Emulation of Transient Software Faults for Dependability Assessment: A Case Study

Roberto Natella, Domenico Cotroneo
Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
{*roberto.natella, cotroneo*}*@unina.it*

***Abstract*—Fault Tolerance Mechanisms (FTMs) are extensively used in software systems to counteract software faults, in particular against faults that manifest transiently, namely *Mandelbugs*. In this scenario, *Software Fault Injection* (SFI) plays a key role for the verification and the improvement of FTMs. However, no previous work investigated whether SFI techniques are able to emulate Mandelbugs adequately. This is an important concern for assessing critical systems, since Mandelbugs are a major cause of failures, and FTMs are specifically tailored for this class of software faults.**

In this paper, we analyze an existing state-of-the-art SFI technique, namely G-SWFIT, in the context of a real-world fault-tolerant system for Air Traffic Control (ATC). The analysis highlights limitations of G-SWFIT regarding its ability to emulate the transient nature of Mandelbugs, because most of injected faults are activated in the early phase of execution, and they deterministically affect process replicas in the system. We also notice that G-SWFIT leaves untested the 35% of states of the considered system. Moreover, by means of an experiment, we show how emulation of Mandelbugs is useful to improve SFI. In particular, we emulate concurrency faults, which are a critical sub-class of Mandelbugs, in a fully representative way. We show that proper fault triggering can increase the confidence in FTMs' testing, since it is possible to reduce the amount of untested states down to 5%.

***Keywords*-Dependability Assessment, Software Fault Injection, Fault Tolerance, Software Faults, Mandelbugs**

I. Introduction

With the recent growing of software complexity, *software faults* (i.e., *bugs*) represent a significant cause of system failures. In the last decades, very interesting studies on software systems contributed to better understand this kind of faults [1]–[4]. As these confirmed, dealing with software faults is quite a hard task: the main problem is the *reproducibility* of the failure, that is the ability to identify the fault activation pattern. Faults whose activation is reproducible are called *Bohrbugs*. They are typically detected and then fixed during testing phases. *Mandelbugs*, instead, are faults whose activation is not systematically reproducible and they typically lead to transient failure manifestations [5][1]. Their activation conditions (namely, *fault triggers*) depend on complex combinations of user inputs, the internal state, and the external environment, i.e., the set composed by other programs, services, libraries, virtual machines, middleware and operating system the application interact with. The activation conditions of Mandelbugs occur during the system operational phase and can be very difficult to reproduce (e.g., a thread scheduling that triggers a concurrency fault). For this reason, testing activities revealed to be not effective for dealing with such a kind of faults. It is worth noting that Mandelbugs account for a significant part of failures in the operational phase, up to 82% in well-tested critical software [2], [5], [6].

Since it is almost unfeasible to avoid the occurrence of Mandelbugs, critical systems adopt *Fault Tolerance Mechanisms* (FTMs). In fact, failures due to Mandelbugs can be masked by means of spatial redundancy (e.g., replication [2], [7]) and temporal redundancy (a retry of the failed action can result in success [8]). However, FTMs in critical systems have to undertake a thorough assessment before acceptance, which can be achieved by means of *Software Fault Injection* (SFI). By deliberate injection of software faults, fault tolerance can be tested and potentially improved [9], [10]; SFI is also useful to assess the impact of software faults on the system during the operational phase [11].

Although several past works focused on software fault emulation techniques [12]–[14], to the best of our knowledge no previous work investigated whether SFI techniques are able to emulate Mandelbugs adequately. We claim that emulation of Mandelbugs is an important concern for assessing critical systems, since Mandelbugs are a significant cause of failures. In this paper, we analyze an existing state-of-the-art SFI technique, namely G-SWFIT [14], in the context of a real-world fault-tolerant system for Air Traffic Control (ATC). In particular, we investigate whether faults injected by G-SWFIT are representative of Mandelbugs. Our analysis is supported by a Finite State Machine (FSM) model of the system, and we use a quantitative metric, namely *state coverage*, to evaluate the confidence of SFI campaigns.

The mentioned analysis highlights limitations of G-SWFIT regarding its ability to emulate the transient nature of Mandelbugs. In fact, most of injected faults are activated in the early phase of execution, and they deterministically affect process replicas in the system. This problem is not easy to be solved, since emulation of Mandelbugs, in addition to fault injection into the executable code, also requires the emulation of fault triggers that characterize them. By

[1]In the taxonomy proposed in [5] the term "Heisenbugs", which was adopted in some past research, denotes the subset of Mandelbugs that change their behavior when probed or isolated.

978-0-7695-4007-8/10 $26.00 © 2010 IEEE
DOI 10.1109/EDCC.2010.13

means of an experiment, we show how emulation of fault triggers is useful to improve SFI. In particular, we focus on the emulation of concurrency faults, which represent a critical sub-class of Mandelbugs because they are hard to be detected [15], [16], they have a severe impact on system availability and reliability [3], [4], and because of the current shift towards multi-threaded software architecture [17]. The representativeness of emulated faults is assured by a field data study on real concurrency faults, and by the precise emulation of their activation conditions (using a technique specifically designed for this purpose).

The case study considered in this paper consists of a mission-critical system for ATC, namely Flight Data Processor System (FDPS), and the middleware on which it is based, namely CARDAMOM[2]. Both CARDAMOM and FDPS have been developed in the framework of an Italian research project, namely COSMIC[3]. From experimental results, we notice that G-SWFIT cannot achieve full state coverage in the case study, leaving untested 35% of states. Moreover, we demonstrate that, by means of proper fault triggering, it is possible to reduce the amount of untested stated down to 5%. In turn, a higher state coverage provides an increased confidence in FTMs' testing.

The paper is organized as follows. Section II discusses relevant work on SFI. Section III describes the case study considered in this work. In section IV, G-SWFIT is analyzed with respect to the mentioned aspects. In section V, we describe a technique for injecting concurrency faults, which is evaluated in section VI. Section VII closes the paper.

