Generation of an Error Set that Emulates Software Faults Based on Field Data

J. Christmansson[Ψ]
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
joc@ce.chalmers.se

R. Chillarege
Center for Software Engineering
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
ramchill@watson.ibm.com

Abstract

A significant issue in fault injection experiments is that the injected faults are representative of software faults observed in the field. Another important issue is the time used, as we want experiments to be conducted without excessive time spent waiting for the consequences of a fault. An approach to accelerate the failure process would be to inject errors instead of faults, but this would require a mapping between representative software faults and injectable errors. Furthermore, it must be assured that the injected errors emulate software faults and not hardware faults. These issues were addressed in a study of software faults encountered in one release of a large IBM operating system product. The key results are:

- *A general procedure that uses field data to generate a set of injectable errors, in which each error is defined by: error type, error location and injection condition. The procedure assures that the injected errors emulate software faults and not hardware faults.*
- *The faults are uniformly distributed (1.37 fault per module) over the affected modules.*
- *The distribution of error categories in the IBM operating system and the distribution of errors in the Tandem Guardian90 operating system reported in [14] were compared and found to be similar. This result adds a flavor of generality to the field data presented in the current paper.*

1. Introduction

In the past decade, fault injection [2],[5],[11] has emerged as an attractive approach to the validation of dependable systems. Fault injection can be used for studying the effects of hardware faults and software faults. However, in both the academic community and industry, most fault injection studies have been aimed at studying the effects of physical hardware faults, i.e. faults caused by wear-out or external disturbances. Only a few studies have been concerned with software faults, e.g. [10],[13], for the reason that knowledge of software faults experienced by systems in the field is limited, making it difficult to define realistic fault sets to inject. This is crucial if an experiment is intended to estimate how well a system is working (fault forecasting). The aim of fault forecasting might be to estimate coverage, which is an important parameter in many analytical dependability models.

Another important factor is the time used, as we want experiments to be conducted without excessive time spent waiting for the consequences of a fault. Much time can be spent conducting a fault injection experiment if the injected faults are rarely activated. Injecting errors rather than faults would be an approach to accelerate the failure process, that is, removing transition 1 (fault activation) and transition 2 (no fault activation) in figure 1. This must be done without affecting the probabilities of the other transitions, e.g. the probability of a failure given that an error exists (transition 3) should be the same as though a fault would have been injected. Furthermore, the injected errors must emulate software faults and not hardware faults. That is, an error can be caused by software faults or by hardware faults. The latter cause must be factored out, as the desire is to emulate software faults.

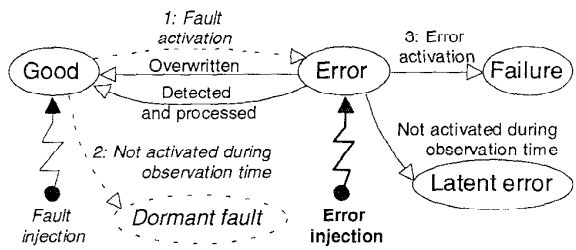

Figure 1. The fault, error, failure process.

Fault forecasting by means of error injection raises a number of important questions?

1. *What* is the appropriate error model that mimics representative software faults?
2. *Where* should the error be injected to emulate a particular software fault?

[Ψ]Was with IBM during the summer of 95.

0731-3071/96 $5.00 © 1996 IEEE
Proceedings of FTCS-26

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

3. *When* should the error be injected?
4. How should a representative *operational profile* (i.e. a probabilistic description of system usage) be designed that will maintain reasonable experiment times?
5. What *readouts* should be collected, and which *measures* should be calculated?
6. How should the calculated measures be related to *analytical models* of dependability?

Provided that these questions are successfully answered, one can start fault/error injection experiments. The experiments may be conducted using any of the fault injection tools that employ software-implemented fault injection and modify the data segment, e.g. [3],[9],[12],[13],[19]. Now we must find out whether the above questions can be answered.

Questions 5 and 6 are addressed by [1],[2] and [18]. These papers provide the theoretical framework needed to answer the questions. Question 4 is partly answered by work on statistical usage testing [16]. However, questions 1 to 3 have been more or less neglected. The current paper will examine whether and to what extent the field data obtained on software faults answer the first four questions. The conclusion to be drawn from that examination is: Yes, the field data provide the information needed to answer the questions. More specifically, this paper proposes a procedure that uses field data to generate a set of errors (i.e. *error type*, *error location* and *injection condition*), which maintains the distribution of the set of faults observed in the field and assures that the injected errors emulate software faults and not hardware faults. The procedure is general, although it is illustrated with data from a specific release of an IBM operating system. The description in this paper is organized as follows. Section 2 describes the procedure used to gather data on field defects. The first four questions are answered and discussed in section 3. A procedure for generating an error set is proposed and illustrated in section 4, and section 5 concludes the paper.

2. Collection of field defect data

Many studies have dealt with software defects detected during the development phase, e.g. [4],[8]. However, our interest is in faults encountered in a system during field operation. A study of the software defects experienced by the IBM products MVS, IMS and DB2 is presented in [20]. The software faults and resulting errors in the Tandem Guardian90 operating system are presented in [14]. The current study is inspired by the latter two papers.

