



Cost Efficient Dependable Electronic Systems

A technique for fault tolerance assessment of COTS

Author	Ruben Alexandersson, Krishna Chaitanya D., Peter Öhman
Document Id	008
Date	28 September 2005
Availability Status	Public Final

A technique for fault tolerance assessment of COTS based systems

Ruben Alexandersson, Krishna Chaitanya D., Peter Öhman

Dept. of Computer Engineering
Chalmers University of Technology, SE-41296, Gothenburg, Sweden

ruben@ce.chalmers.se

Telephone: +46(0)31-7721685

Fax: +46(0)31-7723663

Abstract. This paper investigates the feasibility of emulation of source code software faults directly in Java byte code. Experimental results show that software defects introduced in source code can be emulated in Java byte code with high level of confidence. This makes it possible to validate the dependability of Java programs with respect to realistic software defects embedded within used COTS components without the need to know the source code. First it is investigated with good results how well the fault locations found at byte code level maps to the source code. Next the behaviors of the byte code level mutants are compared with the corresponding source code mutant behavior. In a back-to-back comparative study with 2210 mutants based on ten representative programming defects, no difference in the program behavior between source and byte code level mutants could be distinguished.

1 Introduction

With software development proceeding at unprecedented speed, in-house development of all system components will be too costly. The use of commercial-off-the-shelf (COTS) components reduces time-to-market, since the components are ready to be used, and save money as the cost of such component acquirement is cheaper than developing from scratch. Quality and risk concerns currently limit the use of COTS components in safety critical applications. In order to increase the level of COTS usage in this application area the dependability assessment of COTS-based systems must be more accurate than today.

Software defects (also called faults or bugs) have been recognized as the major cause of computer outages [1]. The elimination of all software defects during development is very difficult to achieve in practice. Therefore, all non-trivial systems contain residual software defects that are activated when an appropriate input pattern is encountered during operation, leading to system failure, thus greatly influencing system dependability.

Furthermore, as larger systems are increasingly built using COTS components, the residual software faults embedded within the COTS represents a growing risk as the quality level of COTS components is difficult to assess [2]. Therefore any critical sys-

tem built on COTS components should have robustness against faults embedded within these components [3]. Since the source codes are, in most cases, not available for COTS components it is not trivial to use fault injection techniques to experimentally verify the robustness of the system. Because of this, earlier efforts have focused on injecting errors on the interface between the COTS component and the rest of the system instead of injecting faults within the COTS, e.g. [3]. Although a useful technique, it can't be known how representative these errors are and if they actually could have been generated by the activation of possible faults within the COTS. Therefore in order to generate error behavior that is representative of real software faults we need a technique that can both determine the faults that could have been made within the COTS and inject them. We present a feasibility study of a mutation-based fault injection technique that shows great promise in achieving exactly that.

Fault injection is a well-known method to study system behavior in the presence of faults. The method is traditionally used to emulate external disturbances causing transient or permanent faults in the system under consideration. Residual faults like software defects, on the other hand, are in most cases not covered when using experimental validation techniques like fault injection. However some studies, e.g. [4, 5, 6, and 7], have discussed the concept of injecting software faults for dependability evaluation. As the residual software defects are hard to activate (since they have passed the normal quality assurance process) they cannot be used to experimentally validate the system dependability since the fault activation speed will be too low. Instead, representative software defects have to be introduced into the system by fault injection in order to evaluate the capability of the system to cope with residual defects. By this a significant speed up of fault activation is achieved.

The mutation technique was originally developed in the testing community as a means to assess test set quality. For the purpose of dependability evaluation, on the other hand, SWIFI has been the traditional technique [8, 9, 10, and 11] for hardware related faults. This technique was extended to include software faults by Costa and Madeira et. al. [6, 7] but showed to have limitations. Because of this later work on software fault injection for dependability evaluation purposes has taken up and used the technique of mutating program code [15].

