Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 381389, 6 pages
http://dx.doi.org/10.1155/2013/381389

*Research Article*

# Metamorphic Testing Integer Overflow Faults of Mission Critical Program: A Case Study

**Zhanwei Hui,[1] Song Huang,[1,2] Zhengping Ren,[1,2] and Yi Yao[1,2]**

[1] *Command Information Institute, PLA University of Science and Technology, Nanjing, Jiangsu Province 210007, China*
[2] *PLA Military Training Software Testing and Evaluation Centre, Nanjing, Jiangsu Province 210007, China*

Correspondence should be addressed to Zhanwei Hui; hzw_1983821@163.com

Received 14 November 2012; Accepted 8 January 2013

Academic Editor: Tsung-Chih Lin



For mission critical programs, integer overflow is one of the most dangerous faults. Different testing methods provide several effective ways to detect the defect. However, it is hard to validate the testing outputs, because the oracle of testing is not always available or too expensive to get, unless the program throws an exception obviously. In the present study, the authors conduct a case study, where the authors apply a metamorphic testing (MT) method to detect the integer overflow defect and alleviate the oracle problem in testing critical program of Traffic Collision Avoidance System (TCAS). Experimental results show that, in revealing typical integer mutations, compared with traditional safety property testing method, MT with a novel symbolic metamorphic relation is more effective than the traditional method in some cases.

## 1. Introduction

Integer overflow is one of the most important dangerous faults, which is crucial for mission critical programs which are usually related to critical functions or services. And more important, it is too late to deal with integer overflow until the program under test halts or collapses, which is catastrophic [1]. Detection of such faults or bugs presents a challenge because conventional software testing processes do not always apply: in particular, it is difficult to detect subtle errors, faults, defects, or anomalies, which could lead to recessive failure in many applications in these domains because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "nontestable programs" [2]. Many of these applications fall into a category of software that Weyuker describes as "programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known" [2].

Even though the ideal oracle is hard to get sometimes, software testing can still be carried out. Baresi and Young [3] proposed to use necessary properties of program to test numerical programs. For example, a common method used for testing numerical programs is to check if the outputs satisfy certain properties, such as the property of $e^x \times e^{-x} = 1$ for exponential calculation programs. Program checkers [4, 5] and self-testing [6, 7] also make use of relations among outputs of object program to check its correctness by themselves. To design programs that check their work, one basic technique used is, first, determine the necessary properties of program, then check if it satisfies these properties with random inputs. The widely used properties are linear consistency and neighbor consistency [6, 7]. In software tolerance, the most relevant technique is data diversity [8], which is redescribing the input in a different format. This technique can lower the cost greatly compared with "N-version programming." Data diversity was first proposed for fault tolerance, not for fault detection. Meanwhile, properties applied by data diversity are restricted by equality relations.

For testing integer defects, which could be considered as one of the most important faults of mission critical software failures and classified into four categories [9], we first enumerate the metamorphic relations that such application would be expected to demonstrate, then for a given implementation determine whether each relation is a necessary property to reveal program correctness. If it is, then an unexpected change could exhibit the violation of the relation,

Table 1: Examples of four types of integer overflow.

| An overflow integer defect | An underflow integer defect |
|---|---|
| (1) int $i$; | (1) int $i$; |
| (2) unsigned int $j$; | (2) unsigned int $j$; |
| (3) $i$ = INT_MAX; //2147483647 | (3) $i$ = INT_MIN; //−2147483648 |
| (4) i++; | (4) $i$− −; |
| (5) printf ("$i$ = %$d$\n" , $i$); /* $i$ = −2147483648* / | (5) printf ("$i$ = %$d$\n " , $i$); /* $i$ = 2147483647* / |
| (6) $j$= UINT_MAX; //4294967295 | (6) $j$ = 0; |
| (7) $j$++; | (7) $j$− −; |
| (8) printf ("$j$ = %$u$\n " , $j$); /* $j$ = 0* / | (8) printf ("$j$ = %$u$\n", $j$); /* $j$= 4294967295* / |
| **A signedness integer defect** | **A truncation integer defect** |
| (1) int $i$ = −3; | (1) unsigned short int $u$ = 32768; |
| (2) unsigned short $u$; | (2) short int $i$; |
| (3) $u$ = $i$; | (3) $i$ = $u$; |
| (4) printf ("$u$ = %hu\n " , $u$); /* $u$ = 65533* / | (4) printf ("$i$ = %$d$\n", $i$); /* $i$ = −32768* / |
| | (5) $u$ = 65535; |
| | (6) $i$ = $u$; |
| | (7) printf ("$i$ = %$d$\n", $i$); /* $i$ = −1* / |