The sources of the data presented in this paper are two IBM internal databases: Orthogonal Defect Classification (ODC) database and REmote Technical Assistance Information Network (RETAIN). All defects experienced by a large operating system product during a two-year period were extracted from the ODC database, and we are thus looking at the full population of software faults experienced by one release over a period of two years. The corresponding defect reports were also pulled from RETAIN. The reports were used to classify the erroneous system state that a particular software fault caused. This section will first present the ODC attributes used, and second RETAIN, and the error types used to classify the erroneous system states that the selected software faults caused.

2.1 Orthogonal Defect Classification

ODC [6] is a measurement technology that is consistently applied to a large number of IBM projects, and the amount of field data is growing. The ODC data contain several attributes of the defects, which are used to provide feedback to the development and verification of a product. ODC can therefore be used as a framework for the evolution of fault and error models. In the current study, two ODC attributes were used to depict a defect: fault type and system test trigger. The selection of the fault type is implied by the eventual correction, and the type is usually chosen by the person that makes the correction. The system test trigger describes the environmental conditions that made the fault surface during operational usage in the field.

2.1.1 Fault types

The fault type represents the defect in the source code, i.e. the cause of an error. ODC employs the following six fault types related to code:

Assignment, value(s) assigned incorrectly or not assigned at all.

Checking, missing or incorrect validation of parameters or data in conditional statements.

Algorithm, efficiency or correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change.

Timing/Serialization, necessary serialization of shared resources is missing, wrong resource has been serialized or wrong serialization technique employed.

Interface, communication problems between users, modules, components or device drivers and software.

Function, affects a sizeable amount of code and refers to capability that is either implemented incorrectly or not implemented at all.

2.1.2 Software triggers

Software triggers are the broad environmental conditions or activities that work as catalysts to assist faults to surface as failures [7]. There are three classes of triggers associated with the three most common activities of verification: Review/inspection, function test and

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

system test. The system test triggers are of interest, as they describe the environmental conditions that were dominant when the fault surfaced in the field, and they are defined as follows:

Startup/Restart, the system was being initialized or restarted due to an earlier shutdown or failure.

Workload volume/Stress, the system was operating near some resource limit, either upper or lower.

Recovery/Exception, an exception handler or a recovery procedure was active. The fault would not have surfaced if some earlier exception had not invoked the handler or the procedure.

Hardware/Software configuration, triggers related to unusual or unanticipated configurations.

Normal mode, a trigger that says that no special conditions must exist for the fault to surface, i.e. the system was working well within upper and lower resource limits. Normal mode trigger implies another underlying trigger, either a review/inspection trigger or a function test trigger.

2.2 RETAIN and error types

RETAIN is maintained by IBM field service and contains world-wide information on hardware problems, software problems, bug fixes, release information etc. Our interest is in software problems during field operation, and these are described by defect reports. Each such report in RETAIN represents a unique defect that surfaced in the field, and contains some standard attributes (e.g. failure severity) of the defect and several pages of text. This text describes the: symptoms of the problem, environmental conditions and correction of the fault. The text was used to classify the erroneous system state that a particular software fault caused.

2.2.1 Failure severity

The failure severity indicates the magnitude of the customer problem, and the following guidelines are used to determine the severity of a field failure.

Severity 1 - The customer is unable to use the product, which has a critical impact on his/her operation.

Severity 2 - The customer is able to use the product, but his/her operations are severely restricted by the problem.

Severity 3 - The customer is able to use the product with some restrictions on the functions that he/she can use. These restrictions, however, do not have a critical impact on his/her operation.

Severity 4 - The problem causes little or no impact on the customer's operation.

2.2.2 Errors

The text in the defect reports pulled from RETAIN was used to classify the erroneous system state that a particular software fault caused. However, the error type was not classified for all defects. The rationales for selecting faults were that they should: (i) be related to the development; (ii) cause an erroneous system state; and (iii) have impact on the user, i.e. severity 1-3. Rationales one and two exclude fault types not related to code (Build/Package/Merge, Documentation), and rationale three, excludes failure severity 4. Thus, the full population of faults was partitioned into: a relevant part (408 defects) and an irrelevant part (279 defects). The erroneous states are grouped into the following four error categories:

Single address error (A). An error falls into this category if the software fault caused an incorrect address word. Errors in this category were further divided into the following types of address word: control block address, storage pointer, module address, linking of data structures and register.

Single non-address errors (N). An incorrect non-address word (e.g. data word) is caused by the software fault. This error was divided into the following types: value, parameter, flag, length, lock, index and name.

Multiple errors (M). A software fault can generate multiple errors at the same time. The errors generated can be any combination of single errors or be related to a data structure. These errors were divided into the following types: values, parameters, address and something else, flag and something else, data structure and random. The observant reader might ask how a multiple error of address and flag was classified, and the answer is simply: we did not encounter such an error. However, this is an oversight that must be corrected before the next empirical study is conducted.

Control errors (C). Some of the software faults analyzed affected the memory content in a very subtle and non-deterministic way. Furthermore, some faults did not affect the memory at all, e.g. interaction with a user or terminal communication. The resulting errors were divided into the following types: program management, storage management, serialization, device management, user I/O and complex.

