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

Roberto Natella, Domenico Cotroneo
{roberto.natella, cotroneo}@unina.it
Outline

- Context and problem statement
  - Software Fault Injection
  - Bohrbugs and Mandelbugs
- Case study from the ATC domain
- Evaluation of state-of-the-art fault injection
- An experiment involving concurrency faults
- Conclusions
Software faults represent an important cause of system failures

Despite of efforts on Verification activities, fault avoidance, and fault removal, software systems are often delivered with residual software faults

Critical systems adopt Fault Tolerance Mechanisms (FTMs) to avoid failures at run-time
FTMs: A few examples
- Spatial redundancy
  - CORBA FT, TANDEM90 Process Pairs
- Temporal redundancy
  - Checkpointing and rollback
Software Fault Injection (SFI) is a valuable approach for the verification and the improvement of FTMs
To correctly emulate software faults, we need to understand their features
Software Faults

**BohrBugs**
- Faults whose activation is reproducible, i.e., it is straightforward to identify its activation pattern
- Typically detected and then fixed during testing phase

**MandelBugs**
- Faults whose activation is transient and not systematically reproducible
- Their activation conditions depend on complex combinations of user inputs, the internal state and the external environment
Mandelbugs represent the major cause of failure in mission-critical system
- up to 82% in well-tested software [2] [5] [6]
Mandelbugs are typically tolerated by the adoption of several redundancy schemes

Are existing SFI techniques able to emulate Mandelbugs adequately?
How should Mandelbugs be emulated?
To date, *representativeness* of injected faults has still not been investigated with respect to:

- Fault manifestation;
- Their effectiveness in testing FTMs (i.e., to emulate faults that most often occur and that they should tolerate)
We aim to investigate this issue with a simple experimental campaign but...
....in a complex and real-world software systems.

- We evaluated G-SWFIT, with respect to Mandelbugs.
- We compared the results with an experiment, specifically designed to emulate Mandelbugs.

Case study: a fault-tolerant system from the Air Traffic Control (ATC) domain.
- It is a Flight Data Processor (FDPS) based on a CORBA-compliant middleware.
Case study (1/2)
We modeled the FDPS as a FSM to support the analysis of faults.

A state consists of the following internal variables:

1) The number of FDP requests queued by the Façade
2) The number of requests under processing
3) The number of requests queued by Processing Servers (PSs)

CR, FR, PSC, ..., are the messages exchanged in the FDPS.
We implemented G-SWFIT fault operators in an open-source fault injection tool.

The tool analyzes a C/C++ source code file, to produce a set of faulty source files.

- Freely available at: http://www.mobilab.unina.it/SFI.htm
533 faults have been injected in the Façade source code

1599 experiments (3 different workloads)

For each experiment, we collected:

- Information about a failure (e.g., Façade crash, switch to the backup, missed FDP requests)
- The state in which a failure occurred
- The state in which the fault was activated
♦ G-SWFIT is useful to test important system states (e.g., the checkpointing mechanism)

♦ However, faults did not emulate well Mandelbugs because:
  ▪ A great amount of faults (56%) manifest themselves during Façade initialization or during the first request (state 0:0:0); but Mandelbugs usually manifest themselves during the operational phase of a system
However, faults did not emulate well Mandelbugs because (CONTINUED):

- In most of cases (93%) in which the backup Façade is activated, the backup also fails (i.e., fault activation is simple to reproduce, like Bohrbugs)

- Some important states (potentially affected by Mandelbugs) are untested (e.g., when one or more requests are queued by the PSs)

- State coverage: 65%
To emulate Mandelbugs, we analyzed the scientific literature on software faults.