which indicates an integer defect. If it is not a necessary property of the software under testing and the properties would still be anticipated to hold in the application or algorithm, then they cannot be used for validation. In addition to validating the effectiveness of metamorphic testing of integer overflow, we conduct a case study, where the authors apply the method to alleviate the oracle problem in a typical mission critical system, Traffic Collision Avoidance System (TCAS), which is used as the critical application for airplane collision management.

The rest of the paper is organized as follows. Section 2 supplies background information about integer overflow defects and introduces the integer defect categories. Some basic definitions are also introduced in this section. Section 3 proposes the relationships between MT and metamorphic relation. Section 4 presents the case study. The results of our experiment demonstrate that the MT could detect integer defects effectively in the mission critical program TCAS. Mutation testing is also introduced to systematically insert integer mutants into the source code and validate the effectiveness of metamorphic testing with conventional ones in integer defects detection. Our conclusions and future works are given in Section 5.

## 2. Preliminaries

*2.1. Integer Overflow Defects.* An integer overflow is caused by the fixed width of the integer data type expression. As the data of operation statements computed is bigger or smaller than the range of the fixed width, then there is an integer overflow. And sometimes, an integer overflow defect is an integer bug in wide sense. For C language, Sercord [1] proposed that integer defects could be categorized into four types, which include overflow integer defects, underflow integer defects, signedness integer defects, and truncation integer defects. Table 1 gives the examples of different kinds of integer defects.

*2.2. Definitions.* The test oracle problem has always been restricting the development of software testing. Researchers have been exploring the solutions to the oracle problem in different applications. Then in 1998, Chen and his fellows [10] proposed the software metamorphic testing technique, which is an effective method for oracle problem. This technique verifies the correctness of the program by checking whether it satisfies some necessary properties of the program, which are called metamorphic relations. The definitions with metamorphic testing techniques are given as follows.

There are two understandings of the test oracle [2]. Some one refer to it as the expected outcome of the program under test. Some others also include the process of comparing the actual outcomes against the expected ones. For example, program $p\_\sin(x)$ is an implementation of $\sin(x)$, the test oracle in a narrow sense is all the $\sin(t)$ for $\forall t \in T$, and the test oracle in a wide sense refers to a mechanism that checks whether expression $p\_\sin(x) = \sin(x)$ is true for $\forall t \in T$. The test oracle used in this paper is referred to the one in a narrow sense without special statements.

*Definition 1* (metamorphic relation (MR) [11]). Suppose program $P$ is the implementation of function $f$ and $x_1, x_2, \ldots, x_n$ ($n > 1$) are $n$ groups of input for $f$, their corresponding outputs are $f(x_1), f(x_2), \ldots, f(x_n)$. If $x_1, x_2, \ldots, x_n$ satisfy relation $r$, it can be referred that $f(x_1), f(x_2), \ldots, f(x_n)$ satisfy relation $r_f$; that is,

$$r(x_1, x_2, \ldots, x_n) \Longrightarrow r_f(f(x_1), f(x_2), \ldots, f(x_n)). \quad (1)$$

Then $(r, r_f)$ is called a metamorphic relation of $P$.

Therefore, if $P$ is correct, then it must satisfy the following comprehension:

$$r(I_1, I_2, \ldots, I_n) \Longrightarrow r_f(P(I_1), P(I_2), \ldots, P(I_n)). \quad (2)$$

$I_1, I_2, \ldots, I_n$ are actual inputs of $P$ corresponding with $x_1, x_2, \ldots, x_n$ and $P(I_1), P(I_2), \ldots, P(I_n)$ are the outputs. People could verify the correctness of $P$ by checking whether expression (2) is satisfied while testing.

