Tool Support for Fuzz Testing of Component-Based System Adaptation Policies

Jean-François Weber

FEMTO-ST UMR 6174 CNRS and Univ. Bourgogne Franche-Comté, Besançon, France
jfweber@femto-st.fr

Abstract. Self-adaptation enables component-based systems to evolve by means of dynamic reconfigurations that can modify their architecture and/or behaviour at runtime. In this context, we use adaptation policies to trigger reconfigurations that must only happen in suitable circumstances, thus avoiding unwanted behaviours. A tool (*cbsdr*, standing for Component-Based System Dynamic Reconfigurations) supporting both the Fractal and FraSCAti component frameworks was developed, but the testing of the robustness of new adaptation policies was not easy. This is the reason to add to our implementation a new behavioural fuzzing tool. While fuzzing consists of sending invalid data to a system under test to find weaknesses that may cause a crash or an abnormal reaction, behavioural fuzzing sends invalid sequences of valid data. Valid traces are modified using fuzzing techniques to generate test cases that can be replayed on a dummy system using the adaptation policies to be tested while focusing on interesting regions of specific data sequences.

1 Introduction

Component-based systems can evolve at runtime using dynamic reconfigurations that can modify their architecture and/or behaviour. A tool (*cbsdr*[1,2], standing for Component-Based System Dynamic Reconfigurations) supporting both the Fractal [3] and FraSCAti [4] component frameworks was developed. This tool uses adaptation policies based on temporal logic to trigger reconfigurations while enforcing some temporal properties; this means that a specific reconfiguration would only be performed if it does not make the system evolve in a configuration that may violate the properties to be enforced. Reflection polices that would generate a reaction when some properties are violated are also part of this tool. In a nutshell, adaptation policies prevent anything bad to happen at the next configuration, whereas reflection policies trigger a pertinent response when something bad has already happened. Nevertheless, the testing of the robustness of new adaptation policies is complicated and time consuming, especially for large systems that would require tailored settings to test specific policies.

Fuzz testing, or fuzzing [5], is a software testing technique that aims at discovering weaknesses by inputting massive amounts of data (often random and/or invalid). Behavioural fuzzing sends (invalid) sequences of valid data. These sequences can either be generated from a model, like in [6], or by re-engineering the result of a previous run of the system, namely its log files. By using specificities of our reconfiguration model to generate the data to be injected, we allow the tester to focus on specific regions of the sequence that would enable adaptation policies to be tested.

We will briefly introduce the *cbsdr* project in Sect. 2 before presenting the way we tackle the problem of the test of adaptation policies in Sect. 3. Finally, Section 4 presents our conclusion and future work.

2 The *cbsdr* Project

We developed a prototype tool, contained in a java package named *cbsdr*, supporting our reconfiguration model to run component-based systems with dynamic reconfigurations. Using generic java classes, independent of any component-based system framework, we can use our implementation to perform reconfigurations on applications deployed using Fractal [3] or FraSCAti [4]. The Fractal framework is based on a hierarchical and reflective component model. Its goal is to reduce the development, deployment, and maintenance costs of software systems in general[1]. FraSCAti is an open-source implementation of the *Service Component Architecture*[2] (SCA). It can be seen as a framework having a Fractal base with an extra layer implementing SCA specifications. In [4], a smart home scenario illustrates the capabilities and the various reconfigurations of the FraSCAti platform.

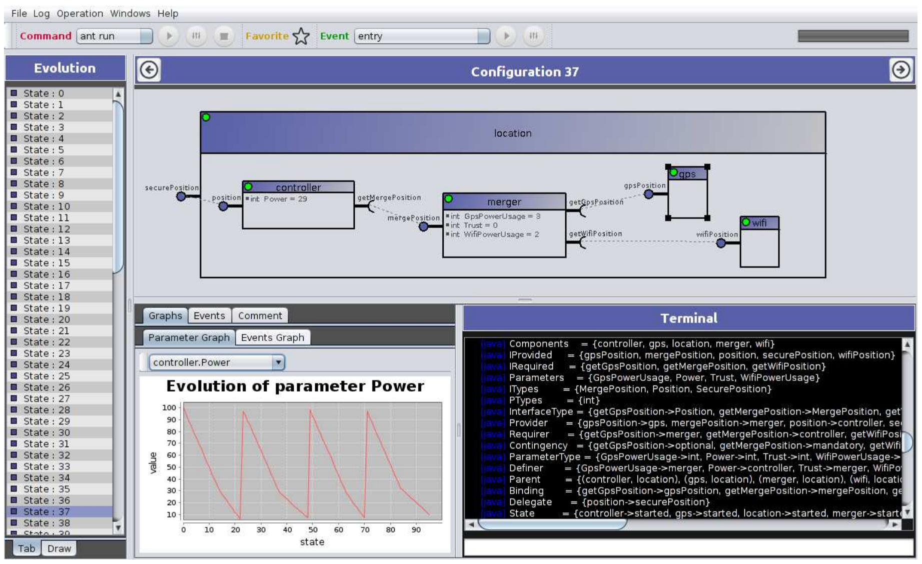
Figure 1 shows the *cbsdr* interface displaying a given state of a component-based system developed using Fractal (top frame). The left frame shows the various states of the run under scrutiny, whereas the bottom frame can be used to display various information such as the evolution of parameters of the model, console output, or the outcome of reconfigurations performed.

