Structuring Software Fault Injection Tools for Programmatic Evaluation

Lukas Pirl, Lena Feinbube, and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute at University of Potsdam
Email: lukas.pirl@student.hpi.de, {lena.feinbube, andreas.polze}@hpi.de

***Abstract*—The increasing complexity of software systems challenges the assurance of the likewise increasing dependability demands. Software fault injection is a widely accepted means of assessing dependability, but far less accessible and integrated into engineering practices than unit or integration testing. To address this issue, we present a dataset of existing fault injection tools in a programmatically evaluable model. Our Fault Injection ADvisor (FIAD) suggests applicable fault injectors mainly by analyzing definitions from Infrastructure as Code (IaC) solutions. Perspectively, FIAD can yield findings on how to classify fault injectors and is extensible in a way that it can additionally suggest workloads or run fault injectors.**

Keywords–fault injection; infrastructure as code; testing; service-oriented systems; distributed systems.

I. Introduction

Facilitated by the advent of cloud computing, our society – politically, commercially and individually – increasingly relies on utility and service computing. The need for growing quality and quantity of such ubiquitous distributed software systems boosts their complexity. There is thus a dire need for dependability assessment of such systems but they are notably hard to test: Complex software systems fail in complicated ways [1] even when fault tolerance is implemented [2]. It is widely recognized that *Software Fault Injection (SFI)* should be part of the development process [3]. Yet, SFI lacks the high grade of automation and adaption of testing. Many fault injectors remain research prototypes, others are used only in high criticality systems.

We assume and address a practical issue with SFI tools (*injectors*): Knowledge about available injectors is not user-friendly accessible and it is thus hard to find applicable ones for a given infrastructure. We elaborate on a dataset consolidating the aforementioned knowledge, its programmatically evaluable model, and the automatic acquisition of information about infrastructures. Ultimately, this paper presents a tool which automatically determines applicable injectors by analyzing systems' IaC descriptions.

After core concepts have been introduced in the remainder of this section, Section II outlines related work on the topic. While Section III defines the research problem and our approach to address it on a theoretical level, the subsequent section describes our concrete realization. Section V demonstrates an example use case including characteristic code excerpts. Finally, Section VI draws conclusions from the findings of this work and proposes areas of possible further investigation.

A. Software Fault Injection

Fault injection testing is a well-established approach to test the fault tolerance of computer systems. We use the term SFI to denote the software-implemented injection of *software faults*. This is not to be confused with Software-Implemented Fault Injection (SWIFI), that mainly refers to the injection of *hardware faults*, implemented in software. In the broad field of SFI, many tools exist that differ in target applications, usability and their implementation strategy. For practitioners wanting to embed SFI in a software development process, finding suitable injectors is tedious and likely results in testing various prototypes of varying fitness and quality.

B. Infrastructure as Code

The increasing complexity of distributed systems has amplified the interest in the IaC paradigm [4]. This paradigm describes the definition of infrastructures in a machine- *and* human-readable format. Ideally, IaC enables fully automated provisioning, as well as deployment. Additionally, it has the advantageous side effect that other tools can reuse the infrastructure definitions.

Ansible [5] is an open source IT configuration management, deployment, and orchestration tool of increasing significance [6]. The desired state of the targeted infrastructure is defined declaratively in the YAML [7] file format. Using the same format, so-called *Playbooks* consolidate the desired state regarding an infrastructure at the topmost level. Enterprises, such as Apple and Juniper [8], as well as large open source projects, such as OpenStack [9], use Ansible actively.

II. Related Work

The concept of dependability [10] is an area of interest for computer scientists since decades. Recently, the awareness that the dependability of distributed systems needs in-depth assessment has pervaded major Web enterprises. Netflix injects faults into its production systems using ChaosMonkey [11], similarly to Amazon [12] and Etsy [13]. While earliest fault injectors focused on hardware, SFI became popular in the 1990s (e.g., FIAT [14]). It has been acknowledged that the dependability bottleneck is not within the hardware but within the software layers [15], [16]. Natella et al. present a comprehensive up-to-date survey of SFI approaches from research [17]. The research areas of SFI and "classical" software testing overlap, and the distinction between "workload" and "faultload" blurs. [18] presents a fault model oriented view on software testing. The approach of fuzz testing (e.g., CRASHME [19]) also exemplifies the overlap of fault injection and testing: Here, the "fault model" consists of special inputs to the same API accessed during unit testing. When cataloging injectors, it is hence imperative to consider a broad spectrum of tools from both domains. Our dataset includes popular fuzzers, such as Trinity [20], which are excluded from traditional surveys. Software testing strategies are further discussed theoretically in [18] and surveyed in [21].