We identified the following fault triggers:

- **Concurrency**
- Timing of external events
- Wrong memory state
- Faulty error handlers
- Complex input sequences
- Software aging
Features of most frequent concurrency faults (from a field data study [29]):

- They are atomicity-violation faults (49%)
- Only 1 shared variable is involved (66%)
- At most 2 threads are needed to trigger the fault (90%)

Our fault model:

- 2 threads access to a shared variable without acquiring a lock (race condition)
We propose a fault emulation technique in two phases:

- **Fault injection:**
  - Collects information about critical regions and their memory accesses
  - Removes lock operations before and after a pair of conflicting critical regions

- **Trigger injection:**
  - Submits an input sequence to drive the system to a target state
  - Schedules 2 threads such that memory accesses interfere with each other
Focusing on fault triggering, we have to profile the system to recognize (and then to drive) the operating state.

An input is associated to:
- A sequence of messages sent and received by the Façade
- A sequence of lock and memory accesses

**Diagram Description**

- **Input message tracing**
  - CR(INSERT, 1), CR(UPDATE, 1), ...

- **Output message tracing**
  - FR(INSERT, 1), FR(UPDATE, 1), ...

- **Memory & lock profiling**
  - 1(W), 2(R), 3(R), 4(R), ...

- List of critical sections and accesses to shared variables
- Correlations between inputs, critical sections and accesses to shared variables
How to trigger a fault?

<table>
<thead>
<tr>
<th>Input</th>
<th>Messages and Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR/CRQ</td>
<td>CR/CRQ</td>
</tr>
<tr>
<td>INSERT</td>
<td>1, 2, 3, 4, 5, 6</td>
</tr>
<tr>
<td></td>
<td>FR/FRQ</td>
</tr>
<tr>
<td></td>
<td>7, 8, 9</td>
</tr>
<tr>
<td></td>
<td>PSC/PSCQ</td>
</tr>
<tr>
<td></td>
<td>10, 11, 12, 13, 14, 7, 8, 9</td>
</tr>
<tr>
<td>CR/CRQ</td>
<td>CR/CRQ</td>
</tr>
<tr>
<td>DELETE</td>
<td>2, 3, 4, 5, 6</td>
</tr>
<tr>
<td></td>
<td>FR/FRQ</td>
</tr>
<tr>
<td></td>
<td>7, 8, 9</td>
</tr>
<tr>
<td></td>
<td>PSC/PSCQ</td>
</tr>
<tr>
<td></td>
<td>10, 11, 12, 13, 14, 15, 7, 8, 9</td>
</tr>
<tr>
<td>CR/CRQ</td>
<td>CR/CRQ</td>
</tr>
<tr>
<td>UPDATE</td>
<td>2, 3, 4, 5, 6</td>
</tr>
<tr>
<td></td>
<td>FR/FRQ</td>
</tr>
<tr>
<td></td>
<td>7, 8, 9</td>
</tr>
<tr>
<td></td>
<td>PSC/PSCQ</td>
</tr>
<tr>
<td></td>
<td>10, 11, 12, 13, 14, 7, 8, 9</td>
</tr>
</tbody>
</table>

- An algorithm identifies (i) which inputs to send and (ii) in which state to send inputs to trigger a fault.
- The algorithm exploits preliminary information to match messages with shared memory accesses.
An example of concurrency fault (1/2)

The CRQ input is sent

Lock operation omitted

Thread 1 is blocked

The CR input is sent

Thread 2 writes an inconsistent value

Thread 1 reads the (faulty) value

An algorithm processes the FSM to find when to send inputs (next slide)
4 injected concurrency faults lead to a failure of primary Façade and not the backup one.

- We covered 13 out of 14 states (93%) in which faults were injected.

- Cumulative state coverage: 95%
  - In particular, states *:3:1 were tested (i.e., one or more requests queued by PSs)
Lessons Learned

Are existing SFI techniques able to emulate Mandelbugs adequately?
- No, G-SWFIT should be complemented by taking into account Mandelbugs

How should Mandelbugs be emulated?
- Our solution is to identify most common fault triggers, and to try to emulate them in addition to modifying the source code
Thank you!
Any questions?
Backup slides
G-SWFIT fault operators were derived from a field data study [14]

Fault activation and manifestation were neglected due to lack of data