Tracing Structural Changes of Adaptive Systems

Kai Nehring
AG Software Engineering: Dependability
University of Kaiserslautern
Kaiserslautern, Germany
Email: nehring@cs.uni-kl.de

Peter Liggesmeyer
AG Software Engineering: Dependability
University of Kaiserslautern
Kaiserslautern, Germany
liggesmeyer@cs.uni-kl.de

***Abstract*—The internal structure of an adaptive system is subject to change. An unforeseen and unwanted component composition can cause serious malfunction in a system, which can be difficult to track. Knowledge of the system's internal structure at runtime helps to understand the system and, in the end, helps to test a system more thoroughly. In this paper, we present the provisional result of our ongoing research on an approach that tests the reconfiguration of adaptive systems.**

***Keywords*-adaptive system; BTrace; component composition; internal structure; state tracking**

I. Introduction

In a demanding world where systems not only have to be available incessantly but also have to adapt to changing environments and/or requirements, (dynamically) adaptive systems gain popularity. Testing such systems is still a challenging task. Several approaches have been published over the years, such as [1][2][3] to name a few. Most approaches test the functionality of an adaptable system but not the quality requirements, such as elapse time of the adaptation or the states which occur during the reconfiguration.

Other approaches take some of these requirements into account [4][5] but often require specialised knowledge and complicated system descriptions, often in terms of temporal logic (A-LTL) [6][7] which can be a challenging task, too.

In our work, we focalise on a test approach for quality requirements of adaptive systems that uses information which are already available, such as component diagrams, and a methodology which does not need in-depth knowledge beyond that of an average system developer.

Our current work is concerned with the internal structure of a system and the representation of changes which occur at runtime. These information can be used to gain knowledge of the system, e.g., the individual states of an application during reconfiguration. Furthermore, these data can be used to evaluate the architecture by applying metrics [8] and eventually to improve the system's design.

In Section II, we present a general approach to track structural changes. Section III introduces the toolset we use to track changes in Java applications. We also applied the suggested approach to a demonstration system and present the result of that experiment in Section IV. Section V gives an overview of the future work.

II. Tracing structural changes

A structural change appears if one component is replaced with another. Several steps are usually necessary to replace a component. At least all references which point to that component have to be reset in order to point to the replacement. This can require many operations, depending on the number of references which have to be updated. Often a dedicated controller is responsible to reconfigure the system, e.g., a component manager or a configuration manager. Various information can be of interest, depending on the point of view, such as:

- How long did the reconfiguration take?
- What components are currently (or previously) connected with the component of interest?
- What configurations passed the system through the reconfiguration?

The reconfiguration elapse time can be of great importance whenever time constrains have to be fulfilled, e.g., the reconfiguration elapse time for an ordering system must not exceed 1500 ms. Since the reconfiguration of a system is usually done within a set of methods, the elapse time of these methods have to be recorded. During application tracing, the time stamp of both the entry-point and return-point of a method have to be recorded in order to calculate the elapse time. Depending on the structure of the system, multiple methods might have to be traced and elapse times have to be summarised.

To answer the second question, an approach is necessary that allows to add probes into the runtime system which fire whenever an attribute of interest will be changed, e.g., references to components which are likely to be replaced at runtime. It is important to observe all components which use these replaceable components in order to track configuration changes. A recorded change (delta set) comprises the

- source of the change, i.e., name of the component and its identification hash code
- attribute that held the reference and then holds the new reference after the modification
- target reference, i.e., name of the new component and its identification hash code
- time of occurrence when the change took place

A probe is a piece of code that will be executed whenever a certain event occurs. This can be

1) a method in the original code, i.e., additional code in a set-Method,
2) an aspect that has been woven around an attribute/method or
3) code that has been injected into the runtime system.

The probe has to log the original value (before-value) and the new value (after-value). The change in value can be used to track configuration changes: a reference of a component now points to a different component; therefore, the configuration has changed.

