A Model Based Framework for Specifying and Executing Fault Injection Experiments

János Oláh and István Majzik
Budapest University of Technology and Economics
Dept. of Measurement and Information Systems
Magyar Tudósok krt. 2., H-1117 Budapest, Hungary
jantsee@gmail.com, majzik@mit.bme.hu

Abstract

Dependability is a fundamental property of computer systems operating in critical environment. The measurement of dependability (and thus the assessment of the solutions applied to improve dependability) typically relies on controlled fault injection experiments that are able to reveal the behavior of the system in case of faults (to test error handling and fault tolerance) or extreme input conditions (to assess robustness of system components). In our paper we present an Eclipse-based fault injection framework that provides a model-based approach and a graphical user interface to specify both the fault injection experiments and the run-time monitoring of the results. It automatically implements the modifications that are required for fault injection and monitoring using the Javassist technology, this way it supports the dependability assessment and robustness testing of software components written in Java.

1. Introduction

In dependable and safety-critical computer systems the validation of the applied error handling and fault tolerance mechanisms is a basic requirement. One of the well-known methods for the experimental validation is the use of *controlled fault injection* in order to study the behavior of the system in case of faults (to test error handling and fault tolerance) or extreme input conditions (to assess *robustness* of system components). Several different approaches exist that can be distinguished on the basis of the techniques and targets of the fault injection (a good overview of these techniques is provided by the AMBER project [1]). To reach the goal of fault-injection, four important questions have to be answered during the planning of a fault-injection experiment: *what kind of* fault model to use; *where* to inject faults; *when* the injected fault should be activated; *how* to plan a workload [2].

To inject faults (or fault effects) into software, two prevalent methods exist: fault-injection and error injection. In the first case, the injected faults are to simulate design faults committed by the developer. This method is also known as mutation. In the other case, error injection, the injected faults affect the operation of the program under test. This can be done in two ways: program state manipulation, and parameter corruption, which is also known as robustness testing. The classic software fault injection tools (e.g., FTAPE [3], DOCTOR [4]) offer possibilities to emulate hardware faults in order to examine the target software, but direct software fault injection to carry out a robustness testing scenario is usually not supported.

The classic tools are mostly configured by using scripts or parameter files. The preparation of these technology-specific scripts and the parameterization of the injector are low-level manual processes. Our goal was to develop a tool that fits to the modern model based development approach. *Model based development* refers to the systematic use of models as primary artifacts throughout the engineering lifecycle [9]. Precise engineering languages (like UML, BPEL, AADL, etc.) allow not only an unambiguous specification and design but serve as the input for subsequent development steps like (automated) code generation, verification, and testing. Our idea is to offer a solution to design the fault injection based validation steps on the basis of the model of the application. The model based approach hides the specific details of the low-level fault injection tool or technology, this way the designer can focus on the important questions mentioned above (location of faults to be injected, activation rate etc.).

In this paper we present our Eclipse-based fault injection framework that implements the model based approach and supports the dependability assessment and robustness testing of software components written in Java. Our framework is characterized by the following features: (1) it provides a graphical user interface to specify both the fault injection experiments and the run-time monitoring of the results *on the basis of a model* that resembles the UML class diagrams, (2) it is able to *construct this model automatically* by processing the Java byte-code of the components, (3) it offers a *library of predefined fault patterns* that can be easily configured on the basis of the model, (4) it *automatically implements the modifications* that are required for fault injection and monitoring using the Javassist technology [6], and finally (5) it runs the experiments and provides the results in a way that allows a rough comparison (but supports more sophisticated analysis of the results as well).

The structure of our paper is as follows. Section 2 discusses the model based approach for designing and executing fault injection experiments. Section 3 presents the initial fault and monitoring library, while Section 4 summarizes the components of our framework. In Section 5 a fault injection scenario is described.

2. The model based approach

Our framework fits to the model based development approach by offering to the tester (validator) the model of the tested application (using UML class diagram model elements) and domain-specific extensions that allow the configuration of the fault injection experiments.

Accordingly, the tester configures the faults on the graphical interface of the framework that is based on the application's model. The specific code snippets that perform the injection of the faults and monitoring of the effects are generated in the background and are injected into the tested application by the framework with the aid of the Java bytecode manipulator technology called Javassist [6].

2.1 Typical use case scenario

The typical use of the framework can be divided to four main steps.

In the first *set up* step, the user creates the new project. This must include the Java class files of the application under test, and optionally the sources files as well. After creating the new project, the framework builds the model of the internal structure of the tested application. The model can be extracted either from the UML based design models or from the implementation of the application automatically.