This interface allows the monitoring of a component-based system and the generation of (external) events during a run of *cbsdr*, but can also be used to analyse the logs of a run already performed.

In addition to the above-mentioned functionalities, adaptation is performed using reconfigurations triggered by temporal properties at runtime, as described in [1]. This works as follows: *a*) adaptation polices are loaded and applied using a control loop, *b*) temporal properties are evaluated and candidate reconfigurations (if any) are ordered by priority using fuzzy logic values embedded in adaptation policies, *c*) these reconfigurations are applied to the component-based system model using our reconfiguration semantics to verify that corresponding target configurations do not violate any of the properties to enforce, and *d*) the target configuration obtained using the reconfiguration with highest priority that does not violate any of the properties to enforce is applied to the component-based system using a protocol similar to the one described in [7].

The test and implementation of adaptation policies being feasible for small systems can become complex and time consuming for larger ones, and may require specific settings to put the system in the conditions enabling such policies.

[1] http://fractal.ow2.org/tutorial/index.html
[2] http://www.oasis-opencsa.org/sca



```
[java] Components    = {controller, gps, location, merger, wifi}
[java] IProvided     = {gpsPosition, mergePosition, position, securePosition, wifiPosition}
[java] IRequired     = {getGpsPosition, getMergePosition, getWifiPosition}
[java] Parameters    = {GpsPowerUsage, Power, Trust, WifiPowerUsage}
[java] ITypes        = {MergePosition, Position, SecurePosition}
[java] PTypes        = {int}
[java] InterfaceType = {getGpsPosition->Position, getMergePosition->MergePosition, get
[java] Provider      = {gpsPosition->gps, mergePosition->merger, position->controller, se
[java] Requirer      = {getGpsPosition->merger, getMergePosition->controller, getWifiPosi
[java] Contingency   = {getGpsPosition->optional, getMergePosition->mandatory, getWifi
[java] ParameterType = {GpsPowerUsage->int, Power->int, Trust->int, WifiPowerUsage->
[java] Definer       = {GpsPowerUsage->merger, Power->controller, Trust->merger, WifiPo
[java] Parent        = {(controller, location), (gps, location), (merger, location), (wifi, locati
[java] Binding       = {getGpsPosition->gpsPosition, getMergePosition->mergePosition, ge
[java] Delegate      = {position->securePosition}
[java] State         = {controller->started, gps->started, location->started, merger->start
```

Fig. 1. Model of a component-based system displayed in our interface

3 Fuzz Testing of Adaptation Policies

Fuzz testing or fuzzing [5] is a software testing technique used to discover coding errors and security loopholes in software, operating systems or networks by inputting massive amounts of (random) data, called *fuzz*, to the system in an attempt to make it crash or at least misbehave. Behavioural fuzzing sends invalid sequences of valid data. These sequences can either be generated from a model, like in [6], or by re-engineering the result of a previous run of the system, namely its log files. Since these tests are not performed during but after the run of the system, they consists of *offline* fuzzing, instead of *online* fuzzing that would be performed at runtime.

We chose to use the best of both approaches (model-based and trace-based fuzz generation) by using specificities of our reconfiguration model to generate the fuzz to be injected. In a nutshell, our reconfiguration model is based on configurations that can be seen as a tuple $\langle Elem, Rel \rangle$, where $Elem$ is made of architectural sets containing elements such as components, (required of provided) interfaces, parameters, etc. and Rel contains relations linking architectural elements, e.g., interfaces binding or wiring, components states (*started* or *stopped*), parameters values, etc. We also use a set CP of configuration properties on the architectural elements and the relations between them. These properties are specified using first-order logic formulae [8]. Therefore, the operational semantics of a component-based system is defined by the labelled transition system $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ where $\mathcal{C} = \{c, c_1, c_2, \dots\}$ is a set of configurations, $\mathcal{C}^0 \subseteq \mathcal{C}$ is a set of initial configurations, $\mathcal{R}_{run}$ is a set of reconfigurations, $\rightarrow \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ is the reconfiguration relation, and $l : \mathcal{C} \rightarrow CP$ is a total interpretation function.