 ISBN: 978-1-61208-528-9

Despite decades of research, there is no "cookbook for SFI testing". To the best of our knowledge, no programmatically evaluable catalog of injectors exists.

III. Problem Statement and Approach

Distributed software systems grow in complexity, also due to the rapid rise of cloud computing. Their dependability demands grow along with their importance for enterprises and customers. Distributed software systems are notoriously difficult to test and fail even when adhering to well-established engineering practices (see Section I). SFI is a scalable and versatile approach to assess their dependability experimentally.

Unit and integration testing is widely incorporated into software development processes and allows for a decent level of automation. In contrast, SFI is yet less established and remains a research rather than an engineering topic. One reason may be that the application of SFI to an existing product currently requires significant knowledge and manual effort. It is laborious to get an overview of available injectors and to check their applicability regarding an infrastructure.

To overcome this burden, we leverage the unprecedented availability of information provided by IaC solutions. In particular, our implementation will initially be based on Ansible because of its growing popularity. Our tool – FIAD – hence analyzes Playbooks for the detection of characteristics required by injectors (hence *targets*). Further, we created a dataset of injectors from research, industry and open source projects. Combining these sources of information, FIAD programmatically identifies applicable injectors. Due to the specificity of dependability requirements, users are obliged to judge the fitness of the injectors' fault models. Possible users include dependability researchers and actors in the software development process, such as testers, developers and engineers.

Given that an infrastructure is represented using Playbooks, this approach introduces no extra effort. Also, there are no minimal requirements regarding the information available via Playbooks but self-evidently, the more complete the Playbooks, the more accurate and complete the results. Since no access to the deployed infrastructure is required, FIAD can as well be used on infrastructures outside the own administrative domain.

Test environments and workloads often differ from the production system, which can lead to uncertainty regarding the fault tolerance of the latter [13]. Since FIAD operates on the basis of infrastructure descriptions, it is workload-independent. Hence, it can analyze the actual production system and does not suffer from the aforementioned problem.

IV. Concepts and Realization

At its core, the realized model is a two-sided object oriented class hierarchy: one for injectors and one for targets. Accompanied by the simplified illustration in Fig. 1, the following Subsections elaborate further on its details. We aimed at a programmer oriented model of injectors and targets which satisfies the requirements listed below:

1) It should allow to **flexibly classify injectors by multiple characteristics** and thereby accumulate knowledge from research and documentation of tools.
2) It should be **explicit and not overly abstract**, allowing users to easily understand, customize and extend it.
3) It should **allow for programmatic evaluation**.

FIAD is a command line interface application that reads Playbooks and outputs a list of applicable injectors. The output enables users to get an initial overview of applicable injectors or to expand their existing SFI setup. FIAD is capable of connecting to hosts referenced in Playbooks for the collection of *Facts* [22]. Facts are pieces of information about the target hosts, such as virtualization, hardware and networking details.

The internal procedure of an execution of FIAD is visualized in Fig. 2. Initially, all models (i.e., injectors and targets) register themselves at a registry and are set up in the second step. If required, FIAD detects Playbooks recursively in a specified directory. Thirdly, Playbooks are loaded from files using Ansible's sophisticated internal mechanisms supporting Playbook dependencies, file inclusions etc. Upon explicit request of the user, FIAD gathers Facts from all hosts referenced in Playbooks. Fourthly, all loaded Playbooks are handed to all registered targets to enable the latter to detect themselves within the former. Detected targets then imply further targets accordingly. Finally, detected and implied targets are matched against injectors and the results are displayed.

FIAD is implemented in a modular structure where the topmost coordination happens within the module *cli*. Alongside with some auxiliary modules (e.g., to load and aid in accessing Playbooks and Facts), the implementations of injectors and targets are kept in the module *models*. This module encapsulates all the domain-specific knowledge and is the main extension point of FIAD's dataset and capabilities. When adding an injector or a target, the use of a registry eliminates the need to alter code outside the corresponding module.

A. Injectors

A central contribution is a programmatically evaluable dataset of existing SFI tools (*injectors*). Previous surveys [17], [23] mainly took research in the form of published conference or journal articles into consideration. Yet, for users searching for pragmatic answers to the question of which injector to use, besides the vast body of research, practical experience of industrial SFI is relevant. Hence, our dataset of injectors consolidates knowledge from research, commercial and open source origins. Developers can additionally include new injectors easily (e.g., a in-house injector of an enterprise).