Mutation-based fault injection of software defects is normally done by modifying elementary program components in the source code which introduces small changes (faults) in the target program code, creating different versions of a program, and observe how each version behave (each has one injected software fault) [12, 13]. The mutation testing community has investigated the method of changing the source code for a long time. Modification of source code requires recompilation, re-linking and re-loading and this introduces a large overhead for fault injection experiments. Traditionally, this is somewhat enhanced by using interpretative tools. A method to avoid this and speed up the process is mutant schema [14] but this still requires the source code to be known. Alternatively, by modifying the machine code using low-level fault models the time-consuming post injection process of compilation and linking would be eliminated. This method is particularly useful if the source code is not available which is the case when using COTS components. Unfortunately this requires thorough analysis in order to inject a set of low level-faults that corresponds to common high-level programming faults.

In a recent study, Madeira et. al. [15] proposed a technique to emulate software faults through educated mutations introduced at the machine-code level. The central idea is to find key programming structures at the machine code-level where high-level software faults can be emulated. The results show that ODC classes [16] of faults such as assignment, checking, interface and simple algorithm faults can be directly emulated using this technique.

We extend these principles to be applied to the byte code for virtual machines like JVM (Java Virtual Machine). Java byte code is the machine-level representation of Java programs just as real machine language is the representation of C or C++ programs. Since the Java byte code preserves the object structure of the source code we get the benefit of being able to emulate object-oriented faults, which is not possible when mutating real machine code. Java byte code is also in other aspects closer to the source code with more instructions that uniquely maps to specific source code constructs. We want to know if this gives us the possibility to mimic a larger set of the actual faults made at source code than what is possible in real machine code. This could be highly usable and give reliable dependability evaluations.

The use of byte code level emulation has been proposed by Ma et. al. [13, 17] for mutant testing purposes. However, since their scope doesn't include the case where the source code is unknown, they investigated a combination of mutant schematic and byte code translation for optimal performance.

In this paper we show that representative software defects in Java source code can be mapped to corresponding structures in the Java byte code. We can thereby emulate realistic software defects in components when the source code is unknown. Furthermore, by this a significant speed up is achieved making it possible to experimentally validate the dependability of large real life software systems with high confidence.

2 Defect mapping

2.1 Source code faults

The purpose of this mutation based method of fault injection is to make it possible to emulate the actual faults that are normally introduced into software during development, and not like other methods, e.g. [3], the error behavior of such faults.

Fault injection experiments using actual software faults should give better confidence in the validity of the results obtained, and could also be used to verify the validity of error based methods. In order to have a fault model that can be considered representative of common software faults, it must be based on field data from actual software projects. To the best of our knowledge no such survey of common faults has been published for the Java language. Duraes et. al. [18] has made a survey for the C language and although it is likely that the fault types found for C will also to a large extent be representative of Java faults this is not known. Because of this we can not base this study on a set of known common faults and verify that these specific faults can be emulated in Java byte code but we must evaluate which classes of faults, in respect to code constructs and structure, can be successfully emulated. Therefore, the source code faults have been selected with emphasis on the criteria that they are suit-

able to test the investigated method rather than they would stipulate a representative fault model for dependability evaluation.

