CHALMERS

An Overview of GOOFI - A Generic Object-Oriented Fault Injection Framework

JONNY VINTER
JOAKIM AIDEMARK
DANIEL SKARIN
RAUL BARBOSA
PETER FOLKESSON
JOHAN KARLSSON

Technical Report No. 05-07

Department of Computer Science and Engineering
Division of Computer Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2005

An Overview of GOOFI - A Generic Object-Oriented Fault Injection Framework

Jonny Vinter, Joakim Aidemark, Daniel Skarin, Raul Barbosa,
Peter Folkesson and Johan Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden
{vinter, aidemark, skarin, rbarbosa, peterf, johan}@ce.chalmers.se

Abstract

This paper presents a framework called GOOFI (Generic Object-Oriented Fault Injection) for experimental dependability validation of embedded computer systems using fault injection. GOOFI is designed to be adaptable to various target systems and different fault injection techniques and is highly portable between different host platforms since it relies on the Java programming language and a SQL compatible database. The most recent version of the framework supports four different techniques for fault injection. They are software implemented fault injection, scan-chain implemented fault injection and fault injection via two on-chip debug interfaces known as BDM (Background Debug Mode) and Nexus (a recently introduced standard interface). Work has also been started on adding support for simulation based fault injection in VHDL models as well as function-level models.

1. Introduction

Embedded computer systems are increasingly being used to protect large investments or human lives. Validating the dependability of such systems is an essential part of the design process. *Fault injection* [1], which is a way of accelerating the occurrences of faults in the system, has become an important method for system engineers to experimentally validate the dependability of computer systems. The main purpose of fault injection is to evaluate and debug the error detection and recovery mechanisms in computer systems.

Fault injection can be used at various abstraction levels depending on the information available about the system and at which stage of the design process the method is applied. Fault injection techniques can be divided into *simulation-based* and *physical* techniques depending on whether faults are injected into a model of a system, or into an actual physical system or prototype. The advantage of simulation-based fault injection is that it can be used early in the development process before the actual system is available which facilitates early discovery of design deficiencies. Physical fault injection is important since it allows the actual implementation of the system to be tested.

So far, 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.

However, a few recent tools have addressed the issues of extension and portability to different target systems. NFTAPE [2] 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. The Xception tool [3] is implemented using a modular design, and has recently been extended to include different types of fault injection techniques.

We have developed a fault injection framework called GOOFI (Generic Object-oriented Fault Injection), which can be used to perform fault injection campaigns using different fault injection techniques on different target systems. A major objective of the framework 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 to new target systems and new fault injection techniques. GOOFI is highly portable between different host platforms, since it is 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 framework.

An early version of GOOFI was first presented in [4]. This paper presents the latest version of the GOOFI framework, where different fault injection techniques can be added as plug-ins, and additional tools such as an error distribution classification tool and a database visualization tool also are provided. In particular, a plug-in for fault injection via the Nexus [5] on-chip debug interface with support for *pre-injection* is presented in detail. Pre-injection is an analysis technique that is applied before any fault injection is performed to focus the injection to specific parts of the fault space and thereby reduce the time and effort of validating dependable systems by fault injection [6]. *Post-injection* is another analysis technique in which the results from fault injection experiments are used to predict the outcome of other experiments in order to speed up the validation process.

The remainder of the paper is organized as follows. An overview of different fault injection techniques is presented in Section 2. The architecture of GOOFI is described in Section 3. Section 4 shows how to use the framework with Nexus-based fault injection on a MPC565 microcontroller. Section 5 describes the pre-injection analysis and Section 6 presents related work on fault injection tools. Finally, conclusions and future extensions to the framework are presented in Section 7.

2. Fault injection techniques

As fault injection has become widely used as an experimental dependability validation method, many different techniques for injecting faults have been developed. This section gives an overview of the most common techniques. The various fault injection techniques can be characterized according to different properties. One such property is *reachability,* expressing the ability of the fault injection technique to reach possible fault locations in the system. Another property is *controllability*, with respect to *space* and *time*, denoting the ability to control *where* and *when* the faults are injected among the reachable locations. *Repeatability* denotes the ability to accurately repeat a single fault injection experiment while *reproducibility* refers to the ability to statistically reproduce the results of several experiments for a given set-up. *Intrusiveness* relates to the level of undesired impact the fault injection technique may have on the behavior of the target system and can be divided into *space* and *time* properties. In order to achieve experiments corresponding to faults in the real world, it is important that the intrusion is low. Intrusiveness in time relates to the temporal overhead caused by the fault injection technique while intrusiveness in space relates to the hardware/software overhead. Other properties include *flexibility*, denoting the ease of changing fault injection targets in the system, *effectiveness* with respect to the ability to activate and exercise various fault handling mechanisms in the system and *efficiency* with respect to the amount of time and effort needed to conduct the experiments. Another important property is *observability*, which refers to the ability to provide means for observing and measuring the effects of faults in the system.

2.1 Simulation-based fault injection

In simulation-based fault injection, faults are injected in a simulation model of the computer system. This technique is often applied in the early design phases to allow test and validation of error handling techniques before a physical prototype is available. This allows design faults to be detected at an early stage which may reduce the cost for correcting such faults. Controllability and observability are typically high in simulation-based fault injection while the efficiency is quite low since there is often considerable time overhead for performing the simulations. Simulation-based fault injection can be made either at the *electrical* (transistor) level, *logic* (gate) level or *function* (algorithm) level depending on the abstraction level employed when simulating the system. The efficiency is generally higher for higher abstraction levels while the reachability is lower. Faults may be injected using additional components (*saboteurs*), via alteration of the simulation model (*mutants*) or through built-in simulator commands. The intrusiveness is obviously much higher in the first case compared to the latter.

2.2 Physical fault injection

Physical fault injection techniques are applied on actual implementations or prototypes of systems during later phases in the development process. There are two main categories of physical techniques; those that rely mainly on hardware for performing the fault injection, and those that rely on software, i.e., *software-implemented fault injection* (SWIFI). While the observability and controllability can be more limited for physical techniques than for simulation-based techniques, the efficiency is often higher. Another advantage of physical techniques is that the actual implementation of the system is validated instead of a system model.

One of the most common hardware-based techniques is *pin-level fault injection*. In this technique, faults are injected by setting the logical values of the pins of digital circuits to 0 or 1 (*stuck-at*) or inverting the current value (*inversion*). Other common fault models include *bridging*, when several pins of the circuit are interconnected and *open connection*, when the faulted pin is essentially tri-stated. There are two main strategies used for injecting faults on the pins: *forcing* and *insertion*. Pin-level forcing uses probes that are applied on the pins and associated equipotential lines. Insertion uses transistor switches connected to the pins of the integrated circuits to provide isolation from the rest of the system.

Another physical fault injection technique is *radiation based fault injection*. This technique injects Single Event Upsets (bit-flips) in integrated circuits by exposing them to high energy particles. Energetic heavy-ion beams which are able to penetrate vulnerable regions of integrated circuits can be generated using cyclotrons (particle accelerators). Another, more cost-effective technique for generating SEUs (Single Event Upsets) relies on the use of heavy-ions emitted from a Cf-252 source [7].

The recent advancements in programmable circuit technologies have promoted the use of FPGAs[1] for fault injection. [8] describes how circuit descriptions may be automatically added to synthesizable designs targeted for FPGAs for injecting faults. The method combines the advantages of simulation-based fault injection, such as high controllability and observability, with the advantages of physical fault injection, such as small temporal overhead. Methods of injecting permanent faults by mutations and reprogramming of FPGAs have also been presented [9].

The *Scan-chain implemented fault injection* (SCIFI) is a physical technique which injects faults via scan-chains, i.e., built-in logic conforming to the IEEE 1149.1 standard. Such built-in test logic is used for testing and on-chip debugging of microprocessors and other VLSI circuits. The SCIFI technique provides better reachability, observability and controllability than many other physical techniques since the technique allows faults to be injected and observed both on the boundary pins and in the internal state elements (latches and flip-flops) of the circuits. The intrusiveness is also lower for SCIFI than for many other physical techniques since the faults are injected using the test logic already available in the system. In order to use the SCIFI

[1] Field-Programmable Gate Arrays

technique efficiently, detailed information about the scan-chains is required. This information is often only accessible to chip designers and third party debug vendors. The implementation of a SCIFI tool may then have to rely on the use of an existing debugger which may have limitited access to the scan-chains.

Other types of *on-chip debug techniques* exist beside those based on IEEE 1149.1. *Background Debug Mode (BDM)* is a proprietary technique from Motorola which has been successfully exploited for fault injection in [10]. As for the SCIFI technique, the intrusiveness is low for BDM fault injection. However, the technique is limited to Motorola devices through the use of the proprietary BDM port. On-chip debugging techniques supporting real-time tracing such as the IBM RISCTrace and the Nexus standard [5] are other viable alternatives for fault injection as they feature minimal time overhead for injecting faults and observing the system as the system does not need to be halted.

Other physical fault injection techniques which have been introduced use *power supply disturbances*, *electromagnetic interference* and *laser beams*. Power supply disturbances can be injected by introducing voltage sags on the power supply lines to integrated circuits [11]. The electromagnetic interference (EMI) technique uses probes or plates connected to a burst generator placed above the target circuits of the target system to inject the faults [12]. Laser fault injection uses laser beams on the parts of VLSI circuits sensitive to SEUs to inject faults [13].

2.3 Software-implemented fault injection

An increasingly popular technique is software-implemented fault injection (SWIFI) which uses additional software for injecting faults into physical systems and thus provides a cheaper and more flexible way of injecting faults than most other physical techniques. However, the effects of physical faults may not always be properly emulated due to a lack of reachability and knowledge of how faults generated by software correspond to actual physical faults. The various SWIFI techniques can be divided into *pre-runtime injection* and *runtime injection* techniques.

In pre-runtime injection, faults are injected by manipulating the software before it is downloaded to the target system. The major advantage of this technique is that the level of time related intrusiveness can be kept low since the faults are injected without the need to stop or divert the program execution.

Runtime injection involves adding the software necessary for performing the fault injection in a pre-runtime initialization step and to use this software during a runtime fault injection step. Runtime injection uses interrupts or breakpoints to divert the program execution to the software that injects the faults. This approach allows faults to be injected not only in the software code but in all parts of the system reachable by the software (memory, CPU registers etc.). The disadvantage is the time overhead induced by handling interrupts and breakpoints.

2.4 Hybrid fault injection

Several techniques using combinations of some of the fault injection techniques mentioned above have been proposed. One example is given in [14] where SWIFI is combined with simulation based techniques which perform the actual fault injection. This *hybrid fault injection technique*, also known as mixed-mode fault injection, allows the advantages of both SWIFI and simulation based fault injection to be utilized, i.e. the actual target system may be executed at full speed except during the injection of a fault when a simulator providing detailed access to the target system is used instead.

3. Overview of GOOFI

The objectives of GOOFI are to provide i) a user-friendly fault injection environment and ii) support for adaptation to new target systems and new fault injection techniques. To achieve the first goal, the GOOFI graphical user interface has been designed to be more or less self-explaining and that fault injection experiments with different fault injection plug-ins are carried out in a similar way. The second goal is achieved by providing a plug-in based framework.

New techniques and target systems are added through the GOOFI plug-in interface. A major advantage of this architecture is that a new plug-in can be added to GOOFI without the need of a regression test[2] since the old system will not be affected by bugs in the added plug-in. GOOFI does not have to be recompiled when a new plug-in is added and the new plug-in will automatically be found when GOOFI is restarted. The target systems and fault injection techniques that GOOFI currently supports are:

- The microcontroller MPC565 from Motorola with Nexus-based fault injection
- The microprocessor HC12 from Motorola with BDM-based fault injection
- The microprocessor Thor Rad Hard from Saab Ericsson Space with scan-chain implemented fault injection (SCIFI)
- The microprocessor MC68340 from Motorola with software-implemented fault injection (SWIFI)

Figure 1 shows the GOOFI tool with different plug-ins and how GOOFI may communicate with a target system and an environment simulator (an optional program emulating the target system environment). In the GOOFI framework, the target system, the environment simulator and the database that stores information and results from the experiment can be located within the host computer or connected via a network.

[2] Regression testing is applied to modified software to provide confidence that the changed parts behave as intended and that the unchanged parts have not been adversely affected by the modifications.

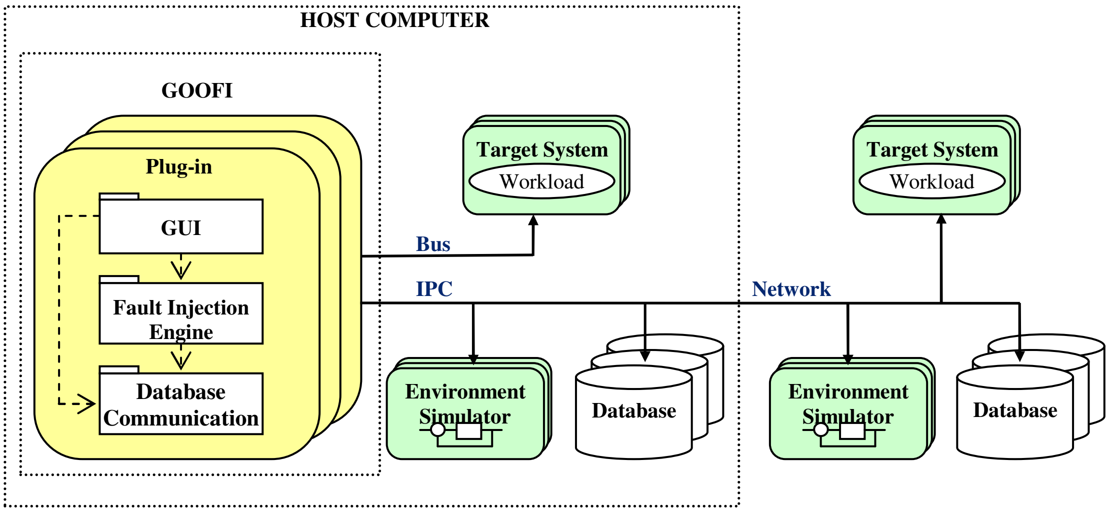


Figure 1. GOOFI overview.

