Using reflective programming to inject faults into object-oriented systems

Amanda Cibele Apolinário Rosa, Eliane Martins
Institute of Computing - State University of Campinas (Unicamp)
P.O Box 6176 Campinas - 13083-970 SP Brazil
{amandapo, eliane}@dcc.unicamp.br

Abstract

Software-implemented fault injection is, nowadays, a largely used technique to validate dependability properties of software systems. To inject faults and monitor their effects some form of instrumentation may be introduced into the system under test (target system). This instrumentation causes some level of intrusiveness, i.e., it imposes some interference upon the target system execution. Therefore, a goal of a software instrumentation approach is to be the least intrusive possible. This requires instrumentation to be functionally independent from the target system. Other important qualities that an instrumentation should present are *modularity*, in order to allow easy incorporation of new features, *reusability*, to allow easy adaptation to different target systems, and *portability*, to allow the use in different hardware/software platforms with minimum changes. To obtain these qualities we propose in this study the use of reflective programming. Intrusiveness in the target system is reduced, in that it allows a clear separation between functional and non-functional aspects, the later being related to fault injection and monitoring aspects. The design of a reflective fault-injection architecture is presented, followed by a brief discussion about the issues of injecting faults into objects. Some aspects of the implementation using the Open C++ 1.2 programming language are presented and future works are also discussed.

Keywords:
Software-implemented fault injection — object-oriented systems — reflective programming — metaobjects

1. Introduction

Fault injection is the deliberate introduction of faults into a system to observe its behavior [AAA+90]. By accelerating the occurrence of errors and failures[1], it is a valuable approach to validate the dependability[2] properties of a system. More recently, this technique has also been used for quantifying worst-case scenarios and risks [VMK+97].

Fault injection can take various forms according to the system's life-cycle phase. In the early phases, when the system may be represented by an abstract model, *fault simulation* (or *simulation-based fault injection* [HTI97]) is generally used. In this case, logical faults are introduced into a model representing a system at a structural (e.g., gates or ports) or functional level. This technique is useful to study error propagation and to forecast system dependability measures. During testing phases, when a hardware prototype, or the software, or both are available, the techniques mostly used are physical (or hardware) fault injection and software-implemented fault injection.

In *hardware fault injection*, a special hardware device introduces physical faults into integrated circuits (ICs). Faults can be injected so as to affect the ICs externally or internally. Pin-level fault injection is an example of external access, in which functional pins of an IC are affected. Submitting components to heavy-ion radiation or electromagnetic interference may generate internal faults.

In *Software-implemented fault injection*, errors induced by the occurrence of hardware or software faults are introduced by software. This technique has deserved great research interest because it has the following advantages over the above mentioned:

1. It is less expensive, because it doesn't require either a special hardware device (as in physical fault injection) or too many machine resources (as in fault simulation);
2. It is suitable to validate target software systems at higher levels (application, operating system), which is hard to do with physical fault injection;
3. Its fault models are more flexible than those of physical fault injection, in that they allow the injection of either software or hardware faults;
4. It is easier to implement and to use;
5. It may be more easily reused to test different systems, and more easily portable to different platforms than hardware injectors or fault simulators.

In spite of these attractive features, software-implemented fault injection has some drawbacks (c.f. [HTI97] for more details). An important one is intrusiveness: system execution can be disturbed or even its original structure may be altered by the instrumentation necessary to inject faults and monitor their effects into the target system. Unless a hybrid approach, in which some functions are performed by special hardware [KKA95] is used, or the environment offers fault injection and monitoring features [CMS95], it is impossible to avoid disturbance to the system under test.

In this study we propose the use of reflective programming to minimize perturbation to the target software when testing object-oriented systems by software-implemented fault injection. Reflection is chosen because it is an easy way to separate instrumentation functionality from those of the target system. Reflection introduces a new architecture model, in that there is a meta-level and a base level. The base level is used in this study to implement the target system objects. The meta-level allows programmers to observe and manipulate data structures and/or actions performed at the base level, and thus, is used here to implement fault injection and monitoring features.

This paper is organized as follows: Section 2 provides some aspects of software-implemented fault injection, presenting some related studies; Section 3 contains a brief overview of reflective

[1] A failure occurs when the service provided by the system doesn't conforms to its specification. A failure can occur because of the presence of errors in the system. A fault is the cause of an error [Laprie92].