The first three error categories are taken from [14]. The similarities of the categories will enable a comparison between a large IBM operating system product and the Tandem Guardian90 operating system. In the current paper, each of these three error categories is supplemented with an error type that gives more specific information on the location of the error (e.g. control block address).

The fourth error category, control errors, is special as these errors affect the memory in a non-deterministic way, that is, it was not possible to identify and classify the error in any of the first three categories. However, it was possible to capture the incorrect internal behavior from the textual description in the defect reports, and the error

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

types of this category depict how that behavior could be provoked. For instance, the error type storage management tells us that one module deallocates a region of memory before it has completely finished using the region. Thus, that behavior could be emulated via an error injection in a table that manages the memory allocation/deallocation.

3. Discussion of the questions

3.1 *What* error model?

This subsection will first compare the distribution of error categories for a large IBM operating system product and the Tandem Guardian90 operating system. It then, presents the fault-error mapping observed in a large IBM operating system and, finally, argues that the data answer the first question.

A large IBM OS		Tandem GUARDIAN90	
Categories	%	Categories	%
Single address	19	Single address	23
Single non-addr.	38	Single non-addr.	38
Multiple	17	Multiple	18
Control	26	Other (9 %) Unclear (12 %)	21

Table 1. Comparison with [14].

Table 1 compares the distribution of error categories for the operating system considered in the current paper and the Tandem Guardian90 operating system [14]. Although these systems are quite different, an interesting observation can be made: the variation of the distributions is not significant ($\chi^2_{0.05}(3)=7.82 > q(3)=2.07$, $p=0.56$)[1]. This suggests that there may be some similarities in the distribution of the error categories for large, robust operating systems. The casual reader might ask whether the classifications are made according to the same definitions of error categories. Recall the description of errors (see section 2.2.2) and note that the we used Lee's definition of the first three error categories. Moreover, to avoid bias, all erroneous system states were classified before the statistics was computed.

Thus, this finding adds a flavor of generality to the data presented in table 2 that are used to answer the first question: *What* is the appropriate error model that mimics representative software faults. Table 2 shows the mapping between the six fault types and the four error categories and, more specifically, the 24 fairly general error types. The bottom row in this table shows the distribution of all relevant faults experienced by a particular release during a two year period. The table gives the distribution of the resulting error types in the second column.

Clearly, these data are representative of the faults caused by this particular release of the product, as they show the observed joint distribution. Whether these data are representative of the faults caused by all versions of the product is another issue. However, the considered version happens to be quite mature, and we claim that the data are representative of faults in mature versions of this operating system. Furthermore, recall the comparison in table 1, showing that the difference in the distribution of error categories is not significant. We do not claim, however that the measured distributions is representative for faults and errors in other systems. Nevertheless, it might by applied to other large robust operating systems if no field data are available. Thus, table 2 clearly supplies an appropriate error model. That model contains 24 different error types, however, and we would prefer to mimic a particular fault distribution with fewer error types. This is due to practical issues, i.e. we want to reduce the complexity of the error set generation and the error injection mechanisms.

A heuristic approach would be to select: (a) the 2-3 *most frequent* error types within each error category, and (b) the error type(s) that has a *large impact* on failure severity. For instance, errors caused by timing/serialization defects are responsible for 30% of the severity 1 failures, although they constitute only 13.2 % of the 408 defects. Applying these heuristics results in the following 12 error types gives:

a) The 2-3 most common within each error category[2] - `et1`, `et2`, `et6`, `et7`, `et8`, *`et15`*, *`et16`*, `et17`, `et19`, `et20` and *`et21`*; and

b) Large impact - `et10`, highly related to timing/serialization defects (see table 2), which have a large impact on the system. Furthermore, `et10` will enable the injection of `et21`, which is also highly related to timing/serialization defects.

These 12 selected error types (50% of the error types) will emulate the mapping between fault types and error categories (see table 2) fairly exactly. Furthermore, they will cover 79.4% of the observed defects and, more specifically, 88.9% of the timing/serialization defects. The conclusion is that the 12 selected error types will be an appropriate error model, and the differences observed in the joint distributions can be reduced during the generation of the set of errors that will be injected (see section 4.1).

[1] Formalism used for the c2 test [17]: $c2\alpha(df)$, critical value of chi-square at the α significance level; $q(df)$, chi-square value computed from the observed data; and p, probability of occurrence for $q(df)$.

[2] Note that the error types in italics can be injected with mechanisms for other error types, e.g. et15 by et1 and et6.

