GOOFI : Generic Object-Oriented Fault Injection Tool

Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson
Laboratory for Dependable Computing
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
+46 31 772 5225, 46 31 772 3663 fax
{aidemark, vinter, peterf, johan}@ce.chalmers.se

Abstract

In this paper, we present a new fault injection tool called GOOFI (Generic Object-Oriented Fault Injection). GOOFI is designed to be adaptable to various target systems and different fault injection techniques. The tool is highly portable between different host platforms since it relies on the Java programming language and a SQL compatible database. The current version of the tool supports pre-runtime Software Implemented Fault Injection and Scan-Chain Implemented Fault Injection.

1. Introduction

Fault injection has become an important method for experimentally validating the dependability of computer systems. It can be used to identify dependability weaknesses in the design of a fault tolerant system. Fault injection can also be used to obtain dependability measures such as the error coverage of a system. The coverage can then be used in an analytical model to calculate the system's availability and reliability.

Many fault injection tools and techniques have been presented. Some examples are: MEFISTO [1], VERIFY [2] and DEPEND [3], which are tools that inject faults into a simulation model of a system. RIFLE [4] and MESSALINE [5] are tools for pin-level fault injection, while Xception [6], FIAT [7] and FERRARI [8] are tools that inject faults into physical systems using software implemented fault injection (SWIFI).

Different techniques are used in different phases of the design cycle. Simulation based fault injection can be used early in the design cycle, while SWIFI and pin-level fault injection requires that a system prototype is available.

Most fault injection tools have been developed with a specific fault injection technique in mind targeting a specific system, and using a custom designed user interface. Extending such tools with new fault injection techniques, or porting the tool to new target systems is usually a cumbersome and time-consuming process.

So far, only a few tools have addressed the issues of extension and portability to different target systems. NFTAPE [9] is a recent fault injection tool that relies on available lightweight fault injectors, triggers, monitors and other components to facilitate porting the tool to new target systems as well as adapting it for different fault injection techniques.

In this paper, we present a new fault injection tool, called GOOFI (Generic Object-oriented Fault Injection), which can perform fault injection campaigns using different fault injection techniques on different target systems. A major objective of the tool is to provide a user-friendly fault injection environment with a graphical user interface and an underlying generic architecture that assists the user when adapting the tool for new target systems and new fault injection techniques.

The GOOFI tool is highly portable between different host platforms since the tool was implemented using the Java programming language and all data is saved in a SQL compatible database. Furthermore, an object-oriented approach was chosen which increases the extensibility and maintainability of the tool.

The current version of GOOFI supports pre-runtime Software Implemented Fault Injection (SWIFI) and Scan-Chain Implemented Fault Injection (SCIFI). The SCIFI technique injects faults via the built-in test-logic, i.e. boundary scan-chains and internal scan-chains, present in many modern VLSI circuits. This enables faults to be injected into the pins and many of the internal state elements of an integrated circuit as well as observation of the internal state [10]. In pre-runtime SWIFI, faults are injected into the program and data areas of the target system before it starts to execute. GOOFI is capable of injecting single or multiple transient bit-flip faults. In

this paper, we focus on the architecture of GOOFI and the implementation of the SCIFI fault injection technique.

The target system considered in this paper uses the Thor RD microprocessor [11]. The Thor RD is developed by SAAB Ericsson Space AB and is intended for highly dependable space applications. It is an improved version of the Thor microprocessor evaluated in [10] featuring parity protected instruction and data caches. Thor RD is manufactured using a radiation-hardened process.

The remainder of the paper is organised as follows. The architecture of the tool is described in section 2. Section 3 shows how to use the GOOFI tool with the SCIFI technique. Conclusions and future extensions of the tool are presented in Section 4.

2. Architectural design

Adapting a fault injection tool to a new fault injection technique or a new target system requires a trade-off between generality and user friendliness. The objective of GOOFI is to provide as much support as possible for adapting the tool to new target systems and new fault injection techniques. When a new fault injection technique is added, a new fault injection algorithm must be implemented and the graphical user interface must be modified to support the new fault injection technique.