That leads to the answer to the third question. To identify and eventually analyse the different states during the reconfiguration, all delta sets have to be applied on the initial state (before the reconfiguration) of the system in order of occurrence, i.e., the final state of the system is built upon the initial state and the delta sets. Based on that, evaluation methods can be applied to find illegal states, e.g., to mark all occurrences of components which hold a reference to an outdated component.

The previously mentioned approach requires the state of the system right before the reconfiguration takes place, i.e., the initial state. Although it is possible to trace an application right from the start, it is often impossible or at least very difficult, especially if an application shall be observed under real-life conditions. The initial state can be calculated, too, by using the before-values of the recorded changes. A before-value correlates with the association to another component before it is going to change. Both the association and the associated component must be preserved in order to calculate the initial state. By iterating over all changes, the state will be built up. Attributes with value `null`, however, must be ignored. Components that are newly created usually have all attributes set to `null`. In that case, the whole component has to be ignored since it was not part of the composition before. The remaining connections among components have to be summed up in order to create the previous state right before the first change. This state, however, does not include components, which hold references to one of the displayed components, that do not anticipate in the reconfiguration process, i.e. the state might be incomplete.

III. Recording changes with BTrace

We use BTrace [9] to trace changes at runtime. BTrace allows to insert probes into the Java Virtual Machine and therefore to observe applications even if their source code is not available by using the Java Instrumentation API. Applications do not need to be modified or recompiled in any way.

BTrace offers a set of probes which fire on certain events, such as entry or return of a method or whenever an observed attribute will be modified. It also allows to record the values of attributes before and after the manipulation. A probe typically consists of an action method which should be executed if a certain event occurs and an annotation which tells BTrace when to fire.

The action method is similar to a Java method but its functionality is reduced to a minimum. It is not possible to overwrite attributes of the application nor to create new objects. BTrace also prohibits loops and some long running operations. It is basically an observation tool designed for minimal impact.

The annotation of an action method tells BTrace what to observe, when to fire and what information to collect. The following annotation causes BTrace to fire whenever the method reconfig in class pwgen.Main is executed. An action method annotated that way would be executed right before (Kind.ENTRY) the execution of the reconfig-method.

```
@OnMethod(clazz="pwgen.Main",
   method="reconfig",
   location=@Location(Kind.ENTRY))
```

The annotation that is necessary to trace field manipulation is insignificantly more complex. Please note, that a wildcard is used to declare the method tag since all manipulation attempts should be tracked regardless which method causes the manipulation of an observed attribute.

```
@OnMethod(clazz="pwgen.Main",
   method="/.*/",
   location=@Location(
      value=Kind.FIELD_SET,
      clazz="pwgen.Main",
      field="pwGen",
      where=Where.BEFORE))
```

A special compiler compiles the script and BTrace will inject the resulting class into the JavaVM for a defined Java process. It is therefore possible to observe several Java processes in parallel, each with a different script. BTrace currently exists as a console application and as an add-on for VisualVM [10]. Latter offers an easy to use graphical user interface.

IV. Trace of a Demo-System

We applied the approach on a demonstration system which is used to teach adaptive behaviour in software. Following the procedure, we have found a so far unknown defect in one of the software components which could cause a failure.

The system under test was a password generator which is comprised of the following components:

SRG: A SimpleRandomGenerator that produces random numbers. It can be connected to only one client at a time.

PwGen2: Password generator with reduced character set (literal only). For further speed optimisations, the length of the password is limited to 6 characters.

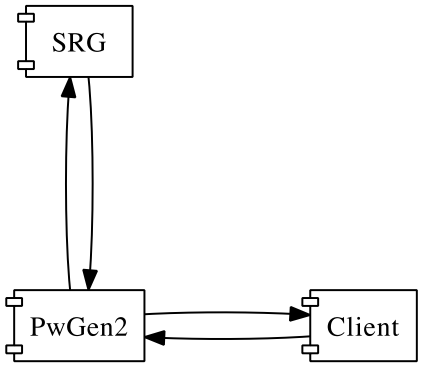

Figure 1. Configuration 1 (initial state): fast, but only suitable for weak passwords

PwGen1: Enhanced password generator that uses an extended character set (literal, digit and special character) and offers a more liberal limitation of the password length. The enhanced feature set results in lower execution speed.