Table 1. The selected fault types

MOI – Missing Object Instantiation: The omission of instantiating an object variable when created. Can lead to failure if accessed prior to given a value	
<i>ODC:</i> Assignment (MVI)	<i>Limitation:</i> No Limitation
OCM - Object Changed by Method: An object given as input is altered by the method although that was not intended.	
<i>ODC:</i> Assignment, Interface	<i>Limitation:</i> Only int variables within the object are altered.
WOM - Wrong Overriding Method name: Typos while writing an overriding method will lead the compiler to treat it as a new one.	
<i>ODC:</i> Interface	<i>Limitation:</i> No limitation
WCT - Wrong Casting type: An object is being cast to a type that it doesn't have or is a subtype of.	
<i>ODC:</i> Assignment	<i>Limitation:</i> No limitation
DHV - Declaration of Hiding Variable: A variable is unintentionally given the same name as an instance variable in its own or an ancestor class thus shadowing it.	
<i>ODC:</i> Assignment, Checking, Interface, Algorithm	<i>Limitation:</i> Only int variables that are accessed for reading are considered.
WVA - Wrong Variable Assignment: Variable is assigned to a wrong constant value.	
<i>ODC:</i> Assignment (WIDI)	<i>Limitation:</i> Only instance variables of type int considered.
WLO - Wrong Logical Operator: Applying the wrong logical operator. For instance using when it should be &&.	
<i>ODC:</i> Assignment (WLEA), Checking (WLEC), Interface (WLEP)	<i>Limitation:</i> Only <i>method && variable</i> and <i>variable && variable</i> expressions are considered.
WEB - Wrong Else Body: Omission of curly brackets surrounding a multi statement body of an <i>else</i> statement.	
<i>ODC:</i> Algorithm (MIEA), Checking (MIA)	<i>Limitation:</i> Only simple unnested if-else statements are considered.
WPO - Wrong Parameter Order: The order of equal type parameters are mixed up when doing a method call.	
<i>ODC:</i> Interface (WPFO)	<i>Limitation:</i> Only int variables are considered.
MBC - Missing Break in Case: A break statement is unintentionally omitted at the end of a case body.	
<i>ODC:</i> Algorithm (MBC)	<i>Limitation:</i> No limitation

The criteria for selecting the faults have been:

- All applicable ODC classes [16] should be represented among the faults (i.e. Assignment, Checking, Interface, and Algorithm)
- The set of faults should include the manipulation of variables, values, interfaces and program flow, spanning both classical procedural faults and object-oriented faults.
- The faults should differ in structure and involved language constructs.
The faults should be likely to occur from a programmer’s mistake.

Using these criteria a set of ten fault types, that can be considered representative of the general case, have been selected for the proof-of-concept experiments (see Table 1.). Most of the fault types correspond to multiple source code patterns that are equivalent in complexity and structure. To speed up the experiment setup phase, only a subset of patterns for each fault have been considered in the experiments, which is adequate for showing the feasibility. For example, only the byte code mapping (i.e. key programming structure) of two types of Wrong Logical Operator (WLO) faults namely *variable && variable* together with *method-call && variable* were implemented. However, the WLO and WEB fault types have inherently significant differences in their respective set of source code patterns. Consequently, the results from these faults cannot directly be applied to the general case, but a more detailed investigation must be conducted (see section 4).

The detailed ODC-classification suggested in [18] is used for the five classical procedural fault types (WVA, WLO, WEB, WPO, and MBC). For most of the object-oriented fault types, with the exception of the MOI fault, this detailed classification is not applicable and therefore only the original ODC classification is used. As can be seen in Table 1 several of the fault types belong to more than one ODC class. This is due to the fact that the classification is decided by the context in which the fault appears. It can be noted that two of the selected faults match the Inter-Class operators by Ma et. al. [17]. The DHV fault is equal to the IHI operator and the WOM fault is very similar to the IOD operator. The difference between WOM and IOD lies in that the mutation pattern for WOM changes the name of the method to simulate a typo and the IOD operator deletes the method altogether. Since the changed method name is never called by the surrounding program the effect of the two is practically the same.

2.2 Fault emulation technique

The investigated technique is an adaptation of the G-SWFIT [15] method to Java programs. By introducing fault-specific changes directly into the byte code, the compiled result of a defect source code is emulated (without the need of an actual compilation). This requires knowledge of how Java source code is translated into byte code, in particular how high-level programming errors translate into specific instruction patterns at byte code level.

The Missing Break in Case (MBC) fault is used as a running example throughout this section to explain the technique of finding (by the concept of key programming

structures) and model (by fault injection patterns) a source code fault at the byte code level. A non-faulty and a faulty source code example are shown in Figure 1.

