Software Implemented Fault Injection Used for Software Evaluation

Andréas Johansson

Department of Computer Engineering
Chalmers University of Technology
Gothenburg, Sweden

aja@ce.chalmers.se

Abstract: It is the sad truth that no matter how good a programmer you are all software systems will contain faults and be exposed to faults from the environment! Therefore, an important aspect of any software component is how resilient it is to faults. The concept of using software implemented fault injection (SWIFI) for software component evaluation seems very appealing. There are however many questions that needs to be resolved before any real use of such a technique can be gained. Questions like what type of faults that arise in real systems and how to emulate them in software must be answered. Is the system at all suitable for testing with SWIFI? What impact does the tool have on the experiment and impact does the workload have? In this paper some of the more important questions are discussed and references are given to previous work done in the corresponding area. Several examples of existing tools are presented and parallels are drawn to the technique presented in the book, IPA, interface propagation analysis.

1 Introduction

As we become more and more dependant on computer-based systems, our dependence on the correct functionality of these systems increases. It is therefore important to have methods for evaluating the dependability of computer systems. One way of increasing the “quality” of software, as well as saving money through a faster process, is the use of better and more sophisticated programming paradigms. One such paradigm is component-based systems. Although the quality of the software may have increased, the need for testing the software has not been erased. Therefore, the need for testing components for dependability arises.

In chapter 10, a technique called IPA (Interface Propagation Analysis) is explained. It is a technique by which the information transfer between different software components is corrupted. The consequences of the perturbation can be anything from a failure of the software to that the fault raised no effect at all. In this paper the scope of the discussion is widen to include all types of software implemented fault injection (SWIFI).

Fault injection has long been used as a technique for “accelerated” testing. Faults or errors are introduced in the system and the effect is studied. The different types of fault injection used are divided by the instrumentation method used, e.g., physical (or hardware based), software implemented or simulation based fault injection. This section will focus on SWIFI, since it is the most convenient method usable in software development.

This paper is divided into five sections which each is a discussion on a specific topic concerning the use of SWIFI for software verification and validation. The second section discusses issues
concerning the non-functional attributes that can be targeted by SWIFI. The third section targets the system under test, what is the motivation for using fault injection? The fourth section discusses the types of faults occurring in real systems and how these can be emulated in a fault injection tool. The sixth section discusses the impact of the tools themselves. The paper is lastly summarized and some conclusions are drawn in the seventh section.

2 Non-functional attributes

A basic question that arises when one is confronted with a new technology/method is what it can be used for. So an obvious question is; which properties of a software system can be estimated using SWIFI? When discussing properties, or behaviors, of a software system one often draws a line between functional and non-functional behaviors. One can also characterize the system with a set of attributes, functional and non-functional. Fault injection is today used for evaluating a certain set of non-functional attributes, namely dependability attributes [1]. Dependability incorporates reliability, availability, safety and maintainability. Of these, reliability is the most commonly targeted attribute.

There are other important non-functional attributes/behaviors apart from dependability. One of the most important ones is performance. Fault injection cannot directly be used for estimating the performance of a system. But, for an important class of systems, namely safety-critical real-time systems, the performance of the system when exposed to erroneous information is very important (this of course holds also for non-safety critical real-time systems, but here the impact of a failure is not a severe). For such systems fault injection can be used to perturb information in the system such that the performance of the system under the presence of faults can be estimated. Note that for many faults the system may not be able to deliver the proper service, i.e., the system fails. Many safety-critical systems (and other as well) contain fault tolerance routines. In Section 3 a discussion on the use of fault injection will be held which will cover this issue.

Performance is often measured with benchmarks. Benchmarks are specific programs that run on top of the system and the execution time of the programs are measured and used as estimates of the performance of the entire system (software + hardware). The natural way of measuring performance under the presence of fault would therefore be to insert faults during the execution of a benchmark.

Other examples of non-functional attributes could be upgradeability, the ability to upgrade a system without compromising the service delivered from it. Other behaviors include unwanted or unexpected behaviors from the system, including undocumented code or malicious code inserted by intruders. These examples are difficult to estimate with fault injection. Some results could possibly be made on the undocumented code in a system, since faults in the system could make it execute undesired parts of the code and therefore possibly discover these behaviors. There are more examples of non-functional behaviors/attributes, but since the scope of this paper is limited to SWIFI I will not dig any deeper into them.