In the next, *configuration* step, the user configures the faults to be injected into the application (i.e., creates the *fault configuration*), and also configures the monitoring points (i.e., creates the *monitoring configuration*). This configuration step is the most important part of the scenario. The fault configuration specifies the type(s) of the fault(s) and the location(s) of injection (designated on the model), the time or rate of activation, and the parameters of the fault(s). The monitoring configuration specifies the information (e.g., a variable or the return value of a method) that has to be observed (also designated on the model by drag-and-drop).

The third step is the *execution*. The tester selects the fault configuration to be injected and the monitoring configuration to observe the application. The framework performs the injection and instrumentation, and launches the instrumented application under test. During the run of the experiment, the instrumentation creates log files according to the monitoring points.

The last step is the *analysis* of the result. In this phase the tester can compare the log files extracted from different experiments. The framework provides a facility to visualize the amount of differences on a pie chart. The content of the log files can be exported to a database using a star schema [10] that allows more sophisticated analysis (like OLAP) as well.

2.2 Metamodel for configuring fault injection and monitoring

To allow model based configuration of fault injection and monitoring, the framework provides a *domain specific visual modeling language*. Naturally, this language has to carry the information necessary to design the fault injection experiments. Moreover, it has to fit to the widely used engineering languages in order to be familiar to the testers and developers (to facilitate the easy configuration of experiments).

According to these requirements, we defined a metamodel, which is presented in Figure 1.