Non-Faulty Source code	Faulty Source code
<pre>public void checkBreak(int s){ switch(s) { case 1: System.out.println("This is OK");break; case 2: System.out.println("Not this one"); } }</pre>	<pre>public void checkBreak(int s){ switch(s) { case 1: System.out.println("This is OK"); case 2: System.out.println("Not this one"); } }</pre>

Fig. 1. Non-faulty and faulty source code for the MBC fault.

The part that requires most effort is that of finding where the faults can actually be injected. As a first step, an environment for the fault at source code level is defined. An environment is a specific set of source code statements where there is some room for programmers to make the fault. For example in the code given in Figure 2, if the programmer forgets the break statement, it leads to a switch fall through. So, the Switch-Case statement as a whole forms the environment thereby giving room for the programmer to make the fault as shown in Figure 2.

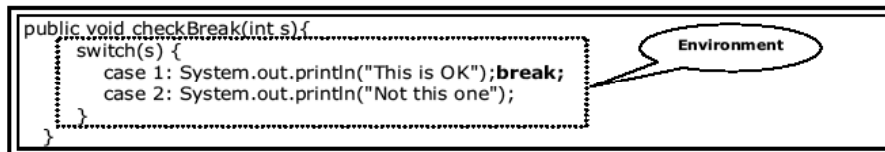


Fig. 2. Environment of non-faulty source code.

There can be more than one environment definition for a given fault type as there could be many such sequences/combinations/patterns that may lead to a fault of that particular type. For example, there are different environment definitions for the two types of WLO faults (see Table 1.) implemented.

Once the environment at the source code level is defined, the comparison of the generated code for both variants (with and without faults) allows us to identify the specific instruction patterns at byte code level that are used to locate each fault (called key programming structure) and the instruction patterns used to mutate the fault free byte code (called fault injection pattern).

This is illustrated in figure 3, which shows the set of correct and faulty byte codes for our running example. The key programming structure is highlighted in the non-faulty code to the left. The faulty byte code to the right is what is generated when the fault is made in the source code and then compiled. To mimic the behavior of the code to the right (and thereby emulating the fault) the parameter for the highlighted *goto* at offset 36 can be changed from 47 to 39. The fault injection pattern is therefore defined as making that change.

Non-Faulty Bytecode			Faulty Bytecode		
Offset	Opcode	Parameters	Offset	Opcode	Parameters
0	iload	1	0	iload	1
1	lookupswitch	1:28, 2:39, default: 47	1	lookupswitch	1:28, 2:36, default: 44
28	getstatic	java.lang.System::java.io.PrintStream out	28	getstatic	java.lang.System::java.io.PrintStream out
31	ldc	"This is OK"	31	ldc	"This is OK"
33	invokevirtual	java.io.PrintStream::void println(java.lang.String)	33	invokevirtual	java.io.PrintStream::void println(java.lang.String)
36	goto	47	36	getstatic	java.lang.System::java.io.PrintStream out
39	getstatic	java.lang.System::java.io.PrintStream out	39	ldc	"Not this one"
42	ldc	"Not this one"	41	invokevirtual	java.io.PrintStream::void println(java.lang.String)
44	invokevirtual	java.io.PrintStream::void println(java.lang.String)	44	nop	
47	nop		45	return	
48	return				

Fig. 3. Key programming structure at the byte code level

All standard source code compilers contain compile time checking mechanisms that verifies certain aspects of a program, e.g. a variable is declared prior to being assigned a value. Therefore some faults made at source code level cannot pass the compilation and result in a faulty byte code executable. When conducting byte code fault emulation these faults must be excluded by analyzing the mutant byte code.

In our running example with the switch fault we need to verify that the removal of the break statement does not lead to any unreachable statements that would have been recognized by the compiler.

The key programming structures and fault injection patterns for each fault type is the fundamental information needed to conduct the fault injections as described in section 3.