II. Related Work

A. State-of-the-art in Software Fault Injection

Software Fault Injection roots date back to 70s, during which *mutation testing* was developed [18]. Mutation testing is based on the transformation of elementary components in a program, such as changing variable names or arithmetic operators in expressions. Synthetically generated faulty programs (namely, *mutants*) are used (i) to create a set of test cases, in the hypothesis that test cases effective against mutants will be able to detect real faults, and (ii) to evaluate and compare the adequacy of testing strategies. Several mutation operators have been proposed, which deal with most common programming constructs. However, since mutants do not occur with equal probability in real software systems, the problem of *fault representativeness* arose in the last decades [19]. Representativeness is a relevant concern for dependability evaluation, since it is a necessary condition to obtain meaningful results (e.g., to evaluate the *fault coverage* of FTMs, the occurrence rate of different faults has to be taken into account).

[2]http://cardamom.objectweb.org
[3]http://www.cosmiclab.it

In several past works, SoftWare-Implemented Fault Injection (SWIFI) was used to emulate both hardware and software faults [10], [20]; this technique consists in the byte-level alteration of register and memory locations at runtime, in order to emulate high-level software faults (e.g., by changing the destination register of an opcode to emulate an incorrect assignment operation) and errors due to software faults (e.g., data corruption). However, this approach does not achieve fault representativeness for two reasons. First, it has been demonstrated that SWIFI is able to emulate only a subset of software faults, due to the semantic gap between high-level programs and their binary translation. Moreover, the error model commonly adopted by SWIFI is only representative of errors due to hardware faults (e.g., bit-flips). Second, injection of faults and errors in memory at arbitrary time does not reflect the characteristics of software faults, which are permanent design/implementation faults into the source code. [13], [21]

Subsequent works pursued error injection by (i) designing a formal framework for error injection experiments, (ii) corrupting the memory state using a realistic error model, and (iii) triggering error injection every time that a target source-code statement (supposed to be faulty) is executed [9], [12], [22]. Error injection techniques are convenient in terms of time and efforts devoted to experiments. Nevertheless, they still lack of a precise mapping between injected errors and source-code faults [13]; they cannot guarantee that an error could have been caused by a residual software fault into the code. This issue also affects *robustness testing*, i.e., testing against exceptional inputs [23], which is not suitable for accurately reproducing the effects of residual software faults in a system [24].

The *Generic Software Fault Injection Technique* (G-SWFIT) [14] was the first proposal in the literature that achieved fault representativeness, by means of a field study on real software faults. It consists of (i) a set of *fault operators*, that is representative mutation operators, and (ii) a technique for injecting software faults at the machine-code level. It was found that a small set of fault operators is able to emulate most of faults occurring in several real-world programs; therefore, G-SWFIT can be used on systems for which field data are not available, as in the case of third-party software [11]. However, no previous work investigated how well faults injected by G-SWFIT emulate Mandelbugs.

B. Fault-Tolerance Testing

Fault injection has been extensively used to evaluate and improve FTMs in hardware and software systems. Therefore, theoretical frameworks have been proposed in past works to make the fault injection process systematic. In [25], [26], formal testing approaches for fault-tolerant protocols are proposed; they are based on a state model of the fault-tolerant system, namely *execution tree*. This model is an abstraction of the events that may occur (i.e., both correct and

faulty inputs) and the actions the system should undertake for each event. The model is used to generate and monitor events (by means of appropriate system instrumentation) for testing an actual implementation of a protocol (e.g., with respect to wrong messages), and to assess the coverage of the tests. In [27], a methodology for conformance testing by fault injection (CoFI) is evaluated in the context of software for space applications. In CoFI, state-based models are derived from specifications, for each service provided by the software. A model describes the expected behavior with respect to normal inputs and external faults; SWIFI is used to inject communication, processor, and memory faults, in order to check the conformance of the system to the specification. In [28], a fault injection tool for distributed systems evaluation (Loki) is described. The tool is designed to trigger faults (e.g., a process crash) during a specific state, with low intrusion and high precision; these goals are achieved by means of partial view of global state and optimistic synchronization (i.e., the tool assumes that the nodes are correctly synchronized, and it performs off-line filtering of experiments in which fault injection is wrongly triggered). In [23], a robustness testing technique is proposed for triggering faulty input injection at the interface between drivers and the operating system. The technique is based on the preliminary analysis of distinct sequences of function calls to a driver (namely *call blocks*); an experiment is performed several times, once for each call blocks containing the target function call. It is shown that triggering injection using call blocks can identify an higher number of robustness vulnerabilities than first-occurrence triggering.

Although the mentioned works demonstrated the usefulness of proper triggering in fault injection experiments, to the best of our knowledge there is not a study about the triggering of injected software faults; instead, existing approaches focus on SWIFI and robustness testing. In this paper, we consider a fault model representative of Mandelbugs, in which the fault activation is taken into consideration together with source-level fault injection; this feature enables to test fault tolerance under different system states.

III. Case Study

This section describes the fault tolerant system considered in this work. In particular, the emphasis is on FTMs adopted by the FDPS, which are provided by the underlying middleware platform (CARDAMOM). In the following sections, we analyze the effectiveness of SFI (both G-SWFIT and Mandelbugs emulation) for testing FTMs of this system.

A. CARDAMOM middleware

CARDAMOM is a middleware platform that provides features to configure, deploy and execute near real-time, distributed and fault-tolerant applications. It is a CORBA-based, OMG compliant platform supporting both the object and the component programming models.

CARDAMOM lies on several off-the-shelf components; in particular, it makes use of the Linux operating system, and the TAO ORB[4]. Moreover, the platform includes an implementation of the OMG Data Distribution Service (DDS) standard for publish-subscribe communication, namely RTI DDS[5]. DDS enables data sharing among distributed processes, without concern for their actual physical deployment.