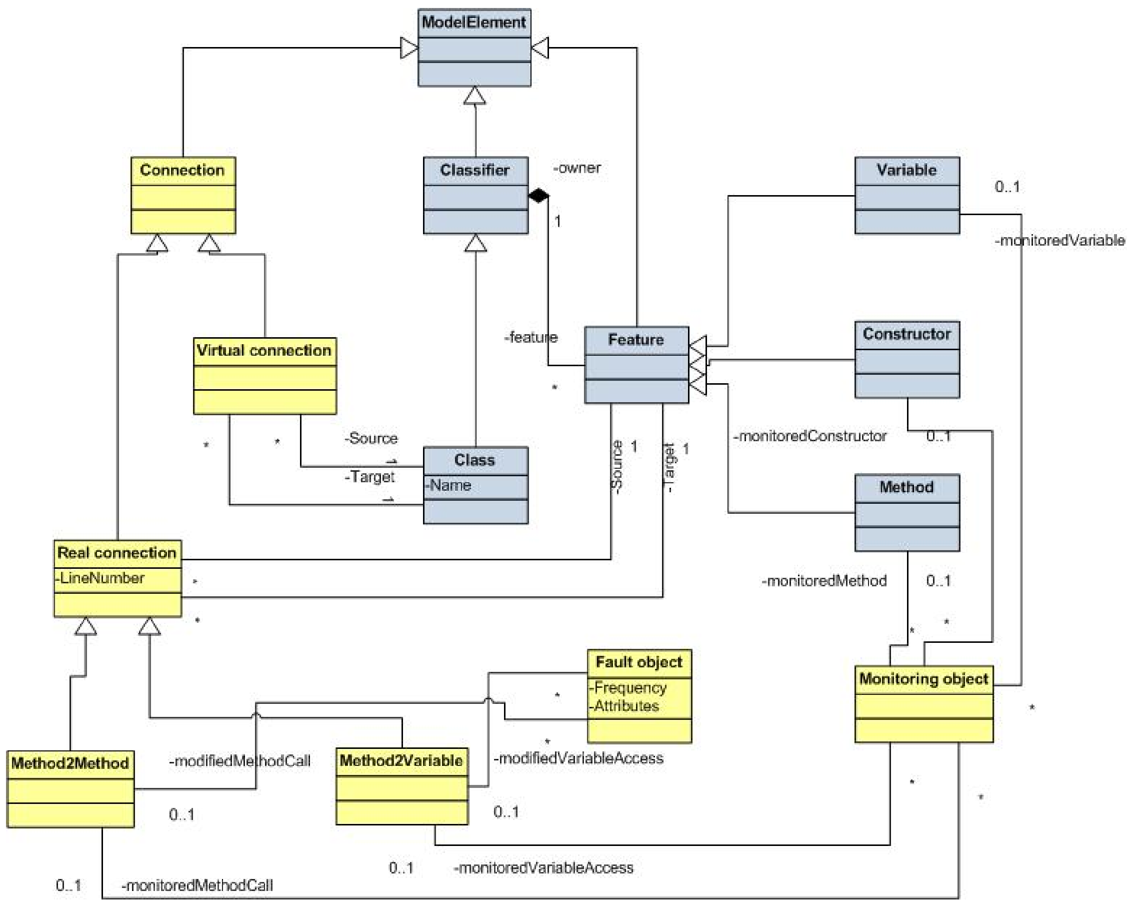

Figure 1. The metamodel

The majority of model elements originate in the well-known UML metamodel. An example is the Class object, which is the descendant of Classifier, and its features include Variables, Constructors and Methods. Accordingly, these elements are presented in the framework on a diagram that resembles UML class diagrams.

The other elements in the metamodel carry the domain-specific extensions, i.e., the information that is specific to fault injection experiments. These domain specific extensions include the *Connections*. A so-called *Real connection* exists between two features, more specifically between two methods or between a method and a variable. In the first case, a target method is called by a source method, in the second case a variable is read or is written by a method. *Virtual connections* exist in the model between classes if there is a real connection between any feature of the source and target classes. (As there would be too many connections simultaneously in the visual representation of the model, the connections may be hidden.) Note that the UML based model elements and the connections represent the internal structure of the application, which can be obtained in two ways (as discussed in Section 2.3).

Finally, the configured *Fault objects* and *Monitoring objects* are also included in the metamodel. Fault objects are associated with the connections they modify, while the monitoring objects are associated with the features and connections they observe. These objects are instantiated by the user of the framework, and they form the fault configuration and the monitoring configuration, respectively.

2.3 Obtaining the structure of the application

There are two different ways to extract the necessary information to visualize the structure of the application.

The first case is the use of the UML model created during the design of the application. The UML class diagrams, collaboration and sequence diagrams can be processed to identify the classes, features and connections, i.e., all model elements included in the metamodel of Figure 1 except the fault and monitoring objects. A disadvantage of this method is that UML models are often not available in the case of off-the-shelf (OTS) or legacy components included in the tested application.

The other possible way to obtain the model is *reverse engineering*. This means that the necessary information for the model is extracted from the actual implementation of the tested application using the Javassist technology that provides effective methods to explore the classes, methods, variables and connections. This way the structure of OTS and legacy components can also be explored. The current version of the framework supports this method for obtaining the model.

3. Fault library and monitoring library

The injection and instrumentation according to the configured fault objects and monitoring objects are performed by the framework in the background. The code snippets belonging to faults and monitors are defined in re-usable and extensible form in two modules called *fault library* and *monitoring library*. These libraries contain Javassist source code templates. In the execution step, the framework parameterizes the templates on the basis of the configured fault/monitoring objects (by setting the different parameters like fault activation frequency, erroneous return value etc.), creates the actual Java byte code with the use of the Javassist technology, and injects them into the application under test.

The current version of the fault library contains six different types of faults. With the combination of these faults, a representative faultload can be configured for robustness testing of a Java application.

- *Return cancelled:* In this case the called method does not return to the caller (return is omitted). These faults are often used to examine the timeout mechanisms in the caller.
- *Method call cancelled:* The effect of this fault is the omission of the configured method call. In this way, for example, a variable set by this method remains uninitialized.
- *NULL value returned:* This fault changes the return value of a method to NULL. It can be used to examine whether a method checks the return value before use.
- *Predefined number returned:* This fault means that the return value of a method is fixed to a predefined number. In this way, for example, the checking of extreme return values can be investigated or specific execution traces can be forced.
- *Predefined string returned:* Similar to the previous, but in this case the predefined return value is a string.
- *Predefined parameters in method call:* This fault sets the parameters of a method call. This kind of fault can be used to examine the input value checking of the corresponding method.

As an example, we present a Javassist code snippet that is taken from the template belonging to the fault specified as "*Method call cancelled*":

```
if("+newFieldName+"=="+fault.getFrequency()+"){
  "+newFieldName+"=1; }
else{
  ;$_ = $proceed($$);"+newFieldName+"++; };
```

This template demonstrates that Javassist is a powerful technology (supporting the addition, removal and modification of methods and insertion of new variables) but the low-level implementation of the fault is not a trivial task (even for Java programmers). Fortunately, these details are hidden from the tester as only the visual configuration is needed in the framework.

The monitoring library is different from the fault library, because it contains complete methods that instrument the application under test and perform the configured logging activities. Monitoring is implemented by calling these methods in the specified situations.