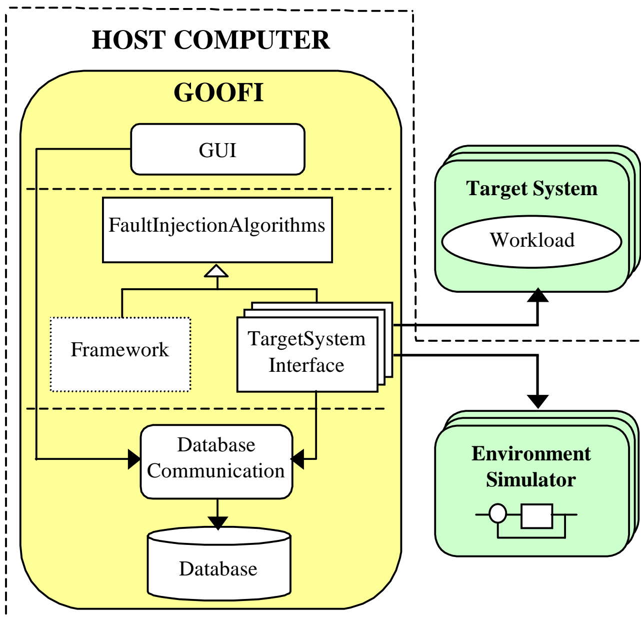

Figure 1. The GOOFI architecture.

Since GOOFI is developed using Java and relies on a SQL compatible database for storing data, the tool is highly portable between host platforms. The current version of GOOFI was developed on a PC running Windows 2000 and was subsequently ported to a Sun workstation running Solaris 8.

The GOOFI architecture can be divided into a three-layered architecture (see Figure 1). At the top layer is the graphical user interface (GUI). From the menus in the GUI, fault injection campaigns can be configured and started for a chosen fault injection technique and for a chosen target system.

The main Java classes in the middle layer include FaultInjectionAlgorithms and Framework, as well as a TargetSystemInterface class for each supported target system. The fault injection algorithms defined in the FaultInjectionAlgorithms class use abstract methods (undefined procedures) that must be implemented in each TargetSystemInterface class. The fault injection algorithm for the SCIFI-technique is shown in Figure 2.

```
abstract class FaultInjectionAlgorithms {
   public abstract void initTestCard();
   public abstract void loadWorkload();
   public abstract void runWorkload();
   public abstract void waitForBreakPoint();
   public abstract void writeMemory();
   public abstract void readMemory();
   public abstract void readScanChain();
   public abstract void injectFault();
   public abstract void writeScanChain();
   public abstract void waitForTermination();
   . . . .

   public void faultInjectorSCIFI(String campaignNr){
      readCampaignData(campaignNr);
      makeReferenceRun();
      for(int i = 0; i < nrOfExperiments; i++){
         initTestCard();
         loadWorkload();
         writeMemory();
         runWorkload();
         waitForBreakPoint();
         readScanChain();
         injectFault();
         writeScanChain();
         waitForTermination();
         readMemory();
         readScanChain();
      }
   }

   public void faultInjectorSWIFI(..){
      . . . .
   }
}
```

Figure 2. The FaultInjectionAlgorithms class.

The Framework class, see Figure 3, is used as a template by the programmer when creating a new TargetSystemInterface class. The TargetSystemInterface class inherits the FaultInjectionAlgorithms class and can therefore use the defined fault injection algorithms directly. Only the abstract methods used by the algorithm need to be implemented in the TargetSystemInterface class.

Finally, the database and the interface to the database are in the lowest layer. The database stores information

about the target system, the fault injection campaigns and data logged from the fault injection experiments.