Two CARDAMOM services are relevant in the context of this paper: the *Fault Tolerance* (FT) and *Load Balancing* (LB) services. The FT Service is compliant to the FT CORBA Specification. It provides redundancy by means of CORBA object replication; when a faulty primary replica is detected (e.g., a replica is terminated unexpectedly), the FT Service elects a new primary replica from a pool (namely *object group*). CARDAMOM implements the *warm passive* replication style, i.e., when the state of the primary replica is modified, it gets recorded and transferred to other members in the object group. The fault-tolerant application is responsible for maintaining consistency between replicas, by means of an API provided by CARDAMOM.

The LB Service allows the distribution of CORBA requests among the members of an object group. A request is redirected to one of the servers, according to an user-defined policy (e.g., Round-Robin, Random). The LB Service is transparent from the client point of view.

B. FDPS

FDPS is a C++ application based on CARDAMOM. It represents the part of an ATC system in charge of managing Flight Data Plans (FDPs). An FDP is a data structure containing information about a flight; the goal of FDPS is to keep FDPs up-to-date. For example, FDPS has to analyze the actual position of aircrafts (retrieved from radar tracks) and update flight routes consequently, in order to efficiently allocate the flight space and to avoid flight collisions. A simplified view of FDPS architecture is shown in Figure 1.

FDPS is composed by a Façade component, replicated by the FT Service, and a set of Processing Servers (PSs), managed by the LB Service. The Façade is in charge of interfacing the FDPS with external systems (e.g., a graphical user client), and to manage the state of FDPs. When the Façade receives a FDP request, it locks the FDP (at most one request at a time for the same FDP can be processed) in an internal data structure (namely, the FDPs table), and it sends a processing request to a PS. The PS retrieves the FDP and radar tracks from the DDS, processes the FDP, and returns the FDP to the Façade. The Façade then updates the FDP on DDS and unlocks it. If several requests are sent to the same PS, they are executed one at a time. The Façade is responsible for queueing concurrent requests, and for checkpointing the FDPs table. For the sake of simplicity, the

[4]http://www.theaceorb.com
[5]http://www.rti.com/products/dds/

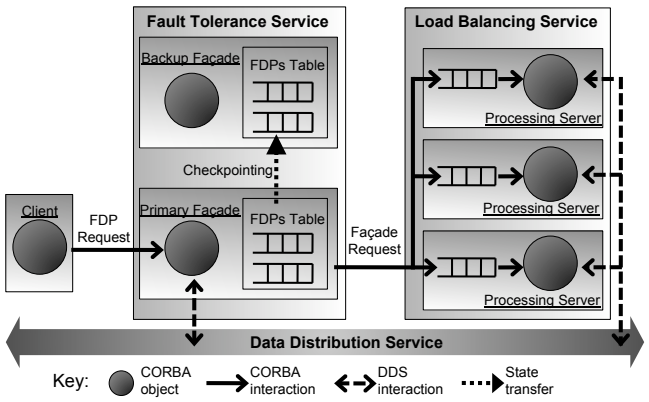

Figure 1. Simplified architecture of FDPS.

only FDP operations we take into account are the insertion, deletion, and update of an FDP.

IV. Evaluation of G-SWFIT

In order to study how G-SWFIT performs with respect to state coverage, we first model the FDPS using a Finite State Machine (FSM). State-based analysis is relevant for dependability assessment, since a fault trigger can unpredictably occur in any system state, even after a long period of uninterrupted execution; in turn, system behavior under faulty execution, and correct failure detection and recovery, depend on the current state. In a FSM, each state represents a different combination of internal system variables. States are connected by transitions, which represent events (e.g., an external input) that change the value of internal variables. When the state is changed, the system performs a computation in answer to the occurred event.

The choice of internal variables is a trade-off between the accuracy and the complexity of the model. A too accurate model suffers from the explosion of the number of states, which makes the analysis unfeasible. Therefore, we select a proper subset of system variables and of their possible values, in order to keep the number of states low, and to still take into account the features of the system most relevant to fault tolerance testing. The considered variables are:

$\#QF$ Number of FDP requests queued by the Façade;
$\#UP$ Number of Façade requests under processing;
$\#QP$ Number of Façade requests queued by PSs.

Moreover, we consider messages exchanged within the FDPS (shown in Table I) as transition events, because they are easy to be collected, and they enable to track the values took by the considered variables.

To make the model finite, the number of requests that can be queued by the Façade is bounded ($\#QF \leq 3$), without any loss of generality. For the same reason, we also choose to assign only two possible values to $\#UP$, respectively the absence of requests queued by PSs ($\#UP = 0$), and the presence of one or more requests queued ($\#UP = 1$). Instead, the number of requests under processing is bounded by the number of PSs ($\#UP \leq 3$ in the current FDPS architecture). The chosen internal variables and events do not take into account which particular FDPs or FDP operations are under processing, but only how many FDPs or FDP operations are involved, and whether a FDP operation involves an already queued FDP (CRQ, FRQ, PSCQ) or not (CR, FR, PSC). This choice reduces the number of states from several thousands to 20 (Figure 2).

Table I
Messages exchanged within the FDPS.

Name	Description
CR	A Client request for an FDP not already requested
CRQ	A Client request for an FDP already requested
FR	A Façade request for an FDP request not in the Façade queue
FRQ	A Façade request for an FDP request in the Façade queue
PSC	A PS returns an FDP, and no other Façade requests are queued by the PS
PSCQ	A PS returns an FDP, and there are Façade requests queued by the PS

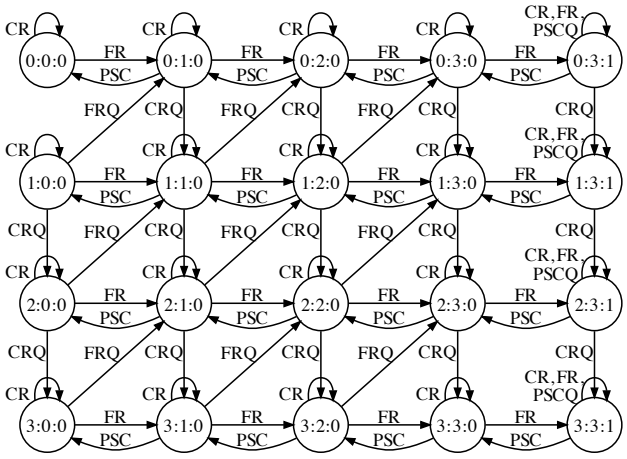

Figure 2. A Finite State Machine that models the FDPS.