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

Error types	#	%	Fault types					
			ft1. Check.	ft2. Assign.	ft3. Algorit.	ft4. Tim/se.	ft5. Interf.	ft6. Func.
Single addr. (**A**)	78	19.1	13	27	26	4	5	3
et1. Cntr. block addr.	(48)	(11.8)	(9)	(16)	(17)	(3)	(2)	(1)
et2. Storage pnt.	(14)	(3.4)	(1)	(5)	(3)	(1)	(2)	(2)
et3. Module addr.	(9)	(2.2)	(3)	(3)	(3)	(0)	(0)	(0)
et4. Link. data struct.	(4)	(1.0)	(0)	(0)	(3)	(0)	(1)	(0)
et5. Register	(3)	(0.7)	(0)	(3)	(0)	(0)	(0)	(0)
Single non-addr. (**N**)	155	38.0	30	38	53	12	15	7
et6. Value	(38)	(9.3)	(10)	(6)	(12)	(3)	(2)	(5)
et7. Parameter	(38)	(9.3)	(8)	(11)	(10)	(1)	(6)	(2)
et8. Flag	(37)	(9.1)	(7)	(10)	(17)	(0)	(3)	(0)
et9. Length	(15)	(3.6)	(4)	(4)	(4)	(0)	(3)	(0)
et10. Lock	(11)	(2.7)	(0)	(1)	(2)	(8)	(0)	(0)
et11. Index	(8)	(2.0)	(1)	(3)	(4)	(0)	(0)	(0)
et12. Name	(8)	(2.0)	(0)	(3)	(4)	(0)	(1)	(0)
Multiple (**M**)	69	16.9	9	6	32	6	4	12
et13. Values	(4)	(1.0)	(0)	(1)	(2)	(0)	(0)	(1)
et14. Parameters	(3)	(0.7)	(0)	(0)	(0)	(0)	(3)	(0)
et15. Address +	(7)	(1.7)	(1)	(1)	(3)	(0)	(0)	(2)
et16. Flag +	(10)	(2.4)	(2)	(1)	(4)	(2)	(0)	(1)
et17. Data structure	(37)	(9.1)	(6)	(3)	(20)	(3)	(1)	(4)
et18. Random	(8)	(2.0)	(0)	(0)	(3)	(1)	(0)	(4)
Control error (**C**)	106	26.0	10	7	43	32	5	9
et19. Program manag.	(35)	(8.6)	(4)	(0)	(19)	(8)	(2)	(2)
et20. Storage manag.	(33)	(8.1)	(3)	(7)	(12)	(5)	(1)	(5)
et21. Serialization	(16)	(3.9)	(0)	(0)	(2)	(14)	(0)	(0)
et22. Device manag.	(11)	(2.7)	(2)	(0)	(7)	(2)	(0)	(0)
et23. User I/O	(6)	(1.5)	(0)	(0)	(2)	(0)	(2)	(2)
et24. Complex	(5)	(1.2)	(1)	(0)	(1)	(3)	(0)	(0)
Total #	408	100	62 (15.2)	78 (19.1)	154 (37.8)	54 (13.2)	29 (7.1)	31 (7.6)

Table 2. Mapping faults - errors[3].

Now, the reader might ask: we know the error types and with what distribution they should be injected, but why are we concerned with the joint distribution between faults and errors? The reason is that we want to mimic software faults and not hardware faults. That is, an erroneous system state can be caused by software faults or by hardware faults. If we simply injected the error types without any connection to a fault (i.e. fault type and fault location), then we could not claim that the injections emulate software faults in a controllable way. Thus, the error model is really a fault and error model. We will elaborate on this when the second question is discussed.

3.2 Where should the error be injected?

This subsection will first show how the defects were distributed over the components (i.e. subsystem) in the system and, second, claim that the data support a random generation of error locations that represent particular fault locations.

Table 3 shows how the 408 faults were distributed over the 48 affected components. The table also presents the number of modules affected in each component. It can be seen that 59.5% of the faults were contained in eight of the affected components. However, it was not possible to pinpoint any module as particularly fault-prone, and the average ratio was 1.37 fault per affected module. The faults experienced are uniformly distributed. That is, the null hypothesis: *the faults are uniformly distributed over the affected modules*, cannot be rejected ($\chi^2_{0.05}(8)=15.51 > q(8)=7.34$, $p=0.50$).

The second questionb - *where* should the error be injected to emulate a particular software fault? - is addressed by the data provided by tables 2 and 3. Table 3 shows how the defects are distributed over the components. It also presents the number of affected

[3] The reader might be concerned with the classical problem of zeros in the cells. However, this is not an issue as the table presents a full population and not a sample.

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

modules and the resulting defect density. For instance, 7.8% of the defects were contained in component D. Furthermore, the 28 modules affected in that component had 1.14 faults/module. Thus, the data give us a good, high-level understanding of how to distribute the injections over the system components and modules. For instance, if the system were subjected to 1000 injections, then about 78 faults would be injected into component D. However, to select the actual error locations, we must consult the data on the fault-error mapping provided by table 2.

Component	# of faults	%	# of affected modules	Fault / module
A	43	10.5	33	1.30
B	35	8.6	31	1.13
C	33	8.1	20	1.65
D	32	7.8	28	1.14
E	30	7.4	24	1.25
F	25	6.1	17	1.47
G	25	6.1	23	1.09
H	20	4.9	14	1.43
The other 40 comp.	165	40.5	108	1.53
Total	408	100	298	1.37

Table 3. The components that contained the faults.