The *cbsdr* tool contains controllers using control loops to monitor the evolution of a component-based system under scrutiny by regularly retrieving its configuration. The sequence of all the configurations retrieved during a run constitute a trace that can be modified either manually for the generation of very specific test cases, or automatically for bulk generation of test cases using random shuffling, duplication, and/or deletion of configurations. Such tests cases (called below *fuzzy logs*) are obtained by transformations that can be automated using a sub-package of *cbsdr* called *cbsdr.fuzzy* and referred below as *Fuzzy Engine*.

The Fuzzy Engine tool is integrated in the *cbsdr* development as shows Fig.2 where light coloured entities are part of the previous developments and the elements of the fuzzing tool are represented in darker colours.

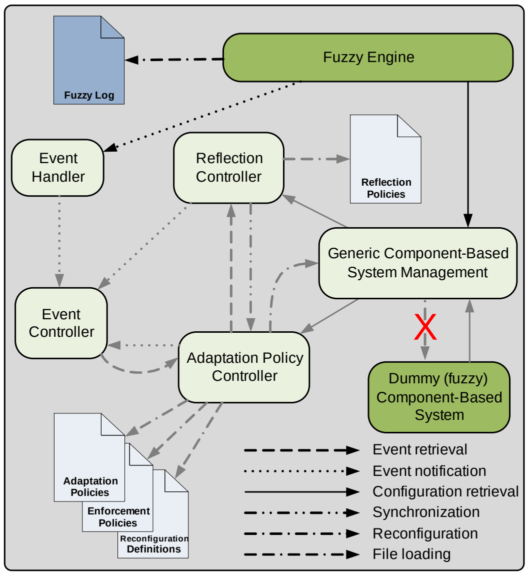


Fig. 2. *cbsdr* Fuzzing Architecture

This informal representation of our implementation displays three controllers: *a*) the *event controller* receives events, stores them, and flushes them after they have been sent to a requester, *b*) the *reflection controller* sends events to the *event controller* when a property of a reflection policy is violated, and *c*) the *adaptation policy controller* manages dynamics reconfigurations triggered by adaptation policies. The reader interested by the interactions between these controllers is referred to [1].

In addition, an *event handler* is used to receive events from an external source and to send them to the *event controller*. All interactions with the *component-based system* take place through the *generic component-based system management* entity (*gcbsm*), a set of Java classes developed in such a way that they can be used regardless of the framework used to design the component-based system without modifying its code.

The *gcbsm* is mainly developed using abstract classes that are used for the reification of other classes specially designed for the handling of Fractal [3] or FraSCAti [4] component frameworks. We just added to the *gcbsm* support for another new component framework that we called *dummy*. This way, each time the *adaptation policy controller* or the *reflection controller* requests the current configuration, the *gcbsm*, when detecting a dummy component, requests the corresponding configuration to the Fuzzy Engine instead of retrieving it from an actual component-based system. Of course, the Fuzzy Engine must always be initialized with a fuzzy log corresponding to the pertinent test case before usage.

We can automatically filter (or put aside for further examination) test cases with an influence on an adaptation policy under test (APUT) by giving unique names to reconfigurations triggered by the APUT. It is also possible to add an additional reflection policy that stops the system (or take any other suitable action) for each success or failure of a reconfiguration triggered by the APUT.

This way, our tool, which can be launched using the interface of Fig. 1, takes fuzzy logs as input to simulate the run of a component-based system using a dummy system. The output consists of a set of trace files containing a subset of traces involving reconfigurations triggered by the APUT. Such traces can be displayed using our interface to verify that the APUT behaves as intended.

As example, we can consider, as in the case study of [1], a component-based system in charge of the location of an autonomous vehicle. To ensure reliability, the position must be computed by using different techniques such as Wi-Fi or GPS signals. When the power level of the vehicle decreases, it may be suitable to remove, for example, the GPS software component to save energy, as long as the other positioning systems keep providing accurate positions. Of course as the batteries can be recharged, when the power level rise above a certain value, the GPS component can be added back using the *addgps* reconfiguration operation. Such a reconfiguration is triggered by an adaptation policy responsible for the management of the GPS component. This policy, among other things, must take into account the low utility of adding back the GPS component to the system when the vehicle is in a tunnel where there is no GPS signal.

