Improving Usability of Fault Injection

D. Cotroneo*, L. De Simone*, A.K. Iannillo†, A. Lanzaro†, R. Natella*†
*Critiware s.r.l. / Università degli Studi di Napoli Federico II, Italy
†Consorzio Interuniversitario Nazionale per l'Informatica (CINI), Italy

***Abstract*—The lack of tools that can fit in existing development practices and processes hampers the adoption of Software Fault Injection (SFI) in real-world projects. This paper presents an ongoing work towards an SFI tool integrated in the Eclipse IDE, and designed for usability.**

Keywords*—*Fault Injection Tools; Eclipse IDE; Usability

I. Introduction

As software is becoming more and more pervasive in many critical systems, it is more important than ever to assure software dependability against threats from the environment, users, other systems, and even design flaws in software components. Software Fault Injection (SFI) is an approach for gaining confidence about fault-tolerance properties of a software system, by deliberately injecting perturbations in the software to emulate faulty components and stressful conditions.

Research efforts in the last decades on SFI focused on extending the scope of fault injection from its traditional focus on the *hardware dimension* (faults that originate in, or affect, hardware, e.g., heavy-ion radiations and electromagnetic interference [3]) to the *software dimension* (faults that affect software code and data, e.g., bugs, configuration faults, resource exhaustion or unavailability). Several studies advocated the use of SFI for:

- **Validating fault-tolerance mechanisms**: SFI can evaluate error detection and handling mechanisms (such as assertions and exception handlers) against faulty hardware and software components, and to improve such mechanisms if needed [17], [13], [2], [4].
- **Aiding FMECAs (Failure Mode, Effects, and Criticality Analysis)**: Developers can quantify the impact of a faulty component on the overall system (e.g., in terms of catastrophic system failures), and mitigate risks by testing the most critical components and by revising the system design [21], [9], [15].
- **Dependability benchmarking**: SFI helps developers to choose among alternative systems or components the one that provides the best dependability and/or performance in the presence of faults [11].

Recent technical advances in SFI include the definition of realistic fault models for software [8], [14] and operator faults [12], the non-intrusive injection of faults using advanced debugging facilities [5], [1], and easy portability to different target systems [20]. Despite these technical advances, SFI is still not widely used among developers. We partially attribute this to the lack of tools that can fit in existing development environments, technologies, practices, and processes that are adopted everyday by developers. For this reason, as will be discussed in the next sections, we aim at compensating this gap by developing a fault injection tool closer to users' practical needs.

II. Fault injection tools

Software Fault Injection includes software-implemented fault injection (SWIFI) techniques and tools, that were initially developed to emulate hardware faults, and were later adopted to also emulate software faults. Moreover, SFI includes more recent, ad-hoc methodologies specifically developed for emulating software faults. The research split into two main directions. While fault injection *techniques* have been evolving to a more accurate emulation of faults, *tools* have been evolving towards better usability. A brief overview of some SFI tools, with emphasis on usability, is reported below:

- **FIAT** [18]: it was one of the first SWIFI tools, aimed to provide an automated testing environment for distributed real-time systems. Based on fault injection techniques, the tool needs to be fed with workload, faultload and experiment descriptions. This information should be written in C files and configuration files with a specific format. These files are automatically transformed into the experimental script that runs the fault injection campaign;
- **GOOFI** [1]: this tool aims at adapting to various target systems and different fault injection techniques. The user has the possibility to customise the tool to target systems, fault injection techniques, workloads and fault models through the implementation of some abstract methods in an object-oriented fashion. Then, a graphical user interface guides the user to define and run a fault injection campaign, storing the system state logs into a database. The user analyses the results by writing SQL queries;
- **NFTAPE** [20]: it is a modular environment whose components (*i.e.*, the fault injection component, the fault trigger component, and the control mechanisms) can be simply added and their combination can be configured. A control host executes the fault injection experiment based on a user-provided script. A *lightweight injector* and trigger can be added (and possibly customised) to different target systems, and controlled by the control host;
- **ORCHESTRA** [7]: this fault injection tool is based on a framework called *script-diven probing and fault-injection*. A probe-injection layer is inserted between any two consecutive layers in a protocol stack, and its behaviour is determined by a user-defined script;