[2] Dependability includes a set of attributes such as reliability, availability, security and safety (c.f. reference above).

programming, focusing mainly on the reflective programming language Open C++ 1.2, used in this study; Section 4 briefly presents a test architecture using meta-level objects to inject faults and monitor an object-oriented system, with a brief discussion on some issues concerning the injection of faults into objects; Section 5 presents implementation issues; and finally, Section 6 concludes this paper, presenting future applications of the test architecture described here.

2. Software-implemented fault injection approaches

Software-implemented fault injection consists of the injection of error patterns into executing software [KKA95]. These error patterns represent errors that can be generated by the occurrence of hardware or software faults. Hardware faults are generally regarded as being memory (alteration of text or data segment), processor (corruption of general or special registers), bus (alteration of instructions or data transmitted through the bus), I/O (generation of errors accessing peripheral devices) and communication faults (alteration, suppression, duplication or delay of messages transferred through a communications channel). Software faults represent design or implementation faults, such as variables/parameters that are wrong or not initialized, incorrect assignments or incorrect condition checks.

Software fault injection allows the injection of higher level faults, being thus, suitable for testing applications and operating systems, which is hard to do with physical fault injection. In order to inject faults and monitor their effects, some form of target system instrumentation is needed. When the target is an application, instrumentation can be inserted into the application, in the operating system or between the application and the operating system [HTI97].

Many tools, which use different approaches to instrument the target system, have been developed to support software fault injection. Some of these tools are presented in what follows.

2.1 Some tools

In FIAT [SVS+88], fault injection and monitoring capabilities are added to the application code and operating system. This tool can inject memory faults at runtime; faults are triggered when target system execution reaches some specified locations (spatial triggering). FTAPE [TI92], also adds drivers at the operating system level to activate faults; besides temporal and spatial triggering, it also considers system stress; that is, the amount of activity performed by the system. This tool can inject processor, memory and I/O faults. For I/O faults, routines are inserted at the disk driver, generating exceptions that simulate device faults.

SFI [RS93] and its successor, Doctor [HRS93], use different approaches to inject different types of faults. They use code alteration to inject processor faults: e.g., to simulate faults in functional units, instructions using that unit are altered at source code level being the program, thus, compiled with the altered instructions. Fault activation occurs whenever the program executes an erroneous instruction. To inject communication faults, they alter library routines so as to provide erroneous communication services, and to inject memory faults they use a process running concurrently with the target application. When experiments require memory faults, a high priority process called the memory fault injection process is started to perform fault injection. This process injects faults at selected addresses by reading, altering, and then writing back the contents of a specified location. Once the fault has been injected, the memory fault injection process will pause automatically for a period of time determined during the experiment setup phase. The process is repeated until the experiment is completed.

Another tool based on the use of an external process is SOFIT [AT95]. It is aimed at testing applications running under Unix or other operating systems that provide virtual image of a process. This virtual image is examined or modified by a process having the privilege to do so - in this case, its parent process. The target process execution must be stopped, in order to allow that its virtual image be examined or modified for testing purposes. This tool that way can inject various types of faults.

In ProFi [LE93] fault injection uses the trace-mode of a processor. When the trace-mode is active, a trace routine is executed before each instruction of the target application. The trace routine runs in

supervisor mode and has access to the whole system memory and all programmer-visible registers. The trace routine is thus, used to inject processor faults.

FERRARI [KKA92] uses software traps to inject processor and bus faults. Software traps are triggered either by the program counter or by a timer. When the traps are triggered, the injector, located at the trap handling routine, injects faults at specified locations by changing the content of elected registers or memory location. The faults can be transient or permanent, and they emulate errors in address line, data line and condition code flags.

Xception [CMS95] uses performance monitoring and debugging features included in many modern processors to inject faults and monitor a target system. Processor's built-in hardware exceptions are used to trigger instrumentation functions. Faults can be triggered not only after a specified time has elapsed, but also when an instruction in a specified location is fetched or yet when data stored in a specified location is accessed. The injection of a fault comprises several steps: 1st - programming the fault trigger condition in the processor debugging logic; 2nd - corrupting specific registers or memory contents when the exception that injects the fault is raised, according to the fault type and the functional unit required; 3rd - restoring the original register or memory content needed in case transient faults are being injected. Although trace mode can also be used when it is required to execute some instructions step-by-step, it is not a main feature of the tool. Xception runs initially on systems based on the PowerPC family. According to a study done by their authors, the special features needed for its implementation are also present in other processor families.