Client: The client uses a password generator, either PwGen2 (default) or PwGen1 (if strong passwords are requested).

Please note that each component comprises its own component manager, not shown in the figures. The component manager sets up each component if necessary and connects it with required services, e.g., the component manager of PwGen2 requests the service of SRG and injects its reference into PwGen2 if SRG is available. It also releases SRG if its service is no longer needed.

Method calls are performed asynchronous. Each component therefore requires not only a reference to a service provider but also needs to inject its own reference to the provider in order to receive the provided service, depicted in Figure 1.

The requirements on the password strength changes at runtime. The generator reconfigures itself to use PwGen1 instead of PwGen2. Several steps are necessary to replace this component:

1) Client: remove client reference in PwGen2 that points to the Client component
2) PwGen2: release SRG, i.e., the client reference of SRG that points to PwGen2 has to be removed
3) PwGen2: release reference to SRG
4) PwGen2: release reference to Client
5) Client: establish connection to PwGen1 and inject reference to Client
6) PwGen1: establish connection to SRG and inject reference to PwGen1
7) SRG: establish connection to PwGen1 and inject reference
8) PwGen1 (now fully functional): inject reference into Client

In the end, the composition of components should look

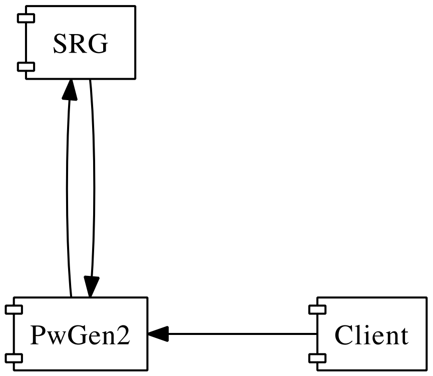

Figure 2. First step of the reconfiguration: Connection between PwGen2 and Client was removed

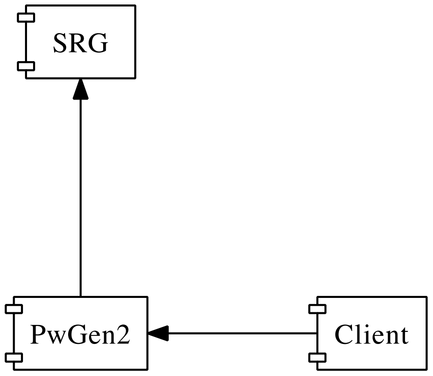

Figure 3. PwGen2 releases SRG; SRG can now be used by another component

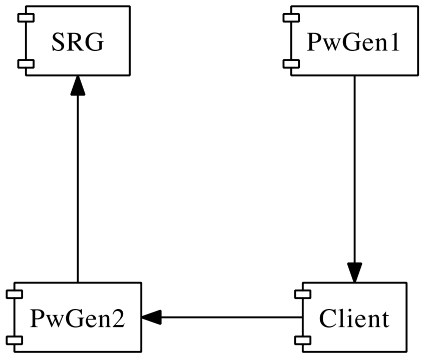

Figure 4. Client requests service of PwGen1 and injects its reference

like Figure 1 but with PwGen1 in place of PwGen2.

We added probes to trace important events, such as changes in component connections and ran the demo application. We then analysed the trace result and generated a visual representation of the component composition before, during and after the reconfiguration, depicted in Figure 2 to Figure 7.