In order to study the manifestation of injected faults with respect to system states, the application has been instrumented to log the contents of input and output messages of the Façade (Table I). Moreover, a log message is produced before a faulty piece of code is executed. By analyzing these logs after a fault injection experiment, it is possible to trace which states were reached during an experiment. A failure has occurred if the sequence of Façade states does not match the state sequence of faulty-free runs.

Regarding the workload used for SFI experiments, we designed a set of 3 workloads. Each workload makes the system to reach all states in the FSM (starting from *0:0:0*). The workloads differ in the direction in which the FSM is visited (e.g., row-by-row, column-by-column) and the type

of the requests (insert, update, delete).

As for the faultload, fault operators encompassed by G-SWFIT (Table II) are used to inject faults into the business logic of the Façade. Fault injection is focused on the Façade, since it is the most complex component in the FDPS, and therefore the most fault-prone [11]; moreover, it represents a fundamental entity in the FDPS architecture. We performed fault injection by means of an automated tool, which has been developed by the authors[6]. The tool is based on a C/C++ parser, which seeks suitable fault locations within a source code file and produces a "patch" file for each injectable fault. Each fault represents an individual SFI experiment. The experimental campaign encompasses 533 faults, and 1599 fault injection experiments; in 521 cases we noticed a failure of the primary Façade.

Table II
FAULT OPERATORS (SEE ALSO [14]).

Acronym	Explanation
OMFC	Missing function call
OMVIV	Missing variable initialization using a value
OMVAV	Missing variable assignment using a value
OMVAE	Missing variable assignment with an expression
OMIA	Missing IF construct around statements
OMIFS	Missing IF construct + statements
OMIEB	Missing IF construct + statements + ELSE construct
OMLAC	Missing AND in expression used as branch condition
OMLOC	Missing OR in expression used as branch condition
OMLPA	Missing small and localized part of the algorithm
OWVAV	Wrong value assigned to variable
OWPFV	Wrong variable used in parameter of function call
OWAEP	Wrong arithmetic expression in function call parameter

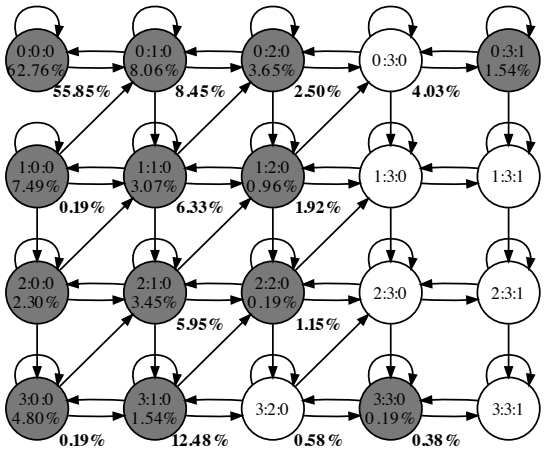

Figure 3. Failure and fault activation distributions of the G-SWFIT campaign. (the percentage of failures for each state is shown in bold; the percentage of faults activated in each state is shown within the node)

For each experiment in which a failure occurs, we consider the state of the system before the failure, and the state in which the faulty piece of code is executed before the failure. Figure 3 shows the 2 distributions of these states. A detailed analysis of the experimental results reveals that:

- A great amount of faults (55.85%) manifest themselves before the Façade is ready to receive and to process requests, or when the Façade receives the first request (state *0:0:0*). Although these faults are useful to test fault tolerance during the initialization phase of FDPS, they are not well representative of Mandelbugs, which can unexpectedly manifest themselves during the operational phase of a system.
- Faults injected by G-SWFIT are useful to test fault tolerance with respect to a subset of important system states. In particular, faults that manifest when at least one request is queued by Façade ($\#QF > 0$) allow testing of the checkpointing mechanism (i.e., whether the FDPs table is correctly sent to the backup Façade) and failure detection mechanisms.
- In most of cases (93.3%) in which the backup Façade is activated, the fault causes the failure also of the replica. This suggests us that injected faults are activated deterministically. In these cases, the fault activation is simple to reproduce (as in the case of Bohrbugs). This result biases evaluation of FTMs, which should instead focus on Mandelbugs in well-tested critical software.
- A significant part of states (35%) is not covered during SFI experiments. Moreover, there are some states in which the percentage of activated faults is very low (i.e., *1:2:0*, *2:2:0*, *3:3:0*), or the faults are activated only under 1 out of 3 workloads (i.e., *3:1:0*, *0:3:1*). Thus, these states are not well tested, since it is unlikely that they would be covered under a different workload. The same holds true for those states that were not covered.
- Part of the injected faults (15.36%, Figure 4) causes a failure only under 1 workload, and a significant part of faults (56.93%) do not cause a failure. Therefore, even if several of the injected faults could be representative of Mandelbugs, they are seldom (or never) activated due to the missing occurrence of their activation conditions.

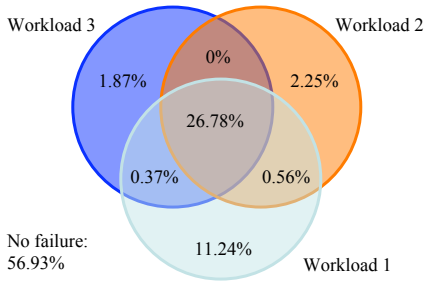

Figure 4. Distribution of faults leading to failures (230 out of 534) among workloads. Overlapping areas are faults activated by more than 1 workload.