That joint distribution can be used to select actual locations for injection from a list of all possible locations. The list of possible locations in the modules that are to be subjected to injection can be generated by a parser. The parser would scan through the modules and, for each statement, generate the triplet: (fault location, fault type, error type). However, the function defects can probably not be identified by a parser. The person setting up an experiment must identify these manually, using the functional requirements on a component or a module. The actual fault locations can be randomly selected from the list of triplets according to the joint distribution provided by table 2. For instance, the probability of selecting a location with the pair: (`ft1`. checking, `et6`. value) would be 2.45% ($10/408$). When the actual fault locations have been selected, loader information can be used to produce a list of actual error locations. Thus, the field data provide the joint distribution needed for a random selection of error type and error location that maintains the distribution of the emulated faults.

3.3 W*hen* should the error be injected?

The system clock could be used to inject errors at randomly selected times. This simple, straightforward solution would clearly produce errors with the desired distribution. However, as mentioned earlier, errors can be caused by software faults or by hardware faults. We intend to emulate software faults, and the randomness of hardware-induced errors must therefore be factored out. Consequently, the error injection must be synchronized with the execution of the emulated statement, i.e. the injection is *even-driven*. Thus, the list of selected errors (type and location), the related mimicked fault (type and location), can be used to generate the *conditions for injection*. This will enable the construction of software traps (e.g. breakpoints used by debuggers) that will synchronize[4] the actual error injection with the execution of the emulated fault location. The usage of these traps will result in two things: a controlled emulation of software faults and a reduction of the proportion of overwritten errors.

3.4 Design of an *operational profile*

Making a trustworthy parameter estimate (e.g. error detection coverage) depends on exercising the product as though it were in the field. We have already covered the error set; now let's consider the generation of test cases. This problem was addressed by work on statistical usage testing (SUT) [15]. In SUT, a probabilistic description of system usage is captured by an operational profile. When a profile is available, sampling techniques are used for the generation of test cases. Do our data support this?

Table 4 gives the distribution of error categories for the *environmental conditions* that made the defect surface during actual usage. For instance, the 78 single address errors (A) are distributed as: 60.2%, 2.6%, 5.1%, 30.8% and 1.3%, for normal mode, startup/restart, workload/stress, recovery/exception and HW/SW configuration, respectively.

			Error categories			
Trigger	#	%	A	N	M	C
Normal mode	284	69.6	47	120	49	68
Startup/Restart	14	3.4	2	1	5	6
Workload/Stress	27	6.6	4	11	3	9
Recovery/Except.	75	18.4	24	19	12	20
HW/SW Configt.	8	2.0	1	4	0	3
Total #	408	100.0	78	155	69	106
		(%)	(19.1)	(38.0)	(16.9)	(26.0)

Table 4. System test triggers by error categories, i.e. Addr.(A), Non-addr.(N), Multiple(M) and Control(C).

[4] We are concerned with the delay introduced by the trap, e.g. a task might miss its deadline. Therefore, hardware support for synchronization would be very convenient, e.g. a watch-point function that is able to detect a predefined bit pattern on an address bus and a data bus, and whose detection signal is connected to a maskable interrupt.

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

Clearly, the data on the system test triggers (see table 4) support the design of an operational profile that represents the *environmental conditions* that made a defect surface. These triggers capture the mix of the customer environment and thereby provide input to the process of developing an operational profile. This process is described in [16]. However, using test cases sampled from an operational profile might result in excessive time being wasted, as the error injection is synchronized with the execution of the emulated fault location. Consequently, one experiment run is divided into two subsequent parts: (i) execution of the path in which the emulated fault would have been located and (ii) representative system usage to obtain valid estimates of execution parameters. The test cases used during part one will, based on the list of fault locations, be selected according to path-based testing techniques. An example of such usage is [13], which successfully uses these testing techniques to activate faults injected into Sun NFS. Input data to the latter part will be generated as stipulated by SUT [15], i.e. sampled from the operational profile. Thus, time is not wasted waiting for an error injection, and the system usage after injection will be representative of field usage.

4. Generation of the error set (what, where and when?)

The data on field defects can, as concluded above, be used to answer four important questions related to error injection. This section will outline a procedure that answers the first three questions, i.e. *what* model, *where* to inject and *when* to inject? The fourth issue, operational profile and generation of input data that exercises particular paths, will not be exemplified, as that issue is thoroughly treated by the testing community. The procedure for generating the error set can be semi-automated, and is:

1. Select components, e.g. component A-H in table 3.
2. Distribute the defects over the selected components according to measured distribution, e.g. table 3 suggests that component A should be exposed to 10.5% of the faults.
3. Distribute the defects of one component randomly among its modules according to measured distribution, i.e. table 3 indicates that the faults should be uniformly distributed.
4. Generate a list of *possible fault locations* within each selected module, e.g. using a parser. Each location in the list is related to a fault type and resulting error type(s).
5. Select the *actual fault locations* within each module randomly according to the measured joint distribution between fault types and error types (e.g. table 2) from the lists of all possible fault locations.
6. Generate a list of *actual error locations* using the list of actual fault locations and loader information.
7. Decide *when* the errors should be injected, based on the list of actual fault locations, i.e. construct the software traps (*injection condition*).