Suppose program $P$ is correct, then the following expression should be satisfied: $r(I_1, I_2, \ldots, I_n) \Rightarrow r_f(P(I_1), P(I_2), \ldots, P(I_n))$, where $I_1, I_2, \ldots, I_n$ is the input of $P$ corresponding to $x_1, x_2, \ldots, x_n$ and $P(I_1), P(I_2), \ldots, P(I_n)$ is the output. We use $x_1, x_2, \ldots, x_n$ to represent the input in this paper. So if the outputs of test cases do not satisfy the above formula, then the hypothesis is wrong and there are faults in the program. MRs are the key to judging the execution of a set of test cases and their quality greatly affects the efficiency of testing. For different SUTs, there is usually more than one MR. Suppose $\mathrm{MR}_i(r_i, r_{fi})$ is the $i$th metamorphic relation of $P$ and MR = $\{\mathrm{MR}_1, \mathrm{MR}_2, \ldots\}$ represents the set of metamorphic relations.

For Definition 1, it is not easy to understand, and more important, it needs more than two formulas to represent a metamorphic relation. Therefore, based on the traditional one, we propose an integrated formula in expression (3) to describe a metamorphic relation:

$$\mathrm{MR}_{f(x)} : \left\{\left(r, r_f\right) \mid \left(r\left(x_1, x_2, \ldots, x_n\right)\right) \Longrightarrow \left(r_f\left(f\left(x_1\right), f\left(x_2\right), \ldots, f\left(x_n\right)\right)\right)\right\}. \tag{3}$$

For example, for program $p\_\sin(x)$, a typical metamorphic relation is

$$\mathrm{MR}_{\sin(x)} : \left\{\left(r, r_f\right) \mid \left(r\left(x_1, x_2\right) = \left(x_2 = \frac{\pi}{2} - x_1\right)\right) \Longrightarrow \left(r_f\left(\sin\left(x_1\right), \sin\left(x_2\right)\right) = \sin^2\left(x_1\right) + \sin^2\left(x_2\right) = 1\right)\right\}. \tag{4}$$

*Definition 2* (original test cases). It is also called original test input recorded as OTI. Suppose there is a metamorphic relation $(r, r_f)$ and its input is $r(x_1, x_2, \ldots, x_n)$, then the OTI is test cases from $r(x_1, x_2, \ldots, x_n)$ which are generated with other testing methods, such as special testing, random testing, and iterative testing.

*Definition 3* (follow-up test cases [12]). It is also called follow-up input (FTI). Suppose one has a metamorphic relation $(r, r_f)$ and its input is $r(x_1, x_2, \ldots, x_n)$. FTI is all the test cases from $r(x_1, x_2, \ldots, x_n)$ except original test cases. Follow-up test cases are generated based on metamorphic relations. Suppose the input of a metamorphic relation is $r(x_1, x_2, \ldots, x_n)$, then it can be recorded as $r$(OTI, FTI).

## 3. Relationships between Metamorphic Testing and Metamorphic Relation

Metamorphic testing technique [10, 11] was proposed by Professor Chen, which checked the relations between inputs and outputs to determine whether the program satisfies the necessary properties, which are called as metamorphic relations. With the relations, it is unnecessary to assume the existence of an ideal oracle for test inputs. It could also check whether the test outputs deviate from the predicted ones. In the previous section, several prerequisite conceptions were proposed. The relationships between these conceptions of metamorphic testing and metamorphic relation will be illustrated in this section.

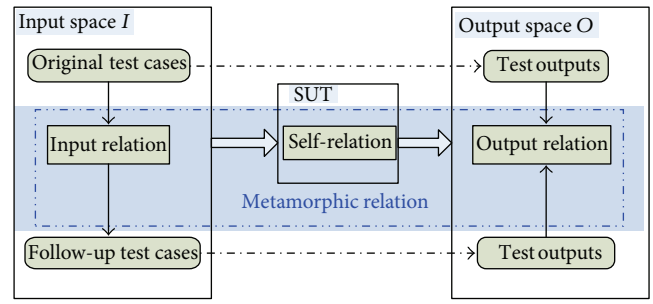


Figure 1: Relationships between several metamorphic testing conceptions.