The library contains five types of monitoring mechanisms. With the proper configuration, the behavior of the application under test, and the effects of the injected faults can be monitored.

- *Parameter monitoring:* This monitoring method logs the parameters of a method call.
- *Return value monitoring:* This method logs the return value of a method call.
- *Try-catch monitoring:* This monitoring method inserts a try-catch block around the selected method call or variable access, and records if an exception is thrown.
- *Variable monitoring:* This monitoring method logs the value of a variable during the execution of the experiment.
- *Method call monitoring:* This monitoring method crates a log entry if a specified method is called (independently of the caller).

4. Architecture and components of the framework

The architecture of the framework (Figure 2) is inspired by the design patterns documented in [5].

The main component of the framework is the *Controller*, which manages the operation of all the other components and communicates with the user interface.

The *User interface* is implemented using the Eclipse RCP [7] and Eclipse GEF [8] technologies. Accordingly, the framework is a stand-alone Eclipse plug-in, which uses the GEF technology to visualize the model of the tested application.

The *Parser* component is responsible for creating the model of the tested application from its implementation. To extract the necessary information that is visualized in the model, it uses the Javassist technology. The structure of the application is stored in an XML based *Model descriptor file*.

The *Fault injector* component has three tasks, and three subcomponents respectively: (1) creating the fault configuration and saving it into an XML based *Fault configuration file*, (2) processing saved fault configurations both for presentation with the model and before the actual injection, and (3) performing the fault injection corresponding to the selected fault configuration. It refers to the *Fault dictionary* which is the library containing the fault templates.

The *Monitor injector* component is similar to the *Fault injector* component; the only difference is that it manages the *Monitor configuration file* and the *Monitor dictionary*.

The *Activator* component identifies the entry point of the application under test and launches the instrumented version of the application (using a simple batch file).

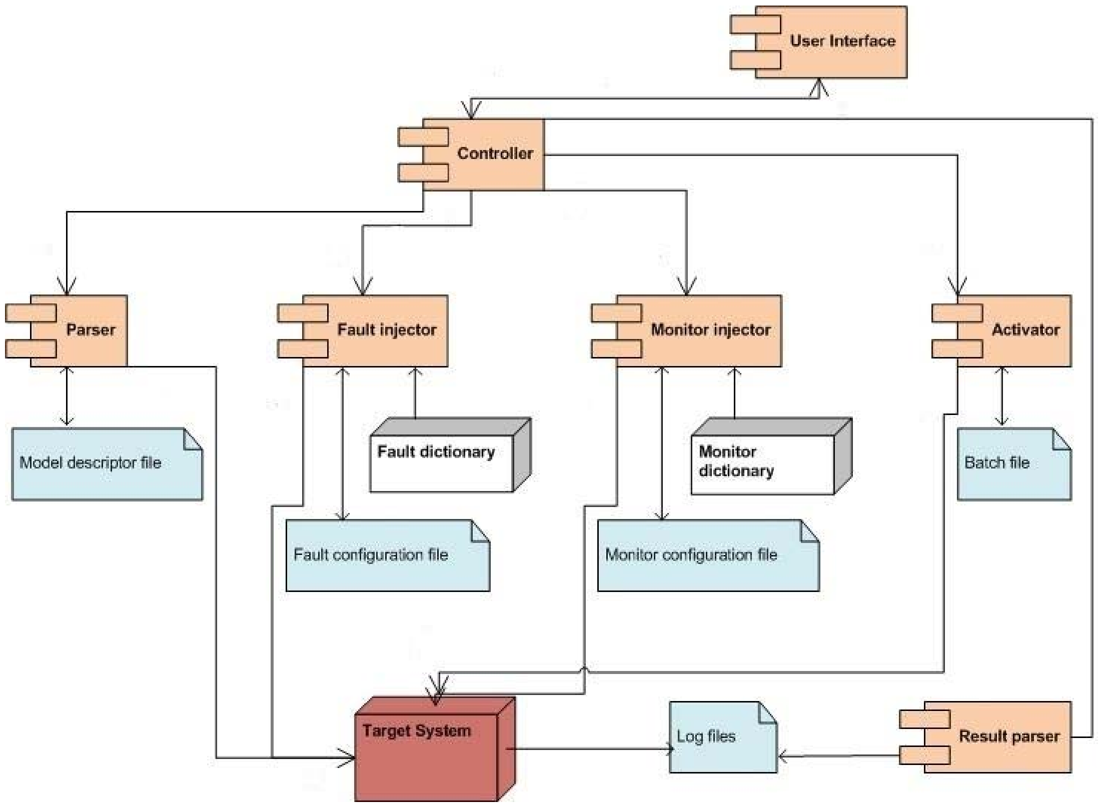

Figure 2. The architecture of the framework

The *Result parser* component is responsible for processing the *Log files*. It is able to present the log files in textual format, and also offers a facility to compare log files.