GOOFI supports the implementation of new plug-ins by supplying a plug-in template, which is a partly implemented plug-in, and documentation on how to develop a complete GOOFI fault injection plug-in. Each plug-in must implement a set of functions that are called by the GOOFI framework. For example, GOOFI can extract campaign data from all plug-ins and control each plug-in to open, start and stop fault injection campaigns (a campaign is a set of experiments). The plug-in should also adhere to the database design given in the database template (see Section 4.6).

When support for a new fault injection technique is implemented using a plug-in template, software code for communicating with the target system (i.e. a microcontroller or microprocessor) has to be added to the template. To do this the programmer has to follow the documentation[3] while using the plug-in template. The support given by GOOFI for the plug-in developer not only simplifies the work but also ensures that all plug-ins have a similar look (e.g. tree structures). The plug-in is compiled separately and placed in the plug-in directory to be automatically found when GOOFI restarts.

[3] The documentation consists of a Software Requirements Specification (SRS) and a Software Design Specification (SDS).

4. The GOOFI MPC565 Nexus plug-in

This section describes the usage of a Nexus-based fault injection plug-in implemented using the GOOFI framework for a specific target system built around the MPC565 microcontroller. A detailed description of this set-up can be found in [15]. Here, we provide an overview of the conceptual design of the MPC565 Nexus plug-in and describe the main functionality supporting the four major phases of a fault injection campaign: the *configuration*, *set-up*, *fault injection* and *analysis phase*.

4.1 Conceptual design of the MPC565 Nexus plug-in

The conceptual design and the data flow of the set-up used for the MPC565 Nexus plug-in are shown in Figure 2. The GOOFI plug-in controls a commercial debugging environment called winIDEA [16] and uses its built-in script functionality to inject faults and observe the behavior of the system through a Nexus standard interface. A MySQL [17] database is used for storing the information required for conducting the fault injection experiments as well as the results of the experiments.

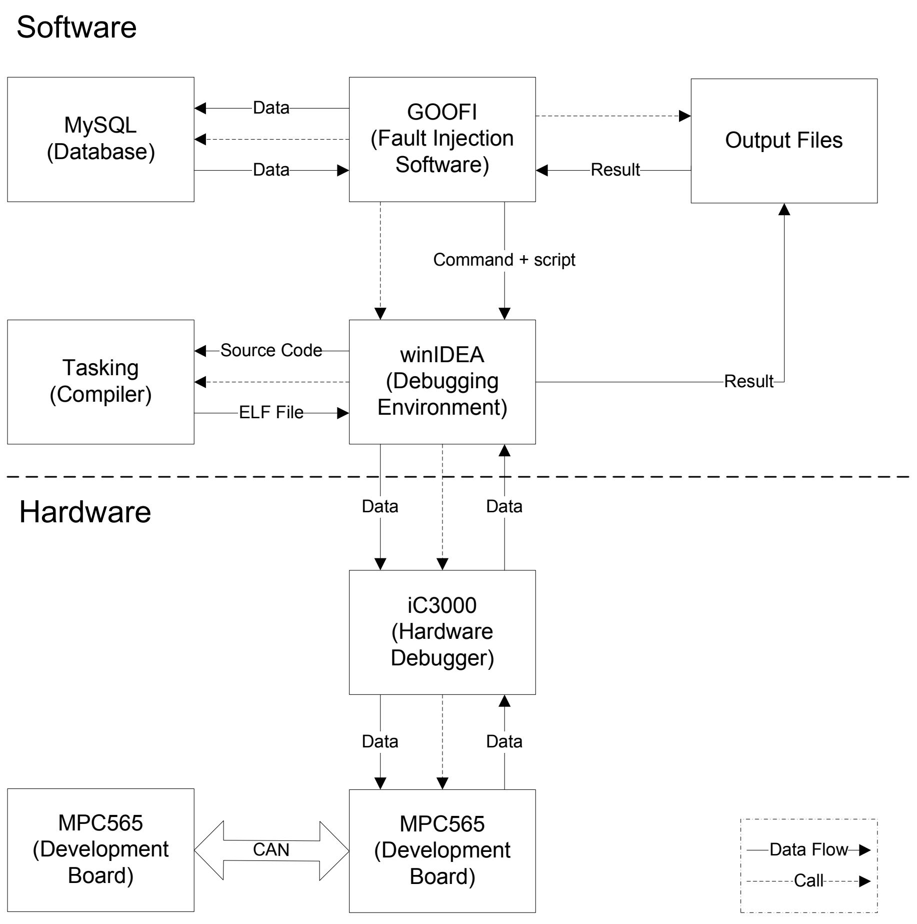


Figure 2. Conceptual design of the MPC565 Nexus plug-in set-up.

The development board used as target system is the phyCORE-MPC565 [18]. MPC565 is one of the first microcontrollers to implement the Nexus standard interface. The Nexus debug interface can be used to observe and manipulate register and memory contents and for creating real-time traces of the program and data flow. The interface implemented for the MPC565 is described in [19]. The board has several different I/O features implemented beside Nexus, which is useful if the design should be extended with additional peripheral hardware.

Some of the features are:

- 32-Bit Motorola MPC565 PowerPC microcontroller on an advanced PCB layout
- UART port: two RS-232 transceivers for channel A and B, also configurable as TTL
- CAN port: Three on-chip CAN controllers, two 82C251 CAN transceivers for channel A and B; also configurable as TTL
- JTAG/BDM/Nexus interfaces
- 40 to 56 MHz clock frequency
- 2 MB external SRAM, 1 MB external Flash memory and 1 MB on chip Flash memory

The winIDEA debugger communicates with the Nexus interface of the development board by using the iSYSTEM's iC3000 [20] active emulator. Features offered by iC3000 are:

- BDM, OnCE[4] and JTAG interface support
- Four hardware breakpoints and an unlimited number of software breakpoints
- On-chip and in-system flash programming
- Multiple voltage support (2V, 3V, 5V)
- Multi processor support
- Universal Serial Bus (USB) and serial (COM) high speed PC interfaces
- Ethernet (TCP/IP) interface
- Source level debugging for C and assembly code.

winIDEA itself does not include a compiler, so in order to compile C code to a PowerPC executable an external compiler is needed. The Tasking compiler v 2.1 for PowerPC from Altium [21] is used.

[4] OnCE is an emulation method where the processor's operation is entirely controlled through a specific port.

4.2 Configuration phase

The configuration phase involves adapting the tool to the target system. GOOFI uses tree structures in the graphical user interface to aid the user in the definition of a fault injection campaign during the set-up phase (see Section 4.3). From these trees, locations to observe and inject faults in can be selected. The user must create these tree structures in the configuration phase by providing information about the target processor (e.g. accessible registers) and the target application (e.g. where the application is located in memory).

The information about the target processor is independent of the target application. For the MPC565 Nexus plug-in, a textual description (*Register info* in Figure 3) of the MPC565 registers is downloaded into the plug-in database table RegisterInfo (see Section 4.6) when the database is created for the first time. The registers are grouped and defined by: *<block name>; <register name>; <read mask>; <write mask>*. For example, only some specific bits in the MPC565 special purpose register XER (Integer Exception Register) are readable and writable and therefore defined as: *Special Purpose Registers; XER; 0xFC00007F; 0xFC00007F.*

Information about where the code and data segments of the application should be located in memory must be defined in the linker file (*Sample.ind* in Figure 3). The Tasking compiler uses this information together with the *start-up code*[5], and the target application *source code*, to produce the executable (*ELF file*) and the *Memory map*. The memory map file contains information about the different memory segments used by the target application, e.g. at which address interval the program code is located. It also contains information about memory addresses, and names, of data variables in the target application.

By using the register description stored in the plug-in database together with the memory map, GOOFI can create trees showing information about the target system (as in Figure 5).

The processor must be initialized before the executable file can be downloaded, which is done by a script (*MPC565.ini* in Figure 3). This script initializes registers containing necessary information for the processor about connected peripheral devices (e.g. size and timing parameters of external memory), and enables the memory to be used (e.g. internal flash or external RAM). The commands in this file are executed by winIDEA/iC3000 before the executable *ELF file* is downloaded to the MPC565 development board. For programs located in the on-board flash memory, the initialization performed in this script must be done from the application running on the development board instead. The *Fault injection script* is used in the fault injection phase (see Section 4.4) and controls winIDEA to carry out the fault injection experiments defined in the set-up phase (see Section 4.3).

[5] The start-up code configures the stack and heap before the main method of the application is called.

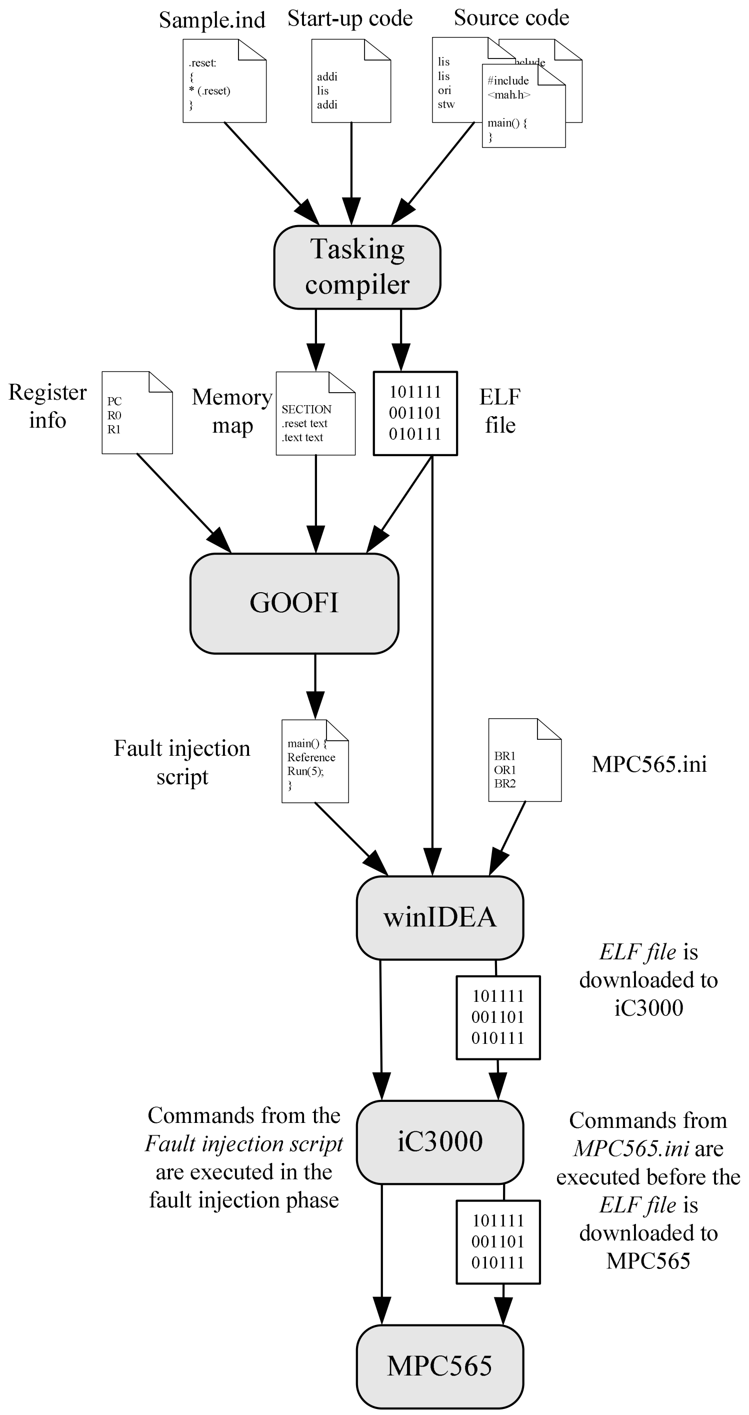


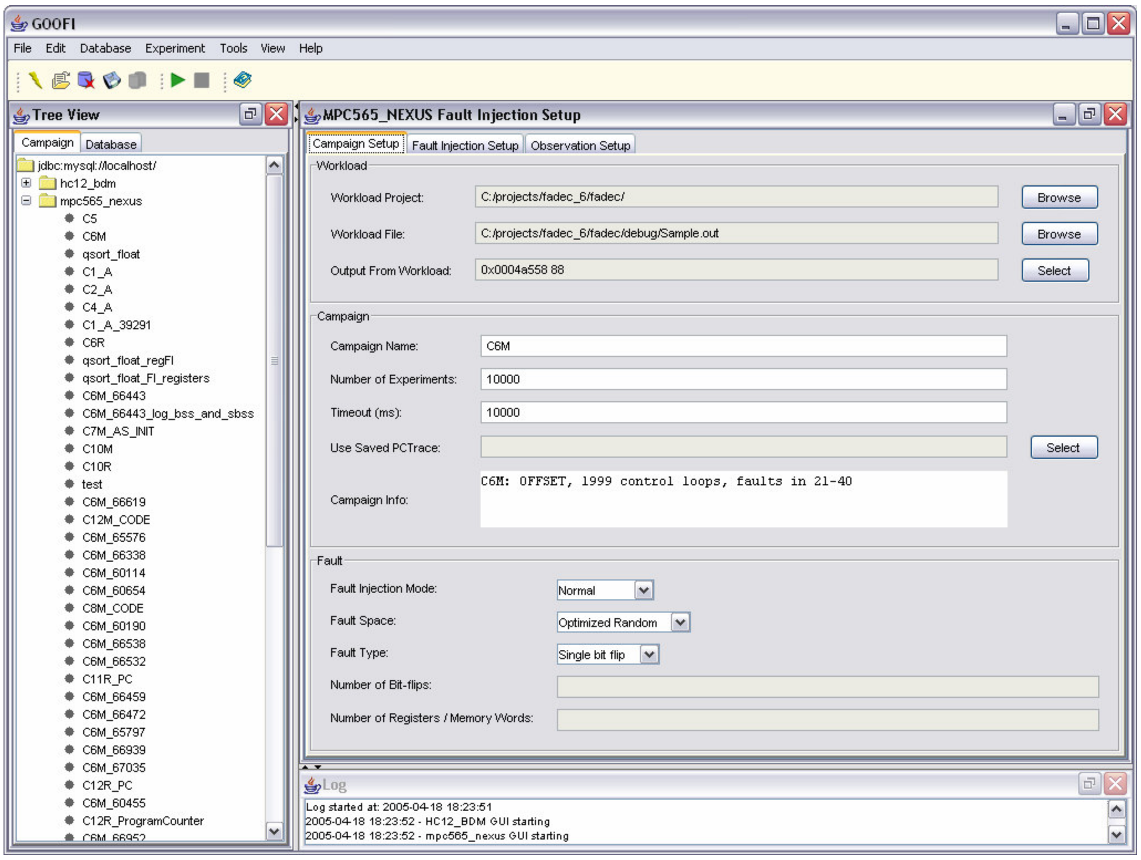
Figure 3. The MPC565 Nexus plug-in configuration set-up.