However, although this procedure is general, it has some limitations: identification of function defects requires manual intervention, and representative data on field faults must be available as the field data presented is not representative for all software systems. The latter limitation is shared with all techniques for the generation of representative fault/error sets.

The following subsections will present a simple example that is meant to illustrate steps 4 to 7 in the procedure for generating the error set. A module (traffic light task, see figure A in the appendix) of a distributed computer control system for traffic control will be used as an example. That module is part of the intersection signal control component. The underlying run-time system will not be considered in this example. Thus, only a subset of the error types given in table 2 is injectable in the traffic light task: value, parameter, flag, values and flag +. Nevertheless, the traffic light task will be used to illustrate some aspects of the design of an error injection experiment.

4.1 Error type and error location (what and where?)

Steps 4 to 6 are relevant to the selection of error type and error location in a module. The prerequisite of steps 4 to 6 in the error-generating procedure is that step 3 has decided that the traffic light task will be subjected to injection.

Step 4: A list of *possible fault locations* is generated from the source code. The source code is given in figure A, and the resulting list of some of the possible fault locations is shown in table 5. The first column in that table shows the locations as the corresponding line number in figure A. The table also gives the fault type and the error type for a particular location. This table can be generated automatically by a parser. However, identification of function defects requires human intervention.

Steps 5 and 6: The *actual fault location* is selected randomly from the list of all possible fault locations, according to the joint distribution between fault types and error types. The mapping between fault types and error types (see table 2) is used to obtain the joint distribution.

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

Fault location	Fault type	Error type
Light,8	ft2. Assign.	et8. Flag
Light,9	ft2. Assign.	et6. Value
Light,11	ft5. Interf.	et7. Parameter
Light,12	ft2. Assign.	et6. Value
Light,16	ft1. Check.	et6. Value
Light,17	ft5. Interf.	et7. Parameter
...	...	...
Light,55	ft2. Assign.	et6. Value
...	...	...
Light,73	ft2. Assign.	et8. Flag

Table 5. Possible fault locations.

Table 6 shows the joint distribution between the fault type and the error categories (extracted from table 2). For instance, assume that we want to inject 1000 errors into the distributed computer control system; then, 74 and 93 single non-address errors will be injected to emulate checking defects and assignment defects, respectively.

% in cells		Errors			
Faults	%	A	N	M	C
ft1. Check.	15.20	3.19	**7.35**	2.21	2.45
ft2. Assign.	19.12	6.62	**9.31**	1.47	1.72
ft3. Algorit.	37.75	6.37	12.99	7.84	10.54
ft4. Tim/ser.	13.24	0.98	2.94	1.47	7.84
ft5. Interface	7.11	1.23	3.68	0.98	1.23
ft6. Function	7.60	0.74	1.72	2.94	2.21
Total %	100.00	19.12	37.99	16.91	25.98

Table 6. Fault types by error categories, i.e. Addr.(A), Non-addr.(N), Multiple(M) and Control(C).

Now, as discussed previously, we want to use a reduced set of error types: for single non-address errors, that set is: et6, et7, et8 and et10. The distribution of those error types over checking and assignment defects is shown in columns one and three in table 7. The number of errors to inject for the emulation of ft1 and ft2 by a reduced set of N errors is shown in columns two and four in table 7. That table is obtained by a simple multiplication of probabilities, for instance, the frequency 29 in cell (ft1, et6) by: $1000 * (30/408) * (10/(10+8+7+0))$.

	ft1. Checking		ft2. Assignment	
Category N, reduced	Prob. (%)	Freq. #	Prob. (%)	Freq. #
et6. Value	2.94	29	2.00	20
et7. Parameter	2.35	24	3.66	37
et8. Flag	2.06	21	3.33	33
et10. Lock	0.0	0	0.34	3
Total	7.35	74	9.31	93

Table 7. Single non-address errors (N) by ft1 and ft2, distribution and injection frequency.

A possible output of steps 5 and 6 is shown in table 8. The table shows the selected error locations and the emulated fault type and fault location. Note that only errors are injected, although the relation to fault type and fault location is needed to assure that software faults are emulated.

Error location	Error type	Fault type	Fault location
REQ_COLOR	et6.	ft1.	Light,16
VEHICLE_LIGHT	et6.	ft2.	Light,55

Table 8. Selected error locations.

4.2 Injection time (when?)

The last step in the generation procedure is to decide the *injection condition.* Errors could be injected at randomly selected times. However, we have two reasons not to make such a choice: reduction of the proportion overwritten errors, and a controlled emulation of software faults. Our approach is therefore to synchronize the injection of an error that mimics a fault type at a particular fault location via software traps.

Step 7: The list of actual fault locations is used to produce the conditions for injection (see table 9). For instance, the injection of et6 at error location VEHICLE_LIGHT related to the write caused by fault location light,55 must take place after VEHICLE_LIGHT has been assigned red, and before time-out Red2Green_Pedestrian has expired. Simply, inject after line 55 (see appendix figure A) has been executed and before line 57 is executed.

Fault location	Condition
Light,16	**not** Error_flag **and** Set_light accepted **and before** Req_color:= Mycolor
Light,55	timeout Green2Red_Vehicle expired **and after** Vehicle_light:= red **and before** timeout Red2Green_Pedestrian expired

Table 9. Conditions for the software traps.