Due to the wide variety of existing injectors, only a semi-structured procedure to assemble the dataset could be followed: Initially, we collected a list of injectors by surveying research publications and by examining open source, as well as commercial products. To cover a broad range of targets, no tools were actively excluded because of their maturity or origin. For every injector found, we identified characterizing properties (e.g., "injects based on a rate"). Subsequently, we deduced more abstract characteristics (e.g., "injection trigger"). Where applicable and possible, we finally determined yet unspecified properties for all combinations of tools and characteristics.

B. Targets

Injectors typically aim at a group of applications or infrastructures sharing certain characteristics, such as "instances on Amazon EC2". We refer to a characteristic required by an injector as *target*. Targets can be at different layers of abstraction (e.g., an operating system versus a Web API).

 ISBN: 978-1-61208-528-9

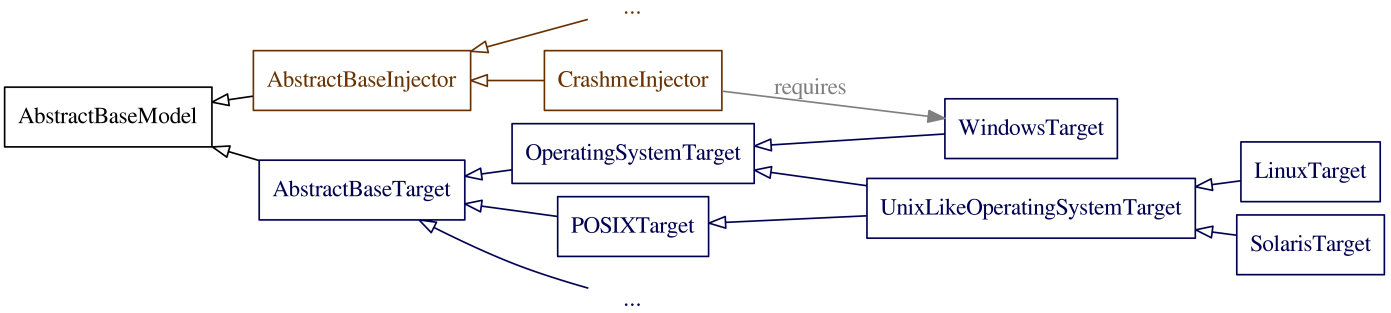

Figure 1. The class hierarchies for injectors (brown) and targets (blue). They interweave through the requirements of injectors on targets (gray).

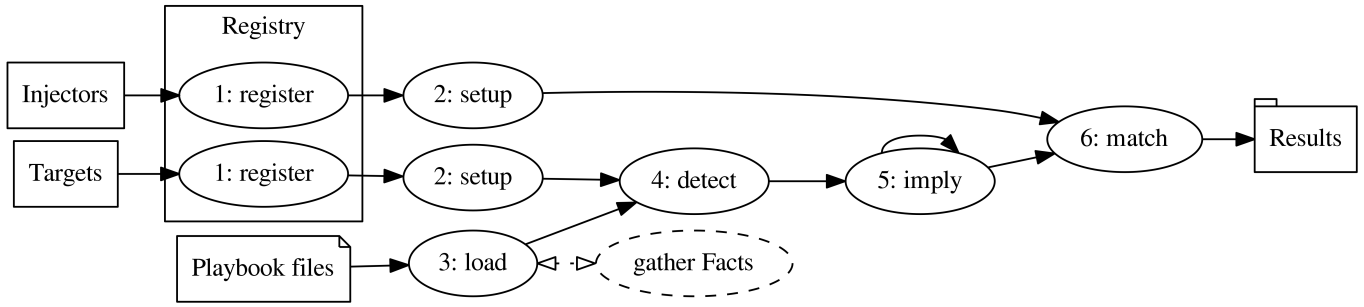

Figure 2. FIAD's execution procedure (excerpt).

Targets can be general, such as "requires a POSIX API", or more specific, such as "requires a Linux operating system". In this example, the latter target implies the former, since Linux implements the POSIX API. We use object oriented inheritance to represent such implications. Consequently, our dataset contains a hierarchy of targets as depicted in Fig. 1.

