Automated Test Driver Generation for *Java* Components

Frank S. de Boer[2] and Marcello M. Bonsangue[1] and
Andreas Grüner[1] and Martin Steffen[3]

[1] LIACS, Leiden, The Netherlands
[2] CWI Amsterdam, The Netherlands
[3] University of Oslo, Norway

Whereas object-orientation is established as a major paradigm for software development, testing methods specifically targeted towards object-oriented, class-based languages are less common. We propose a testing framework for object-oriented programs, based on the *observable trace semantics* of class components, i.e., for black-box testing. In particular, we propose a test specification language which allows to describe the behavior of the component under test in terms of the expected interaction *traces* between the component and the tester. The specification language is tailor-made for the programming language *Java* e.g., in that it reflects the nested call and return structure of thread-based interactions at the interface. From a given trace specification, a testing environment is automatically *synthesized* such that component and environment represent an executable closed program.

The design of the specification language is a careful balance between two goals: using programming constructs in the style of *Java* helps the programmer to specify the interaction without having to learn a completely new specification notation. On the other hand, *additional* expressions not provided in *Java* itself allow to specify the desired trace behavior in a concise, abstract way, hiding the intricacies of the required synchonization code at the lower-level programming language.

In our presentation, we will propose a *test specification language* for describing the desired interface behavior of a component. The language is designed under consideration of the following aspects:

- The behavior is formalized on the basis of interface traces, that is, the sequence of method calls and returns between the component and its environment. The language provides interaction statements for *incoming* (i.e. expected) and *outgoing* (i.e. committed) method calls and returns, which can be nested to describe an expected sequence of component-environment interactions.
- To allow specifications which correspond to possibly infinite sets of traces we add *Java*-like language constructs like variable declarations, conditionals, and while-loops.
- Certainly, there exist sequences of interactions that cannot be realized by any component. The grammar of the language rules out most of these faulty specifications.

Moreover, we will explain, by means of a simple example, how to automatically *generate* Java methods from a test specification. The resulting tester classes drive the test by checking the behavior of the component and, at the same time, by providing the needed stimuli and collaborator objects. Thus, our approach can be considered as a *mock object framework*. In contrast to existing mock object frameworks, which do not provide a test specification language, however, we had to tackle three main problems:

- *control flow:* The code at the Java-level must be contained in bodies of methods, corresponding to the *incoming* method-labels of the trace specification, i.e., the test-code must be appropriately "distributed" over different method bodies and classes. Furthermore, the order of accepting incoming communications and generating outgoing ones must be realized as given by the specification.
- *variable binding:* The parameters of a specification's incoming method call introduce a *scope* that resembles the scope of a method declaration's formal parameter in Java. A direct translation of these scopes to Java is not possible, however, as this would need Java to have dynamic scoping.
- *realizability:* It is impossible to identify all unrealizable specifications, statically. Instead, we detect at runtime if the tester expects an impossible behavior of the component. In these cases, the tester stops the execution and reports a faulty specification.

Andreas Grüner received his *Diplom* (Master's degree) in computer science from the Christian-Albrechts-University in Kiel, Germany. He is currently working on his Ph.D. at the University of Leiden. His scientific interests lie in the domain of semantics of and testing approaches for object-oriented programming languages, compositionality, and concurrency.