Although these results are influenced by the particular

[6]Available at: http://www.mobilab.unina.it/SFI.htm

FSM model and workloads, we can conclude that G-SWFIT alone does not easily allow testing of FTMs under some specific states. In particular, no fault manifests itself when one or more requests are queued by the PSs ($\#QP > 0$). This fact prevents testing of whether the system correctly tolerates a failure in this scenario. We believe that the problem is due to the lack of control on the fault activation. This result justifies the study of whether faults can manifest in the remaining states, and how they can be activated.

V. Concurrency Fault Injection

In order to identify which factors lead to a transient manifestation of Mandelbugs, and the related nature of faults, we analyzed the scientific literature on field data of software faults. Results of field analysis guided us to the design of a SFI technique which, by taking into account the fault activation process, is able to precisely emulate the manifestation of faults.

A. Fault Model

From past works on field data [2], [3], [6], [29], we identified the following transient fault triggers:

- concurrency;
- timing of external events;
- wrong memory state;
- faulty error handling routines (the fault is triggered by another one);
- complex input sequences;
- software aging (e.g., resource leaks).

As discussed in Section I, we focus on faults related to concurrent programming. In order to emulate the most frequent concurrency faults, we take into account the results of a study on concurrency faults [29], which analyzed 105 faults from 4 real complex and concurrent systems. This study pointed out that the most frequent synchronization faults are[7]:

- *atomicity-violation faults* (48.57%), i.e., non-atomic execution of concurrent memory accesses;
- *deadlock faults* (29.52%);
- *order-violation faults* (22.86%), i.e., execution of a set of operations in the wrong order.

In particular, we focus on atomicity-violation faults, since they occur most frequently. Moreover:

- non-deadlock faults involved one variable (66.22%);
- 2 threads are needed for triggering a fault (89.52%).

There are several strategies to enforce the atomicity of a group of memory accesses. However, it is well known that the use of *locks* is the most common strategy used by programmers. Although [29] does not provide the precise amount of atomicity-violation faults fixed by means of lock primitives (i.e., by adding or changing lock operations), the study recognizes that different fix strategies are not mandatory, except when a precise ordering among operations is needed (e.g., order-violation bugs). Because in this work we focus on atomicity-violation faults, we will consider faults related to the wrong usage of lock primitives.

To emulate the features of most frequent atomicity-violation faults, we adopt a fault model in which the access to a shared variable by 2 threads is non-atomic (by omitting lock operations), one of whose is a write access to the variable (in order to cause an interference). This fault is more commonly known as *race condition*.

B. Overview of the Concurrency Injection Technique

In order to emulate the adopted fault model, we adopt a SFI technique consisting of two phases. In the first phase (namely *fault injection*, Section V-C), we identify two critical sections, both containing a memory access to the same shared variable. The fault model requires the removal of lock operations before and after a critical section.

Since this kind of fault, like Mandelbugs in general, has a low probability to be activated, the technique includes a second phase (namely *trigger injection*) for activating the injected fault during a test. There are tools that can be used for triggering concurrency faults, such as Chess [15] and CTrigger [16]; they aim to detect residual faults in concurrent programs, by systematically forcing several thread interleavings. However, they do not assure that the injected fault will be activated. Moreover, in this work we want to investigate whether proper fault triggering can improve SFI, by triggering software faults in those states not covered during the first experimental campaign. To this aim, we design a novel fault triggering technique (Section V-D).

Before fault injection, the technique requires to trace input and output messages. For each message, we profile shared memory accesses and lock usage (Figure 5). This is information is used for removing lock operations between and after conflicting memory accesses, and for identifying which inputs can be used to trigger a conflict in a specific state. Memory and lock profiling can be made by means of profiling tools like Valgrind[8].

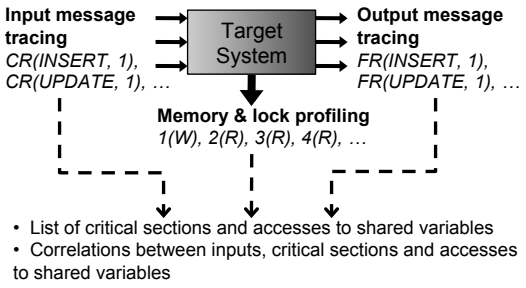

Figure 5. Preliminary data collection.

[7] 2.86% of the faults are classified as both atomicity- and order-violation.

[8] http://www.valgrind.org

C. Fault Injection Phase

In the fault injection phase, the information about memory accesses and lock operations is analyzed to identify those critical sections that can be conflicting. In particular, the analysis is based on the list of memory accesses to shared variables, the type of access (read or write), and the set of locks held during the access. A critical section is a piece of code that atomically accesses shared memory variables; we take into account those critical sections beginning with a lock acquisition, and ending with a lock release. Moreover, to keep low testing efforts, we take into account only memory accesses performed by the application under a given set I of inputs. The required information consists of:

- The set of critical sections accessed for each input i;
- The set of memory accesses to shared variables made in the jth critical section;
- The lock acquired before the jth critical section;
- The shared variable sv_a accessed by the access a;
- The type (read or write) of each access a, that is, type$(a) = R$ or type$(a) = W$.

Table III shows the shared variables for the case study, by using their symbolic name in the source code (although no information about the source code is required for fault injection). Moreover, 15 critical sections are identified; the table shows the shared variables accessed by each critical section, and the type of access (e.g., if critical section 11 both reads and writes a shared variable, we write "11-RW"). The *locks* variable is an array representing the FDPs table. Each element stores a set of attributes related to a FDP request (e.g., the *callback* attribute is an identifier of the requester). The shared variable *state* is used by the FT Service for checkpointing; *m_state_changed* is a boolean flag which is set when *state* is updated.

Table III
SHARED VARIABLES AND CRITICAL SECTIONS IN THE FDPS.