2.2 Final comments

From what has been said, software-implemented fault injection approaches can be categorized according to the phase faults are injected in [HTI97]: at compile-time or during runtime.

Compile-time injection presents low intrusiveness, as no extra software is needed during runtime to control experiments, or any instrumentation has to be introduced into the target system for injection or monitoring purposes. This approach presents, however, low controllability and observability: it is not easy to know whether the injected faults will be activated or not, nor to monitor their effect.

Runtime injection presents better controllability and observability; however, intrusiveness increases unless fault injection and monitoring use an extra hardware device [KKA95] or special features of the target system environment [CMS95].

Our aim in this study is to show how object-orientation and its properties can be used to implement runtime fault injection so that the necessary instrumentation could be introduced in a transparent, easy-to-use way. Another goal is to enhance reusability by implementing test features regardless of the application. The following section describes how reflective programming can be used to achieve these goals.

3. Reflection and object-oriented programming

3.1 Reflection in object-oriented languages

Computational reflection (from here on called just *reflection*) is the capability of a computational system to reason about or act upon itself [Mae87]. Conventional systems manipulate data that represents entities external to the system. A reflective system, in addition to that, contains data that represents structural and behavioral aspects of the system itself (*meta-information*). This meta-information must be causally connected to the actual behavior of the system, i.e., changes in system behavior cause changes in its meta-information and vice-versa.

Reflection is achieved by subdividing a system into two levels: *functional* (or *base-*) level, in which the application's functionality is implemented, and *non-functional* (or *meta-*) level, which performs observation and manipulation of the base-level structure or behavior.

In object orientation Patti Maes [Mae87] introduced the *metaobject approach* to implement reflection. An object x can be associated with a metaobject ^x that represents the meta-information of x.

A metaobject is an object, thus, it can be manipulated in the same manner as a base-level object. Therefore, metaobjects can have one or more other metaobjects associated to them. In its concept, it would be possible to build an infinite hierarchy of meta-levels in which metaobjects in a given layer control the ones in a layer bellow. However, in a study presented by Ancona et al[3], referenced in [Lis97], the number of layers of such a hierarchy may not exceed 3, for complexity and performance concerns.

Many applications have been using reflection to implement different mechanisms such as distribution [CM93], concurrency [MWY91], and fault-tolerance [FNP+95, Rub97] among others. Reflection presents several advantages for software-implemented fault injection. First, the separation in base level and meta-level allows the independence of injection/monitoring features from the application's functionality's, which aids in reducing intrusiveness. Second, implementing injection and monitoring features at the meta-level enhances modularity and reusability once these features can be developed regardless of the target application, being it possible to incorporate and remove whenever necessary. And third, for object-oriented applications, the use of metaobjects makes it possible to inject different types of faults into different base-level objects, which allows the emulation of more complex failure modes.

3.2 The Open C++ 1.2 language

OpenC++ 1.2 [Chi93], an extension of the C++ that supports reflection, is being used in this study. Its main features are [Chi93, Lis97]:

1. Objects in base-level can be reflective or normal (non-reflective). Reflective objects have an associated metaobject which controls its behavior. A non-reflective objective is a standard C++ object;
2. Each base-level reflective object can be associated with only one metaobject. Furthermore, metaobjects cannot be shared;
3. Metaobjects can control method invocation and attribute access of its base-level object;
4. A metaobject is an instantiation of a meta-level class. Meta-level classes are associated with base-level classes at compile-time;
5. Meta-level classes inherit from the predefined *MetaObj* class. The subclasses of *MetaObj* can redefine its methods. Such methods allow a metaobject to control method calls, and read or write access to attributes at the base level, among others.

A program in OpenC++1.2 is a C++ program that contains clauses to declare classes, methods, and attributes as reflective. These clauses are embedded as C++ comments that start with //MOP. To declare methods and attributes as reflective, one must precede them with the clause: //MOP reflect. To associate a base-level class with a meta-level class, the clause //MOP reflect class must be used.