3 Fault injection and evaluation

The fault injection technique is divided into two phases. First, the *fault analysis phase* which searches for fault-related information in the byte code and mark the fault location. Second, the *fault emulation phase* that mutates the non-faulty byte code in a fault-specific way. The technique takes the non-faulty byte code as input and generates the mutated byte code that emulates the fault.

During the fault analysis phase the target application byte code is scanned for all the key programming structures defined for a fault. The search is based on a simple regular expression or a complex algorithm, depending on how the structure is defined. For each such hit, the corresponding fault location information is stored and passed on to the fault emulation phase.

During the fault emulation phase, the fault location information from the fault analysis phase is collected and mutated in a specific way defined by a fault injection pattern, such that the resultant mutated byte code emulates the fault.

3.1 Prototype description

For the proof-of-concept experiments a fault injection tool was developed using a pipe-and-filter architecture. This gives us the opportunity to collect and analyze the data output from the various components of the tool independently.

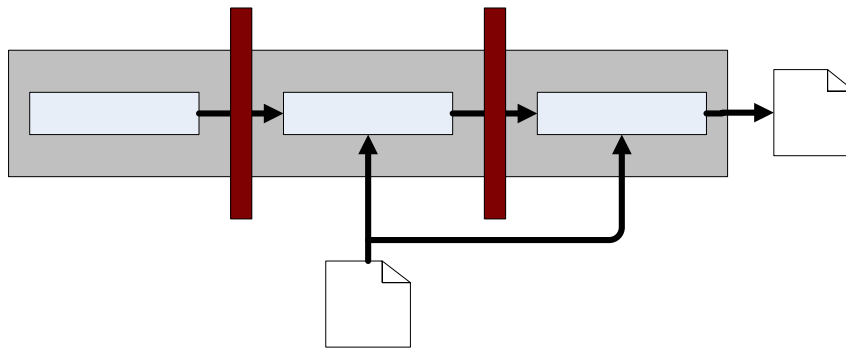


Fig. 4. Architecture of the fault injection tool.

As can be seen in Figure 4, the principal components of the architecture are:

Fault Library The fault library component is the container of the fault objects. Each fault object represents a high level fault.

Fault Analysis The fault analysis component is responsible for analyzing the original byte code for faults and producing formatted fault location information (XML) by searching for faults in `.class` files (byte code) extracted from the jar file supplied.

Fault emulation The fault emulation component performs fault-specific mutation on a given jar file according to the fault location information supplied by the analysis phase and produces a mutated jar file as result. Fault emulation can either be run in pipeline with fault analysis phase or can be run independently. In the latter case, persisted fault location information should be given. The fault mutation component of the tool produces, along with the mutant, a status report containing trace information about the mutations.

Filters The set of faults and fault locations used in the experiment can also be restricted with the aid of fault and fault location filters.

3.2 Experimental feasibility evaluation

The experiment aims at determining to what extent the byte code fault patterns emulate the corresponding source code defects. The fault injection process consists of both finding the correct locations (analysis phase) and manipulating the byte code (emulation phase). Consequently, the experiment is conducted in two parts. First it is investigated how well the fault locations found at byte code level maps to the source code. Next the behaviors of the byte code level mutants are compared with the corresponding source code mutant behavior.

Two target programs (named tp1 and tp2) are used in the experiment. The programs are selected on the criteria that it should be possible to manually find all possible source code fault locations and only a small number of test vectors should be needed for defect activation. Therefore the two target programs are fairly small (of size 20 and 70 KLOC) and sequential in structure so that the output is a strict function of the input. Both programs are implementations of a source code analysis tool and offer the same functionalities but using different user interfaces (command line vs. graphical) and were implemented by different teams using different off-the-shelf parsers (with known source code).

3.2.1 Fault location experiment

As can be seen in Table 2, all the potential locations can be found, without any incorrect hits, for a vast majority of the investigated fault types. This means that by only analyzing the byte code it will be possible to determine the actual number (and locations) of possible programming errors at the source code level of the investigated types.