We experienced that most targets are able to detect themselves through indicating commands (C_i), Ansible modules (M_i), packages to install (P_i) or Facts gathered from hosts (F_i). As illustrated in equation (1), the default implementation considers a target detected if any of the indicators found (C_f, M_f, P_f, F_f) is present. This enables defining most targets with little effort. If a more advanced detection is required, the default can be overridden and optionally be reused selectively.

$$detected \quad \Leftrightarrow \quad \bigvee_{X \in \{C,M,P,F\}} (X_i \cap X_f \neq \emptyset) \tag{1}$$

Every detected target determines implied targets by traversing up the class hierarchy. The support for recursive traversal, taking multiple inheritance into account, allows for an even more efficient and flexible notation of targets.

C. Matching Targets with Injectors

Once all detected and recursively implied targets are known, FIAD is able to determine which injector is applicable to which infrastructure (hence a *match*). In analogy to the detection of targets, we observed that the majority of injectors share logic concerning their required targets. Again, a default behavior to determine matches is provided. It enables the definition of injectors with little effort and can be overridden and reused selectively. To use the default implementation, an injector must define a set T_{all} of targets that are *all* required to be detected, or a set T_{any} of targets of which *any* is required to be detected, or both. If both sets are provided, requirements regarding T_{any} *and* T_{all} must be fulfilled to yield a match. Equation (2) summarizes this default implementation.

$$\begin{aligned} match \quad \Leftrightarrow \quad & (\ T_{detected} \supseteq T_{all} \ ) \quad \wedge \\ & (\ (T_{any} = \emptyset) \ \vee \ (T_{detected} \cap T_{any} \neq \emptyset) \ ) \end{aligned} \tag{2}$$

V. Example

The two following listings exemplify a fraction of the dataset of FIAD and also demonstrate the characteristically concise notation of injectors and targets. Listing 1 shows FIAD's model representing a Java Runtime Environment (JRE), Listing 2 the model representing the FATE injector [14]. Since a JRE is present when it is installed, the *JavaTarget* defines *INDICATING_PACKAGES_ANY* accordingly. In analogy, since FATE targets distributed systems written in Java, the *FATEInjector* defines *REQUIRED_TARGETS_ALL* and *REQUIRED_TARGETS_ANY* accordingly.

```
class JavaTarget(LanguageTarget):
  """
  Represents a Java Runtime Environment.
  """
  INDICATING_PACKAGES_ANY = {
      re_compile_ci(r'^jdk($|[-_][^\s]*$)'),
      re_compile_ci(r'^java($|[-_][^\s]*$)'),
  }
  # ...
```

Listing 1. The model *JavaTarget* defines a list of (regular expressions to detect) packages that indicate a JRE's presence.

```
class FATEInjector(AbstractBaseInjector):
  """
  FATE aims at a high coverage of failure scenarios
  in cloud systems, including multi-fault ones. ...
  """
  FRIENDLY_NAME = "FATE"
  PUBLICATION_BIBTEX = # ...
  WEBSITE_URL = # ...
  REQUIRED_TARGETS_ALL = {JavaTarget}
  REQUIRED_TARGETS_ANY = {EC2VMTarget,
                          AzureVMTarget, ...}
  # ...
```

Listing 2. The model of the FATE fault injector. FATE is potentially applicable if for example virtual machines in cloud environments and a JRE are detected.

We further assume actors of a software project wanting to assess the dependability of their product. To find applicable injectors effortlessly, they run FIAD on the Playbooks that exist to deploy the product. FIAD would then, for instance, find an Ansible task in which a JRE is installed as depicted in Listing 3. In accordance to the algorithm described in Section IV, the *JavaTarget* is detected and assuming the *EC2VMTarget* is detected as well, FATE emerges as a potentially promising injector. Besides suggesting FATE, FIAD can provide additional information, such as the URL of the injector's Web site or the scientific publication which described it initially.

```
- name: Install Java 1.7
  yum: name=java-1.7.0-openjdk state=present
```

Listing 3. An Ansible task which installs a JRE.

VI. Conclusion and Outlook

The presented tool FIAD provides a knowledge base to explore applicable injectors. By requiring no pre-existing experiences and by being automated, it can be integrated in software development processes and thereby raise the awareness of SFI.

The underlying model for cataloging injectors including their requirements, serves as an extensible framework and allowed us to efficiently define them as a uniform dataset. We are currently completing the dataset and hope that maintaining and updating it becomes an open source community effort. Research-wise, the dataset and its model may yield findings on how to formally classify injectors. Reusing information provided by IaC solutions for the determination of applicable injectors proved to be a powerful approach. However, further research is necessary to fully understand its versatility and possible limitations. For this purpose, an in-depth case study on OpenStack [24] is planned. By adding a new layer of abstraction, other IaC solutions could be additionally integrated, what would allow for recommendations of increased quality.

