A Concept for Dynamic Wiring of Components

Correctness in Dynamic Adaptive Systems

Dirk Niebuhr
Clausthal University of Technology
P.O. Box 1253
38670 Clausthal-Zellerfeld, Germany
dirk.niebuhr@tu-clausthal.de

Andreas Rausch
Clausthal University of Technology
P.O. Box 1253
38670 Clausthal-Zellerfeld, Germany
andreas.rausch@tu-clausthal.de

ABSTRACT

Component-based Systems in our days tend to be more and more dynamic. Due to the increased mobility of devices hosting components, components have to be attached or detached to respectively from a system at runtime. This dynamic adaptation of the system configuration imposes several correctness issues. In general it is not possible to determine a correct system configuration without wiring and executing the system in advance. We will discuss approaches how to improve this situation. Finally we will focus on our favorite approach based on runtime testing.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Software/Program Verification*; D.2.11 [**Software**]: Software Engineering—*Software Architectures*; F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*Specifying and Verifying and Reasoning about Programs*

General Terms

Design, Reliability, Verification

Keywords

Dynamic Adaptive Systems, Reconfiguration, Runtime Testing, Correctness, Component, Adaptation

1. INTRODUCTION

To produce systems out of *IT components* component-based development approaches have been developed and successfully applied over the past years changing the predominant development paradigm: Systems are no longer redeveloped from scratch, but composed of existing components [4, 1]. Nowadays, these IT components are being more and more used within an organically grown, heterogeneous, and dynamic IT environment. Users expect these IT components to collaborate autonomously with each other and provide a real added value to the user. On the other hand, we depend more and more on these organically grown IT systems. Hence their correctness has to be guaranteed even though these systems are never developed and tested in advance. These *dependable adaptive IT systems* need to have the ability to dynamically attach and detach dynamic adaptive IT components during runtime. Moreover they need to detect and avoid possible resulting incorrect system configurations during runtime.

In this paper we present our approach of achieving runtime dependability of these systems by runtime testing. We will sketch the proposed runtime testing approach illustrated by a small example in Section 3.

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

2. DYNAMIC WIRING OF COMPONENTS

Imagine a very simple system containing three components *ComponentA*, *ComponentB*, and *ComponentC*. *ComponentA requires* a component providing *InterfaceA* whereas *ComponentB* respectively *ComponentC provide InterfaceB* respectively *InterfaceC*. Moreover each of the interfaces comes along with it's own specification (t_A, t_B, t_C) of required respectively assured properties. This component landscape is depicted in Figure 1.

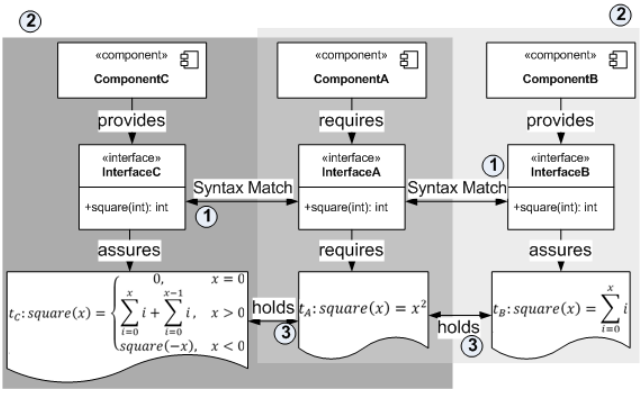

Figure 1: The Artifacts of the Example System

During configuration we have to decide at runtime, whether *holds(prov, req)* is valid for a specific configuration in general, which is depicted in number 3 of Figure 1. This predicate is valid, if the specification of the provided interface implies the specification of the required interface which means proving the refinement relation or showing the equivalence of two Turing machines. This requires the evaluation of implications using second-order logic which refers to the

decision problem. This has been proven to be not decidable by Turing and Church in 1936 [5, 2]. Therefore proving the correctness of a component wiring at runtime in general is not possible. There are several approaches to provide (limited) statements regarding the correctness of the wiring:

1. Using a specification language like regular expressions or finite state machines, which is more restricted and not as powerful in order to get a calculable holds-predicate. In this case you will have to answer the question, whether this specification language is still capable of specifying the desired dynamic adaptive systems respectively you get a less valuable proof in case of an incomplete specification.