Shared variable	Critical sections and access type
locks	1-RW 2-R 3-R 4-R 5-R 10-R 11-R 12-R 13-R 15-R
locks[i].id	1-W 2-R 11-R 15-R
locks[i].flag	1-W 2-RW 3-RW 11-RW 12-W 15-W
locks[i].callback	1-W 4-W 10-R 15-W
locks[i].mutex	1-W 2-R 3-R 11-R 12-R 15-W
locks[i].cond	1-W 2-R 3-R 11-R 12-R 15-W
state	5-W 8-R 11-R 12-R 13-W
m_state_changed	6-W 7-R 9-W 12-W 14-W

Subsequently, data on locks and memory accesses are used to identify faults to be injected. A fault injection experiment consists of the following steps:

1) Select any pair of critical sections such that:
   a) They contain an access (respectively, a' and a'') to the same shared variable, that is $sv_{a'} = sv_{a''}$;
   b) The accesses a' and a'' are conflicting, that is type$(a') = W$ and type$(a'') = R$;
   c) They hold the same lock l.
2) Remove the acquisition of lock l before the critical section, and its subsequent release;
3) Run the test using the trigger injection technique.

Removal of lock operations can be made statically, i.e., by modifying the binary or source code. Another option is to make lock operations ineffective at run-time, by wrapping functions representing primitives for lock acquisition and release. Wrappers skip a lock operation when it is made in a particular location. This technique can be ported to several hardware/software platforms, and it introduces a negligible overhead, without slowing down the system and keeping its behavior representative [30]. Moreover, as discussed in the following, this option is convenient for fault triggering; therefore it has been preferred in this work.

D. Trigger Injection Phase

The fault triggering technique identifies which inputs to submit to the system to activate an injected fault in a target state. Figure 6 shows an example of an input sequence (the technique will be explained in detail in the following) to trigger a fault in *2:1:0*. When the system is in state *1:1:0*, the input CRQ UPDATE is submitted. In answer to this input, the Façade executes the critical section with the memory access a''. In order to enforce a specific thread scheduling to trigger the fault, we block the thread that is going to make the a'' access (by means of a breakpoint[9]). Subsequently, a second input CR DELETE is sent; then, the system performs the memory access a' (which conflicts with a''); this event is intercepted by means of another breakpoint, in order to unblock the thread that is going to make the access a''. From this moment, the fault is active; in this example, threads interfere with each other because a shared variable is overwritten by thread 2 before being read by thread 1. The system moves to state *2:1:0* when the conflicting accesses occur.

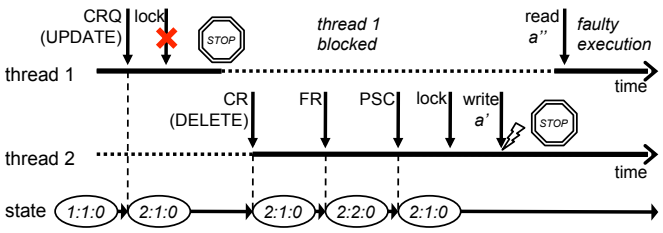

Figure 6. Input timing that triggers a fault in state *2:1:0*.

[9]A breakpoint is an instruction (provided by most computer architectures) that can be inserted in a program to temporarily interrupt its execution.

To avoid fault activation before the target state is reached, lock operations are made ineffective at run-time by means of wrappers (Section V-C). In particular, wrappers enable lock operations when fault triggering should be avoided, and disable them when the fault triggering procedure begins. It should be noted that this solution does not harm fault representativeness; this scenario is equivalent to a dormant fault, i.e., thread scheduling never preempts a thread when it executes a non-atomic critical section. In this way, it is possible to trigger a fault in a specific state.

Input selection. To identify inputs for fault triggering, we design a procedure based on the FSM system specification (Section IV). The procedure is based on the correlation between input messages and memory accesses, and it is obtained by preliminary analysis of the target system (Section V-B). This information is needed to identify which inputs the tester can submit, and how they can be used to enforce a particular path within the FSM.

Table IV
INPUTS, MESSAGE SEQUENCES AND CRITICAL SECTIONS IN THE FDPS.

Input	**Messages and Memory Accesses**		
CR/CRQ INSERT	CR/CRQ 1, 2, 3, 4, 5, 6	FR/FRQ 7, 8, 9	PSC/PSCQ 10, 11, 12, 13, 14, 7, 8, 9
CR/CRQ DELETE	CR/CRQ 2, 3, 4, 5, 6	FR/FRQ 7, 8, 9	PSC/PSCQ 10, 11, 12, 13, 14, 15, 7, 8, 9
CR/CRQ UPDATE	CR/CRQ 2, 3, 4, 5, 6	FR/FRQ 7, 8, 9	PSC/PSCQ 10, 11, 12, 13, 14, 7, 8, 9

Table IV shows the correlation between inputs, message sequences (see Table I) and critical sections occurring after each input. Executions of the same critical section are represented by the same integer number. Depending on the input, a different sequence of critical sections is executed. The proposed procedure performs the following steps:

1) Find the message sequence in Table IV that makes the memory access a';
2) For this message sequence, find a sub-path (*first backward path*, *FBP*) in the FSM that ends in the target state, and that contains the message sequence (recall that edges in the FSM is associated to messages of the system); let S be the first state of this sub-path;
3) Find the message sequence in Table IV that makes the memory access a'';
4) For this message sequence, find a sub-path (*second backward path*, *SBP*) in the FSM that ends in the state S, and that contains the message sequence;
5) The final path is obtained by connecting the *SBP* to the *FBP* in the state S.

The sequences of messages and critical sections may also depend on the current state of the system. This issue can be solved during the preliminary analysis by repeating, under every state of the system, the analysis of sequences caused by each input. Using the additional information about states, the algorithm can be extended to cope with state dependence (by exploring only those paths allowed under a given state). However, the sequences in the FDPS do not exhibit any state dependence, and this extension was left as future work.