As pointed out in section 2.1, some fault types correspond to multiple source code structures. For the WLO and WEB fault types, only the basic and least complex structure has been considered. That might indicate that we will have less optimistic data for the general case for these two fault types. A more comprehensive discussion about the generalization is conducted in section 4.

The missed locations for the Missing Object Instantiation (MOI) fault are because of the concept of `StringBuffer`. An instance of `StringBuffer` will be created from both the code:

```
str = str + "hello";  
and from  
StringBuffer bf = new StringBuffer();
```

The later is a correct fault location and the former is not. Since these are not distinguished at byte code level by our search pattern there will be either some incorrect or some missed locations when these constructs are present in the code.

The missed and incorrect locations for the Missing Break in Case (MBC) fault type are the result of a fundamental difficulty with byte code level fault mapping. Many of the basic language constructs like switch-case, if-else, loops and logical operation are translated to the same byte code pattern (a limited set of branch instructions) which in some cases cannot be distinguished (see section 4 for a detailed discussion).

Table 2. Byte code level fault location precision

Fault			Nr. of correct location identifications	Nr. of incorrect locations	Nr. of missed locations	Ratio of correct locations
MOI	Missing Object Instantiation	tp1	68	0	3	96%
		tp2	204	0	20	91%
OCM	Object Changed by Method	tp1	34	0	0	100%
		tp2	439	0	0	100%
WOM	Wrong Overriding Method name	tp1	84	0	0	100%
		tp2	118	0	0	100%
WCT	Wrong Casting type	tp1	270	0	0	100%
		tp2	457	0	0	100%
DHV	Declaration of Hiding Variable	tp1	552	0	0	100%
		tp2	93	0	0	100%
WVA	Wrong Variable Assignment	tp1	15	0	0	100%
		tp2	49	0	0	100%
WLO	Wrong Logical Operator	tp1	1	0	0	100%
		tp2	5	0	0	100%
WEB	Wrong Else Body	tp1	4	0	0	100%
		tp2	22	0	0	100%
WPO	Wrong Parameter Order	tp1	21	0	0	100%
		tp2	33	0	0	100%
MBC	Missing Break in Case	tp1	306	20	38	84%
		tp2	1043	0	64	94%

3.2.2 Fault mutation experiment

As Java programs are executed on a virtual machine there are inherent run time mechanisms for fault detection (i.e. exception handling). Successful fault detection by these mechanisms is manifested by an error signal like a message on the stderr stream. In addition to this, application specific fault detection (e.g. boundary checking of variable values) mechanisms also use error signaling such as stderr or alert dialogue screens for fault detection signaling. When a fault is not detected an application can fail according to the well-known semantics of value and/or timing failures. Therefore it is natural to classify the application failure mode based on these three parameters (i.e. timing, output value and error signaling) as can be seen in Table 3.

Table 3. Application failure mode classification

Timing	Output	Error signal	Classificaton
OK	OK	No	Correct
OK	OK	Yes	Tolerated fault
OK	NOK	No	Undetected value failure
OK	NOK	Yes	Detected value failure
NOK	OK	No	Undetected timing failure
NOK	OK	Yes	Detected timing failure
NOK	NOK	No	Undetected arbitrary failure
NOK	NOK	Yes	Detected arbitrary failure

Since there is no timing constrains for the target programs the only possible timing fault will be when the programs do not terminate (i.e. they “hang”).

The purpose of the experiment is to verify that the byte code mutants behave in the same way as the corresponding source code mutants. During the experiment, a detailed comparison between the output of the byte code mutants and source code mutants were conducted. Also the outputs of the mutated program were evaluated by comparison with a reference of correct outputs and the corresponding application failure mode was determined.

Instead of using a large number of random test vectors, only one specifically selected for utilizing a large proportion of the program functionality was used. By this approach, a large number of fault activations were obtained with a minimum number of target program executions.