A category_name can be associated with reflective methods and attributes; that way, a metaobject recognizes that the reflective method or attribute has a category and changes method invocations and attribute access accordingly. These features will become clearer in the next section, with the examples showing their use in the fault injection architecture proposed.

4. Fault injection and monitoring using reflective programming

In this section we present a reflective architecture to implement software fault injection based on the metaobject approach presented previously. Although OpenC++1.2 offers limited reflection capabilities,

[3] Ancona, M. et al. Reflective Architecture for Reusable Fault Tolerant Software. In proc. Of Latin American Conference in Informatics, Canela, Rs, Brazil, 1995, pp. 87-98.

it is useful in our study to show the feasibility of implementing injection and monitoring features at the meta-level.

4.1 Architecture

The reflective architecture for software fault injection proposed in this study is shown in Figure 1. Its main components are: an *application*, representing the target system application implementing the fault-tolerance mechanisms to be validated; a *meta-level library*, a *controller* and a *user-interface module*.

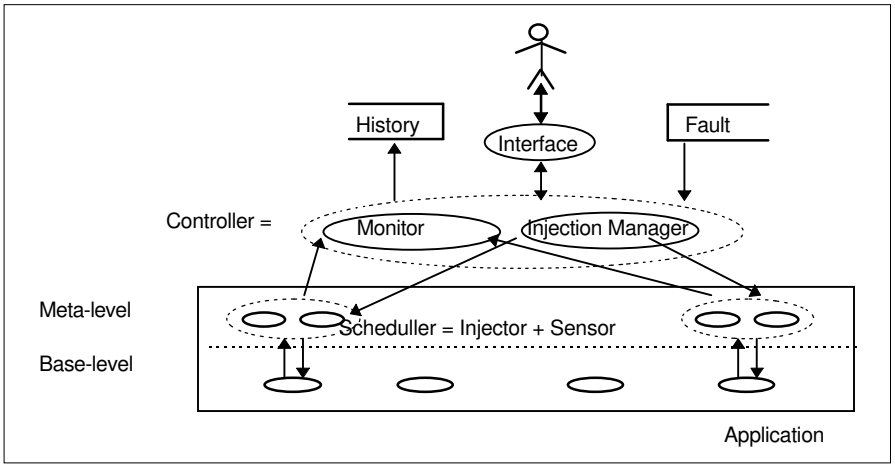

Figure 1: A reflective architecture for fault injection and monitoring.

The *application* is an object-oriented fault-tolerant application, which must be submitted to a preparation phase before testing. In this phase the clauses (see section 3 above) declaring the reflective components (classes, methods, or attributes) are introduced into the source code. As in OpenC++1.2 instances of a reflective class can be reflective or not. In this phase, it is also necessary to declare which objects will be reflective and which will not. In the following the reflective objects will be called *injectable objects*.

The *meta-level library* contains metaobjects called *schedulers*; each composed of an *injector* and a *sensor*[4] . An *injector* is responsible for the introduction of pre-specified faults into an injectable object, and a *sensor* is responsible for readout collection. As a metaobject, a *scheduler* is created when its associated injectable object is instantiated. Its function will be further explained in 4.2.1.

The *controller* controls the experiments: it starts the application, controls *injectors* and *sensors* and stores results. The *controller* is composed of two objects: the *injection manager* and the *monitor*. The *injection manager* reads the pre-specified faults from a *Fault* file and transfers them to *schedulers*, which transfer them to *injectors* at injection time. The *monitor* receives and stores the collected data in the *History* file.

The *user interface* helps users to observe and control the execution of experiments at runtime. Among other functions, users can turn on/off *injectors* and *sensors*. This feature enables the user to adjust injection or monitoring according to the system activity.

4.2 Fault Injection

In the approach presented here, faults are injected during runtime, so a mechanism is necessary to trigger faults during execution. Faults can be triggered on temporal or spatial basis [CMS95]. On

[4] The *scheduler* is a composite object created because of a limitation of OpenC++ 1.2: each base-level object can be associated with one and only one meta-object. Separation of injection and monitoring functions is necessary for modularity and reusability concerns, thus, the need of a composite meta-object.

temporal basis, faults are activated when a specified time has elapsed. On spatial basis faults can be activated when application execution reaches a specified address [KKA92], and/or access to certain data is performed, as in [CMS95]. Another important characteristic of a software fault injection tool is the fault types that can be considered. Both aspects are discussed in what follows.