Fault triggering. The inputs to be submitted by the tester for fault triggering are the initial messages of the *SBP* and the *FBP*, respectively. More specifically, the tester has to:

1) Send the first message associated to the *SBP*;
2) Wait for the memory access a''; the thread should be blocked before the access;
3) Send the first message associated to the *FBP*;
4) Wait for the memory access a'; subsequently, unblock the previous thread.

Since the tester can control fault triggering, the experiment is repeatable (e.g., to enable *a posteriori* analysis).

Breakpoint setup. Breakpoints are exploited to drive thread scheduling, in order to cause a faulty interleaving between memory accesses. The first breakpoint is inserted before the memory access a'', in order to stop the execution of a thread until another thread makes the memory access a'. The second breakpoint is inserted after a', in order to re-enable the thread interrupted by the previous breakpoint. It should be noted that the intrusiveness due to breakpoints is negligible, since their delay (less than 1 ms in modern CPUs) is much littler than other delays in complex systems (e.g., communication and processing delays).

An example of concurrency fault trigger. We describe an example of fault trigger identified by the proposed procedure. The fault triggering path is shown in Figure 7, which contains a subset of the FSM in Figure 2. Let suppose to inject a fault in the state *2:1:0* between the critical section 8 that follows a PSC DELETE message, and critical section 2 that follows a CRQ UPDATE message (as in Figure 6). The two critical sections make respectively a write (a') and a read (a'') memory access to the same shared variable. First, the procedure finds a sub-path (*FBP*) ending with a PSC transition in *2:1:0* (steps 1, 2). The sub-path should start with a CR or a CRQ transition, since all message sequences in Table IV start with a CR or a CRQ request. In the example, a suitable *FBP* is: *2:1:0* → CR → *2:1:0* → FR → *2:2:0* → PSC → *2:1:0*. Subsequently, the procedure finds a second sub-path (*SBP*) ending with a CRQ transition in S (i.e., *2:1:0*), and starting with a CR or CRQ transition (steps 3, 4). In the example, the *SBP* is: *1:1:0* → CRQ → *2:1:0*. The final path is obtained by connecting the two sub-paths in reverse order (step 5): *1:1:0* → CRQ → *2:1:0* → CR → *2:1:0* → FR → *2:2:0* → PSC → *2:1:0*.

To trigger the fault using this path (Figure 7), the tester has to send to the system a CRQ UPDATE and a CR DELETE request, in accordance to the timing shown in Figure 6. The system should first reach the initial state of the path (*1:1:0* in our example). The choice of the workload used to reach the initial state does not affect the experiment; in our case

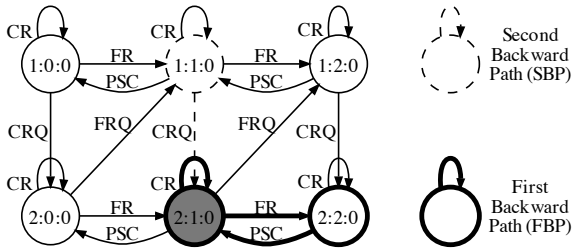

Figure 7. An example of path in the FSM that can trigger a fault in *2:1:0*.

study, we one of the workloads described in Section IV. The sequence of inputs and messages in Figure 6 are the same of the *SBP* and *FBP*, respectively.

There can be several paths that can be used to trigger a fault in a given state. However, it is better to choose the shortest path. In fact, it is possible that blocking a thread may cause the stall of the program, even if lock operations are made ineffective. For example, the thread may have acquired a logical resource by means of a boolean flag, which prevents further execution of other threads. By choosing the shortest path, such a scenario will be less likely. Nevertheless, it is neither possible to avoid this issue in all cases, nor to know *a priori* if it will happen. Therefore, a timeout has to be enforced when it is not possible to cause a specific thread scheduling [15], [16]. If memory accesses are not made within the timeout, the experiment is aborted.

VI. Evaluation of Concurrency Fault Injection

The concurrency fault injection technique has been used to evaluate the FDPS in those states not adequately tested during the first campaign (i.e., states in which the fault activation distribution is null or very low). To this aim, we applied the triggering technique (Section V-D) in those states. From the experimental campaign, we identified four pairs of critical sections that are able to cause a failure in at least one state (Table V). Figure 8 shows a detailed view of states in which one or more conflicting critical sections caused a failure. During the experiments, all failures occur in the same state in which a fault is activated, and the failures manifest through invalid pointer dereference.

Table V
Critical section pairs leading to a failure.

Shared variable	Critical sections	Messages
locks[i].cond	2-R, 1-W	CRQ, CR
locks[i].cond	11-R, 15-W	PSC, PSC
locks[i].mutex	2-R, 15-W	CR, PSC
locks[i].mutex	11-R, 15-W	PSC, PSC

From the analysis of results, we conclude that:

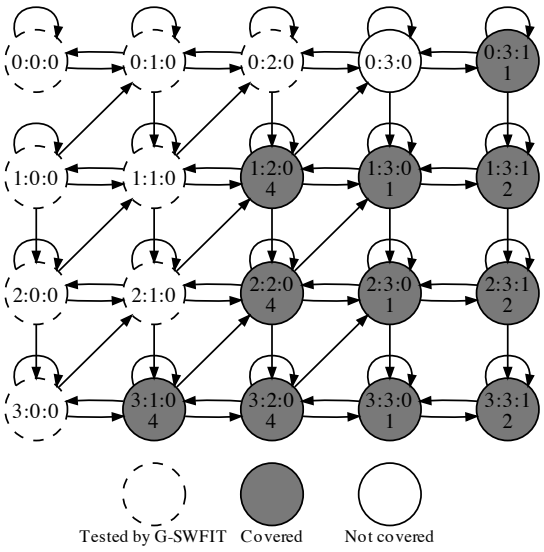

Figure 8. Fault activation distribution of the concurrency fault injection campaign. (this distribution is the same of the failure distribution)

- There are some conflicts that can be activated only in specific states (e.g., in *2:3:0*, only 1 out of 4 failure-prone conflicts is activated). This is because it is not possible to find a fault activation path in those target states, or the fault is not activated within a timeout. The problem of not activated faults is common among fault injection techniques (e.g., G-SWFIT).
- By proper fault triggering, it is possible to trigger a fault in almost all states not adequately tested by G-SWFIT. Even if these states are actually failure-prone, faults injected by G-SWFIT are seldom activated in them. Therefore, faults by G-SWFIT do not cover all the scenarios in which the considered system can potentially fail.
- Only the state *0:3:0* is not covered by both experimental campaigns. It is still possible that the system can be affected by software faults in state *0:3:0* (e.g., by different kinds of Mandelbugs). However, the increased state coverage (95%) brings a significant improvement on the confidence of the overall SFI campaign.