5. Using the framework

The four main steps of a typical application scenario were already introduced in Section 2.1. Here we present the main features and usability aspects of the implemented framework.

In the first step the user (tester) creates a new project in the workspace giving the root folder of the application under test (all binary files under that folder will be added automatically to the new project). Source files can also be added, in this way location of the faults can be shown in the Java source code as well.

The next step is the building of the model of the application. By clicking on the *"Create model"* button in the framework, the structure of the application is explored and a model is created according to the metamodel presented in Section 2.2 and saved in an XML file. It is then visualized in the editor area of the framework (see Figure 3).

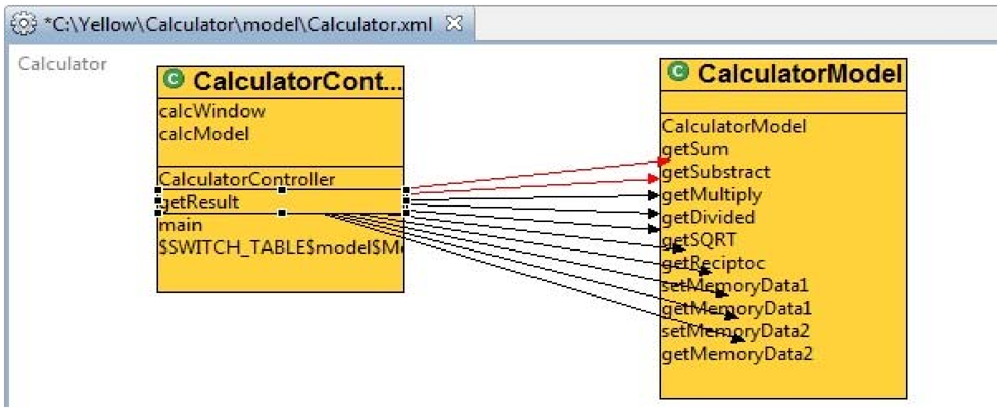

Figure 3. Presentation of the model (with a selected connection)

Having the model in the editor area, the configuration of the faults and monitoring objects becomes possible. Faults may be configured on real connections of the model, while monitoring objects may be placed on real connections, methods or constructors, depending on the actual monitoring object's specification.

When a connection is selected the fault configuration view becomes active (see in the middle of Figure 4). The tester may choose any kind of fault, and may set the suitable parameters (frequency, value if needed, etc.). Finally, the newly created configuration is saved.

The next step is to create a monitoring configuration. This can be done by dragging and dropping objects from the palette of the editor area (see on the left part of Figure 4) to the model. At the end, the monitoring configuration is to be saved.

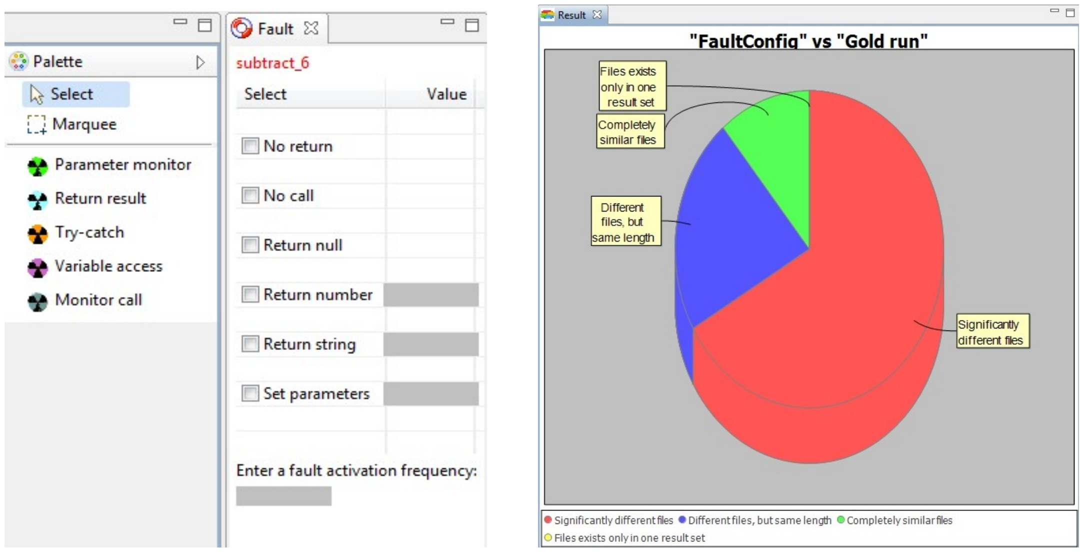
Palette
Select
Marquee
Parameter monitor
Return result
Try-catch
Variable access
Monitor call
Fault
subtract_6
Select
Value
No return
No call
Return null
Return number
Return string
Set parameters
Enter a fault activation frequency:
Result
"FaultConfig" vs "Gold run"
Files exists only in one result set
Completely similar files
Different files, but same length
Significantly different files
● Significantly different files ● Different files, but same length ● Completely similar files
● Files exists only in one result set