3 System under test

Now that we know which attributes of a system that can be targeted with SWIFI, the next question that arises is which systems are suitable to test with SWIFI? In the chapter it is stated that IPA, which is a special case of general SWIFI, is very well suited for testing COTS
components\(^1\). Since IPA only perturbs the states flowing between components only the interfaces of the components are needed. For general SWIFI tools this is not the case. But, many successful SWIFI tools are very well suited for COTS components. For instance the Xeption tool [2] allows corruption of internal registers in the processor. This means that even though we do not know the source code of the component we can insert fault inside it. Note that such low level fault might not correspond to more complex high level faults. A discussion on the different types of fault is in the following section.

If we have access to the source code of the components we can do more detailed analysis of the behavior in presence of faults. We have more precise locations to insert faults as well as better visibility of the effects. Another important aspect is of course that we can also do something about the behavior, i.e., make changes or introduce fault detection and recovery mechanisms. Work conducted at Chalmers [3,4] has shown that SWIFI can be used to characterize the propagation of errors between modules of software. The information can then be used to decide where to put error detection and error recovery mechanisms.

For component based systems the scenario will be that you use some components that are COTS, and you want to integrate them together with your own in-house component(s). In such a case IPA or similar techniques can be very useful. Techniques where faults are introduced in the interfaces of “components” have been used in other situations. The Ballista tool [5] uses the interface of an operating system, feeds it with erroneous data (in the context of the call) and observes the consequences. This technique was later used to test C-libraries and CORBA implementations across operating systems and vendors.

In hardware design of CPU’s the testing part of development has increased significantly since the beginning of the modern processor. Manufacturers have identified testing as an equally important task as the design. Recent processors even include special logic for making testing and fault injection easier, for instance scan-chains [6]. An interesting idea would be to transfer this idea to the field of component based software engineering. Can one write components with additional interfaces that can facilitate testing? A major difference between software components and hardware components, like CPU’s, is that the basic internal structure of a CPU is known, e.g., we know that it contains a certain set of general purpose registers, pipeline registers, floating-point units etc. It is therefore easy to decide where to inject fault and where to observe the consequences. Can this sort of setup be transferred to software? If a rough sketch of the internal structure, along with a set of interfaces to reach certain critical areas, is included in the component, can this facilitate the testing of the component? How large does a component have to be to make it feasible? How many times does it have to be re-used in order to make it pay-off? What types of systems (critical?) demands this type of testing? There are a lot of unresolved questions in this area.

IPA effectively tests the robustness of components when the input parameters are corrupted. Some systems are designed to handle such scenarios, i.e., they have error detection and error recovery mechanisms built in. Fault injection is then the technique used to test these mechanisms. From such experiments the fault coverage of these mechanisms can be estimated. Also, which might be very important, the error latency can be estimated. The latency is the time

\(^1\) COTS stand for Commercial Off The Shelf. This implies that the internal structure of a COTS component is unknown. Only the interface is known to the user of such a component.
from the moment the fault first occurs till it is discovered as an error. As we shall see in the next section the choice of fault to insert greatly impacts the result you get and the significance of the result. This is of course expected as some types of faults are more common in some systems than in other.

4 Types of Faults

When conducting any fault injection experiments, a decision on what type of faults are to be injected, needs to be done. A natural question is then what fault types are there? Are there faults that are common across several different types of systems or are all system specific? Which types can be injected using SWIFI tools?

The classical division of fault is made between physical faults and human-made faults. Physical faults are either internal, like short-circuits or threshold changes, or external like radiation or temperature variations. Human-made faults are either design faults or interaction faults. A rough list of faults that can be expected from a good SWIFI tool could be:

- Physical faults
- Design faults
- Interaction faults

Physical fault have been studied for long in the fault injection community, for example in [7], and it is well accepted that a major set of physical faults can be injected using SWIFI tools, such as Xeption [2] and FERRARI [8]. It becomes more interesting when dealing with design faults and interaction faults.