2.1. Adding a new fault injection technique to GOOFI

To adapt the tool to a new fault injection technique a new window in the GUI must be implemented and a new fault injection algorithm must be added to the FaultInjectionAlgorithms class. By combining different abstract methods we can define algorithms for fault injection techniques such as SCIFI, SWIFI or pin level fault injection (see for example the method *faultInjectorSCIFI* in Figure 2). When adding a new fault injection technique to GOOFI, the abstract methods are used as building blocks. Many of the abstract methods used by one fault injection technique are reusable when defining the algorithm for another fault injection technique, e.g. the abstract method *loadWorkload()* in Figure 2. Other abstract methods need to be implemented specifically for each new fault injection technique, such as the method *injectFault()*. The previously undefined abstract methods needed for defining the new fault injection technique are added to the Framework class.

2.2. Adapting GOOFI to new target systems

When support for a new target system is added to GOOFI, a new TargetSystemInterface class must be created. To do this the programmer uses the Framework class as a template. This means that the programmer only needs to implement the abstract methods used by the fault injection algorithms.

```
public class <FrameWork> extends FaultInjectionAlgorithms {

        public void initTestCard(){
            // Write your code here!
        }
        public void loadWorkload(){
            // Write your code here!
        }
        . . . .
    }
}
```

Figure 3. The framework used for implementing fault injection on a new target system.

2.3. GOOFI database

The GOOFI database contains the tables shown in Figure 4. The *TargetSystemData* table stores all information about the target system required for setting up new fault injection campaigns and the *CampaignData* table stores all the information needed to conduct a campaign. The target system data and campaign data is provided by the user via the GUI. The *LoggedSystemState* table stores the system state during and after an experiment. The results of a fault injection campaign are primarily obtained by analysing the *LoggedSystemState* table. The relations between the tables in the database are designed to use foreign keys (shown as arrows in Figure 4). Through the foreign keys, we prevent inconsistencies in the database and minimize the information stored in the tables while still being able to track all information about the campaign and the target system.

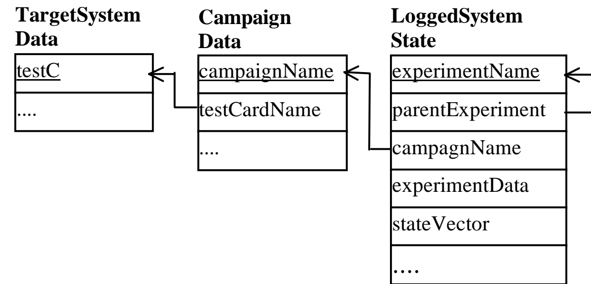

Figure 4. References between the tables in the database.

The attribute "experimentName" in table *LoggedSystemState* holds a unique name for each experiment. To exemplify the use of the second attribute "parentExperiment", assume that one fault injection experiment E1 shows an interesting result such as a fail-silence violation, and we want to investigate the reason for this violation by re-running the experiment logging the system state after each machine instruction. (Such logging is normally not done for each fault in a campaign because it is too time-consuming.) The same campaign data for the new experiment E2 as for E1 must then be used. Thus, the name E1 in "parentExperiment" is saved so that the information about the campaign data for E1 can be tracked. The "experimentData" attribute contains information about the experiment such as the fault injection location, while the "stateVector" attribute contains the logged system state information from the fault injection experiment.

3. Using GOOFI with the SCIFI technique

This section describes how the SCIFI technique is implemented in the GOOFI tool for a specific target system built around the Thor RD microprocessor. Conducting fault injection campaigns using GOOFI involves four phases: the *configuration*, *set-up*, *fault injection* and *analysis* phase.

3.1. Configuration phase

The configuration phase involves adapting the tool to a certain target system. The Thor RD features advanced scan-chain logic, i.e. built-in test logic primarily intended for testing integrated circuits or printed circuit boards, conforming to the IEEE 1149.1 standard for boundary scan. The scan-chain logic can also be used to perform fault injection; it allows access to almost 25000 of the 48100 state elements of Thor RD.