5. Summary of results

This paper provides answers to four key questions related to fault forecasting by means of error injection. The questions posed and resolved are: (i) *what* is the appropriate error model that mimics representative software faults; (ii) *where* should the error be injected to emulate a particular software fault; (iii) *when* should the error be injected; and (iv) how should a representative *operational profile* be designed that will maintain reasonable experiment times?

More specifically, the paper illustrates how the data on all faults experienced by one release of a large IBM operating system product during a two-year period can be

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

used to generate a set of errors that maintains the distribution of the set of faults observed in the field, and the key results are:

1. The experienced faults are uniformly distributed (1.37 fault per module), that is, the null hypothesis: *the faults are uniformly distributed over the affected modules* cannot be rejected ($q(8) = 7.34$, $p = 0.50$). Furthermore, the data show how the faults are distributed over the affected components. Thus, the data give us a good, high-level understanding of how to distribute the injections over the system components and modules.
2. The heuristics proposed that selects 50% of the error types such that 79.4% of the observed defects is covered. Furthermore, the selection assures that 88.9% of the timing/serialization defects are covered, as they tend to cause high severity failures. Thus, the joint distribution between the twelve selected error types and six fault types clearly supplies an appropriate error model.
3. A general procedure that uses field data to generate a set of injectable errors, in which each error is defined by: *error type*, *error location* and *injection time*. Furthermore, the procedure assures that the injected errors emulate software faults and not hardware faults. The main features of that procedure are:
   - The distribution of faults over the affected modules is used to select modules that are to be subjected to error injection.
   - The joint distribution between the error types and fault types is used to sample actual locations for error injection from a list of all possible locations (*error type*, fault type, fault location). The list of possible fault locations in the modules that are to be subjected to injection can be generated by a parser. The actual *error locations* are obtained from the sampled fault locations via loader information.
   - The list of sampled fault locations is used to generate the conditions that control the event-driven error injection, that is, a software trap (e.g. breakpoint) synchronizes the injection with the execution of a selected fault location.
4. One experiment run is divided in two subsequent parts: (i) execution of the path in which the emulated fault would have been located, and (ii) representative system usage to obtain valid estimates of execution parameters. The input data to the first part are chosen such that they satisfy the *injection condition, i.e.* the software trap. The list of fault locations and the trap conditions form the basis of test case selection according to path-based testing techniques. Input data to the latter part will be generated as stipulated by statistical usage testing by means of the system test triggers. Consequently, time is not wasted waiting for an error injection, and the system usage after injection will be representative of field usage.
5. A simple example that gives insight into the procedure for generating the error set. Furthermore, the example indicates how that procedure can be automated.
6. The distribution of error categories for a large IBM operating system and the distribution of errors in the Tandem Guardian90 operating system reported in [14] were compared and found to be similar, as the variation of the distributions was not statistically significant ($q(3) = 2.07$, $p = 0.56$). This result adds a flavor of generality to the field data presented in the current paper.

Acknowledgment

Thanks go to Peter Santhanam, Katty Bassin and Shriram Biyani, IBM, for their help with the field data. We also acknowledge comments by Peter Dahlgren and Marcus Rimén, Chalmers University of Technology.

References

[1] J. Arlat, et al., "Fault Injection for Dependability Validation: A Methodology and Some Application", IEEE Trans. Software Eng., Vol. 16, No. 2, pp. 166-182, February 1990.

[2] J. Arlat, et al., "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems", IEEE Trans. Comp., Vol. 42, No. 8, pp. 913-923, August 1993.

[3] J. Carreira, et al., "Xception: Software Fault Injection and Monitoring in Processor Functional Units", Proc. Fifth IFIP Working Conference on Dependable Computing for Critical Applications, pp 135-149, September 1995.

[4] J.K. Chaar, M.J. Halliday, I.S. Bhandari, R. Chillarege, "In-Process Evaluation for Software Inspection and Test", IEEE Trans. Software Eng., Vol. 19, No. 11, pp. 1055-1070, November 1993.

[5] R. Chillarege, N. S. Bowen, "Understanding Large System Failures - A Fault Injection Experiment", Proc. 19th Int. Symp. Fault Tolerant Computing, pp. 356-363, June 1989.

[6] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements", IEEE Trans. Software Eng., Vol. 18, No. 11, pp. 943-956, November 1992.

[7] R. Chillarege, K.A. Bassin, "Software Triggers as a Function of Time - ODC on Field Faults", Proc. Fifth IFIP Working Conference on Dependable Computing for Critical Applications, pp 188-197, September 1995.

[8] A. Endres, "An Analysis of Errors and Their Causes in System Programs", IEEE Trans. Software Eng., Vol. 1, No. 2, pp. 140-149, 1975.

[9] S. Han, et al., "DOCTOR: An IntegrateD SOftware Fault InjecCTiOn EnviRonment for Distributed Real-time Systems", Proc. IPDS'95, pp. 204-213, 1995.

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE

[10] J. Hudak, B.H. Suh, D. Siewiorek, Z. Segall, "Evaluation & Comparison of Fault-Tolerant Software Techniques", IEEE Trans. on Reliability, Vol. 42, No. 2, pp. 190-204, June 1993.