The inclusion of fault models and workloads is another compelling future extension. Filtering by injectors' fault models would help users to focus on assessing specific aspects of infrastructures' dependability. Through FIAD's capability of interacting with hosts, it could automatically apply injection campaigns and possibly even analyze their effects.

Acknowledgement & Disclaimer

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866. This paper reflects only the authors' views. The European Commission is not responsible for any use that may be made of the information it contains.

References

[1] N. R. Council, D. Jackson, and M. Thomas, Software for Dependable Systems: Sufficient Evidence? Washington, DC, USA: National Academy Press, 2007.

[2] H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins, "Failure as a service (faas): A cloud service for large-scale, online failure drills," University of California, Berkeley, Berkeley, vol. 3, 2011.

[3] H. Madeira and P. Koopman, "Dependability benchmarking: making choices in an n-dimensional problem space," in First Workshop on Evaluating and Architecting System Dependability (EASY), 2001.

[4] M. Hüttermann, Infrastructure as Code. Berkeley, CA: Apress, 2012, pp. 135–156.

[5] Red Hat Inc., "Ansible is Simple IT Automation," visited on 2017-02-12. [Online]. Available: https://www.ansible.com

[6] RightScale, Inc., "2016 state of the cloud report," RightScale, Inc., Tech. Rep., 2016.

[7] C. C. Evans, "The Official YAML Web Site," visited on 2017-02-12. [Online]. Available: http://yaml.org/

[8] S. Ziouani, "Enterprise companies welcoming Ansible IT automation," https://www.ansible.com/blog/enterprise-ansible, Oct. 2014, visited on 2017-01-04.

[9] OpenStack, "OpenStackAnsible," visited on 2017-02-12. [Online]. Available: https://wiki.openstack.org/wiki/OpenStackAnsible

[10] J.-C. Laprie, "Dependable computing and fault-tolerance," Digest of Papers FTCS-15, 1985, pp. 2–11.

[11] A. Tseitlin, "The antifragile organization," Commun. ACM, vol. 56, no. 8, 2013, pp. 40–44.

[12] T. Limoncelli, J. Robbins, K. Krishnan, and J. Allspaw, "Resilience engineering: learning to embrace failure," Commun. ACM, vol. 55, no. 11, 2012, pp. 40–47.

[13] J. Allspaw, "Fault injection in production," Commun. ACM, vol. 55, no. 10, 2012, pp. 48–52.

[14] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," Computers, IEEE Transactions on, vol. 39, no. 4, 1990, pp. 575–582.

[15] J. Gray, "Why do computers stop and what can be done about it?" in Symposium on reliability in distributed software and database systems. Los Angeles, CA, USA, 1986, pp. 3–12.

[16] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in USENIX symposium on internet technologies and systems, vol. 67. Seattle, WA, 2003.

[17] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," ACM Computing Surveys (CSUR), vol. 48, no. 3, 2016, p. 44.

[18] A. Pretschner, D. Holling, R. Eschbach, and M. Gemmar, "A generic fault model for quality assurance," in Model-Driven Engineering Languages and Systems. Springer, 2013, pp. 87–103.

[19] G. J. Carrette, "CRASHME: Random input testing," http://people.delphiforums.com/gjc/crashme.html, 1996, visited on 2017-01-04.

[20] D. Jones, "Trinity : A Linux system call fuzzer." visited on 2017-02-12. [Online]. Available: http://codemonkey.org.uk/projects/trinity/

[21] A. Orso and G. Rothermel, "Software testing: A research travelogue (2000–2014)," in Proceedings of the on Future of Software Engineering, ser. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 117–132.

[22] Red Hat Inc., "Variables – Ansible Documentation," visited on 2017-02-12. [Online]. Available: https://docs.ansible.com/ansible/playbooks_variables.html#information-discovered-from-systems-facts

[23] H. Ziade, R. A. Ayoubi, R. Velazco et al., "A survey on fault injection techniques," Int. Arab J. Inf. Technol., vol. 1, no. 2, 2004, pp. 171–186.

[24] OpenStack, "Open source software for creating private and public clouds." visited on 2017-02-12. [Online]. Available: https://www.openstack.org/

 ISBN: 978-1-61208-528-9