The scan-chains are configured via a graphical user interface (see Figure 5). Here, the user enters the name and the position of possible fault injection locations. This information is stored in the *TargetSystemData* database table. Some locations in the scan-chain are read-only and can therefore only be used to observe the state of the microprocessor.

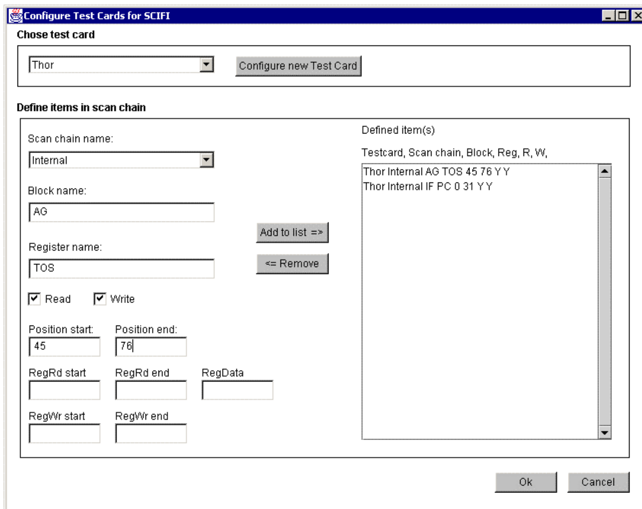

Figure 5. Configuring a target system for the SCIFI technique.

3.2. Set-up phase

In the set-up phase, the user selects a target system. Then the corresponding target system data is interpreted presenting the user with an overview of the possible fault locations and fault models available. The selections made by the user in the set-up phase are stored in the database table *CampaignData*. During the set-up phase, the user may also modify already stored campaign data created for earlier fault injection campaigns or merge campaign data from several fault injection campaigns into a new fault injection campaign.

After selecting the target system, the user chooses the fault injection locations from a hierarchical list of possible locations presented in a window (see Figure 6) as well as the fault models to use and the points in time the faults should be injected. The tool currently supports the bit-flip fault model. The user also selects the target system workload and the number of *fault injection experiments* to perform, i.e. the total number of faults to be injected on the chosen locations during the fault injection campaign.

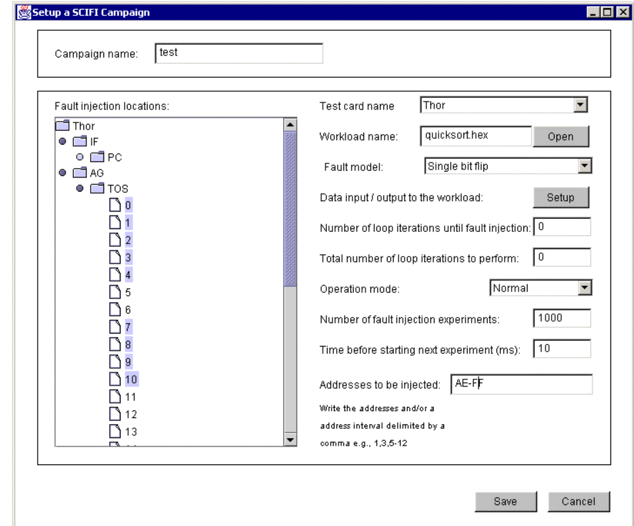

Figure 6. Fault injection campaign definition

The *termination conditions* for the experiments are also selected. A fault injection experiment can be terminated by a debug event generated via the scan chains i.e., when a time-out value has been reached, an error has been detected or the execution of the workload ends, whichever comes first. The workload may consist of a program that either terminates by itself or is executed as an infinite loop. In the latter case, the user must specify the maximum number of iterations that should be executed before a fault injection experiment is terminated. During each loop iteration, data may be exchanged with a user provided environment simulator emulating the target system environment (see Figure 1). Information about which environment simulator program to use, the memory locations holding output and input data within the target system as well as the points in time the data exchange occurs, e.g. when each loop iteration finishes, must also be given. The generated set-up data is stored in the database table *CampaignData* to be used in the fault injection phase.