4.3 Set-up phase

The set-up phase is used for setting up fault injection campaigns and generally involves three steps. In the first step, the user has to enter data about the campaign in the *campaign setup* tab of the plug-in window, see Figure 4. Then, specific information about where and when faults should be injected are defined in the *fault injection setup* tab. Finally, the registers and memory positions that the user wants to observe are defined in the *observation setup* tab. These steps are described in more detail in the following subsections.

4.3.1 Campaign set-up

From the menus in the GUI, fault injection campaigns can be configured by starting the corresponding plug-in for a chosen target system and fault injection technique. In Figure 4, the MPC565 Nexus plug-in has been chosen.


GOOFI
File Edit Database Experiment Tools View Help
Tree View
Campaign Database
MPC565_NEXUS Fault Injection Setup
Campaign Setup Fault Injection Setup Observation Setup
Browse
Browse
Select
Select
Log

```
jdbc:mysql://localhost/
  hc12_bdm
  mpc565_nexus
    C5
    C6M
    qsort_float
    C1_A
    C2_A
    C4_A
    C1_A_39291
    C6R
    qsort_float_regFI
    qsort_float_FI_registers
    C6M_66443
    C6M_66443_log_bss_and_sbss
    C7M_AS_INIT
    C10M
    C10R
    test
    C6M_66619
    C12M_CODE
    C6M_65576
    C6M_66338
    C6M_60114
    C6M_60654
    C8M_CODE
    C6M_60190
    C6M_66538
    C6M_66532
    C11R_PC
    C6M_66459
    C6M_66472
    C6M_65797
    C6M_66939
    C6M_67035
    C12R_PC
    C6M_60455
    C12R_ProgramCounter
    C6M_66952
```

Workload

Workload Project:	C:/projects/fadec_6/fadec/
Workload File:	C:/projects/fadec_6/fadec/debug/Sample.out
Output From Workload:	0x0004a558 88

Campaign

Campaign Name:	C6M
Number of Experiments:	10000
Timeout (ms):	10000
Use Saved PCTrace:	
Campaign Info:	C6M: OFFSET, 1999 control loops, faults in 21-40

Fault

Fault Injection Mode:	Normal
Fault Space:	Optimized Random
Fault Type:	Single bit flip
Number of Bit-flips:	
Number of Registers / Memory Words:	

```
Log started at: 2005-04-18 18:23:51
2005-04-18 18:23:52 - HC12_BDM GUI starting
2005-04-18 18:23:52 - mpc565_nexus GUI starting
```

Figure 4. Campaign set-up.

After selecting the path to the winIDEA workload project and workload (the target system executable), via the file system browser, the output variables from the workload has to be selected via a tree structure listing the memory addresses and the corresponding variable names in the program.

The campaign name, the number of experiments in the campaign and the time-out value for the experiments must also be entered. A fault injection experiment can be terminated 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 or is executed as an infinite loop.

A fault injection campaign requires a reference run (fault-free run), described in Section 4.4. A reference run from an earlier campaign can be reused by pressing the *Select* button to the right of *Use Saved PC Trace*. The user can select a reference run from a campaign using the same workload and settings as the one being configured. When the campaign is saved, the PC trace (the values of the program counter logged during the execution of the reference run) and logged registers from the old reference will be copied to a new reference experiment belonging to the new campaign.

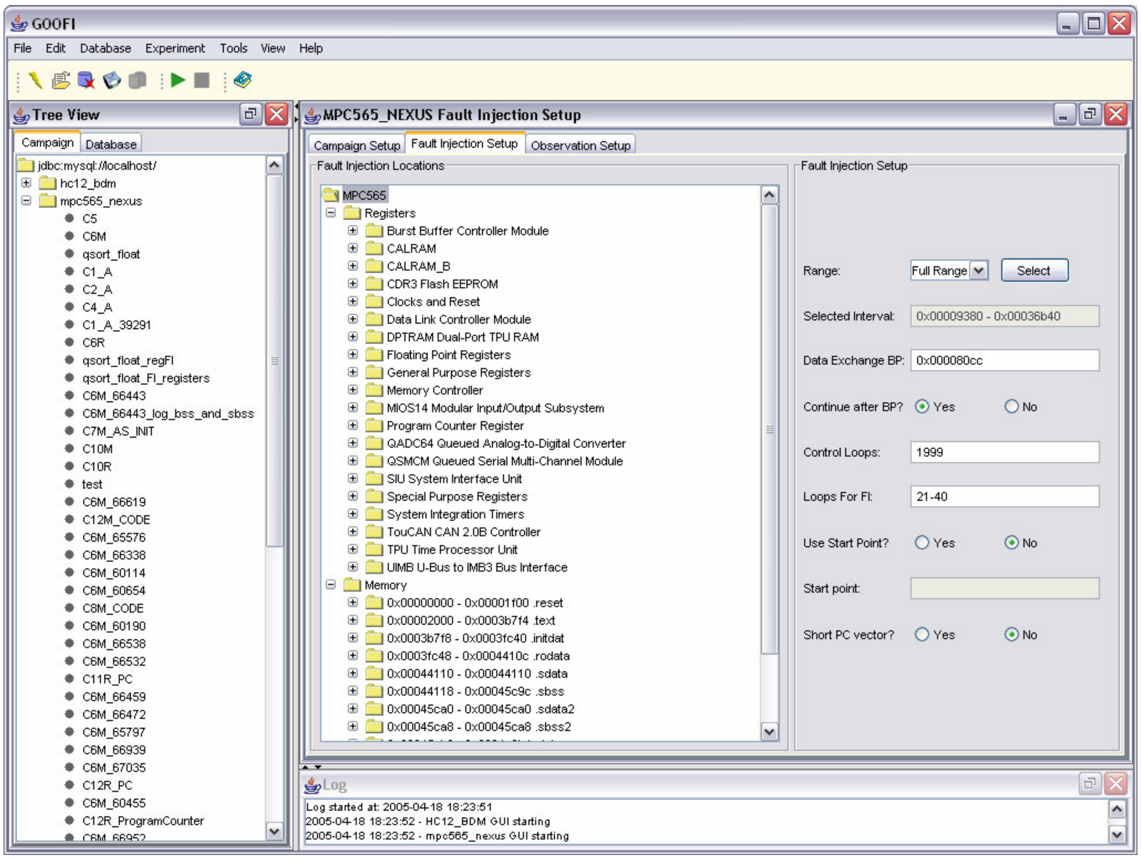
The user can choose between three fault injection modes:

- *Normal* – User-selected memory is saved after each control loop and user-selected registers are saved at the end of the execution.
- *Normal and Trace* – The program flow is saved in addition to the Normal mode.
- *Detailed* – The program flow and user-selected registers and memory locations are saved after each executed instruction.

The user may also choose if a pre-injection analysis should be carried out to prevent injection of faults that are overwritten or never activated by the system (see Section 5). The user can also choose whether to inject single or multiple bit-flip faults. 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 by open a defined campaign, change the contents and save the campaign by a new name.

4.3.2 Fault injection set-up

By clicking the *fault injection setup* tab in Figure 5 the user can select where and when the faults should be injected.


GOOFI
File Edit Database Experiment Tools View Help
Tree View
Campaign Database
jdbc:mysql://localhost/
hc12_bdm
mpc565_nexus
C5
C6M
qsort_float
C1_A
C2_A
C4_A
C1_A_39291
C6R
qsort_float_regFI
qsort_float_FI_registers
C6M_66443
C6M_66443_log_bss_and_sbss
C7M_AS_INIT
C10M
C10R
test
C6M_66619
C12M_CODE
C6M_65576
C6M_66338
C6M_60114
C6M_60654
C8M_CODE
C6M_60190
C6M_66538
C6M_66532
C11R_PC
C6M_66459
C6M_66472
C6M_65797
C6M_66939
C6M_67035
C12R_PC
C6M_60455
C12R_ProgramCounter
C6M_66952
MPC565_NEXUS Fault Injection Setup
Campaign Setup Fault Injection Setup Observation Setup
Fault Injection Locations
MPC565
Registers
Burst Buffer Controller Module
CALRAM
CALRAM_B
CDR3 Flash EEPROM
Clocks and Reset
Data Link Controller Module
DPTRAM Dual-Port TPU RAM
Floating Point Registers
General Purpose Registers
Memory Controller
MIOS14 Modular Input/Output Subsystem
Program Counter Register
QADC64 Queued Analog-to-Digital Converter
QSMCM Queued Serial Multi-Channel Module
SIU System Interface Unit
Special Purpose Registers
System Integration Timers
TouCAN CAN 2.0B Controller
TPU Time Processor Unit
UIMB U-Bus to IMB3 Bus Interface
Memory
0x00000000 - 0x00001f00 .reset
0x00002000 - 0x0003b7f4 .text
0x0003b7f8 - 0x0003fc40 .initdat
0x0003fc48 - 0x0004410c .rodata
0x00044110 - 0x00044110 .sdata
0x00044118 - 0x00045c9c .sbss
0x00045ca0 - 0x00045ca0 .sdata2
0x00045ca8 - 0x00045ca8 .sbss2
Fault Injection Setup
Select
Log

Range: Full Range

Selected Interval: 0x00009380 - 0x00036b40

Data Exchange BP: 0x000080cc

Continue after BP? [✓ Yes] No

Control Loops: 1999

Loops For FI: 21-40

Use Start Point? Yes [✓ No]

Start point:

Short PC vector? Yes [✓ No]

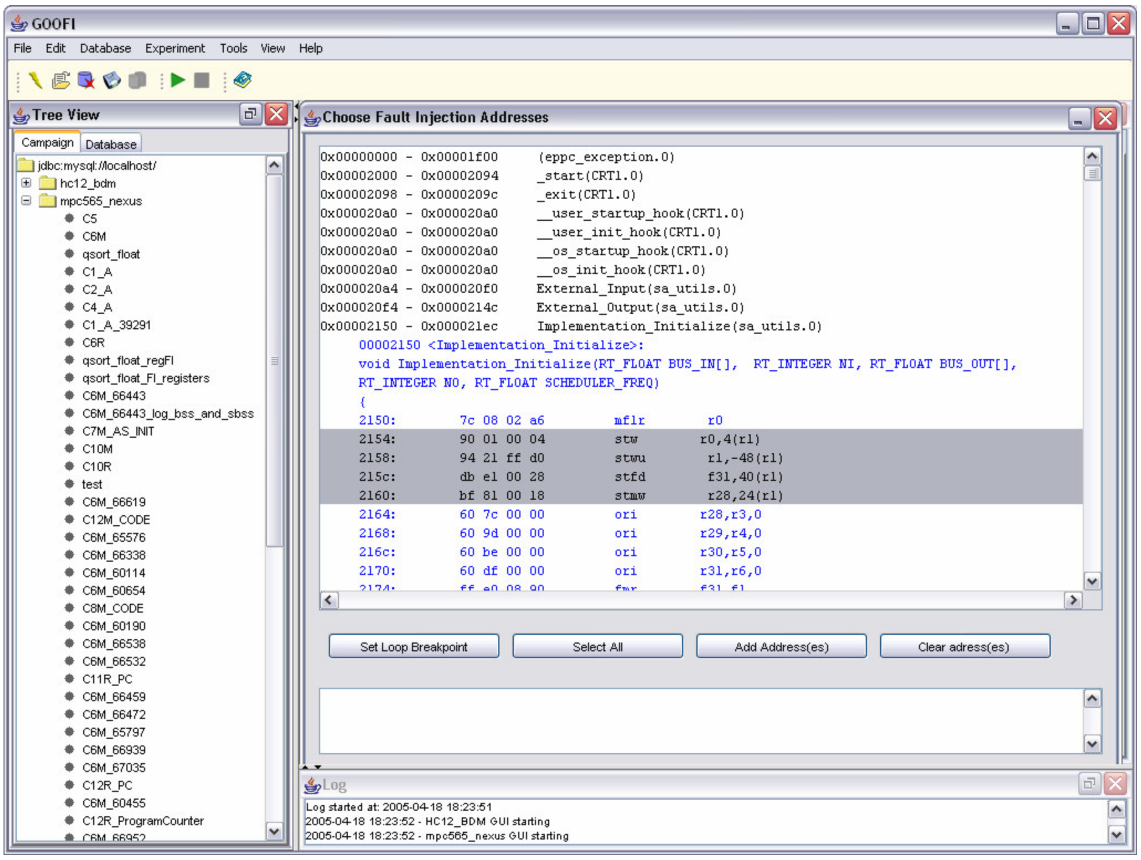
```
Log started at: 2005-04-18 18:23:51
2005-04-18 18:23:52 - HC12_BDM GUI starting
2005-04-18 18:23:52 - mpc565_nexus GUI starting
```

Figure 5. Fault injection setup.

The tree shows all registers and memory positions in which faults can be injected. The user can select to inject faults in registers at the bit level and in memory words. Multiple selections can be made by clicking the left mouse button and holding down the *Shift* key.

The code range, used to decide which instructions that are executed when faults may be injected (corresponding to the time interval for fault injection), is set to *Full Range* by default, allowing faults to be injected during execution of any instruction. This range can be changed by pressing the *Select* button. The window in Figure 6 will appear and allowing the code range to be selected. By double clicking on a name of a function in the source code, the assembly code representing the function will be shown.