In a back-to-back comparative study with 2210 mutants obtained by the ten fault types injected into the target programs, no difference in the program behavior between source and byte code level mutants could be distinguished. Not only did they show the same application failure profile but also had the exact same outputs. This strongly indicates that the byte code level is feasible for emulating the erroneous behaviors of programs containing residual programming defects.

As an example of the type of results that can be obtained by using the proposed method, Table 4 shows the failure mode classification of the conducted experiment. Since the emulation of source code level faults is perfect, identical application failure modes are obtained for both source and byte code level mutants. Hence, only one set of data is presented in the table.

The detection coverage for the value faults is ranging between 50% and 100% depending on fault type. This indicates that a large proportion of the faults are detected by exception handling mechanisms, as these were the only detection mechanisms available in the test programs.

The large number of correct outputs (i.e. the fault has not been activated or has been masked) is mainly owing to the use of off-the-shelf parsers in the test programs as they contain a large proportion of unused functionality. A separate investigation of the experimental results gave at hand that only 4% of the faults injected into the parser code lead to a program failure whereas the corresponding figure for the application specific code was 53%. Fault activation is a well-known concern when conducting fault injection. The conducted experiment indicates that this problem is accentuated when using fault injection techniques on COTS based software systems.

Table 4. Aggregated failure profile based on the proof-of-concept experiment

Fault		Correct	Tolerated Faults	Detected value failures	Undetected value failure	Timing failure (hang)	Total
MOI	Missing Object Instantiation	139	0	129	4	0	272
OCM	Object Changed by Method	469	0	3	1	0	473
WOM	Wrong Overriding Method name	196	0	3	3	0	202
WCT	Wrong Casting type	137	0	124	0	0	261
DHV	Declaration of Hiding Variable	612	0	26	7	0	645
WVA	Wrong Variable Assignment	48	0	11	5	0	64
WLO	Wrong Logical Operator	4	0	1	1	0	6
WEB	Wrong Else Body	26	0	0	0	0	26
WPO	Wrong Parameter Order	48	0	5	1	0	54
MBC	Missing Break in Case	144	0	62	1	0	207

4 Discussion

As seen in the experimental results the object oriented fault types (MOI, OCM, WOM, WCT, and DHV) are successfully found and emulated at byte code level. There are some missed locations for the MOI fault type. However, as explained in 2.1, these are related to a special case and can't be considered to have any impact on the overall feasibility of the method.

Concerning the basic (non-object oriented) faults they fall into three categories. The first are associated with data or data containers (WVA) and the second concerns the manipulation of interfaces (WPO). Both containers and interfaces are very visible in Java byte code and the identification and modification of these haven't presented any difficulties in this study.

As already mentioned we only implemented a specific subset of source patterns for each fault for this experiment. For all fault types discussed so far the subset that we used is representative of the general case and consequently the method has a general feasibility for these fault types.

However, for the third category of basic faults, namely the ones associated with program flow (i.e. WLO, WEB and MBC) the case is a bit more complicated. The subsets that are implemented for two of these (WLO and WEB) are the basic and least complex ones, and therefore the experimental results for these can't automatically be scaled to the general case. But from implementing these subsets and the full implementation of the Missing Break in Case (MBC) fault type the problems associated with the general case have been identified. It should be noted that the problems dis-

cussed below concerns local program flow (not passing any interfaces). If a fault stipulates mutation of program flow at interface level it falls under the second category of basic faults and is easily achievable.

When conditional statements like logical operations, if-else, switch-case, and loops are translated into byte code they all get a similar structure (i.e. a series of branches) that makes it very hard, and in many cases probably impossible, to distinguish them from each other. The problem is even more accentuated when these statements are nested together which is very common in normal programs. This is a serious problem when working on real machine code and to some extent it seems to be a problem for Java byte code as well. However there are some constructs in Java byte code that we lack in real machine code that can be of aid in this process. As an example there are unique codes that are present whenever there is a switch statement. Naturally this is helpful when distinguishing a switch-case from a series of if-else and it is the reason for the good result with the MBC fault type.