3.3. Fault injection phase

In the fault injection phase, the SCIFI Fault Injection Algorithm (see FaultInjectorSCIFI in Figure 2) starts by reading the campaign information from the database

table *CampaignData*. The target system is initialised and the workload and initial input data is downloaded to the system. Then a reference execution of the workload is made, logging the fault-free system state to the database table *LoggedSystemState*. After this, each fault injection experiment begins by reinitialising the target system and downloading the workload and initial input data.

The SCIFI fault injection algorithm requires break-points to be set according to the points in time when the fault should be injected stated in the database table *CampaignData*. The breakpoint is obtained by analysing the workload code and is set via the scan-chains. When a break-point condition has been fulfilled, execution of the workload stops and the chosen faults are injected by reading the contents of the scan-chains, inverting the bits stated in the campaign data and writing back the fault injected scan-chains to the system.

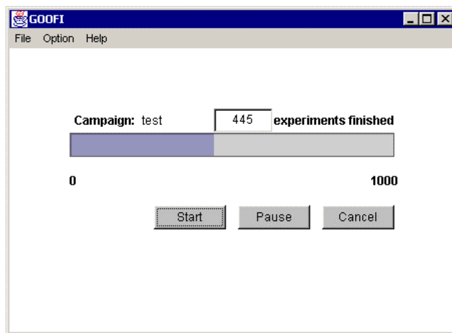

Figure 7. Progress window, showing the number of experiments conducted.

After the fault has been injected, the execution starts from where the target system was halted and continues until the termination condition occurs. The system state is logged to the database table *LoggedSystemState*. The target system is then reinitialised and a new fault injection experiment begins.

GOOFI can be operated in either *normal* or *detail* mode. In normal mode, the system state is logged only when the termination condition is fulfilled. In detail mode the system state is logged as frequently as the target system allows, typically after the execution of each machine instruction, which increases the time-overhead. The detail mode operation is used to produce an execution trace, allowing the error propagation to be analysed in detail. The logged system state typically includes the contents of all the locations in the target system that are observable (the locations to observe can be selected by the user in the set-up phase) as well as the workload input and output values, together with information about when and where any faults were injected.

During the fault injection campaign, a progress window (see Figure 7) is shown enabling the user to monitor the experiments, e.g. getting information about the number of faults injected and also to pause, restart or end the campaign.

3.4. Analysis phase

The data in the database table *LoggedSystemState* is analysed in the analysis phase in order to obtain various dependability measures. The kind of measures obtainable depends on the target system. Currently, there is no support for automatic generation of software that analyses the *LoggedSystemState* table. The user must write tailor made scripts or programs that query the database for the required information. However, this is typically done once for each new target system. The user can then choose which analysis software to use, and where to store the results from a menu in the GOOFI graphical user interface. Typical results obtained include the number of:

Effective errors:

- *Detected errors:* Errors that are detected by the error detection mechanisms of the target system. These errors can be further classified into errors detected by each of the various mechanisms.
- *Escaped errors:* Errors that escapes the error detection mechanisms causing failures such as incorrect results or timeliness violations.

Non-effective errors:

- *Latent errors*: The fault injection experiments where differences between the correct system state logged after the reference execution and the system state logged after the fault injection experiment terminated could be observed, but which could not be identified as either Detected errors or Escaped errors.
- *Overwritten errors:* The fault injection experiments for which there is no difference between the correct system states logged after the reference execution and the system state logged after the fault injection experiment terminated.

4. Conclusion and future extensions

This paper described the GOOFI (Generic Object-Oriented Fault Injection) tool. The tool is implemented in the Java language to support maintainability and portability between different host platforms. All data used by the tool is stored in a portable SQL-database. An object-oriented approach was used to minimize the programming effort needed for adding a new fault

injection algorithm, or adapting the tool to a new target system. Two fault injection techniques have been implemented in the current version of the tool. These are pre-runtime Software Implemented Fault Injection and Scan-Chain Implemented Fault Injection. So far, GOOFI has been used with the SCIFI technique for a control application executing on the Thor microprocessor [12]. In this paper, the target system uses the Thor RD microprocessor.