The loop breakpoint (*data exchange BP)* is also selected in this window by pressing the *Set Loop Breakpoint* button. This address identifies where the workload should stop its execution and log user-selected memory locations and registers. For control applications, this address identifies a specific location in the control loop. The breakpoint address can also be entered in the *Data Exchange BP* field in the window shown in Figure 5.



```
jdbc:mysql://localhost/
  hc12_bdm
  mpc565_nexus
    C5
    C6M
    qsort_float
    C1_A
    C2_A
    C4_A
    C1_A_39291
    C6R
    qsort_float_regFI
    qsort_float_FI_registers
    C6M_66443
    C6M_66443_log_bss_and_sbss
    C7M_AS_INIT
    C10M
    C10R
    test
    C6M_66619
    C12M_CODE
    C6M_65576
    C6M_66338
    C6M_60114
    C6M_60654
    C8M_CODE
    C6M_60190
    C6M_66538
    C6M_66532
    C11R_PC
    C6M_66459
    C6M_66472
    C6M_65797
    C6M_66939
    C6M_67035
    C12R_PC
    C6M_60455
    C12R_ProgramCounter
```

```
0x00000000 - 0x00001f00     (eppc_exception.0)
0x00002000 - 0x00002094     _start(CRT1.0)
0x00002098 - 0x0000209c     _exit(CRT1.0)
0x000020a0 - 0x000020a0     __user_startup_hook(CRT1.0)
0x000020a0 - 0x000020a0     __user_init_hook(CRT1.0)
0x000020a0 - 0x000020a0     __os_startup_hook(CRT1.0)
0x000020a0 - 0x000020a0     __os_init_hook(CRT1.0)
0x000020a4 - 0x000020f0     External_Input(sa_utils.0)
0x000020f4 - 0x0000214c     External_Output(sa_utils.0)
0x00002150 - 0x000021ec     Implementation_Initialize(sa_utils.0)
     00002150 <Implementation_Initialize>:
     void Implementation_Initialize(RT_FLOAT BUS_IN[],  RT_INTEGER NI, RT_FLOAT BUS_OUT[],
     RT_INTEGER NO, RT_FLOAT SCHEDULER_FREQ)
     {
     2150:        7c 08 02 a6        mflr        r0
     2154:        90 01 00 04        stw        r0,4(r1)
     2158:        94 21 ff d0        stwu        r1,-48(r1)
     215c:        db e1 00 28        stfd        f31,40(r1)
     2160:        bf 81 00 18        stmw        r28,24(r1)
     2164:        60 7c 00 00        ori        r28,r3,0
     2168:        60 9d 00 00        ori        r29,r4,0
     216c:        60 be 00 00        ori        r30,r5,0
     2170:        60 df 00 00        ori        r31,r6,0
```

```
Log started at: 2005-04-18 18:23:51
2005-04-18 18:23:52 - HC12_BDM GUI starting
2005-04-18 18:23:52 - mpc565_nexus GUI starting
```

Figure 6. Window for selecting breakpoints and address intervals for fault injection.

If the workload is a control application, the *Continue after BP* should be enabled. The number of control loops that should be executed is entered in the *Control Loops* field and the control loop interval during which faults may be injected should be entered in the *Loops for FI* field.

In certain cases it may be useful to use the *Start Point*. When this alternative is selected, the PC trace will start when this point has been reached. For example, if *main* is specified as a *Start Point*, the initialization[6] of the processor will not be included in the PC trace which may decrease the time the reference run takes. The start point should be a function name defined in the workload.

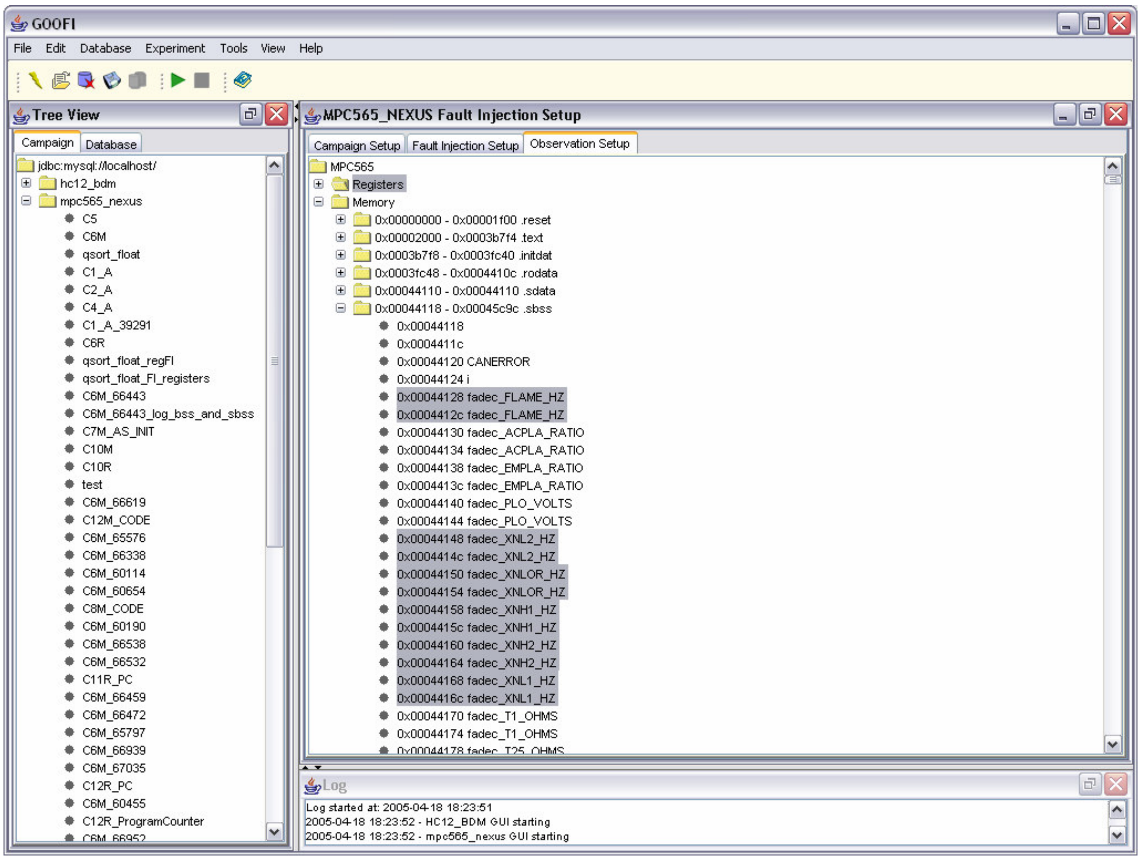
If a single function is used for the fault injection range, the *Short PC Vector* might be

[6] Configuration of the stack pointer, initialization of the heap and other necessary initializations of the processor before control is passed to the application.

useful. If this is selected, only a PC trace of the selected range will be included in the reference. If *No* is selected, the PC trace will include all instructions executed during the control loops selected for fault injection.

4.3.3 Observation set-up

From the GUI in Figure 7 the user selects the locations to observe from a tree. The selected registers and memory positions will be logged at the selected loop breakpoint (the *Data Exchange BP* in Figure 5) in the fault injection set-up.


GOOFI
File Edit Database Experiment Tools View Help
Tree View
Campaign Database
MPC565_NEXUS Fault Injection Setup
Campaign Setup Fault Injection Setup Observation Setup
Log

jdbc:mysql://localhost/
- hc12_bdm
- mpc565_nexus
  - C5
  - C6M
  - qsort_float
  - C1_A
  - C2_A
  - C4_A
  - C1_A_39291
  - C6R
  - qsort_float_regFI
  - qsort_float_FI_registers
  - C6M_66443
  - C6M_66443_log_bss_and_sbss
  - C7M_AS_INIT
  - C10M
  - C10R
  - test
  - C6M_66619
  - C12M_CODE
  - C6M_65576
  - C6M_66338
  - C6M_60114
  - C6M_60654
  - C8M_CODE
  - C6M_60190
  - C6M_66538
  - C6M_66532
  - C11R_PC
  - C6M_66459
  - C6M_66472
  - C6M_65797
  - C6M_66939
  - C6M_67035
  - C12R_PC
  - C6M_60455
  - C12R_ProgramCounter
  - C6M_66952

MPC565
- Registers
- Memory
  - 0x00000000 - 0x00001f00 .reset
  - 0x00002000 - 0x0003b7f4 .text
  - 0x0003b7f8 - 0x0003fc40 .initdat
  - 0x0003fc48 - 0x0004410c .rodata
  - 0x00044110 - 0x00044110 .sdata
  - 0x00044118 - 0x00045c9c .sbss
    - 0x00044118
    - 0x0004411c
    - 0x00044120 CANERROR
    - 0x00044124 i
    - 0x00044128 fadec_FLAME_HZ
    - 0x0004412c fadec_FLAME_HZ
    - 0x00044130 fadec_ACPLA_RATIO
    - 0x00044134 fadec_ACPLA_RATIO
    - 0x00044138 fadec_EMPLA_RATIO
    - 0x0004413c fadec_EMPLA_RATIO
    - 0x00044140 fadec_PLO_VOLTS
    - 0x00044144 fadec_PLO_VOLTS
    - 0x00044148 fadec_XNL2_HZ
    - 0x0004414c fadec_XNL2_HZ
    - 0x00044150 fadec_XNLOR_HZ
    - 0x00044154 fadec_XNLOR_HZ
    - 0x00044158 fadec_XNH1_HZ
    - 0x0004415c fadec_XNH1_HZ
    - 0x00044160 fadec_XNH2_HZ
    - 0x00044164 fadec_XNH2_HZ
    - 0x00044168 fadec_XNL1_HZ
    - 0x0004416c fadec_XNL1_HZ
    - 0x00044170 fadec_T1_OHMS
    - 0x00044174 fadec_T1_OHMS
    - 0x00044178 fadec_T25_OHMS

```
Log started at: 2005-04-18 18:23:51
2005-04-18 18:23:52 - HC12_BDM GUI starting
2005-04-18 18:23:52 - mpc565_nexus GUI starting
```

Figure 7. Observation setup.

4.4 Fault injection phase

Fault injection experiments are defined as scripts which are executed by winIDEA. The scripts are created by GOOFI based on a script template. The template is designed primarily for handling typical control algorithms, i.e. software that runs in an infinite loop and calculates new output each loop. However, the script is generic and can thus be used with applications which are not running eternally.

A GOOFI fault injection campaign requires one reference run (fault-free run) in which necessary information is collected, e.g. the program flow as a program counter (PC) trace, and to store the contents of the user-selected memory and registers. This information is later compared with the experimental results to derive dependability measures such as error detection coverage. The reference run is illustrated in Figure 8.

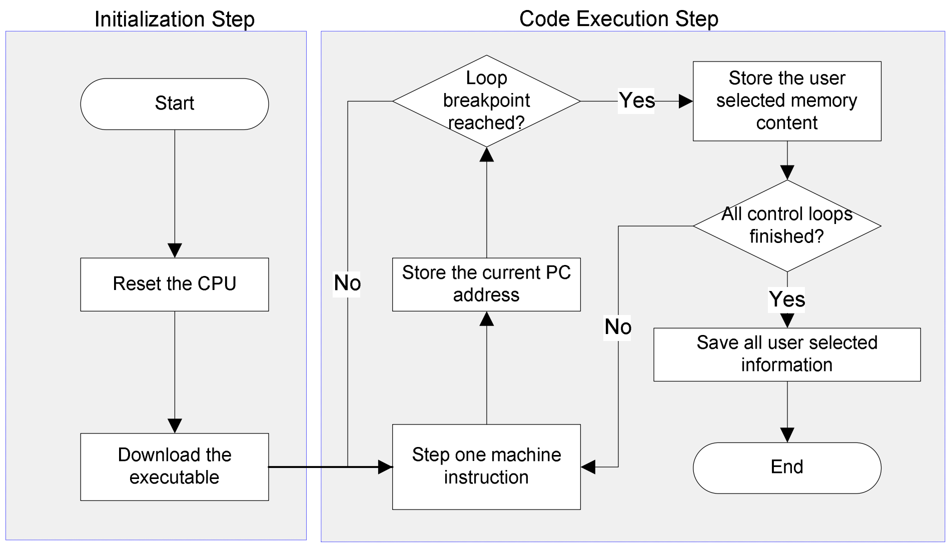

Initialization Step
Code Execution Step
Start
Reset the CPU
Download the executable
Step one machine instruction
Store the current PC address
Loop breakpoint reached?
No
Yes
Store the user selected memory content
All control loops finished?
No
Yes
Save all user selected information
End

Figure 8. Reference run procedure.

A PC trace is needed in order to decide when a fault should be injected and to allow fault injection experiments to be repeated. The precise time and location of the fault injection must be determined and these two parameters are translated to the parameters listed below.

- Control loop index
- Breakpoint address
- Number of breakpoint invocations within the control loop
- The fault injection location

The control loop index is specific for control applications which execute in cycles. It defines the loop during which a fault should be injected. For applications that do not execute in loops, the control loop index is always set to one. The breakpoint address specifies the breakpoint position inside the control loop and the number of breakpoint invocations specifies the number of times this breakpoint should be reached before fault injection. When the user selected breakpoint has been reached, the instruction at the breakpoint address has not yet been executed. Thus, the fault injection occurs before the user selected breakpoint instruction executes.

A typical fault injection experiment running in normal operation mode can be represented in four major steps as described in Figure 9.