4.2.1 Fault triggers

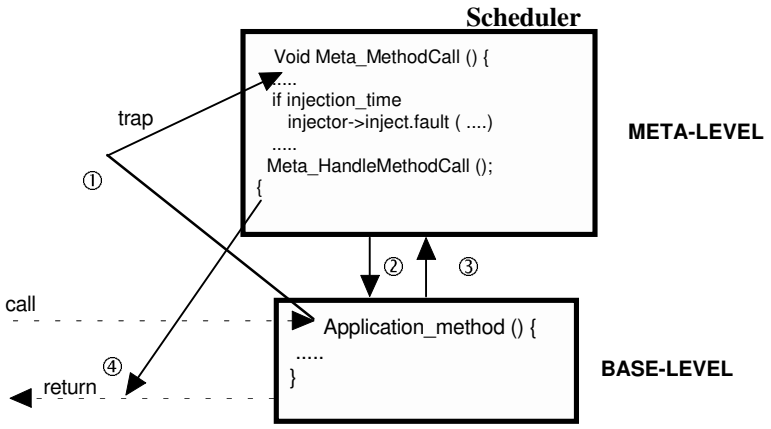

Figure 2: Injection activation via reflective method invocation.

Faults are triggered on spatial basis, using the invocation trap facility offered by metaobject protocol of OpenC++1.2. Invocation traps are used to control base-level objects, and can occur either when a reflective method is called or when a reflective attribute is accessed. Figure 2 shows how the injection is triggered by a method call using reflection. When a reflective method is called at the base level, the call is trapped and handled at the meta-level (by a *scheduler*) by a `Meta-MethodCall` (①). This meta method verifies whether a fault has to be injected. If so, it invokes the *injector* to inject the specified fault. Then, the *scheduler*, using the meta-method Meta-HandleMethodCall, invokes the application method (②③). At the end of `Meta-MethodCall` the results are returned to the caller in the application as in a normal method call (④). The process is the same for access to a reflective attribute.

As an injectable object can be both injected and/or monitored by the metaobject, the *scheduler* needs a category-name (c.f. section 2) to identify the action to be taken according to the method called or attribute accessed.

4.2.2 Fault types

In our approach to testing we are doing a parallel with hardware chips by considering objects as software IC's, as proposed in [Hof89]. When considering objects as software IC´s, we can use the same correspondence as Hoffman; that is, each public method (or attribute) can correspond to a pin, or to a related set of pins. Therefore, a parallel with hardware fault injection can be made, and we can have the introduction of internal faults (affecting an object's private features) or external faults (affecting an object's interface)[5].

Another dimension of fault classification is based on the repetition pattern; that is, faults can be transient (never repeated), intermittent (repeated based on activation injector number; i.e. a fault can be reinjected between *N* injector invocation) or permanent (always repeated). The proposed tool can inject transient, intermittent and permanent faults.

---

[5] Although OpenC++ 1.2 does not allow private features of an object to be reflected, internal faults can still be considered, for protected features.

4.3 Monitoring

Monitoring is an important issue in fault injection experiments. Data collection is useful not only to detect weaknesses of fault tolerance mechanism design or implementation, but also to obtain parameters (e.g. fault coverage) used for dependability evaluation.

Data on fault activation is collected by the *injectors* and sent to the *monitor*. The sensors collect data on fault tolerance mechanism activities; they also send their readouts to the *monitor* that stores all data in the *History* file. A *scheduler* invokes *sensors* in the same way as *injectors. Sensors* can be turned on/off during runtime, which enables the user to tune monitoring according to the system activity.