Therefore, it is not an easy task to evaluate the effectiveness of different relations, because the definition of metamorphic relation is so complex. So, in Figure 1, the metamorphic relation decomposed into three subrelations. Original test cases and follow-up test cases come from input space $I$, and their corresponding test outputs should in output space $O$.

If $P$ is a program, then its program function is denoted by $[P]$ [13]. $x_1, x_2, \ldots, x_n$ represent finite numerable test inputs. $[P](x_1), [P](x_2), \ldots, [P](x_n)$ represent the outputs of corresponding program function $[P]$.

First of all, the metamorphic relation could be denoted as

$$\mathrm{MR} : \frac{P\left(x_i\right) = [P]\left(x_i\right)}{r\left(x_1, x_2, \ldots, x_n\right) \longrightarrow r_f\left([P]\left(x_1\right), [P]\left(x_2\right), \ldots, [P]\left(x_n\right)\right)} \quad (i = 1, 2, \ldots, n), \tag{5}$$

where the numerator $P(x_i) = [P](x_i)$ is the hypothesis of MR, which means that the MR based on the hypothesis that the output of program $P$ is equal to the output of program function $[P]$, and if the input satisfied $r(x_1, x_2, \ldots, x_n)$, the output of program function can satisfy the relation $r_f([P](x_1), [P](x_2), \ldots, [P](x_n))$.

*Definition 4* (input relation of metamorphic relation: IR). Given a program $P$, which implements program function $[P]$, $x_1, x_2, \ldots, x_n\ (n > 1)$ are different input variables and $[P](x_1), [P](x_2), \ldots, [P](x_n)$ are the corresponding outputs. If there is an MR:

$\mathrm{MR} : \{(r, [P], r_f) \mid r(x_1, x_2, \ldots, x_n) \xrightarrow{P(x_i)=[P](x_i)} r_f([P](x_1), [P](x_2), \ldots, [P](x_n))\}$ $i = 1, 2, \ldots, n$, then the relation $r$ is called as IR. It could also denote as IR = $\{(x_1, x_2, \ldots, x_n) \mid r(x_1, x_2, \ldots, x_n)\}$.

*Definition 5* (output relation of metamorphic relation: OR). The relation $r_f$ is called as output relation. It could also denote as OR $= \{([P](x_1), [P](x_2), \ldots, [P](x_n)) \mid r_f([P](x_1), [P](x_2), \ldots, [P](x_n))\}$.

In this paper, the program function $[P](x)$ is also called as self-relation (SR), which could be denoted as SR $= \{(x_1, x_2, \ldots, x_n) \mid ([P](x_1), [P](x_2), \ldots, [P](x_n))\}$. With this definition, it would be easy to understand the structure of MR, shown in Figure 1.

So a metamorphic relation can be represented as MR = [IR, SF, OR].

For example, for formula (4), $\mathrm{IR}(\mathrm{MR}_{\sin(x)}) = \{(x_1, x_2) \mid (x_2 = \pi/2 - x_1)\}$, $\mathrm{SR}(\mathrm{MR}_{\sin(x)}) = \{(x_i) \mid ([P](x_i) = \sin(x_i))\}$, $\mathrm{OR}\ (\mathrm{MR}_{\sin(x)}) = \{(\sin(x_1), \sin(x_2)) \mid (\sin^2(x_1) + \sin^2(x_2) = 1)\}$.

## 4. A Case Study

In Section 3, we introduce the relationships between metamorphic testing and metamorphic relation, which could be used to instruct the testing of this section, in which we conduct a case study, aiming to investigate the effectiveness of our method in verification of integer faults. We choose a kernel component of TCAS, named tcas.c which can be downloaded from the Software-artifact Infrastructure Repository [14].