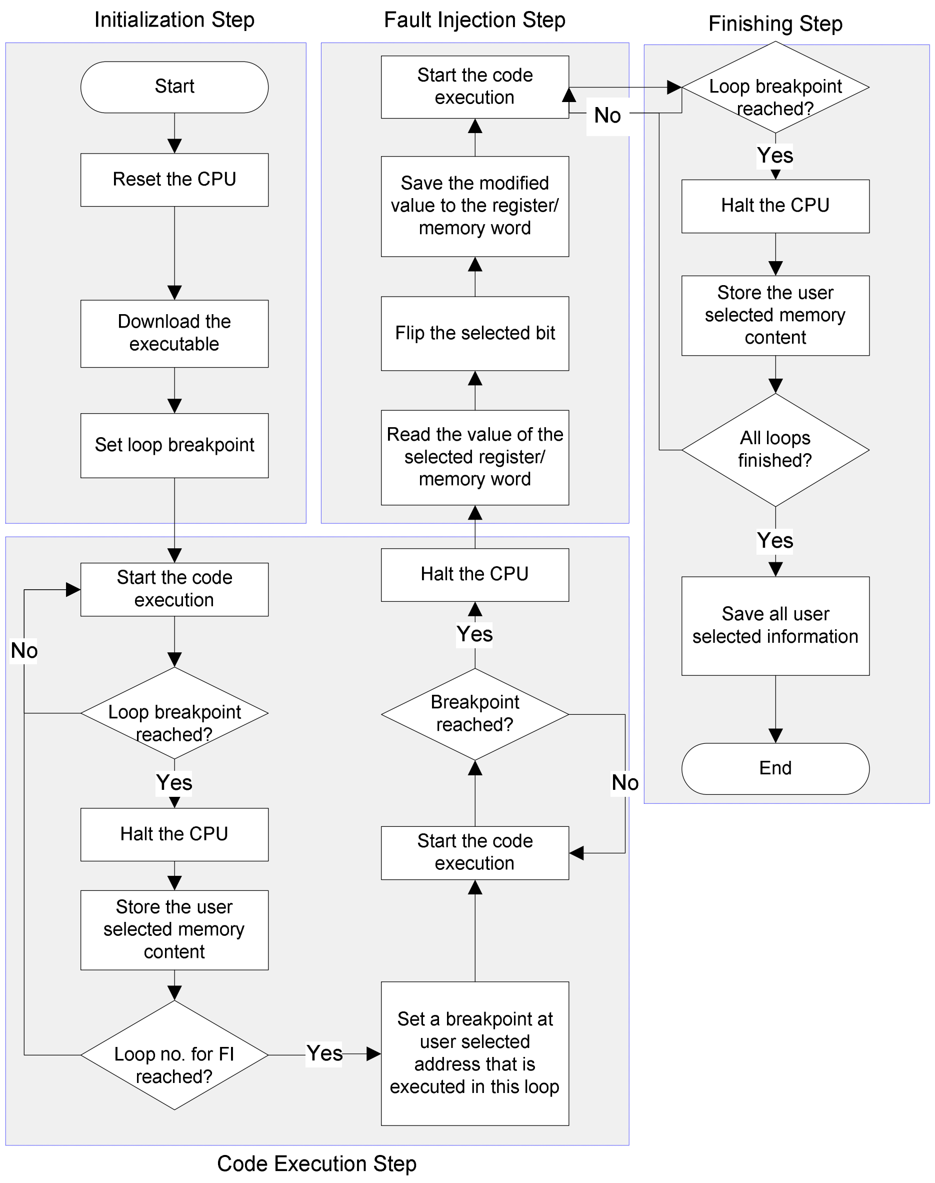


Figure 9. Fault injection procedure for normal mode.

4.5 Analysis phase

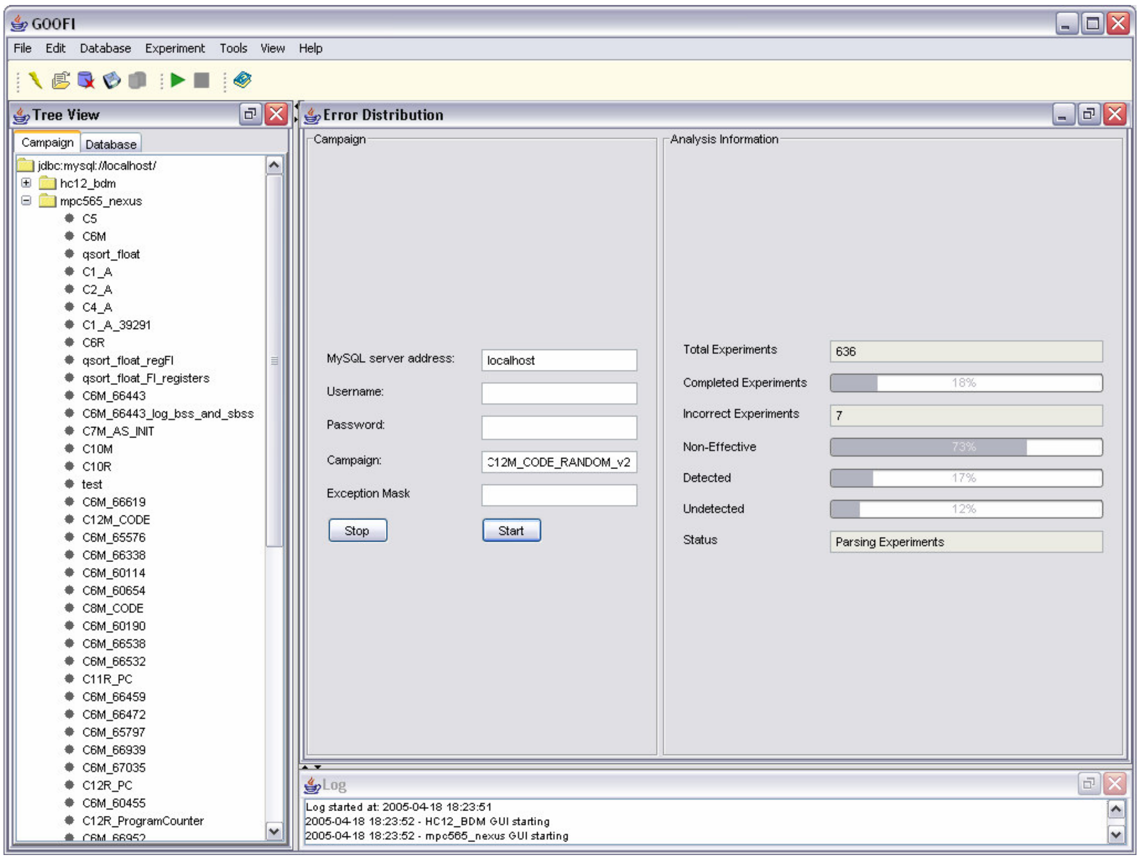
The logged data is analyzed in the analysis phase in order to obtain various dependability measures. The kind of measures obtainable depends on the target system. There are two tools included in the GOOFI tools menu that supports analysis of the experiments; an error distribution classification tool and a database visualization tool.

4.5.1 The Error Distribution tool

The Error Distribution tool classifies experiments in a campaign into the appropriate type of error and stores this information in the database table *ErrorDistribution*. This data can be used to create the distribution of errors and the distribution of exceptions for errors that have been detected. Experiments are classified into one of the following three types of errors:

- *Non-effective* – The fault had no effect during the time the experiment elapsed.
- *Detected* – The hardware mechanisms of the processor has caught the error.
- *Undetected* – No hardware mechanisms have been triggered but the output from the system is incorrect.

An error distribution can be created by selecting *ErrorDistribution* under the *Tools* menu in GOOFI. The window in Figure 10 will appear showing the progress and the percentage of each type of error. This information is updated after each classified experiment. Incorrect Experiments are experiments that stopped before a fault was injected.



MySQL server address: localhost
Username:
Password:
Campaign: C12M_CODE_RANDOM_v2
Exception Mask

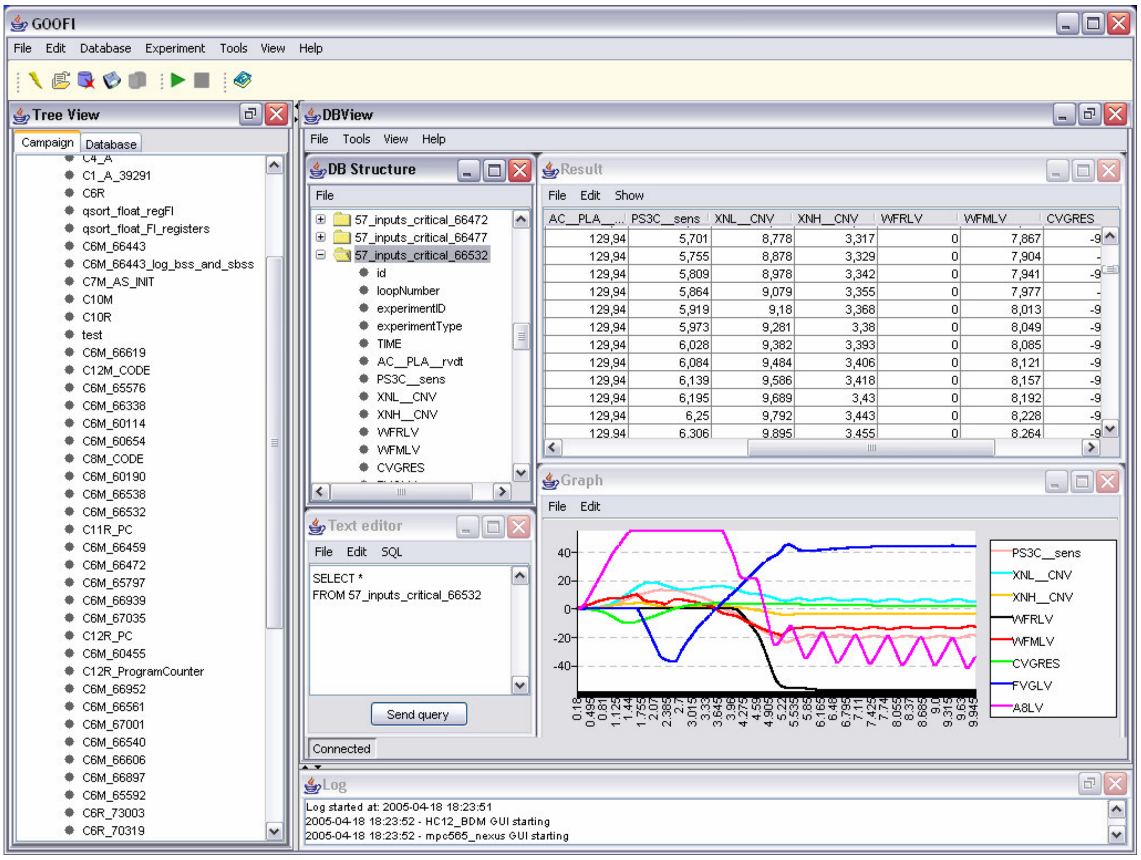
Total Experiments 636
Completed Experiments 18%
Incorrect Experiments 7
Non-Effective 73%
Detected 17%
Undetected 12%
Status Parsing Experiments

```
Log started at: 2005-04-18 18:23:51
2005-04-18 18:23:52 - HC12_BDM GUI starting
2005-04-18 18:23:52 - mpc565_nexus GUI starting
```

Figure 10. The Error Distribution tool.

4.5.2 The Database Visualization tool

The Database Visualization tool is a general tool to communicate with MySQL databases and is started from the *Tools* menu in GOOFI. The main feature with this tool is that experiment results can be analyzed visually. Figure 11 shows a tree view of a database containing measured sensor values from a jet engine, a user entered SQL query, the result table and a graphical representation of the result table.



AC__PLA__...	PS3C__sens	XNL__CNV	XNH__CNV	WFRLV	WFMLV	CVGRES
129,94	5,701	8,778	3,317	0	7,867	-9
129,94	5,755	8,878	3,329	0	7,904	-
129,94	5,809	8,978	3,342	0	7,941	-9
129,94	5,864	9,079	3,355	0	7,977	-
129,94	5,919	9,18	3,368	0	8,013	-9
129,94	5,973	9,281	3,38	0	8,049	-9
129,94	6,028	9,382	3,393	0	8,085	-9
129,94	6,084	9,484	3,406	0	8,121	-9
129,94	6,139	9,586	3,418	0	8,157	-9
129,94	6,195	9,689	3,43	0	8,192	-9
129,94	6,25	9,792	3,443	0	8,228	-9
129,94	6,306	9,895	3,455	0	8,264	-9

```
SELECT *
FROM 57_inputs_critical_66532
```

```
Log started at: 2005-04-18 18:23:51
2005-04-18 18:23:52 - HC12_BDM GUI starting
2005-04-18 18:23:52 - mpc565_nexus GUI starting
```

Figure 11. The Database Visualization tool.

4.5.3 Repeating experiments

During the analysis phase, some interesting result may require a more detailed investigation. Therefore, it is possible to repeat an experiment by selecting the *Repeat Experiment* from the GOOFI menu. This will create a new fault injection campaign which can be configured as usual but certain values can not be modified. A campaign that repeats an experiment does not need a new PC trace and the reference run will therefore only run the control loops and log the user-selected observable registers and memory locations. New campaigns can be configured based on the configuration for an existing experiment. Selecting *Open* instead of *Repeat* in the *Repeat Experiment* window will load a campaign based on the selected experiment. A new campaign can now be configured as usual.

4.6 The GOOFI database

A model of the GOOFI database used by the MPC565 Nexus plug-in is shown in Figure 12. All plug-ins creates and uses their own database and GOOFI can extract campaign information from all databases and e.g. create database backups.