5. An example

Figure 3 shows a simple example of the use of the features described in the previous section.

```
1.  class Example{
2.  public:
3.      Example( );
4.      . . .
5.      // MOP reflect(injection):
6.      int var_ex;
7.      . . .
8.  }

9.  //MOP reflect class Example: SchedulerMetaObj

10. class SchedulerMetaObj: public MetaObj{
11. public:
12.     void Meta_Read(Id var_id, Id category, ArgPac& reply){
13.     . . .
14.     if ((category = = injection | category = = both) &&
15.         (injector->Is_On( )) && Injection_time( ))
16.     {
17.         // code to invoke injector to perform data corruption
18.         . . .
19.         Meta_HandleRead(var_id, reply);
20.     }
21. }


22. main( ) {
23.     Example e1;
24.     refl_Example e2;
25.     x = e2.var_ex;
26.     y = e1.var_ex;
27. }
```

Figure 3: Example of fault injection in a public attribute.

First, a public attribute of the class *Example* is declared as being reflective by the use of the clause: *//MOP reflect (injection)* in line 5. *Injection* is the category_name (see section 3.2) associated with the attribute, which indicates to the *scheduler* that a fault has to be injected when this attribute is accessed. Other possible category_names are: *monitoring*, to indicate methods/attributes to be monitored, or *both*, when both injection and monitoring are to be performed.

The use of the clause *//MOP reflect class* in line 9 indicates that objects of the class *Example* can be reflective and the associated metaobject are instances of the class *SchedulerMetaObj*. In fact, when the OpenC++ pre-processor finds this clause in the source code, it creates a new subclass for *Example*, named *refl_Example*, which is reflective. In this way, normal objects are instances of the class *Example*, and reflective objects are instances of the class *refl_Example*.

In line 10 the declaration of a meta-class, *SchedulerMetaObj,* is shown. This class has MetaObj, a standard class, as its ancestor and must redefine methods offered by that class to implement fault injection and monitoring features. One of these methods is Meta_Read, which is invoked before a reflective attribute is read. This method is redefined, as shown in lines 12 to 21, so that the *injector* is called to corrupt the attribute at the appropriate moment. The details of this invocation are not shown here for the sake of conciseness. Note that before calling the *injector*, the *scheduler* must check the category to determine what action it should undertake. The method *Meta_HandleRead* executes the actual read access after fault injection.

Object creation is shown in lines 23 and 24: normal objects are instances of class *Example* (line 23) and reflective objects are instances of the class *refl_Example* (line 24). Once *var_ex* is declared as reflective and *e2* is a reflective object the access to *var_ex* (line 25) is intercepted by *Meta_Read* and its value may be corrupted by the injector. The corrupted value is then read by *Meta_HandleRead* and assigned to *x*. As *e1* is a normal (non-reflective) object, the access to *var_ex* in line 26 is performed as usual.

6. Conclusions and future works

Software-implemented fault injection has become a valuable technique to validate dependability properties of software systems. It allows the validation of actual implementations and not of models, as happens in fault simulation. It is also more portable and cost effective than physical fault injection besides allowing injection of high-level error patterns; thus, being suitable for validating software systems.

In this paper we propose the use of computational reflection to implement software fault injection for validation of object-oriented systems. By using the metaobject approach the implementation of fault injection and monitoring features can be carried out regardless of the application's functionality.

One important characteristic of this approach is that instrumentation is transparently introduced, from the programmer's point of view: there's no need either to insert software trap instructions explicitly into application source code or to run the application in trace mode. Reusability is also an important feature of this approach, in that fault injection and monitoring metaobjects can be easily incorporated to other applications. Another advantage is that it is simple to use once there is no need to run the program in privileged mode, nor is it necessary to use special hardware features that are not made available by vendors to ordinary users.

Although some limitations still remain, like interference with program execution, inherent to software fault injection and monitoring, and the need of the application source code, this approach is rather attractive in that it allows the development of a framework for building software-implemented fault injection tools.

The architecture described in the previous section is implemented in OpenC++1.2 and runs in a workstation under the UNIX operating system. Some preliminary tests are being run at the moment, using an implementation of a fault-tolerant stack [RC97] as target application. Other applications that run in a distributed environment are also being considered. Therefore, our next step is to develop a distributed architecture for the tests.

7. References

[AAA+90] Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.-C.; Laprie, J.-C.; Martins, E.; Powell, D. "Fault injection for dependability validation - a methodology and some applications". *IEEE Transactions on Software Engineering*, vol. 16, n. 2, Feb. 1990, pp. 166-182.

