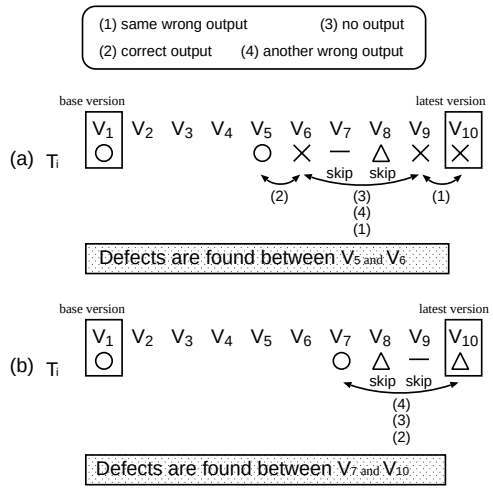

Figure 4. Sample results of localization test

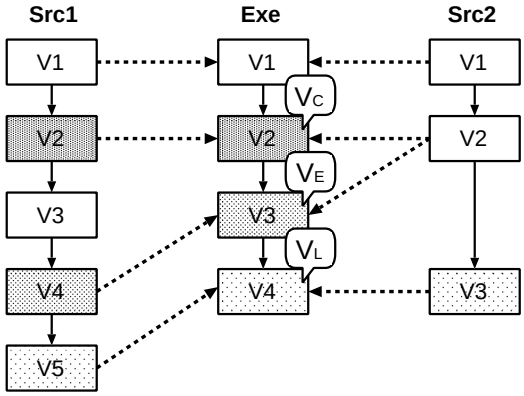

Figure 5. Example of indication

3.3 Indication

In indication phase, code deltas that are specified in detection phase are mapped to the latest version of codes. At first, using the relationship between source codes and executable, source code versions associated with executable version are detected. For example in Figure 5, Exe version V_C (decided as ◯ in previous phase) consists of version V2 of Src1 and Src2, and Exe version V_E (decided as × in previous phase) consists of Src1 version V4 and Src2 version V2. In this example, code delta between V2 and V4 of Src1 should be mapped to V5 (the latest version of Src1).

In general, there are some modification between V_E and the latest version V_L, the delta cannot be applied to easily. In the indication phase, modifications in the delta are classified as "code insertion" and "code deletion", and then some correction is performed to each modification.

Inserted codes by detected delta may be included in the latest version, so indication phase analyzes the further code modification and specifies which line in a delta is the line in the latest version. However, deleted codes are not shown in the latest version; the phase checks the *deleted point* of source code, tracks the further modification, and specifies the line in the latest version if the deleted code existed.

3.4 Reflection

In reflection phase, the modification that should fix the detected defects is applied to the every former version until the version that is not affected with the defect.

This phase should be needed for further cycle of debugging. It is assumed that there are two or more defects in the target software; if you fix *one* defect in the latest version, other defects existed from *all* versions. Further detection phase does work effectively when removing the detected defects to *all* versions.

This phase consists of two procedures, apply a fix

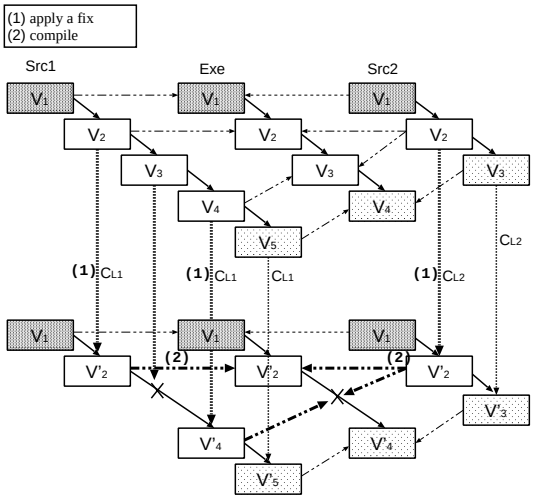

Figure 6. Example of reflection phase

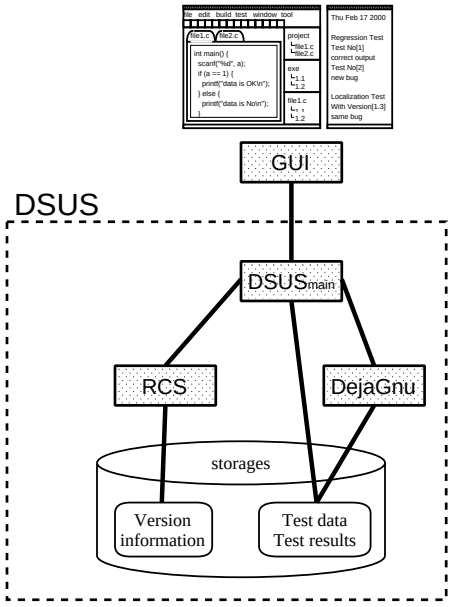

Figure 7. DSUS structure

and compile again. If either of this procedure fails (does not apply a fix, or cannot compile the code), such version is ignored in the further phases.