*4.1. TCAS.* TCAS [15] is an on-board aircraft conflict detection and resolution embedded system. The system is intended to alert the pilot to the presence of nearby aircraft that pose a midair collision threat and to propose maneuvers so as to resolve these potential conflicts. In cases of collision threats, TCAS estimates the time remaining until the two aircrafts reach the closest point of approach (CPA) and presents two main levels of alert. When an intruder aircraft enters a protected zone, the TCAS issues a Traffic Advisory (TA) to inform the pilot of potential threat. If the danger of collision increases, then a Resolution Advisory (RA) is issued, providing the pilot with a proposed maneuver that is likely to solve the conflict. It is possible to download a C component from the Software-artifact Infrastructure Repository [14], which is called tcas.c. It is a preliminary version of TCAS, which is freely and publicly available and is responsible for the Resolution Advisories issuance. The component is (modestly) made up of 173 lines of C code. In our experiment, the environment is Linux operation system and GCC compiler.

*4.2. Designing Typical Metamorphic Relations.* According to the safety properties which is formalized described in [15], if the relative location between TCAS equipped airplane and threat one keeps fixed, no matter how high the altitudes (altitude is high enough) of the two airplanes are, the outputs are equal. After control flow and data flow analysis on the source code of tcas.c, we find that input parameters Own_Tracked_Alt (the altitude of the TCAS equipped airplane) and Other_Tracked_Alt (the altitude of the "threat") are independent from other input parameters, so we conclude that outputs keep equal if the relation between Own_Tracked_Alt and Other_Tracked_Alt is not changed; meanwhile other parameters' values are not changed too.

Above all, we propose a typical metamorphic relation $\mathrm{MR}_{\mathrm{TCAS}}$ based on both black box and white box information, where $a$ represents Own_Tracked_Alt and $b$ is Other_Tracked_Alt as follows:

$$\begin{aligned}
&\mathrm{MR}_{\mathrm{TCAS}} \\
&:\left\{\left(r, r_f\right) \mid r\left(A_1\left(x_1, x_2, x_3, a, x_5, b, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}\right),\right.\right. \\
&\qquad A_2\left(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8,\right. \\
&\qquad \left.\left. y_9, y_{10}, y_{11}, y_{12}\right)\right) \\
&= \left(\left(y_4 = a - \gamma\right), \left(y_6 = b - \gamma\right)\right) \\
&\Longrightarrow r_f\left(f\left(A_1\right), f\left(A_2\right)\right) = \left(f\left(A_1\right) = f\left(A_2\right)\right)\Big\}.
\end{aligned} \tag{6}$$

$\gamma$ is the offset; so in this experiment, let $\gamma = 300, 500, 800, 1000, 2000$ separately. $x_i\ (i = 1, 2, 3, 5, 7, \ldots, 12)$ and $y_i\ (i = 1, 2, \ldots, 12)$ represent other parameters, and their values do not influence the metamorphic relation. $A_1$ and $A_2$ represent individual input parameter combination of the TCAS.

*4.3. Mutant Generation.* To gain an understanding of how effective metamorphic testing is at detecting integer defects in applications without test oracles, we use mutation testing to systematically insert defects into the applications of interest. Mutation testing has been shown to be suitable for evaluation of effectiveness, as experiments comparing mutants to real faults have suggested that mutants are a good proxy for comparisons of testing techniques [16].

For this experiment, the algebra operations are not many for the source code of tcas.c; so the most possible integer bugs are that programmers do not estimate the range of the inputs by defining a wrong type that represents a shorter bit vector. As the range of integer type is from −2147483648 to +2147483647 and char type is from −128 to +127 in GCC standard, we introduce three types of integer mutants shown in Table 2. For the first mutant, we replace the type of parameter Cur_Vertical_Sep from "int" to "char," which is the mutant 1 version program. For the second one, we replace the type of "int" to "short int," which is mutant 2 version program. And for the third one, we replace the type of "int" to "long int," which is mutant 3 version program. A mutant code segment of tcas.c is showing in Box 1, which is a kind of width overflow fault.