[AT95] Avresky, D.R.; Tapadiya, P.K. “A Method for Developing a Software-Based Fault Injection Tool”. Technical Report, Departement of Computer Science, Texas A&M University n° 95-021, 1995.

[CMS95] Carreira, J.; Madeira, H.; Silva, J.G. “Xception: a software fault injection and monitoring in processor functional units”. 5th IFIP International Working Conference on Dependable Computing for Critical Applications, Urbana-Champaign, Illinois, USA, 1995, pp. 135 -149

[CM93] Chiba S.; Masuda T., “Designing an Extensible Distributed Language with Meta-level Architecture”. *Proc. ECOOP’93 LNCS 707*. 1993. p. 28-37.

[Chi93] Chiba S., *“Open-C++ Release 1.2 Programmer’s Guide”*, Technical Report n^{o} 93-3, Dept. Information Science, University of Tokyo, 1993.

[FNP+95] Fabre, J-C; Nicomette, V.; Perennou, T.; Stroud, R.; Wu, Z. “Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming”. *Proc. FTCS-25*, Pasadena (CA), June/1995, pp.489-498.

[HRS93] Han, S.; Rosenberg, H.A; Shin, K.G.. “DOCTOR: an Integrated Software Fault Injection Environment”. Technical Report, University of Michigan n° CSE-TR-192-93, 1993.

[HTI97] Hsueh, M.C.; Tsai, T.K.; Iyer, R.K.“Fault Injection Techniques and Tools”. *IEEE Computer*,April/1997, pp.75-82.

[KKA92] Kanawati, N.; Kanawati, G.; Abraham, J. “FERRARI: A Tool for the Validation of System Dependability Properties”. *Proc. FTCS-22*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 336-344.

[KKA95] Kanawati, N.; Kanawati, G.; Abraham, J. “Dependability evaluation using hybrid fault/error injection”. *IEEE Transactions on Computer*, February/1995, pp. 224-233.

[Lap92] Laprie, J.-C.(ed.). “Dependability: basic concepts and terminology”, vol. 5 of Dependable Computing and Fault Tolerance, Springer Verlag, 1992.

[Lis97] Lisboa, M.L.B. “Computational Reflection in the Object Model”. Tutorial presented at II Brazilian Symposium of Programmin Languages, Campinas, SP, Brazil, 1997, 53 pages. (in portuguese)

[LE93] Lovric, T.; Echtle, K. “ProFI: a Processor Fault Injection for Dependability Validation”. *IEEE International Workshop on Fault and Error Injection for Dependability Validation*, Gothenburg, Sweden, Jun 1993.

[MWY91] Matsuoka, S.; Watanabe, T.; Yonezawa, A. “Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming”. *Proc. ECOOP’91*, Springer-Verlag, 1991, pp.213-250.

[RS93] H.A.Rosenberg, K.G.Shin. “Software Fault Injection and its Application in Distributed Systems”. *Proc. FTCS-23*, Toulouse, França, 1993.

[RC97] Rubira C. M. F., Corrêa S. L.. “FORD: a Reflective Object-Oriented Framework for the Construction of Fault-Tolerant Software”. *Proc. VII Brazilian Symposium on Fault-Tolerant Computing*. Campina Grande, PB, Brazil, 1997, pp.75-89. (in portuguese)

[SVS+88] Z.Segall, D.Vrsalovic, D.P.Siewiorek, D.Yaskin, J.Kownacki, J.Barton, R.Dancey, A.Robinson, T.Lin. “FIAT-Fault Injection Based Automated Testing Environment”. *Proc. FTCS-18*, Tokyo, Japão, 1988, pp102-107.

[TI92] Tsai, T.K; Iyer, R.K. “FTAPE: a Fault Injection Tool to Measure Fault Tolerance”. *Proc. Of AIAA Computing in Aerospace 10*, San Antonio, Texas, USA, 1992, pp.339-346. Obtainable also via Internet: fttp://www.crhc.uiuc.edu/pub/ttsai/cia95.ps.gz.

[VMK+97] Voas, J.; McGraw, G.; Kassab, L.; Voas, L. “A ‘crystal ball’ for software liability”. *IEEE Computer*, June/1997, pp.29-36.