Thus, for this class of faults, Java byte code is still better suited for fault injection than real machine code but it is uncertain whether it is suited enough. A further and more detailed study of this class of faults is needed to fully understand the limitations of Java byte code in this regard.

5 Conclusion

In this paper we investigate a mutation-based fault injection technique that can be used for dependability evaluations. We show that representative software defects at the Java source code level can be mapped to corresponding structures at the byte code, thereby emulating realistic software defects in components where the source code is unavailable.

It is shown that byte code level mutants emulates source code level mutants for a set of ten representative programming source code defects with some minor restrictions in finding all fault locations in the byte code.

Firstly it is investigated how well the fault locations found at byte code level maps to the source code. The study shows that for fault types related to object orientation, data/data container and interfaces a one to one mapping in fault location can be obtained. The program flow fault types, on the other hand, are in some situations difficult to locate at byte code level which is a general problem for all low level mapping methods.

Secondly the behaviors of the byte code level mutants are compared with the corresponding source code mutant behavior. In a back-to-back comparative study with 2210 mutants no difference in program behavior between source and byte code level mutants could be distinguished. Not only did they show the same application failure profile but also had the exact same outputs. This strongly indicates that the byte code is feasible for emulating the erroneous behaviors of programs containing residual programming defects.

The investigated mutant based fault injection technique can therefore be used to validate the dependability of Java COTS based systems with respect to realistic software defects without the need to know the source code. By this also a significantly

speed up is achieved compared to source code based methods making it possible to experimentally validate the dependability of large real life software systems with high confidence.

The results also indicate that a large proportion of the faults are detected by exception handling mechanisms. Furthermore, we also show that the fault activation level is significantly lower in the off-the-shelves part compared to the application specific part of the code.

6 References

1. U. D. Commerce, "The Economic Impacts of Inadequate Infrastructure for Software Testing," RTI, Research Triangle Park, NC 27709, US 2002.
2. S. Sedigh-Ali, A. Ghafoor, and R. A. Paul, "Metrics and models for cost and quality of component-based software," vol. -, pp. - 155, 2003.
3. J. Voas, F. Charron, and K. Miller, "Robust Software Interfaces: Can COTS-based Systems be Trusted Without Them?, In Proc. Of 15th Int'l Conf. on Computer Safety, Reliability, and Security (SAFECOMP'96), Vienna, Austria, October 1996. Springer-Verlag.
4. J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," vol. -, pp. - 313, 1996.
5. E. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting how badly "good" software can behave," vol. - 14, pp. - 83, 1997.
6. H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," presented at Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on, 2000.
7. Diamantino Costa, Tiago Rilho, M. Vieira and Henrique Madeira, "ESFFI – A novel technique for the emulation of software faults in COTS components", Conf. on the Engineering of Computer-Based Systems, ECBS 2001, Washington, DC, USA, April, 2001
8. J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: generic object-oriented fault injection tool," 2001.
9. J. Carreira, H. Madeira, and J. G. Silva, "Xception: a technique for the experimental evaluation of dependability in modern computers," vol. - 24, pp. - 136, 1998.
10. G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: a flexible software-based fault and error injection system," vol. - 44, pp. - 260, 1995.
11. Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "FIAT-fault injection based automated testing environment," vol. -, pp. - 107, 1988.
12. R. A. DeMillio, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11(4), pp. 34-41, April 1978.
13. J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "An Experimental Mutation System for java," 2004.
14. R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, 1993.
15. J. Duraes and H. Madeira, "Emulation of software faults by educated mutations at machine-code level," presented at 13 th International Symposium on Software Reliability Engineering (ISSRE'02), 2002.
16. R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong, "Orthogonal defect classification-a concept for in-process," vol. - 18, pp. - 956, 1992.

17. Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon, "MuJava: An Automated Class Mutation System," *The Journal of Software Testing, Verification, and Reliability*, 2004 to appear.
18. Duraes, Definition of software fault emulation operators: a field data study