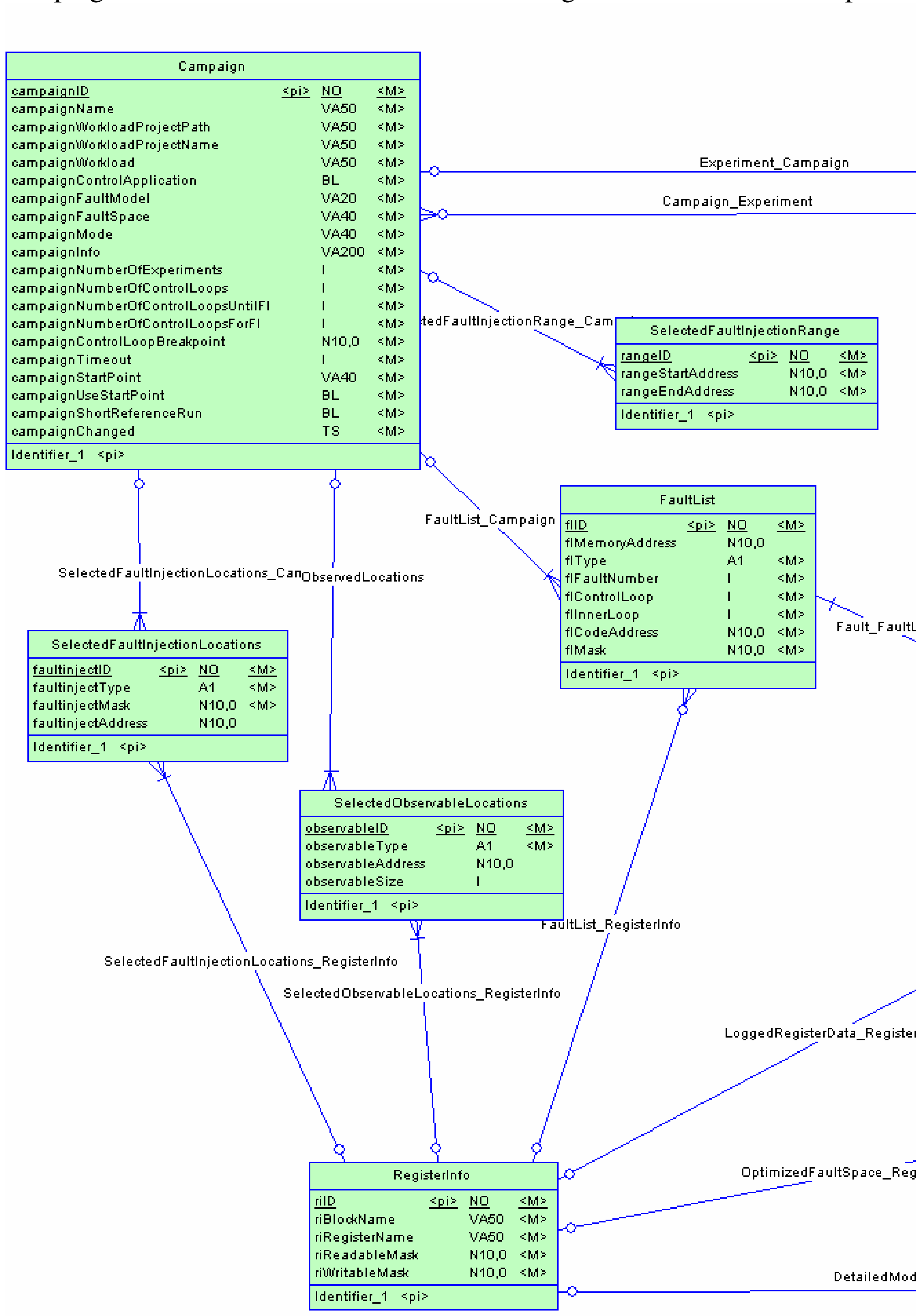

Campaign
campaignID <pi> NO <M>
campaignName VA50 <M>
campaignWorkloadProjectPath VA50 <M>
campaignWorkloadProjectName VA50 <M>
campaignWorkload VA50 <M>
campaignControlApplication BL <M>
campaignFaultModel VA20 <M>
campaignFaultSpace VA40 <M>
campaignMode VA40 <M>
campaignInfo VA200 <M>
campaignNumberOfExperiments I <M>
campaignNumberOfControlLoops I <M>
campaignNumberOfControlLoopsUntilFI I <M>
campaignNumberOfControlLoopsForFI I <M>
campaignControlLoopBreakpoint N10,0 <M>
campaignTimeout I <M>
campaignStartPoint VA40 <M>
campaignUseStartPoint BL <M>
campaignShortReferenceRun BL <M>
campaignChanged TS <M>
Identifier_1 <pi>
Experiment_Campaign
Campaign_Experiment
tedFaultInjectionRange_Cam
SelectedFaultInjectionRange
rangeID <pi> NO <M>
rangeStartAddress N10,0 <M>
rangeEndAddress N10,0 <M>
Identifier_1 <pi>
FaultList_Campaign
FaultList
flID <pi> NO <M>
flMemoryAddress N10,0
flType A1 <M>
flFaultNumber I <M>
flControlLoop I <M>
flInnerLoop I <M>
flCodeAddress N10,0 <M>
flMask N10,0 <M>
Identifier_1 <pi>
Fault_FaultL
SelectedFaultInjectionLocations_Can
ObservedLocations
SelectedFaultInjectionLocations
faultinjectID <pi> NO <M>
faultinjectType A1 <M>
faultinjectMask N10,0 <M>
faultinjectAddress N10,0
Identifier_1 <pi>
SelectedObservableLocations
observableID <pi> NO <M>
observableType A1 <M>
observableAddress N10,0
observableSize I
Identifier_1 <pi>
FaultList_RegisterInfo
SelectedFaultInjectionLocations_RegisterInfo
SelectedObservableLocations_RegisterInfo
LoggedRegisterData_Registe
OptimizedFaultSpace_Reg
DetailedMod
RegisterInfo
riID <pi> NO <M>
riBlockName VA50 <M>
riRegisterName VA50 <M>
riReadableMask N10,0 <M>
riWritableMask N10,0 <M>
Identifier_1 <pi>

Figure 12. The GOOFI plug-in database.

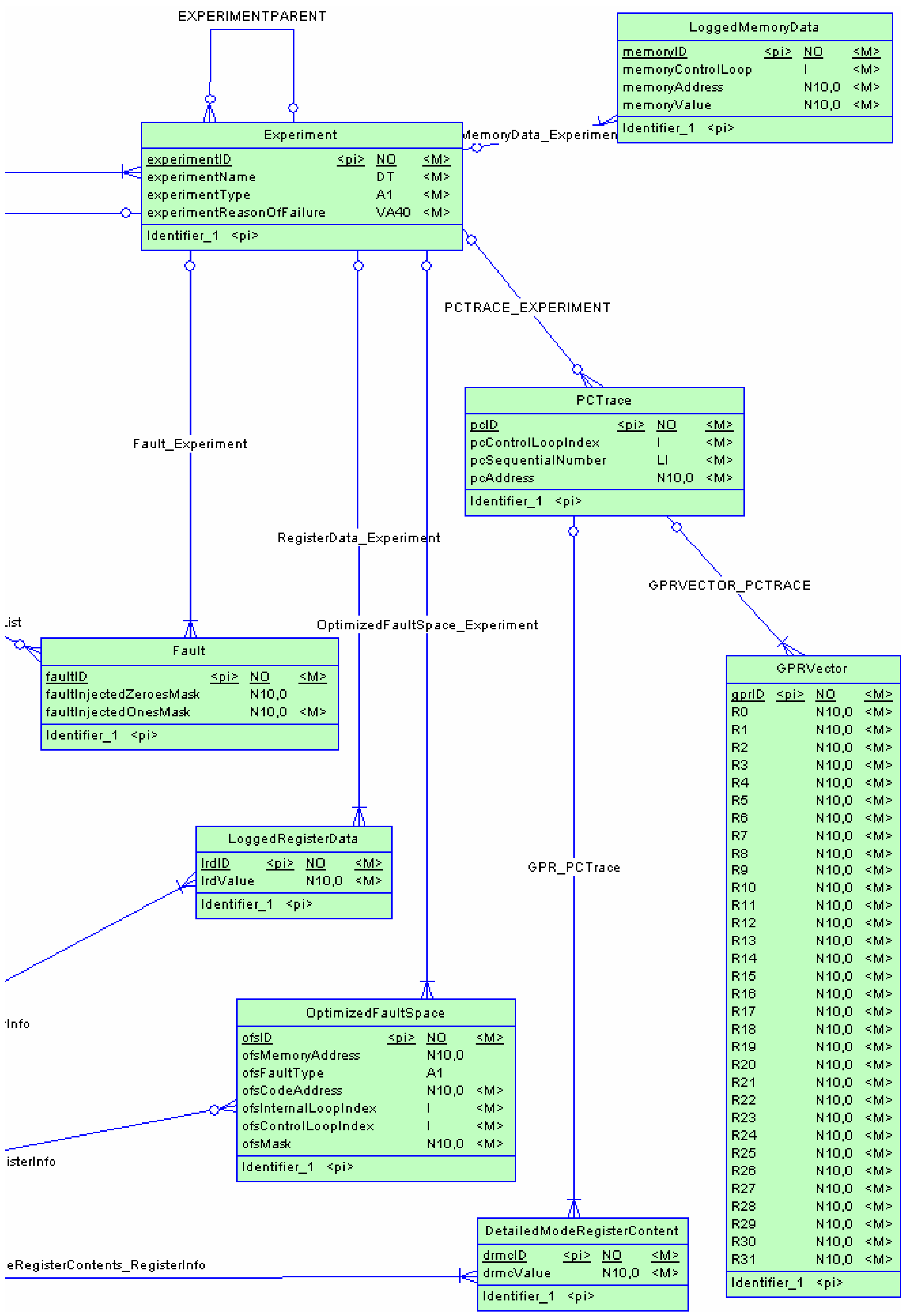


Figure 12, contd. The GOOFI plug-in database.

5. Extending GOOFI to support pre-injection analysis

A problem commonly observed during fault injection campaigns is that not all faults fulfill the purpose of disturbing the system. Often 80-90% of randomly injected faults are not activated. A fault placed in a register just before the register is written or faults that are injected into unused memory locations are examples of faults with no possibility of activation. In most tools the location and the time for fault injection are chosen randomly from the complete fault-space, which is typically extremely large. The statistical implication of this is that the cost of obtaining appropriate confidence levels of the dependability measures becomes unnecessarily high.

The resources (registers and memory) available in computers are, usually, greater than what the applications executed require. This fact motivates a fault injection optimization strategy in which faults are injected only in used resources.

Avoiding unused memory regions might be done manually by analyzing the memory map of the application and choosing the segments (stack, heap, etc.) as valid locations for fault injection. This approach is quite simple but does not consider the way resources are used along the time dimension. The following graph of the memory reads by a jet engine controller [22] during one control loop of execution clearly illustrates this fact.

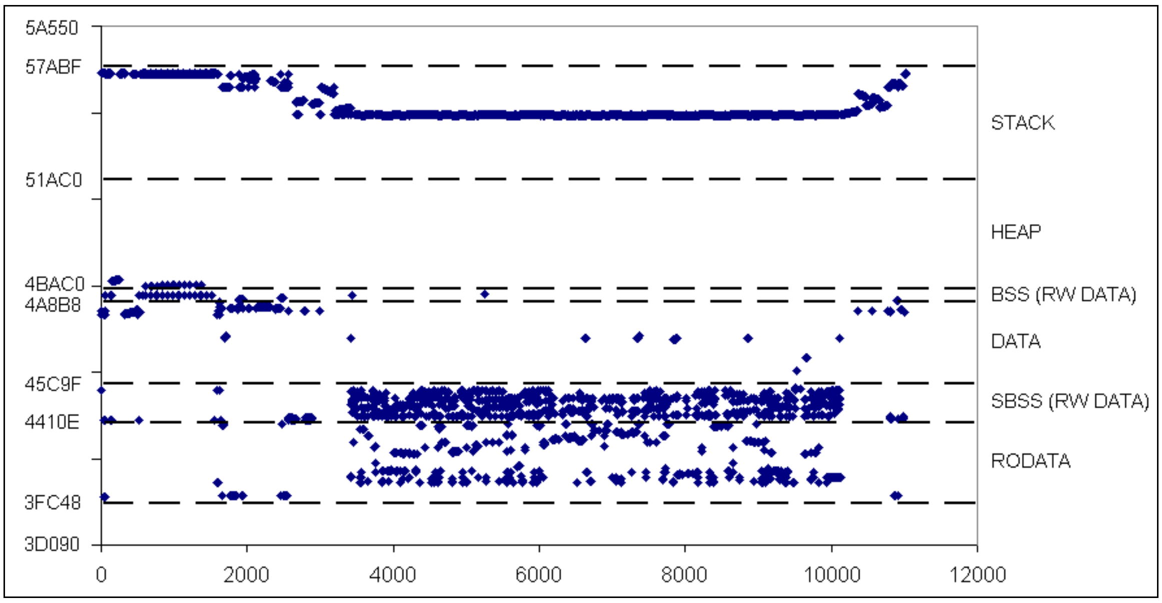


Figure 13. Memory read accesses for a Motorola MPC565 based jet engine controller (1 control loop – 11028 memory accesses).

Reducing the fault-space to the memory segments presented in the graph increases the probability of fault activation. Further optimization would be achieved by considering only the memory locations within these segments that are actually used. Not only are the available resources greater than the needs of the application but the segments are also over-dimensioned. Reducing the granularity to the level of individual memory locations would yield better results, although it would be even better to use techniques that consider the dynamic usage of the resources.

During the past years two main classes of analysis techniques for reducing the cost of validating dependable systems by fault injection have been studied – pre-injection analysis and post-injection analysis. Post-injection analysis [23] aims at predicting the outcome of fault injection experiments using the results from other experiments. Pre-injection analysis uses knowledge of program flow and resource usage to choose the location and time where faults should be placed, before any experiment is performed.

The GOOFI tool currently supports a pre-injection technique [24] for the MPC565 Nexus plug-in. The technique is briefly presented in the following subsections.

5.1 Optimization input

In order to determine the optimized fault-space it is necessary to gather information about the code of the application and the computer system executing it. More precisely, the input required by the method is:

- Assembly code of the application
- The Program Counter (PC) trace over time
- The values of the General Purpose Registers before each memory read access[7]
- The definition of which resources are read by each assembly instruction

In our experimental setup, the *assembly code* is textual information obtained by disassembling the executable (ELF) binaries of the application, processed automatically by the optimization program. The *Program Counter trace* and the *values of the General Purpose Registers* (GPRs) are stored during the execution of the reference run. Even though the values of the GPRs are only required before each memory access, they were stored for every instruction executed to simplify implementation. The effective address of each memory read access is calculated with the GPR values. The *definitions* of which resources are read by each assembly instruction were obtained from Motorola's RISC CPU Reference Manual [19].

[7] The values of the GPRs are required to calculate the *effective address* for memory read instructions.