*4.4. Test Cases Execution and Results Comparison.* Subtest sets in testplans-bigcov.tar cover all the branches in the program and the tests are generated randomly for each branch. We select suite 8 (including 87 test cases) in testplans-bigcov.tar as test inputs. We execute these test cases in the three mutant version programs. We say these different outputs between these versions are actual mutants because of mutant injection. Then forgeting the execution information, we insert several "if" statements into the end of the program with formal safety properties as test oracle. A test is "failure" if

Table 2: Mutation operators used in this experiment.

| Integer mutants | Operators | Description |
|---|---|---|
| Width overflow | Integer truncation | Short-cut data type definition replacement |
| Signedness defect | Unsigned to signed | {unsigned, short, unsigned int, unsigned long} → {signed long, signed int, signed short} |
| | Signed to unsigned | {signed long, signed int, signed short} → {unsigned, short, unsigned int, unsigned long} |

```
...
char Own_Tracked_Alt; // correct version int Own_Tracked_Alt;
int Own_Tracked_Alt_Rate;
int Other_Tracked_Alt
...
```

Box 1: Mutation code segment of tcas.c.

Table 3: Testing execution results.

| | Mutant 1 | Mutant 2 | Mutant 3 |
|---|---|---|---|
| Safety property | 2/87 | 1/87 | 2/87 |
| MR ($\gamma = 300$) | 10/87 | 10/87 | 10/87 |
| MR ($\gamma = 500$) | 20/87 | 20/87 | 25/87 |
| MR ($\gamma = 800$) | 25/87 | 23/87 | 26/87 |
| MR ($\gamma = 1000$) | 25/87 | 23/87 | 26/87 |
| MR ($\gamma = 2000$) | 15/87 | 15/87 | 10/87 |

and only if none of these properties are satisfied and "pass" if and only if at least one is hold. At last, we use MR and original test cases suite 8 to generate follow-up test cases and test the results.

For evaluating the effectiveness of different MRs, we adopt the fault detection ratio (FDR) [8], the percentage of test cases that could detect certain mutant $m$; that is, FDR $(m, T) = N_f/(N_t - N_e)$, where $N_f$ is the number of times SUT fails, $N_t$ is the number of test cases, which include original test cases and follow-up test cases, and $N_e$ is the number of infeasible test cases. The testing execution results are shown in Table 3. For the value in the table, numerator is the number of unsatisfied test cases and denominator is the number of total test cases.

In this experiment, we select $\gamma = 300, 500, 800, 1000, 2000$, separately. $\gamma = 500$ is the medium of the values of two parameters of Own_Tracked_Alt and Other_Tracked_Alt individually in suite 8. The test result is affected by $\gamma$, because after subtracting the value $\gamma$, some of Own_Tracked_Alt and Other_Tracked_Alt are changed from the invalid domain to the valid domain. If we select an appropriate $\gamma$, we may get a higher FD.

As Gotlieb [15] proposed ten safety properties of TCAS, which are more effective than traditional testing methods, so we compare our metamorphic testing with the safety properties testing. The average integer fault detection ratio of metamorphic testing with different MRs and a safety property is shown in Figure 2. From the figure, we can see that metamorphic testing method is more effective than the formal safety property method for the mutant in this experiment. For the

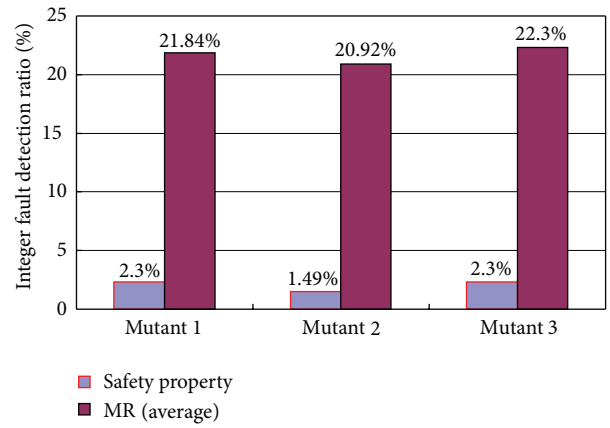


Figure 2: FDR of safety property and metamorphic relation for different mutants.

mutant 1 version program, FDR o MR in average is 21.84%, which is much higher than the FDR of the formal safety property method, which is only 2.30%. And for the other two mutant versions, the FDR of MR in average is 20.92% and 22.30%, and 1.49% and 2.30% for the safety property method.