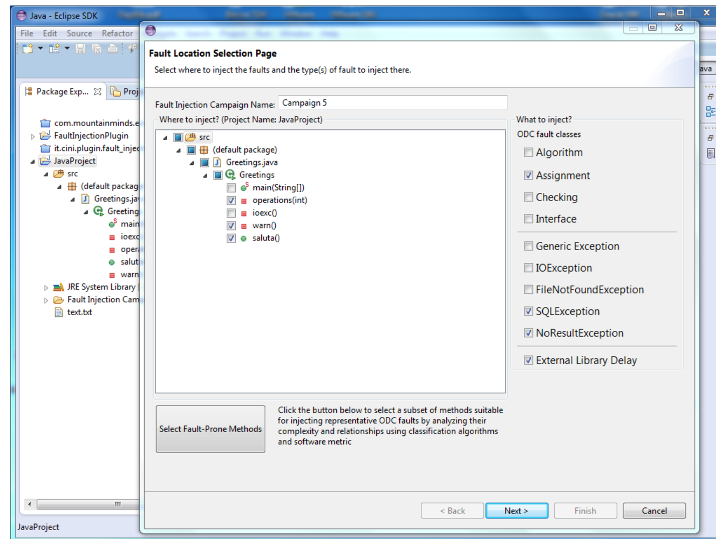

Fig. 1. Fault Injection Tool - Plug-in Wizard

- **Loki** [6]: based on a partial view of the global state of a distributed system, this tool injects faults in one node of the system according to the states of all the nodes. A fault injection campaign is specified through more than one graphical user interface. The state machine specification is easily performed due to a visual editor, while the fault specification exploits boolean expressions, in a very simple language, in order to formalize the fault trigger;
- **PreFail** [10]: it is a programmable fault-injection tool designed for experiments with multiple injections, and it avoids the combinatorial explosion of the number of experiments by using sampling heuristics. The user should provide the tool both failure-injection points (where to inject the failures) and failure-injection tasks (how to inject these failures). In addiction to these, the user should use a special declarative language to write filter and cluster policies in order to reduce the experimental space.

There are still open issues with respect to usability. In the tools mentioned above, a user needs to write scripts or configuration files, to extends programs or to use specific APIs in order to perform the desired fault injection test. All these tasks may impose significant efforts on the user due to the adoption of an ad-hoc formalism.

Another open issue is the lack of integration of FI in existing Integrated Development Environments (IDEs) and development processes. Fault injection is mainly performed after the development phase, before deployment, along with other testing activities. Integrated environments are widely used for software development, as they provide comprehensive facilities and they are moving into the integration of software testing tools (*e.g.*, JUnit for Eclipse) that developers expect to find there. To the best of our knowledge, there is no IDE that integrates a fault injection tool and let a user exploit these innovative techniques in a simple way. Notable exceptions are represented by tools such as MODIFI [19], which is integrated in Simulink, but focuses only on model-based projects.

III. KEY FEATURES

A SFI tool should guide practitioners through three fundamental phases: (1) create the fault injection tests by selecting *what*, *where* and *when* to inject, (2) build and execute the fault injection tests, and (3) gather, analyse and present the results. Both the way these three phases are automated and the effort a user should make to reach his/her goal influence the tool usability and, thus, have an impact on productivity. The more the tool is usable, the more it facilitates the adoption of fault injection in the software life cycle, by reducing costs in terms of time or human resources. In our tool, we consider the following key features for a usability-oriented design:

Integration of the tool in an IDE speeds up the setup of the fault injection testbed, and the learning process for the application of fault injection. The user works with the same familiar interface, enhanced for SFI tasks, to create the fault injection tests and execute them. The tool exploits the IDE procedures to automate the build, execution, and monitoring of the target system.

An exhaustive **set of injectors**, for different types of faults and target systems (e.g., systems based on different programming languages or operating systems). The injectors allow the user to adopt the fault types that fulfill the fault-tolerance requirements for the system (e.g., the user can select to inject a specific type of fault, such as select whether to inject hardware or software faults, see Fig. 1). Moreover, the tool aids the user in the selection of *what*, *where*, and *when* to inject. For instance, if the user wants to inject a specific type of exception or error at library interfaces, the tool automatically detects where this kind of faults can be injected (e.g., focus the injection on specific methods/library function that can throw that error). Or, if the user wants to inject faults for a specific method or library interface, the tool automatically detects what types of errors can be injected. Finally, the injectors aids the developers at sampling the space of injections, by using software complexity metrics to identify which components are most fault-prone.

Finally, the fault injection tool includes procedures to **gather, analyse and present** the results of the fault injection tests in a way the practitioner can get useful feedback for improving the system under test. Our tool provides the user with brief statistics about the *failure modes* exhibited by the system during the experiments. Analyzing failure modes is useful for studying the severity of failures of the target system, and to point out faults that are not tolerated by the system (and which require to improve fault-tolerance mechanisms to be tolerated). Two types of failure modes are considered: "standard" failure modes that are application-independent (e.g., crash or hang failure), and application-dependent failure modes. For the latter, we require the user to provide limited information: (1) the list of application-dependent failure modes, and, for each failure mode, (2) the I/O channels to be monitored during the experiment (e.g., output log files, standard output/error, calls to I/O methods or system calls), and (3) the criteria for correlating outputs and failure modes (e.g., to identify whether a fault has been tolerated, the user can specify to look for a keyword in a log file produced by a fault-tolerance mechanism; or, the tool can compare the contents of an output file with a fault-free run).