5.2 Optimization output

The resulting output (the optimized fault-space) consists of a list of possible locations and times for fault injection. Each element in the optimized fault-space contains the following information:

- Control loop index
- Breakpoint address
- Number of breakpoint invocations within the control loop
- The fault injection location

The control loop index is specific for control applications which execute in cycles. It defines the loop during which a fault should be injected. For applications that do not execute in loops, the control loop index is always set to one. The breakpoint address specifies the breakpoint position inside the control loop and the number of breakpoint invocations specifies the number of times this breakpoint should be reached before fault injection.

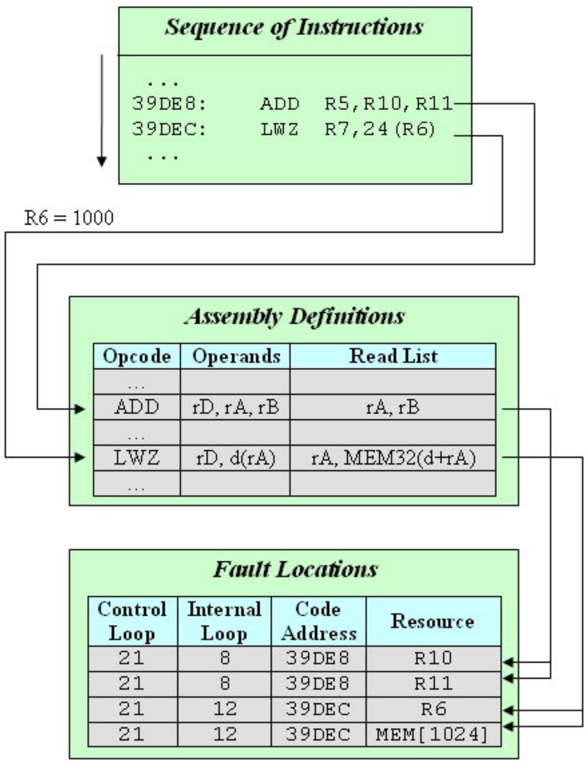
5.3 Performing the optimization

Using the Program Counter trace over time, the disassembled code of the application is parsed to obtain the sequence of assembly instructions executed. Each of the instructions is then analyzed in order to determine which resources the instruction reads. The pseudo-code for this procedure is presented in Figure 14.

```
FOREACH pc_value IN program_counter_trace DO
    control_loop_index ← current_control_loop ()
    breakpoint_invocation ← breakpoint_invocations_count (pc_value)
    instruction ← instruction_at_code_address (pc_value)
    instruction_read_list ← resources_read_by_instruction (instruction)
    FOREACH resource IN instruction_read_list DO
        useful_fault ← [control_loop_index, pc_value, breakpoint_invocation, resource]
        store_in_database (useful_fault)
    ENDFOREACH
ENDFOREACH
```

Figure 14. Pseudo-code for the optimization procedure.

The most important stage (shown in bold in the pseudo-code) is the identification of the resources read by each instruction. To accomplish this, the first step is to find the definition on the list matching the given instruction. This is done by matching the *opcode* and the *operands*. Then, by examining the possible assembly constructs, the symbols available in the *read list* of the definition are replaced by the resources actually read by the given instruction. Figure 15 illustrates this process.



```
 ...
39DE8:     ADD  R5,R10,R11
39DEC:     LWZ  R7,24(R6)
 ...
```

Opcode	Operands	Read List
...		
ADD	rD, rA, rB	rA, rB
...		
LWZ	rD, d(rA)	rA, MEM32(d+rA)
...		

Control Loop	Internal Loop	Code Address	Resource
21	8	39DE8	R10
21	8	39DE8	R11
21	12	39DEC	R6
21	12	39DEC	MEM[1024]

Figure 15. Example of the optimization procedure.

The instruction at address 39DE8 adds R10 to R11 and stores the result in R5. The definition for this instruction is found in the table and the read list contains rA and rB, respectively, R10 and R11. Since these are the two resources read by this instruction, two new lines are inserted into the fault locations for code address 39DE8 (the control loop and the internal loop are assumed to hold the values specified).

The second instruction, at address 39DEC, fetches the memory word addressed by the effective address (R6) + 24 and stores it in R7. Its definition in the table specifies rA and MEM32(d+rA), respectively, R6 and the 32-bit word at 1000[8]+24, as being read. These two resources are then inserted into the list of fault locations.

[8] The value of R6 is obtained during the reference run.

6. Related work on fault injection tools

This section gives an overview of some of the many tools which have been developed for injecting faults in computer systems. The presented tools are also compared with the GOOFI tool.

6.1 Simulation-based fault injection tools

One tool for injecting faults using electrical level simulations is FOCUS [25]. FOCUS has been used for validating a digital jet-engine controller installed in the Boeing 747 and 757 aircrafts. The tool uses a hierarchical simulation environment for tracing the impact of transient faults (modeled as current sources) caused by lightning effects, electrical power surges etc.

Common modeling languages for developing logic level simulation models of systems are the Very high-speed integrated circuit Hardware Description Language (VHDL) and Verilog. A methodology for injecting faults using logic level simulations is demonstrated by the MEFISTO (Multilevel Error and Fault Injection Simulation TOol) [26]. MEFISTO injects faults by either modifying VHDL models of systems with mutations or adding saboteurs, or by using VHDL simulator commands for manipulating signal and variable values in the models. A post-injection analysis technique called path-based error coverage prediction (PBECP) has also been implemented in MEFISTO [23]. The PBECB technique predicts the error coverage for a specified input sequence based on fault injection data obtained for another input sequence.

The simulation based fault injector VERIFY (VHDL-based Evaluation of Reliability by Injecting Faults efficientlY) is presented in [27]. The tool uses an extension of VHDL for describing faults correlated to a component, enabling hardware manufacturers which provide the design libraries to express their knowledge of the fault behavior of their components. Multi-threaded fault injection which utilizes checkpoints and comparison with a golden run is used for faster simulation of faulty runs. Unfortunately the tool is not compatible with current VHDL simulators since the VHDL language has to be modified. VFIT (VHDL-based Fault Injection Tool) [28] is another logic level simulation based fault injection tool with features similar to those of MEFISTO.

ALIEN [29] is another logic level tool implemented for the VHDL language. In contrast to many other such tools, ALIEN relies solely on mutations of VHDL models to inject faults.

The DEPEND tool [30] is an example of a function level simulation tool that has been used for studying the effects of correlated errors and errors with various latencies on a TMR system similar to Tandem Integrity S2. Functional fault models representing manifestations of faults at lower levels are used, e.g. memory bit-flips or CPU register errors.

6.2 Physical fault injection tools

Pin-level fault injection experiments have been conducted on a computerized interlocking system for railway control applications and the ESPRIT Delta-4 distributed system, using the MESSALINE tool [31]. This study demonstrated the usability of the technique for pointing out design/implementation deficiencies and characterizes the behavior of systems in the presence of faults despite limitations such as inadequate fault selection, synchronization problems, testbed efforts etc. Another pin-level forcing tool is the Advanced Fault InjecTion (AFIT) tool [32] which supports the injection of faults at a frequency of up to 40 MHz.

The RIFLE tool [33] is a pin-level insertion tool which uses dedicated logic that intercepts the pins of the target device to inject faults and to record the effects of the faults.

The FIST (Fault Injection system for Study of Transient fault effects) tool presented in [7] uses radiation induced fault injection on the MC6809E CPU. FIST uses heavy-ion radiation from a Cf-252 source to inject the faults and the golden chip technique, in which the target circuit operates synchronously with a reference circuit, not subjected to fault injection, to detect fault occurrences. Error recording is performed using a logic analyzer.

In [12] an electromagnetic interference (EMI) tool is presented and a comparison of the EMI technique with radiation induced fault injection, pin-level forcing and SWIFI is made showing that these techniques are complementary, i.e., they generate, to a large extent, different types of errors.

A laser fault injection (LFI) tool is described in [13]. The fault injection locations are identified using a CAD tool layout of the circuit under test and a high precision (0.1 μm) transition table is used for directing the laser beam on the sensitive locations. Obviously, this requires detailed knowledge of the design of the circuit under test.

Several fault injection tools based on on-chip debugging hardware have also been developed. FIMBUL (Fault Injection and Monitoring using BUilt-in Logic) is a SCIFI tool based on the IEEE 1149.1 standard hardware. The tool is presented in [34] where it is compared with the simulation based fault injection tool MEFISTO. The comparison shows that the SCIFI technique can be more than 100 times faster than simulation based fault injection and yet produce similar results. The tool has also been used to investigate the effects of varying the system input on fault injection results. The tool implements a post-injection analysis technique which uses results from fault injection experiments with a single input sequence to identify other input sequences to the system which give high, medium or low error coverage [35].

In [10], a tool based on the Motorola proprietary BDM debug hardware is presented. The tool relies on a pre-injection analysis technique called fault-list collapsing [6] to identify and eliminate from the complete list of injectable faults, several classes of faults which do not have any effect on the target system or which are equivalent to other faults.

A Nexus-based fault injection tool is INERTE (Integrated NExus-based Real-time fault injection Tool for Embedded systems) [36] which has been used for dependability validation of a diesel engine electronic control unit [37].

6.3 Software implemented fault injection tools

A pre-runtime SWIFI tool called DOCTOR (integrateD sOftware fault injeCTiOn enviRonment) which uses generation of synthetic workloads for system dependability evaluation is presented in [38]. Transient, intermittent or permanent faults can be injected into memory, communication or CPU. The tool has been used for evaluating the HARTS real-time distributed system. Faults injected into memory and communication subsystems can be injected during runtime while faults injected into the CPU are modeled via alteration of the executable workload image using a pre-runtime approach.

One of the earliest examples among the large number of SWIFI tools that have been developed is FIAT (Fault Injection-based Automated Testing) [39]. The tool uses additional software for injecting faults into user application code and data during runtime.

FERRARI [40], a Fault and ERRor Automatic Real-time Injector, is another runtime SWIFI tool which injects faults using UNIX process handling system calls. Here, a fault injection process spawns the target workload process that executes in a special trace mode enabling the injection of transient and permanent faults by the parent process.

The Xception tool [3] uses the debugging and monitoring features of the PowerPC processor to enable spatial, temporal and data manipulation fault triggers. The tool facilitates low intrusion on the target system since most of the fault injection software is running on an external host which only requires software for handling the exceptions to be added to the system. A variant called RT-Xception [41] features an upper bound for the maximum number of executed instructions in the fault injection code, which makes the execution time for each fault injector predictable and thus applicable for real-time systems. The most recent version of Xception is a commercially available tool implemented using a modular design supporting different types of fault injection techniques.

The MAFALDA (Microkernel Assessment by Fault injection Analysis for Design Aid) tool [42], which also uses processor debugging facilities for injecting both transient and permanent faults during runtime, has been used for investigating the benefits of adding fault containment wrappers (extra software for checking calls and the results of the calls) for micro kernels.

FIRE (Fault Injection using a REflective architecture) [43] uses reflective programming in C++ to inject and monitor object-oriented software. The user decides whether the target software should be monitored, fault injected or both. Using a special pre-processor, additional objects are created in the source code, so called meta-level objects. These are compiled together with the original source code enabling the target software to be manipulated at runtime. A similar tool for the Java programming language, called Jaca [44], has also been developed.

In [45], the SWIFI tool ProFI (Processor Fault Injection tool) is presented. The tool is able to inject transient, intermittent and permanent faults on a MC 68000-based add-in card running the A/Rose OS plugged into a Macintosh computer. Faults are injected using a trace routine executed after each machine instruction of the application program.

[46] presents a SWIFI tool called FTAPE (Fault Tolerance And Performance Evaluator) which is used on two prototypes of TMR-based Tandem machines running a synthetic workload. Bit-flips and stuck-at faults are injected into CPU registers and memory. The tool is also used for injecting disk errors to evaluate the mirrored disk system of the Tandem computers.

The BOND tool [47] is an agent-based SWIFI tool which uses a fault injection agent for injecting transient and intermittent bit-flip faults into the applications of a Windows NT PC and a logger agent for monitoring the effects of the faults.

6.4 Hybrid fault injection tools

In [48], the hybrid fault injection tool LIVE is presented which uses pin-level forcing or generates interrupts to activate software fault injection procedures (SWIFI). The tool has been used to evaluate a computer based railway control system based on the Motorola 68040 CPU.

The NFTAPE tool [2] specifically addresses the issues of extension and portability to different target systems. NFTAPE relies on lightweight fault injectors, triggers, monitors and other components to facilitate porting to different target systems as well as adapting it to different fault injection techniques.

6.5 Overview of fault injection tools

Table 1 gives an overview of some characteristics of the fault injection tools presented in this section.

Table 1. Overview of fault injection tools.

