Test Driver Generation from Object-Oriented Interaction Traces

Frank S. de Boer[2] and Marcello M. Bonsangue[1] and
Andreas Grüner[3] and Martin Steffen[4]

[1] LIACS, Leiden, The Netherlands
[2] CWI Amsterdam, The Netherlands
[3] Christian-Albrechts-University Kiel, Germany
[4] University of Oslo, Norway

1 Introduction

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 formal 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 object-oriented thread-based programming languages like *Java* and *C#*, 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 *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 the target language helps the programmer to specify the interaction without having to learn a completely new specification notation. On the other hand, *additional* expressions in the specification language which are usually not provided by the target language 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.

2 Test Driver Generation

Target Language We aim at a testing framework with a formal basis for testing software components written in object-oriented languages with synchronous message passing like *Java* or *C#*. A typed concurrent object calculus, based on impς introduced in [1] (see also [3]), and a corresponding open operational semantics for components[2] serve as a formal common ground for these languages. A component of the calculus consists of threads and classes. The semantics is given in two stages. The component-internal steps are defined without reference

to the environment. The external steps, in contrast, describe the component-environment interactions, i.e., interactions between the component and its environment. They are formalized by a labeled transition system, where the label of a transition represents the actual interaction.

Interface behavior A component-environment interaction is either a method call or a method return. A method call occurs at the interface if the callee is part of the component and the caller resides in the environment or vice versa. We call the first kind of method calls *incoming* and the latter *outgoing method calls*. An incoming method call results in an *outgoing return* (in case the method returns). The dual holds for outgoing calls.

The interface behavior of a component can be described by its set of interface traces. An interface, or interaction, trace of a component is the syntactical representation of a sequence of component-environment interactions that occur when executing the operational semantics. They are expressed by the corresponding sequence of transition labels.

Specification Language We propose a 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. To this end, the language provides interactions statements for incoming and outgoing method calls and returns, which can be concatenated to describe an expected sequence of component-environment interactions.
- To allow specifications which correspond to possibly infinite sets of traces we add language constructs like variable declarations, conditionals, and tail recursion. To ease the use of the specification language for software developers, these language constructs are the same as in the target language.
- Certainly, there exist sequences of interactions that cannot be realized by any component. For instance, an incoming return cannot occur before the corresponding outgoing method call. The grammar and the type system of the specification language filters out most of these faulty specifications.

We give a semantics for the specification language which is similar to the semantics of the target language. However, the labeled transition system is different in that the possible incoming interactions are confined by the incoming interaction statements in the specification. Moreover, an important consequence of the above mentioned design decision is that we can specify a sequence of interaction statements of different threads such that the same order in the resulting trace is ensured.

Transformation We propose a transformation algorithm which generates target language code from a test specification for the methods of the tester classes. Basically, we had to tackle three main problems:

– To provide code which checks whether the component realizes an expected trace, we have to determine the possible orders of execution of interaction statements in the specification. We do this, by analyzing the possible control flow of the given specification.
– The tester should drive and observe the test, so that the component-tester interactions are realized in the specified order. However, our underlying theory shows that there always exist re-orderings which cannot be avoided but which are observable equal to the original order anyway[4]. We use a synchronization mechanism that ensures an order which is observable equal to the specified one.
– Unfortunately, we cannot identify all unrealizable specifications, statically. However, we can detect at runtime if in a certain situation the tester expects an impossible behavior of the component. In these cases, the tester stops the execution and reports a faulty specification.

3 Results

Specification language for a concurrent object calculus We formalize a specification language for object-oriented class-based concurrent components which specifies a component's behavior based on its interface trace and which, at the same time, has a similar look-and-feel as the target language. Moreover, we propose a transformation algorithm which synthesizes a tester program from a given specification, such that the tester and the component under test form an executable closed program.

Soundness of the transformation Our full research program also includes proofing soundness of the transformation in a later stage. This comprises

– *preservation of well-typedness* i.e., from a given well-typed specification, the transformation yields a well-typed tester program in the target language.
– *satisfaction* i.e., the traces accepted by the tester satisfy the specification.
– *detection of faulty specifications* i.e., irrealizable specifications are detected either statically or at run-time.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer-Verlag (1996)
2. Ábrahám, E., Grüner, A., Steffen, M.: Dynamic heap-abstraction for open, object-oriented systems with thread classes. Accepted to be published in SoSYM journal (2006).
3. Jeffrey, A., Rathke, J.: A fully abstract may testing semantics for concurrent objects. In: Proceedings of LICS '02. IEEE, Computer Society Press (2002)
4. Steffen, M.: Object-connectivity and observability for class-based, object-oriented languages. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel (2006).