VII. Conclusions and Future Work

In this paper, we analyzed an existing state-of-the-art SFI technique, namely G-SWFIT, in the context of a real-world fault-tolerant system for Air Traffic Control (ATC). The analysis pointed out that faults injected by G-SWFIT are not well representative of Mandelbugs, because most of them manifest in the early phase of the execution, and they deterministically affect both replicas. This has negative effects on the state coverage, because not covered states (35%) can potentially be affected by software faults, as demonstrated by the second SFI campaign. The technique

for concurrency fault injection was able to inject and trigger faults in most of not covered states, reducing them down to 5%, and thus improving the confidence in FTM assessment. Future work will encompass a broader analysis of G-SWFIT on different systems, since it is likely that SFI campaigns will suffer the same limitations of the ones observed in this work, due to lack of control on fault activation by G-SWFIT. Moreover, we will devote efforts to make the technique more general, by increasing the set of faults and fault triggers, and by adapting it to the features of more complex systems.

ACKNOWLEDGMENT

This work has been partially supported by the project "CRITICAL Software Technology for an Evolutionary Partnership" (CRITICAL-STEP, http://www.critical-step.eu), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, within the context of the Seventh Framework Programme (FP7).

REFERENCES

[1] J. Gray, "Why Do Computers Stop and What Can Be Done About It?" in *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, 1985.

[2] I. Lee and R. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System," in *Proc. 23th Symp. on Fault-Tolerant Computing*, 1993.

[3] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems," in *Proc. Intl. Symp. on Fault-Tolerant Computing*, 1991.

[4] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software," in *Proc. 1st workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[5] M. Grottke and K. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate," *IEEE Computer*, vol. 40, no. 2, 2007.

[6] S. Chandra and P. Chen, "Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2000.

[7] S. Maffeis and D. Schmidt, "Constructing Reliable Distributed Communication Systems with CORBA," *IEEE Communications Magazine*, vol. 35, no. 2, 1997.

[8] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures," in *Symp. on Operating Systems Principles*, 2005.

[9] J. Christmansson and P. Santhanam, "Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms–Criteria for Error Selection using Field Data on Software Faults," in *Proc. of Intl. Symp. on Software Reliability Engineering*, 1996.

[10] W. Ng and P. Chen, "The Systematic Improvement of Fault Tolerance in the Rio File Cache," in *Proc. 29th Intl. Symp. on Fault-Tolerant Computing*, 1999.

[11] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira, "Experimental Risk Assessment and Comparison using Software Fault Injection," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2007.

[12] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data," in *Proc. Intl. Symp. on Fault-Tolerant Computing*, 1996.

[13] H. Madeira, D. Costa, and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2000.

[14] J. Durães and H. Madeira, "Emulation of Software faults: A Field Data Study and a Practical Approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, 2006.

[15] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs," in *Proc. 8th Symp. on Operating Systems Design and Implementation*, 2008.

[16] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places," in *Proc. 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[17] D. Dig, J. Marrero, and M. Ernst, "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries," in *Proc. 31st Intl. Conf. on Software Engineering*, 2009.

[18] W. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, 1982.

[19] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," in *Proc. Intl. Symp. on Software Testing and Analysis*, 1996.

[20] W.-I. Kao, R. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, 1993.

[21] J. Christmansson, M. Hiller, and M. Rimen, "An Experimental Comparison of Fault and Error Injection," in *Proc. Intl. Symp. on Software Reliability Engineering*, 1998.

[22] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting How Badly "Good" Software Can Behave," *IEEE Software*, vol. 14, no. 4, 1997.

[23] A. Johansson, N. Suri, and B. Murphy, "On the Impact of Injection Triggers for OS Robustness Evaluation," in *The 18th Intl. Symp. on Software Reliability Engineering*, 2007.

[24] R. Moraes, R. Barbosa, J. Durães, N. Mendes, E. Martins, and H. Madeira, "Injection of Faults at Component Interfaces and Inside the Component Code: Are They Equivalent?" in *European Dependable Computing Conference*, 2006.

[25] D. Avresky, J. Arlat, J. Laprie, and Y. Crouzet, "Fault Injection for Formal Testing of Fault Tolerance," *IEEE Transactions on Reliability*, vol. 45, no. 3, 1996.

[26] A. Arazo and Y. Crouzet, "Formal Guides for Experimentally Verifying Complex Software-Implemented Fault Tolerance Mechanisms," in *Proc. Intl. Conf. on Engineering of Complex Computer Systems*, 2001.

[27] A. Ambrosio, F. Mattiello-Francisco, V. Santiago, W. Silva, and E. Martins, "Designing Fault Injection Experiments Using State-Based Model to Test a Space Software," *Lecture Notes in Computer Science*, vol. 4746, 2007.

[28] R. Chandra, R. Lefever, K. Joshi, M. Cukier, and W. Sanders, "A Global-State-Triggered Fault Injector for Distributed System Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, 2004.

[29] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics," in *Proc. Intl. Conf. on Architecture Support for Programming Languages and Operating Systems*, 2008.

[30] P. Marinescu and G. Candea, "LFI: A Practical and General Library-Level Fault Injector," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2009.