Fault injection tool	Year	Fault injection techniques supported	Target systems	Fault models/ strategies	Support for adding new target systems	Support for adding new fault injection techniques	Built-in pre-injection analyses	Built-in post-injection analyses
AFIT	1997	pin-level forcing	FASST, TTP/C	semi-permanent (stuck-at)	yes	no	none	none
ALIEN	2003	logic level simulation	VHDL models	permanent (mutation)	yes	no	none	none
BDM	1999	BDM	MC68332	transient (single bit-flips)	no	no	fault-list collapsing	none
BOND	2003	runtime SWIFI	Windows NT 4.0 OS PC	transient, intermittent (bit-flips)	no	no	none	none
DEPEND	1997	function level simulation	Tandem Integrity S2	transient	yes	no	none	none
DOCTOR	1993	SWIFI	HARTS, VxWorks	transient, intermittent, pseudo-permanent	yes	no	none	none
EMI	1993	EMI	MARS	transient, intermittent	yes	no	none	none
FERRARI	1992	SWIFI	Unix workstation	transient, pseudo-permanent	no	no	none	none
FIAT	1988	runtime SWIFI	IBM RT PC	transient (bit-flips)	no	no	none	none
FIMBUL	1998	SCIFI	Thor	transient (bit-flips)	no	no	none	execution profile
FIRE	2000	runtime SWIFI	Unix workstation	transient, intermittent, pseudo-permanent	no	no	none	none
FIST	1989	radiation induced	MC6809, MC68020, MC68070	transient (bit-flips)	yes	no	none	none
FOCUS	1989	electrical level simulation	HS1602	transient	yes	no	none	none
FTAPE	1995	runtime SWIFI	Tandem S2, Tandem TMR, Tandem Duplex	transient (bit-flips, disk errors)	yes	no	none	none
GOOFI	**2001**	**SCIFI, BDM, Nexus, runtime SWIFI**	**Thor Rad Hard, HC12, MPC565, MC68340**	**transient (bit-flips)**	**yes**	**yes**	**OFFSET**	**none**
INERTE	2003	Nexus	MPC565	transient (single bit-flips)	no	no	none	none
JACA	2002	runtime SWIFI	Java Virtual Machines	transient, intermittent, pseudo-permanent	yes	no	none	none
LFI	1997	laser	32-bit RISC CPU	transient (single bit-flips)	yes	no	none	none
LIVE	1996	pin-level forcing, SWIFI	MC68HC11, MC68302, MC68332, MC68360, MC68040	transient (bit-flips), pseudo-permanent	yes	no	none	none
MAFALDA	1999	runtime SWIFI	Chorus ClassiX PC	transient (single bit-flips)	yes	no	none	none
MEFISTO	1994	logic level simulation	VHDL models of DP32, ERC32, Thor, etc.	transient (saboteurs, simulator commands), permanent (mutation)	yes	no	none	path-based error coverage prediction
MESSALINE	1990	pin-level forcing, pin-level insertion	PAI, Delta-4, MARS	semi-permanent (stuck-at, bridging, inversion,open connection)	yes	no	none	none
NFTAPE	2000	SWIFI, logic level simulation, pin-level	VHDL models, Myrinet LAN, Unix workstation	transient, intermittent, permanent	yes	yes	none	none
PROFI	1993	runtime SWIFI	MC68000	pseudo-permanent	no	no	none	none
RIFLE	1991	pin-level insertion	MC680X0, Intel 286/386	semi-permanent (stuck-at, bridging, inversion,open connection)	yes	no	none	none
VERIFY	1997	logic level simulation	VHDL model of DP32	transient, intermittent, permanent (model extensions)	yes	no	none	none
VFIT	2000	logic level simulation	VHDL models	transient and intermittent (saboteurs, commands), permanent (mutation)	yes	no	none	none
XCEPTION	1995	runtime SWIFI, SCIFI, pin-level forcing	PowerPC, Intel Pentium, SPARC	transient (bit-flips)	yes	yes	none	none

7. Conclusion and future work

This paper describes the GOOFI (Generic Object-Oriented Fault Injection) framework. GOOFI is implemented in the Java language to support maintainability and portability between different host platforms. All data used by the framework is stored in a portable SQL-database. GOOFI provides a plug-in based framework which facilitates the addition of new fault injection techniques as well as adaptation to new target systems. The target systems and fault injection techniques that GOOFI currently supports are:

- The microcontroller MPC565 from Motorola with Nexus-based fault injection
- The microprocessor HC12 from Motorola with BDM-based fault injection
- The microprocessor Thor Rad Hard from Saab Ericsson Space with scan-chain implemented fault injection (SCIFI)
- The microprocessor MC68340 from Motorola with software-implemented fault injection (SWIFI)

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

- Support for additional fault models such as software bugs, intermittent and permanent faults
- 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
- A general data analysis tool supporting all plug-ins
- Simulation-based fault injection (both VHDL models via Modelsim [49] and function-level models via Simics [50])
- Further improvements to the framework by e.g. moving required functionality from a plug-in to the GOOFI framework to further simplify the development of a new GOOFI plug-in
- Design of our own hardware/software environment supporting the Nexus standard
- And, of course, extending support for new target systems and fault injection techniques via development of new plug-ins

Acknowledgements

We would like to thank the master thesis students working with the GOOFI tool under our supervision; Stefan Nilsson and Erik Nordström for the implementation of the HC12/BDM plug-in and Ruyin Lai for the work on the first version of the plug-in architecture. Thanks are also due to Nasser Mahmoudi and Peter Gunnarsson for implementing vital parts of new GOOFI plug-ins and Sven-Arne Andréasson of the Department of Computer Science and Engineering at Chalmers for his valuable comments on Object-Oriented analysis and design. This work was supported by NFFP, VINNOVA, ARTES and the Swedish Foundation for Strategic Research (SSF).

References

[1] R.K. Iyer, *Experimental evaluation.* Special Issue of Proc. Twenty-Fifth Int. Symp. on Fault-Tolerant Computing, 1995. pp. 115-130.

[2] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R.K. Iyer. "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors", in *Proceedings of IPDS 2K: IEEE International Computer Performance and Dependability Symposium.* Chicago, IL, USA. 27-29 March, 2000. pp. 91-100.

[3] J. Carreira, H. Madeira, and J.G. Silva, *Xception: a technique for the experimental evaluation of dependability in modern computers.* IEEE Transactions on Software Engineering, 1998. **24**(2): pp. 125-136.

[4] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. "GOOFI: generic object-oriented fault injection tool", in *Proceedings International Conference on Dependable Systems and Networks.* Göteborg, Sweden. 1-4 July, 2001. pp. 83-88.

[5] IEEE-ISTO, *The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface.* 1999. pp. 9-10.

[6] A. Benso, M. Rebaudengo, L. Impagliazzo, and P. Marmo. "Fault-list collapsing for fault-injection experiments", in *Proceedings of the 1998 Reliability and Maintainability Symposium.* Anaheim, CA, USA. 19-22 January, 1998. pp. 383-388.

[7] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, *Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms.* IEEE Micro, 1994. **14**(1): pp. 8-23.

[8] P. Civera, L. Macchiarulo, M. Rebaudengo, M.S. Reorda, and M. Violante, *An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits.* Journal of Electronic Testing: Theory and Applications, 2002. **18**(3): pp. 261-271.

[9] R. Leveugle. "Fault injection in VHDL descriptions and emulation", in *The IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems.* Yamanashi, Japan. 25-27 October, 2000. pp. 414-419.

[10] M. Rebaudengo and M. Sonza Reorda. "Evaluating the fault tolerance capabilities of embedded systems via BDM", in *Proceedings 17th IEEE VLSI Test Symposium*. Dana Point, CA, USA. 25-29 April, 1999. pp. 452-457.

[11] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin. "Two fault injection techniques for test of fault handling mechanisms", in *Proceedings. International Test Conference 1991 (IEEE Cat. No.91CH3032-0)*. Nashville, TN, USA. 26-30 October, 1991. pp. 140-149.

[12] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G.H. Leber, *Comparison of physical and software-implemented fault injection techniques*. Computers, IEEE Transactions on, 2003. **52**(9): pp. 1115-1133.

[13] J.R. Samson, Jr., W. Moreno, and F. Falquez. "A technique for automated validation of fault tolerant designs using laser fault injection (LFI)", in *Proceedings of 28th International Symposium on Fault Tolerant Computing*. Munich, Germany. 23-25 June, 1998. pp. 162-167.

[14] J. Guthoff and V. Sieh. "Combining software-implemented and simulation-based fault injection into a single fault injection method", in *Twenty Fifth International Symposium on Fault Tolerant Computing. Digest of Papers*. Pasadena, CA, USA. 27-30 June, 1995. pp. 196-206.

[15] D. Skarin, J. Vinter, P. Folkesson, and J. Karlsson, *Implementation and usage of the GOOFI MPC565 Nexus fault injection plug-in*. Technical Report No. 04-08, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2004.

[16] iSYSTEM, http://www.isystem.com, October 11th, 2004.

[17] MySQL AB, http://www.mysql.com, October 11th, 2004.

[18] PHYTEC Technology Holding Company, http://www.phytec.com/sbc/32bit/pc565.htm, October 11th, 2004.

[19] Motorola, *MPC565/MPC566 User's Manual*. 2003.

[20] iSYSTEM AG, http://www.isystem.com/Products/Emulators/iC3000/ic3000_overview.htm, October 11th, 2004.

[21] Altium Limited, http://www.altium.com/tasking/products/ppc/index.html, October 11th, 2004.

[22] J. Vinter, O. Hannius, T. Norlander, P. Folkesson, and J. Karlsson. "Experimental dependability evaluation of a fail-bounded jet engine control system for unmanned aerial vehicles", in *Proceedings International Conference*

on Dependable Systems and Networks. Yokohama, Japan. June 28 - July 1, 2005.

[23] J. Aidemark, P. Folkesson, and J. Karlsson, *Path-based error coverage prediction [microprocessors]*. Journal of Electronic Testing: Theory and Applications, 2002. **18**(3): pp. 343-349.

[24] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson. "Assembly-level pre-injection analysis for improving fault injection efficiency", in *Proceedings of the Fifth European Dependable Computing Conference (EDCC-5)*. Budapest, Hungary. 20-22 April, 2005.

[25] G.S. Choi and R.K. Iyer, *FOCUS: an experimental environment for fault sensitivity analysis*. Computers, IEEE Transactions on, 1992. **41**(12): pp. 1515-1526.

[26] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. "Fault injection into VHDL models: the MEFISTO tool", in *Proceedings of IEEE 24th International Symposium on Fault Tolerant Computing*. Austin, TX, USA. 15-17 June, 1994. pp. 66-75.

[27] V. Sieh, O. Tschache, and F. Balbach. "VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions", in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. Seattle, WA, USA. 24-27 June, 1997. pp. 32-36.

[28] D. Gil, J.C. Baraza, J. Gracia, and P. Gil. "VHDL simulation-based fault injection techniques", in *Fault injection techniques and tools for embedded systems reliability evaluation*, Boston: Kluwer Academic, 2003.

[29] C. Robach and M. Scholive. "Simulation-based fault injection and testing using the mutation technique", in *Fault injection techniques and tools for embedded systems reliability evaluation*, Boston: Kluwer Academic, 2003.

[30] K.K. Goswami, *DEPEND: a simulation-based environment for system level dependability analysis*. IEEE Transactions on Computers, 1997. **46**(1): pp. 60-74.

[31] 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, 1990. **16**(2): pp. 166-182.

[32] P. Gil, J.C. Baraza, D. Gil, and J. Serrano. "High speed fault injector for safety validation of industrial machinery", in *EWDC-8 (8th European Workshop on Dependable Computing). Experimental validation of dependable systems*. Chalmers University of Technology, Göteborg, Sweden, April 1997.

[33] H. Madeira, M. Rela, F. Moreira, and J.G. Silva. "RIFLE. A general purpose pin-level fault injector", in *Proceedings of the 1st European Dependable Computing Conference (EDCC-1)*. Berlin, Germany. 4-6 October, 1994. pp. 199-216.

[34] P. Folkesson, S. Svensson, and J. Karlsson. "A comparison of simulation based and scan chain implemented fault injection", in *Proceedings of 28th International Symposium on Fault Tolerant Computing*. Munich, Germany. 23-25 June, 1998. pp. 284-293.

[35] P. Folkesson and J. Karlsson. "Considering workload input variations in error coverage estimation", in *Proceedings of the Third European Dependable Computing Conference (EDCC-3)*. Prague, Czech Republic. September, 1999. pp. 171-188.

[36] P. Yuste, D. De Andres, L. Lemus, J.J. Serrano, and P. Gil. "INERTE: Integrated NExus-based Real-Time fault injection tool for Embedded systems", in *Proceedings of the International Conference on Dependable Systems and Networks*. San Francisco, CA, United States. 22-25 June, 2003. pp. 669.

[37] J.-C. Ruiz, P. Yuste, P. Gil, and L. Lemus. "On benchmarking the dependability of automotive engine control applications", in *Proceedings International Conference on Dependable Systems and Networks*. Florence, Italy. June 28 - July 1, 2004. pp. 857-866.

[38] H. Seungjae, K.G. Shin, and H.A. Rosenberg. "DOCTOR: an integrated software fault injection environment for distributed real-time systems", in *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. Erlangen, Germany. 24-26 April, 1995. pp. 204-213.

[39] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. "FIAT - fault injection based automated testing environment." in *FTCS-18: Eighteenth International Symposium on Fault-Tolerant Computing, Digest of Papers*. Tokyo, Japan. 27-30 June, 1988. pp. 102-107.

[40] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, *FERRARI: a flexible software-based fault and error injection system.* IEEE Transactions on Computers, 1995. **44**(2): pp. 248-260.

[41] J.C. Cunha, M.Z. Rela, and J.G. Silva. "Can software implemented fault-injection be used on real-time systems?" in *Proceedings of EDCC-3: Third European Dependable Computing Conference*. Prague, Czech Republic. 15-17 September, 1999. pp. 209-226.

[42] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat. "MetaKernels and fault containment wrappers", in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, 1999. pp. 22-29.

[43] E. Martins and A.C.A. Rosa. "A fault injection approach based on reflective programming", in *Proceedings of International Conference on Dependable Systems and Networks (includes FTCS 30 30th Annual International Symposium on Fault Tolerant Computing and DCCA 8)*. New York, NY, USA. 25-28 June, 2000. pp. 407-416.

[44] E. Martins, C.M.F. Rubira, and N.G.M. Leme. "Jaca: a reflective fault injection tool based on patterns", in *Proceedings International Conference on Dependable Systems and Networks*. Washington, DC, USA. 23-26 June, 2002. pp. 483-487.

[45] T. Lovric and K. Echtle. "ProFI: Processor fault injection for dependability validation", in *IEEE Int. Workshop on Fault and Error Injection for Dependability Validation of Computer Systems*. Göteborg, Sweden, 1993.

[46] T.K. Tsai and R.K. Iyer. "Measuring fault tolerance with the FTAPE fault injection tool", in *Proceedings of Joint Conference PERFORMANCE TOOLS '95 and MMB '95*. Heidelberg, Germany. 20-22 September, 1995. pp. 26-40.

[47] A. Baldini and P. Prinetto. "Bond: An agents-based fault injector for Windows NT", in *Fault injection techniques and tools for embedded systems reliability evaluation*, Boston: Kluwer Academic, 2003.

[48] A.M. Amendola, L. Impagliazzo, P. Marmo, and F. Poli. "Experimental evaluation of computer-based railway control systems", in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, 1997. pp. 380-384.

[49] Mentor Graphics, http://www.model.com, October 11th, 2004.

[50] Virtutech Inc., http://www.virtutech.com, October 11th, 2004.