Figure 6 shows the example of reflection phase. In this example, there is a defect between V_1 and V_2. At first, DMET applies a fix to V_2, V_3, and V_4 of Src1 and V_2 of Src2. Unfortunately, V_3 is failed to apply, so this version is ignored. Then, compilation is done to all versions. However, V_3' cannot be produced. Therefore, version V_1, V_2', and V_4' are used in next localization test

4 DSUS

In this section, we propose debug supporting prototype system DSUS based on DMET proposed in section 3.

4.1 Features

Before designing DSUS, we specify DSUS features to support software maintenance activities as follows:

- Independent from programming language
  It is possible that more detailed support can be done with language dependent system. However, there are many languages used in actual software development environment, so the system should be language independent to support lots of software development environment.

- Supports both testing and error correction
  The debugging system supports not only test execution, but also fixing defects. It should be enough to use DSUS while debugging for engineers.

- Executes tools automatically
  Routing work such as doing a regression test is tired for engineers. All activities in DMET should be automatically done by DSUS.

4.2 System overview

DSUS system consists of DSUS_{main}, RCS[12], DejaGnu[10], and graphical interface (Figure 7).

DSUS employs two external systems, RCS and DejaGnu. RCS is a version controlling system based on check-in/check-out model[3]. DSUS wraps these dangerous operations with GUI; miss-operation of RCS tools can be eliminated, and periodical check-in can be performed by DSUS; code delta should be kept to small. DejaGnu is a testing framework and DSUS uses this as regression testing tool. DejaGnu setup is automatically done with configuration templates by DSUS.

DSUS_{main} is the main engine for DSUS. GUI management, RCS operation, DejaGnu environment setup and test management are performed by this component. GUI component (Figure 8) is composed of editor window and status window. User can edit arbitrarily version of code, and command DSUS to execute the procedures defined in DMET.

Most of system is implemented in C, and GTK+[15] is used in GUI component. Configuration templates are written in Tcl to adapt the DejaGnu requirements. System scale is about 20000+ LOCs.

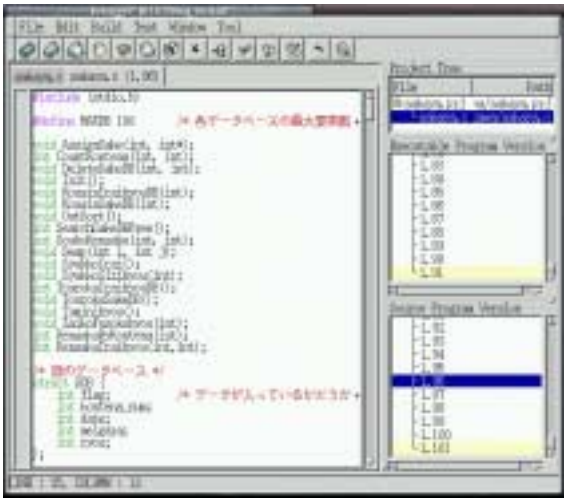

Figure 8. GUI component

Table 1. Software used in the experiments

	versions	tests	detected?
A	28	25	◯
B	91	10	◯
C	72	70	×

Table 2. Debugging time of G_1 (with DMET)

	A&B	C
testee1	75	25
testee2	62	60
testee3	65	57
testee4	79	34
testee5	54	21
average	67.0	39.4

5 Evaluation

In this section, we evaluate DMET through the experimentation of DSUS. The object of this evaluation is to confirm DMET is effective for debugging activities that fix defects in unchanged features.

5.1 Experiments

There are 10 testees (undergraduate and graduate students in a university, have some skill of C programming), and it divides into two group G_1 and G_2. Group G_1 uses whole DSUS features including automated DMET procedures, and group G_2 uses restricted DSUS system which does only regression test, editing code, and check-in/check-out arbitrarily version of source code.

In this experiment, we should prepare software which contains only one defects that comes from the inventory control program of wine shop[6]. To collect them, we have done a preliminary experimentation. At first, we prepare a specification and a program which comes from the specification. Next, we add one feature (checking empty wine container) to the specification. All testees do develop a software based on the one we provide, with a new specification. We have gathered software, and check if a defect exists in the features which is in the original specification. Note that the program we have prepared with original specification has no defect; after the modification by testee, the defect is appeared because testee forget to reflect the changes of the specification to the existing part of software.