The result of the reconfiguration, depicted in Figure 7,

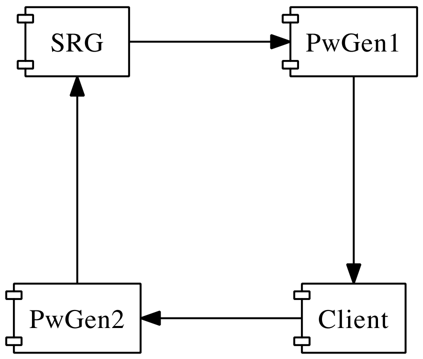

Figure 5. PwGen1 requests service of SRG and injects its reference

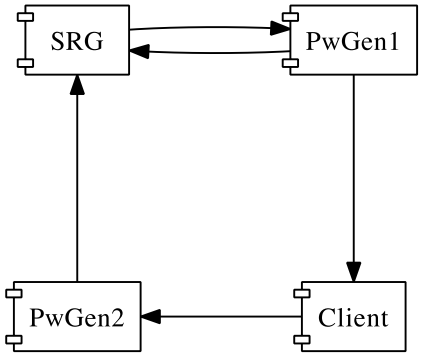

Figure 6. SRG accepts PwGen1's service request and injects its reference to PwGen1

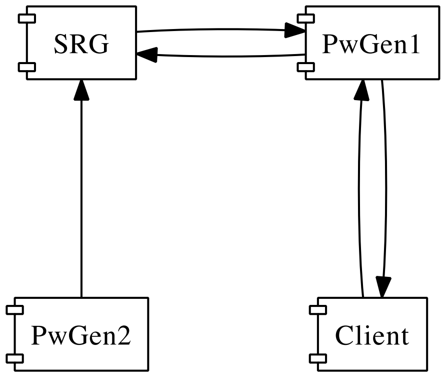

Figure 7. Configuration 2 (final state): resulting composition after reconfiguration (PwGen1 has injected its reference to Client)

derives from the expected result. It turned out that the component manager of PwGen2 did not release the reference to SRG. The resulting composition can cause serious malfunction if PwGen2 is used accidentally. In that case, PwGen2 would request a new random number from SRG. The request would not fail since the reference to SRG is still valid. SRG would generate a new random number but would send it to PwGen1 since its client reference points to PwGen1, which doesn't expect a new number. The impact of this event depends on the implementation of PwGen1. In our case, it added a new character to the current password and, if the minimum password length was (already) reached, sends the password to the Client. The client overwrote the old password with the new one although it didn't request that password.

V. Conclusion and Future Work

As the demonstration shows, serious malfunctions can appear due to erroneous composition of components. Misbehaviour can be difficult to track because its origin was a component that ought to be not active anymore. Knowledge of the actual component composition can help to understand certain failures and can help to identify defects. Furthermore, it can help to create a new class of test cases to test a system more thoroughly.

In future work, we will define metrics and a methodology to determine whether a reconfiguration process satisfies given quality requirements, such as reconfiguration elapse time or latency caused by the adaptation. Target of our research will be the adaptation process itself.

References

[1] X. Bai, Y. Chen, and Z. Shao. Adaptive web services testing. *Computer Software and Applications Conference, Annual International*, 2:233–236, 2007.

[2] J. Grundy, G. Ding, and G. Ding. Automatic validation of deployed j2ee components using aspects. In *In Proc. 2002 IEEE International Conference on Automated Software Engineering*, pages 47–58. IEEE CS Press, 2001.

[3] component+ Partners. Built-in testing for component-based development. *EC IST 5th Framework Project IST-1999-20162 Component+*, Technical Report D3, 2001.

[4] H. J. Goldsby, B. H. Cheng, and J. Zhang. Amoeba-rt: Runtime verification of adaptive software. pages 212–224, 2008.

[5] K. N. Biyani and S. S. Kulkarni. Assurance of dynamic adaptation in distributed systems. *J. Parallel Distrib. Comput.*, 68(8):1097–1112, 2008.

[6] J. Zhang and B. H. C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, Volume 79(10):1361–1369, 2006.

[7] J. Zhang, H. J. Goldsby, and B. H. Cheng. Modular verification of dynamically adaptive systems. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 161–172, New York, NY, USA, 2009. ACM.

[8] C. Raibulet and L. Masciadri. Evaluation of dynamic adaptivity through metrics: an achievable target? In *WICSA/ECSA*, pages 341–344, 2009.

[9] http://kenai.com/projects/btrace 06.18.2010

[10] https://visualvm.dev.java.net 06.18.2010