[11] R.K. Iyer, "Experimental Evaluation", Special Issue FTCS-25 Silver Jubilee, 25rd IEEE Int. Symp. on Fault Tolerant Computing, pp. 115-132, June 1995.

[12] G.A. Kanawati, et al., "FERRARI: A Flexible Software-Based Fault and Error Injection System", IEEE Trans. Computers, Vol. 44, No. 2, pp. 248-260, February 1995.

[13] W.L. Kao, "Experimental Study of Software Dependability", Ph.D. thesis, Technical report CRHC-94-16, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, 1994.

[14] I. Lee, R.K. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System", Proc. 23rd Int. Symp. Fault Tolerant Computing, pp. 20-29, June 1993.

[15] J.D. Musa, A.F. Ackerman, "Quantifying Software Validation: When to stop testing?", IEEE Software, Vol. 6, No. 3, pp. x-y, May 1989.

[16] J.D. Musa, "Operational Profiles in Software Reliability Engineering", IEEE Software, Vol. 10, No. 2, pp. 14-32, March 1993.

[17] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 3rd ed., McGraw-Hill, 1991, ISBN 0-07-100870-5.

[18] D. Powel, et al., "Estimators for Fault Tolerance Coverage Evaluation", IEEE Trans. Comp., Vol. 44, No. 2, pp. 261-274, February 1995.

[19] Z. Segall, et al., "FIAT - Fault Injection Based Automated Testing Environment", Proc. 18th Int. Symp. Fault Tolerant Computing, pp. 102-107, June 1988.

[20] M. Sullivan, R. Chillarege, "A Comparison of Software Defects in Database Management Systems and Operating Systems", Proc. 22nd Int. Symp. Fault Tolerant Computing, pp. 475-484, July 1992.

Appendix

```
1   task type TRAFFIC_LIGHT_TASK is
2     entry INITIALIZE(MYDIR : in direction);
3     entry SET_LIGHT(MYCOLOR: in color);
4     entry RESET;
5   end TRAFFIC_LIGHT_TASK;
6   task body TRAFFIC_LIGHT_TASK is
7     DIR: direction;
8     ERROR_FLAG : boolean := false;
9     REQ_COLOR, PEDESTRIAN_LIGHT,
      VEHICLE_LIGHT: color := flashing;
10  begin
11    accept INITIALIZE(MYDIR:in direction) do
12      DIR := MYDIR;
13    end;
14    loop
15      select
16        when not ERROR_FLAG =>
17        accept SET_LIGHT(MYCOLOR:in color) do
18         REQ_COLOR = MYCOLOR;
19        end;
20        case VEHICLE_LIGHT is
21         when flashing =>
22           if REQ_COLOR = red then
23            PEDESTRIAN_LIGHT := red;
24            VEHICLE_LIGHT := red;
25            delay RED2GREEN_PEDESTRIAN;
26            PEDESTRIAN_LIGHT := green;
27           elsif REQ_COLOR = green then
28            PEDESTRIAN_LIGHT := red;
29            VEHICLE_LIGHT := yellow;
30            delay RED2GREEN_VEHICLE;
31            VEHICLE_LIGHT := green;
32            delay MIN_GREEN_TIME;
33           end if;
34        when red =>
35           if REQ_COLOR = green then
36            PEDESTRIAN_LIGHT := red;
37            delay GREEN2RED_PEDESTRIAN;
38            VEHICLE_LIGHT := yellow;
39            delay RED2GREEN_VEHICLE;
40            VEHICLE_LIGHT := green;
41            delay MIN_GREEN_TIME;
42           elsif REQ_COLOR = flashing then
43            VEHICLE_LIGHT:= flashing;
44            PEDESTRIAN_LIGHT:= flashing;
45           end if;
46        when yellow => ;
47           -- This case should not occur
48           PEDESTRIAN_LIGHT := flashing;
49           VEHICLE_LIGHT := flashing;
50           ERROR_FLAG := true;
51        when green =>
52           if REQ_COLOR = red then
53            VEHICLE_LIGHT := yellow;
54            delay GREEN2RED_VEHICLE;
55            VEHICLE_LIGHT := red;
56            delay RED2GREEN_PEDESTRIAN;
57            PEDESTRIAN_LIGHT := green;
58           elsif REQ_COLOR = flashing then
59            PEDESTRIAN_LIGHT:= flashing;
60            VEHICLE_LIGHT:= flashing;
61           end if;
62        end case;
63      or
64        accept RESET do
65         PEDESTRIAN_LIGHT := flashing;
66         VEHICLE_LIGHT := flashing;
67         ERROR_FLAG := false;
68        end;
69      or
70        delay WATCHDOG_TIMEOUT;
71        PEDESTRIAN_LIGHT := flashing;
72        VEHICLE_LIGHT := flashing;
73        ERROR_FLAG := true;
74      end select;
75    end loop;
76  end TRAFFIC_LIGHT_TASK;
```

Figure A. The code for the traffic light task.

Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing
0731-3071/96 $10.00 © 1996 IEEE