Figure 4. The monitoring and fault configuration palette (on the left) and the comparison of the log files (on the right)

During the execution phase, the application under test has to execute a predefined workload, which duly represents the use of the application. The construction of the workload is the responsibility of the tester (scripts or GUI-tester tools can be used for this purpose).

In order to assess the results obtained in presence of injected faults, it is useful to have a set of log files obtained during a golden run. Choosing the *Golden run* option of the framework the application is executed without injecting faults.

The log files recorded by the monitoring objects in the golden run and in the presence of injected faults (according to the selected fault configuration file) are collected in separate folders and may be opened in a simple textual viewer. The built-in *analyzer component* of the framework is able to compare the contents of the set of log files belonging to two fault configurations (or to a fault configuration and to the golden run) and summarize the deviations between the log files in a pie-chart (see on the right part of Figure 4).

6. Conclusion

The main advantage of the model-based configuration and execution of fault injection experiments is that the tester does not have to be familiar with the low-level configuration format of the fault injection technology (e.g., the Javassist language extensions). Instead of learning a specific language for this purpose, she/he can configure the faults and monitors in a model based framework. The framework presents the structure of the application in a well-known (UML based) model view and allows the configuration by easy-to-use visual objects and graphical facilities.

The framework is based on a domain-specific metamodel that ensures the unambiguous configuration of the faults and monitoring objects. The semantics of these objects is intuitive, and the implementation is made available in separate fault and monitoring libraries in code templates that are configured, applied and compiled by the framework automatically. The extension of the libraries necessitates the preparation of the code templates (by Javassist experts) and specification of the corresponding configuration templates as well. After this, the templates can be easily re-used in the framework. The current set of faults can be used to examine the effects of interaction faults in Java applications and test the robustness of Java components. Experiments with the tool (in the frame of the AMBER project [1]) and extension of the tool with other fault types are in progress. The first validation of the tool in Java applications was encouraging, as several robustness problems were detected.

References

[1] AMBER Consortium, “State of the Art”. Deliverable D2.1, http://www.amber-project.eu/

[2] J. Christmansson and R. Chillarege, “Generation of an Error Set That Emulates Software Faults Based on Field Data”, 26th Int. Symp. on Fault Tolerant Computing, 1996, pp. 304-313.

[3] T. K. Tsai, R. K. Iyer, and D. Jewitt, “An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems”, 26th Int. Symp. on Fault Tolerant Computing, 1996, pp. 314-323.

[4] S. Han, K. G. Shin, and H. A. Rosenberg, “Doctor: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems”, Int. Computer Performance and Dependability Symposium, Erlangen, Germany, 1995, pp. 204-213.

[5] E. Martins, C.M.F. Rubira, N.G.M. Leme, “Jaca: A reflective fault injection tool based on patterns”. Proc. DSN'02, Bethesda, MD, USA, 2002.

[6] Javassist: www.csg.is.titech.ac.jp/~chiba/javassist/ (2008. 09. 22.)

[7] Eclipse Rich Client Platform. http://wiki.eclipse.org/index.php/Rich_Client_Platform, (22 Sept, 2008).

[8] Eclipse Graphical Editing Framework. http://www.eclipse.org/gef/overview.html

[9] D. C. Schmidt, “Model-Driven Engineering”, IEEE Computer 39 (2), February 2006.

[10] H. Madeira, J. Costa and M. Vieira, “The OLAP and Data Warehousing Approaches for Analysis and Sharing of Results from Dependability Evaluation Experiments”, Proc. DSN’03, San Francisco, USA, 2003, pp 22-25.