The current version of GOOFI can be improved and extended in several ways. We are currently working on the following extensions:

- Support for runtime SWIFI, where the target system workload is instrumented with additional software for injecting faults.
- Runtime and pre-runtime SWIFI support for other microprocessors.
- Support for additional fault models such as intermittent and permanent faults.
- Use of pre-injection analysis to improve fault injection efficiency. The purpose of this analysis is to determine when registers and other fault injection locations hold *live data*. Injecting a fault into a location that does not hold live data serves no purpose, since the fault will be overwritten.
- Additional fault triggers such as access of certain data values, execution of branch instructions or subprogram calls, when task switches occur, or at specific times determined by a real-time clock.
- Automatic generation of software for analysing the database table *LoggedSystemState*.

Acknowledgements

We would like to thank Stefan Asserhäll and Torbjörn Hult at Saab Ericsson Space AB for providing the target system with the Thor RD processor, technical assistance and many valuable suggestions. We would also like to thank the anonymous reviewers for their many helpful suggestions. This work was supported by VINNOVA, ARTES and the Swedish Foundation for Strategic Research (SSF).

References

[1] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, “Fault Injection into VHDL Models: The MEFISTO Tool”, in *Proc. 24th Int. Symp. on Fault-Tolerant Computing* (FTCS-24), (Austin, TX, USA) June 1994, pp. 66-75.

[2] V. Sieh, O. Tschache, F. Balbach, “VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions”, *Proc. 27th Int. Symp. on Fault-Tolerant Computing,* (FTCS-27), Digest of Papers, June 1997, pp. 32 –36

[3] Goswami, K. K. and Iyer, R. K., "A simulation-based study of a triple modular redundant system using DEPEND," in *Proceedings of the 5th International Conference on Fault-Tolerant Computing Systems*, Sept. 1991, pp. 300-311.

[4] H. Madeira, M. Rela and J. G. Silva, RIFLE: A General Purpose Pin-Level Fault Injector, in *Proc. EDCC-1, Springer LNCS*, Vol. 852 (1994), pp.199-216.

[5] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, “Fault Injection for Dependability Validation — A Methodology and Some Applications”, *IEEE Transactions on Software Engineering*, 16 (2), February 1990, pp.166-182.

[6] J. Carreira, H. Madeira and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Trans. on Software Engineering*, vol. 24, February 1998, pp. 125-136.

[7] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson and T. Lin, “FIAT --- Fault Injection Based Automated Testing Environment”, in *Proc. 18th Int. Symp. On Fault-Tolerant Computing* (FTCS-18), 1988, pp102-107.

[8] G. Kanawati, N. Kanawati, and J. Abraham, “FERRARI: A Flexible Software-Based Fault and Error Injection System”, *IEEE Transactions on Computers*, 44 (2), Feb. 1995, pp. 248-260.

[9] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, R.K. Iyer, "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors", in *Proc. IEEE international Computer Performance and Dependability Symposium*, 2000 (IPDS 2000), 2000, pp: 91 -100

[10] P. Folkesson, S. Svensson, and J. Karlsson, “A Comparison of Simulation Based and Scan Chain Implemented Fault Injection”, in *Proc. 28th Int. Symp. on Fault-Tolerant Computing* (FTCS-28), (Munich, Germany), June 1998, pp. 284-293.

[11] Saab Ericsson Space AB, “Rad Hard Thor Microprocessor Description”, *Document No P-TOR-NOT-0004-SE*, 20 Jan 1999.

[12] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson, “Reducing Critical Failures for Control Algorithms Using Executable Assertions and Best Effort Recovery ”, *Proceedings International Conference on Dependable Systems and Networks*, DSN 2001, Gothenburg, Sweden, July 2001.