Design faults can be on different levels. The first type of design faults one thinks of is software faults, i.e., bugs. These are introduced by the programmer when he/she is constructing the program. But design faults could also be faults in the overall design. These might be harder to find using fault injection, because it is impossible to introduce fault at this level. In between there is a level, referred to as the “algorithm” level. This level can be accessed, at least when the source code of the component is available.

Interaction faults are faults that are induced by the user. This type of faults have been studied for database systems, were mistake by database administrators can cause severe damage to the system or even loss of vital data. A recent study of outages in computer networks show that more than 50% of outages are caused by human introduced faults [9]. How to introduce this type of fault is not obvious. For most components it is also not too important since they have no real connection with the end-user. However, testing operator faults implies testing the entire system, which means that the component is a part of the system under test.

When characterizing the types of faults occurring in a real system, not only the types of faults are important but also the arrival rate and persistence. For some fault injection experiments it might suffice saying that a set of fault do occur in the real system. The component(s) are then tested with these faults and their robustness against them is estimated.
Finding all the relevant fault types is fine, but you need a tool that can inject the faults as well. Research tools have been developed to inject physical faults and even software faults [10]. It is important to note though that you might want to test the system against certain sets of faults. If you for instance want to test how a COTS component fits in your system, all you have to work with is the interface, i.e., the only changes you can make to the component is to put wrappers at the interfaces. Therefore, inserting software fault into such a component is pointless because it is “as is”, meaning that inserting a fault would lead to testing another component than the intended. The introduced bug will never be there in the real component so why test with it?

In a test of the same sort as in IPA, you want to inject faults into one module, emulating faults occurring in another module or fault occurring on lower levels in the system. One thing that must be considered then is the insertion point. Inserting the fault as near as possible to the “source” is desirable. For example, introducing a hardware fault, like a bit-flip, on a high level might not be meaningful, although such faults are common on the lower levels. This is because on the levels between the physical ones and zeros and the software component might include fault tolerance mechanisms. Using COTS components might even make it worse, since they might include such mechanisms that the system designer doesn’t know of. Therefore, studies must be conducted beforehand that give the relevant faults to inject. SWIFI works as low as the RTL (Register Transfer Level), which means that a number of fault tolerance mechanisms in hardware might interfere.

5 Trusting the Result

An important aspect in any experiment is how much the method used interferes with the result of the experiment. It is desired that any SWIFI tool interferes as little as possible with the system under test. A simple method like IPA interferes very little with the components under test, which is good. More complicated methods interfere more, and for real-time systems even a little interference might mean that the result is completely different. In a setup where a new task has to be added to conduct the experiments the schedulability of the entire system might be compromised.

When conducting a fault injection experiment, the fault load, i.e., the set of faults, their location, appearance and duration, is not all, also the workload is important. The workload is the basic program(s) that run on the system when the experiment is conducted. In the case of IPA the workload is the program that uses the desired component. Naturally, different workloads use the system differently and consequently give different results. With any fault injection experiment you can basically have one of two intentions; either your intention is to find as many flaws as possible in the system or your intent is to see how the system works in a real environment with faults. When conducting tests like the ones in IPA you have the first intent. When conducting tests with the second intention they become like benchmarks. Depending on which intent you have the workload will change correspondingly. For COTS components, finding a special workload that exercises the component in a special way might be difficult because the lack of knowledge about the internals, therefore a more general workload might be chosen.

6 Summary

This paper is a discussion around a set of questions concerning the concept of software implemented fault injection (SWIFI). The first question is what attributes/behaviors of a software system that can be estimated using SWIFI. SWIFI has long been used for estimating
dependability attributes but also other attributes like performance under the presence of faults can be estimated. Another question is what type of systems that are suited for SWIFI. It is concluded that COTS based systems definitely can take use of fault injection during evaluation. But also in-house software can be tested using such techniques. An interesting idea about including “test code” to components to facilitate testing arises from the field of hardware testing. The third question is what types of faults that occur in real systems. Here a set of three types is presented, including physical faults, software faults (bugs) and operator faults. The natural follow-up question is whether all of these can be emulated in a SWIFI tool. Previous research shows that physical faults and software faults can (at least partially) be emulated in software. The last question concerns the factors influencing the result. Two important factors are the tool itself and the workload that runs when an experiment is conducted.

7 References