*4.5. Experiment Conclusions.* By this experiment, we could see that it is possible to detect latent fault by a typical symbolic metamorphic relation, which represents a series of relations. And we should admit that it is not easy to design this typical metamorphic relation, which needs an in-depth understanding of the software under test. In other words, this kind of MRs, which is related to the kernel function of the SUT, is more effective than other relations. And the result of this experiment is also consistent with the conclusion of [17].

However, in this case study, we suppose that there is only one integer overflow defect in the program under test by mutant injection. With one metamorphic relation, it is easy to detect only one integer defect, when the relation is not satisfied. So, it is hard to detect more faults by only one metamorphic relation. At the same time, safety property method can detect a design failure which is not detected

by MT. This is a design bug that the programmer cannot take into account in the implementation or design process, and this goes beyond our discussion about integer defects detection.

## 5. Conclusions

In this paper, we have presented a case study on the mission critical program MT, which could detect integer faults effectively. Integer fault detection ratios contrasting to traditional safety property technique are discussed. The results show that the proposed metamorphic testing is more effective than the safety property technique, which is based on in-depth understanding of the program structure and algorithm to generate this kind of novel relations.

## Acknowledgments

This work was supported by the National High Technology Research and Development Program of China Project (no. 2009AA01Z402) and the Natural Science Foundation of Jiangsu Province, China (no. BK2012059, no. BK2012060).

## References

[1] R. C. Sercord, *Secure Coding in C and C++*, Addison Wesley& Person Education Asia, 2006.

[2] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[3] L. Baresi and M. Young, "Test oracles," Tech. Rep. CIS-TR01-02, Department of Computer and Information Science, University of Oregon, Eugene, Ore, USA, 2001.

[4] M. Blum and S. Kannan, "Designing programs that check theirwork," in *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '89)*, pp. 86–97, ACM Press, New York, NY, USA, 1989.

[5] M. Blum and S. Kannan, "Designing programs that check their work," *Journal of the ACM*, vol. 42, no. 1, pp. 269–291, 1995.

[6] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," in *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90)*, pp. 73–83, ACM Press, New York, NY, USA, 1990.

[7] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 549–595, 1993.

[8] P. E. Ammann and J. C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, 1988.

[9] D. Brumley, T. Chiueh, and R. Johnson, "RICH: automatically protecting against integer-based vulnerabilities," in *Proceedings of the 14th Annual Network and Distributed System Security (NDSS '07)*, 2007.

[10] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Tech. Rep. HKUST-CS98-01, Departement of Computer Science, Hong Kong University of Science and Technology, 1998.

[11] F. Chan, T. Y. Chen, S. C. Cheung, M. Lau, and S. Yiu, "Application of metamorphic testing in numerical analysis," in *Proceedings of the IASTED Conference in Software Engineering*, pp. 191–197, Acta Press, 1998.

[12] T. Y. Chen, D. H. Huang, and T. H. Tse, "Case studies on the selection of useful relations in metamorphic testing," in *Proceeding of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC '04)*, pp. 569–583, Polytechnic University of Madrid, Madrid, Spain, 2004.

[13] R. C. Linger, H. D. Mills, and B. I. Witt, *Stuctured Programming Theory and Practice*, Addison-Wesley, Reading, Mass, USA, 1979.

[14] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[15] A. Gotlieb, "TCAS software verification using constraint programming," *Knowledge Engineering Review*, vol. 27, no. 3, pp. 343–360, 2012.

[16] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 402–411, May 2005.

[17] T. Y. Chen, F. C. Kuo, and Y. Liu, "Metamorphic testing and testing with special values," in *Proceedings of the 5th International Conference on Software Engineering, Artificial Intelligence, Networking, and parallel/Distributed Computing*, pp. 128–134, 2004.