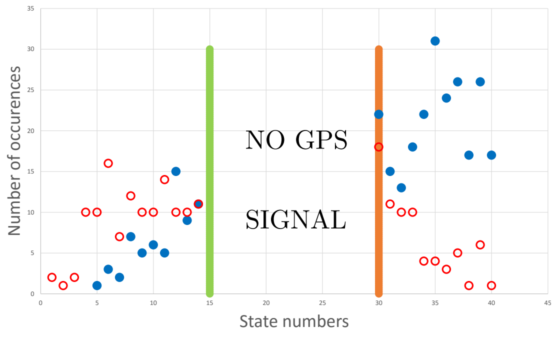


Fig. 3. Occurrences of the *addgps* reconfiguration

Starting from a trace of a run of the system, we generated with our fuzz test tool 1000 test cases that were used to run the GPS adaptation policy with a dummy component-based system. Among these tests, 203 were selected because they were involving the *addgps* reconfiguration operation. The results are summarised in Fig. 3, where the horizontal and vertical axes represent respectively the states number increasing over time and the cumulated number of occurrences of the *addgps* reconfiguration for each state. Vertical lines symbolise the entrance and exit of a tunnel where there is no GPS signal, plain blue dots represent the successful application of the *addgps* reconfiguration, and hollow red dots show that its application failed[3].

These tests show that none of the *addgps* reconfiguration operations were attempted inside the tunnel where there is no GPS signal, which is the way the GPS adaptation policy was supposed to behave.

[3] Because of the random nature of fuzzing, the configuration following the application the *addgps* reconfiguration may not contain a fully functional GPS component, which leads the reconfiguration to be diagnosed as failed.

Finally, fuzzing makes the test and implementation of adaptation policies easier by allowing the tester to focus on specific regions of the sequence of configurations that would enable these policies. Also, as an interesting secondary benefit, in the early stages of development of the Fuzzy Engine tool, by running fuzz testing against some adaptation polices, we were able to identify and correct several bugs in the *cbsdr* implementation.

4 Conclusion and Future Work

The work presented in [1,2] enables component-based systems dynamic reconfigurations guided by adaptation policies. Whereas the test and implementation of these policies were possible for small systems, this was complicated and time consuming for larger systems as specific settings were required in order to put the system in the conditions that would enable such policies. The usage of fuzzing makes such tests easier by allowing the tester to focus on specific regions of the sequence of configurations that would enable these policies.

As a future work, we are planning to perform more evaluations on various case studies. We are also contemplating the possibility to integrate online fuzzing, as in [9], to the *cbsdr* project. To do so, we would use *fuzzy policies* to generate test cases at runtime, focusing on interesting regions of specific data sequences.

References

1. Kouchnarenko, O., Weber, J.F.: Adapting component-based systems at runtime via policies with temporal patterns. In Fiadeiro, J.L., Liu, Z., Xue, J., eds.: FACS'13. Volume 8348 of LNCS. Springer (2014) 234–253
2. Kouchnarenko, O., Weber, J.F.: Practical analysis framework for component systems with dynamic reconfigurations. In Butler, M., Conchon, S., Zaïdi, F., eds.: ICFEM'15. Volume 9407. Springer (2015) 287–303
3. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java. Software: Practice and Experience **36** (2006) 1257–1284
4. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A component-based middleware platform for reconfigurable service-oriented architectures. Software: Practice and Experience **42** (2012) 559–583
5. Takanen, A., Demott, J.D., Miller, C.: Fuzzing for software security testing and quality assurance. Artech House (2008)
6. Schneider, M., Großmann, J., Tcholtchev, N., Schieferdecker, I., Pietschker, A.: Behavioral fuzzing operators for uml sequence diagrams. In: International Workshop on System Analysis and Modeling, Springer (2012) 88–104
7. Boyer, F., Gruber, F., Pous, D.: Robust reconfigurations of component assemblies. In: Int. Conf. on Software Engineering, ICSE '13, Piscataway, NJ, USA, IEEE Press (2013) 13–22
8. Hamilton, A.G.: Logic for mathematicians. Cambridge University Press (1988)
9. Schneider, M., Großmann, J., Schieferdecker, I., Pietschker, A.: Online model-based behavioral fuzzing. In: Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, IEEE (2013) 469–475