At time of writing, the tool is being implemented as an Eclipse plug-in. It supports both Java and C/C++ applications, focusing on C embedded applications, and can inject errors from the environment (e.g., I/O exceptions) and software bugs.

IV. USE CASE EXAMPLE

In the common use case, we expect that the user selects and configures fault injection tests using a wizard (see Fig. 1). The plug-in tool gets information about by the project opened in the current workspace, and adds a fault injection menu for each project. The user can start a new fault injection campaign, or load a previously saved one. In both cases, the wizard:

1) shows the elements of the selected project in a check box tree (e.g., Java, packages, classes, methods);
2) requires the user to insert information about desired injection points, fault types and triggers, with the aid of hints and, possibly, automated filtering tasks (e.g., during the selection of injection points [16]);
3) creates fault injection test plans, i.e., a set of experiments which specifies *what* to inject (i.e., which kind of faults to inject), *when* to inject (i.e., the timing of the injection), and *where* to inject (i.e., in which part of the target system the user wants to inject).

The plug-in saves a test plan as an Eclipse *run configuration*, an abstraction that allows the user to run FI tests as it does with usual applications. This execution is controlled by the tool, which saves outputs from the target system. The plug-in, then, analyzes these data and reports results to the user. These results are presented to highlight, for example, the distribution of failure modes, which reflects the effectiveness of fault-tolerance mechanisms. Moreover, the raw data from the campaign will be stored and made available to the user for in-depth analysis (e.g., the user could be interested in looking at system logs, or to repeat a specific fault injection experiment).

ACKNOWLEDGMENT

This work has been partially supported by the SVEVIA PON Project (PON02 00485 3487758) funded by the Italian Ministry of Education, University and Research.

REFERENCES

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic Object-Oriented Fault Injection Tool. In *Proc. Int. Conf. on DSN*, pages 83–88, 2001.

[2] J. Arlat, J. Fabre, M. Rodríguez, and F. Salles. Dependability of COTS Microkernel-Based Systems. *IEEE TC*, pages 138–163, 2002.

[3] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE TDSC*, pages 11–33, 2004.

[4] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano. Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications. *IEEE TDSC*, pages 223–237, 2004.

[5] J. Carreira, H. Madeira, and J. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE TSE*, pages 125–136, 1998.

[6] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In *Proc. Int. Conf. on DSN 2000.*, pages 237–242, 2000.

[7] S. Dawson, F. Jahanian, T. Mitton, and T. Tung. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *Proc. Intl. Symp. on Fault-Tolerant Comp.*, pages 404–414, 1996.

[8] J. Durães and H. Madeira. Emulation of Software faults: A Field Data Study and a Practical Approach. *IEEE TSE*, 32(11):849–867, 2006.

[9] M. Hiller, A. Jhumka, and N. Suri. EPIC: Profiling the propagation and effect of data errors in software. *IEEE TC*, pages 512–530, 2004.

[10] P. Joshi, H. S. Gunawi, and K. Sen. Prefail: a programmable tool for multiple-failure injection. *ACM SIGPLAN Notices*, 46(10):171–188, 2011.

[11] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.

[12] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proc. Intl. Conf. on DSN*, pages 157–166, 2008.

[13] P. Koopman and J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE TSE*, 26(9):837–848, 2000.

[14] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. An Empirical Study of Injected Versus Actual Interface Errors. In *Proc. Int. Symp. on Software Testing and Analysis*, pages 397–408, 2014.

[15] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental Risk Assessment and Comparison using Software Fault Injection. In *Proc. Int. Conf. on DSN*, pages 512–521, 2007.

[16] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. On Fault Representativeness of Software Fault Injection. *IEEE TSE*, pages 80–96, 2013.

[17] W. Ng and P. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proc. 29th Int. Symp. on Fault-Tolerant Comp.*, pages 76–83, 1999.

[18] 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 *Proc. Intl. Symp. on Fault-Tolerant Comp.*, pages 102–107, 1988.

[19] D. Skarin, J. Vinter, and R. Svenningsson. Visualization of model-implemented fault injection experiments. In *Computer Safety, Reliability, and Security*, pages 219–230. Springer, 2014.

[20] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer. A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proc. Int. Computer Performance and Dependability Symp.*, pages 91–100, 2000.

[21] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly "Good" Software Can Behave. *IEEE Software*, 14(4):73–83, 1997.