Table 1 shows the summary of software we have collected. There are three software (A, B, and C), and software A has 28 versions (including base version), and DMET does 25 times of localization test, and the detected delta contains the defects. Note that the delta of software C is large (70 tests executed although there is 72 versions), and detected delta does not contain the defects.

Both G_1 and G_2 members do debugging with these software, and records the total time of testing and error correction of each testee. We assume that the elapsed time of G_1 is shorter than G_1.

5.2 Results and investigations

Table 2 and 3 show the elapsed time results of G_1 and G_2.

A, B, and C show the software described before, and each line shows the result of each testee. These tables summarize the time of A and B; the detected delta

Table 3. Debugging time of G_2 (without DMET)

	A&B	C
testee6	237	20
testee7	107	5
testee8	237	15
testee9	69	5
testee10	165	8
average	163.0	10.6

contains the defect of software.

According to the Welch's test (using 5% differences), there is a significant difference of the total time of A and B between both groups. However, the time of software C and the total time of A, B, and C does not introduce significant difference between groups. Therefore, DMET/DSUS provide effective features for debugging if the detection is successfully established. Moreover through this experience, we confirm that DMET/DSUS support whole procedures of software debugging.

6 Conclusion

In this paper, we propose DMET debugging method to detect where is a defect in source code. DMET consists of three phases,including detecting a delta through versions, indicating the delta to the latest version, and reflecting the changes that fix the defect. We also implement prototype system DSUS based on DMET. Using DSUS, we evaluate DMET method. As a result, we find out that DMET/DSUS support whole debugging activity, and help maintaining and evolving the software.

As further works, improvements of DMET detection phase to reduce the number of localization tests and to detect correct code delta are required. Moreover, we are going to expand DMET method to apply not only debugging for unchanged features in the maintenance phase, but also for newly created features in the developing phase.

References

[1] Babich, W. A., "Software Configuration Management," Addison-Wesley, Reading, Massachusetts, 1986.

[2] Dogsa, T. and Rozman, I., "CAMOTE - Computer Aided Module Testing and Design Environment," In Proceedings of the Conference on Software Maintenance - 88, pp.404-408, Phoenix, Ariz., 1988.

[3] Feiler P. H., "Configuration Management Models in Commercial Environments," Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, March 1991.

[4] Hetzel, W., "The Complete Guide to Software Testing," QED Informaion Sciences, Wellsley, Mass., 1984.

[5] IEEE., "Test Methods for Mesureing Conformance to POSIX," ANSI/IEEE Standard 1003.3-1991, ISO/IEC Standard 13210-1994.

[6] Kudo, H., Sugiyama, Y., Fujii, M. and Torii, K., "Quantifying a design process based on experiments". In Proceedings of the 21th International Conference on System Sciences, Hawaii, pages 285–292, 1988.

[7] Ness, B. and Ngo, V., "Regression containment through source code isolation," In Proceedings of the 21st Annual Internatial Computer & Applications Conference (COMPSAC '97), IEEE Computer Society Press, pp.616-621, 1997.

[8] Ostrand, T. and Weyuker, E., "Using Data Flow Analysis for Regression Testing," 6th Annual Pacific Northwest Software Quality Conference, pp.1-4, Portland, Oreg., 1988.

[9] Raither, B. and Osterweil, I., "TRICS : a Testing Tool for C," In Proceedings of the First European Software Engineering Conference, pp.254-262, Strasbourg, France, 1987.

[10] Savoye, R., "Test DejaGnu Testing Framework for DejaGnu Version 1.3," Free Software Foundation. Inc., January, 1996.

[11] Teraguchi, M., Matsushita, M., and Inoue, K., "A Proposal of Debugging Method with Changes between Versions," Technical Report of IEICE, SS-99-52, pp.17-24, Naha, Japan, 2000.

[12] Tichy, W. F., "RCS - A System for Version Control," Software-Practice and Experience, Vol.15, No.7, pp.637-654, 1985.

[13] Yau, S. and Kishimoto, Z., "A Method for Revalidating Modified Programs in the Maintenance Phase," In Proceedings of the 11th Annual Internatial Computer & Applications Conference (COMPSAC '87), pp.272-277, Tokyo, Japan, 1987.

[14] Zeller, A., "Yesterday, my program worked. Today, it does not. Why?," Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE '99), Toulouse, France, September 1999.

[15] GTK+ : http://www.gtk.org/