2. Checking the correct wiring of components by bisimulation [3]. In this case you would need to compare the states of two simulated system executions *for every system execution step*: one system is containing the requiring component and performs changes to its system state as specified in the required interface specification, the other one is containing the providing component and performs changes to its system state as specified in the provided interface specification. In this case you need to argue, whether the performance of a system using this approach would still be sufficient due to the massive simulation. In addition you only get a proof of correctness for the following execution step.

3. Performing runtime testing during the reconfiguration process: whenever two components should be wired together, test cases are executed within a testbed which check, whether they fit together. In this case you need to show, that the test cases executed during reconfiguration are good enough to expose mismatches of components. Moreover you have to argue, that the testbed is a sufficient representation of the real system environment.

3. RUNTIME TESTING APPROACH

Since we don't want to restrict the specification language and want to retain a good system performance, our approach is using runtime testing in a testbed during the reconfiguration of a system. This reconfiguration may occur, whenever a component appears within a system or a component disappears or fails. In general we use a three-step process. First of all we derive an ordered set of valid system configurations from the set of available components. Then we wire these system configurations within a testbed and check whether all test cases pass. Finally, we transfer this configuration to the production system. We will describe these steps shortly based on an example system.

As you can see, *InterfaceA* and *InterfaceB* respectively *InterfaceA* and *InterfaceC* match syntactically, since they provide syntactically identical methods[1]. This is checked by the syntax match depicted in number 1 of Figure 1. Based on this, two valid system configurations are identified: C_1 wiring *ComponentA* and *ComponentB* and C_2 wiring *ComponentA* and *ComponentC*. The subsets of involved components in these configurations are depicted in number 2 of Figure 1 and are ordered in a way preferring C_1.

We need to test each of these configurations in a testbed. Therefore we duplicate the components, wire the duplicates together and execute test cases. These test cases can be brought by the component user (here: *ComponentA*) since he knows the usage scenarios for the used interface (here: *InterfaceA*) best. They could be provided by the used component as well. A third option would be to generate test cases from the interface specifications (here: t_A, t_B, t_C) of one (or both) of the components. For simplicity we assume, that *ComponentA* provides a test case containing three method calls: $square(0) : 0$, $square(3) : 9$, and $square(-3) : 9$. Within the testbed this test case is executed.

First of all, a duplicate of *ComponentB* is wired together with a duplicate of *ComponentA*. When executing the second method call, the test case fails, since $square(3) = 6$ which contradicts the expected result of 9. Therefore this configuration is marked as invalid. When executing the test cases on the second system configuration, which wires a duplicate of *ComponentA* and a duplicate of *ComponentC*, all test cases pass and therefore this configuration is established in the following. If we want to assure, that the used component behaves as required during execution, we can ensure this as well by additionally checking this after each method call during the system execution in the following. This would cause a large overhead during system execution and therefore may not be applicable for all types of systems. This would correspond to the bisimulation approach.

[1]If components should be wired though their interface methods are not syntactically equal, one could use ontologies.

4. CONCLUSIONS AND FURTHER WORK

Reconfiguration, which means changing the component wiring, is necessary for dynamic adaptive systems since components may enter or leave a system at runtime. However proving the correctness of a component wiring at runtime is not possible in general. Our approach is based on runtime testing component duplicates in a testbed during reconfiguration. This enables us to recognize semantical mismatches of provided and required interfaces at runtime. Therefore we can mark system configurations, wiring these incompatible interfaces, as invalid and chose a valid configuration instead. However we did not take care about cyclic dependencies of components, where an interface provided by *ComponentA* is required by *ComponentB* and vice versa. Moreover we need to investigate test case generation, to enable component developers to provide a single specification of their components and assure good test cases. Moreover we need to check, whether it is sufficient, to execute only test cases involving newly introduced components during reconfiguration.

5. REFERENCES

[1] K. Bergner, A. Rausch, M. Sihling, and A. Vilbig. Putting the parts together – concepts, description techniques, and development process for componentware. In *HICSS 33, Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, Jan 2000.

[2] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[3] E. Estévez and P. R. Fillottrani. Bisimulation for component-based development. *Journal of Computer Science & Technology*, 1(6), May 2002.

[4] C. Szyperski. *Component Software*. Addison Wesley Publishing Company, 2002.

[5] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, pages 230–265, 1936.