Testing object Interactions

Grüner, A.

Chapter 8

Concluding remarks

In this thesis, we presented a novel unit testing approach for object-oriented multi-purpose programming languages in the style of *Java* and C♯. Analyzing existing unit testing approaches we identified three goals to be melted into our testing framework:

- Due to the tendency that software developers do not only write the unit code but likewise are responsible for specifying and executing the corresponding unit test cases, the test specification language should be easily *accessible* by software developers. In particular, referring to the increasing popularity of agile software development methodologies, a specification language that is completely different to the programming language may hamper the propagated short test-and-develop cycles that many developers embark on.
- Object-orientation entails heavy collaboration among objects. As a consequence, within an object-oriented context, unit testing coincide with integration testing. Hence, an object-oriented unit test does not merely consist of relating input with output data but instead the proper *interaction of objects* itself is to be verified.
- Finally, not only due to the interdependency among objects, specifying interaction-based test cases can easily become quite complex. Therefore, it is useful if a test specification language allows to formalize test case specifications on a high level.

Our idea for combining these, partly contradicting, features was to define a test specification language by *extending* the programming language with dedicated *specification statements*. Aiming at interaction-based testing, the specification statements basically express expectations regarding the *observable behavior* of the unit under test.

Based on this, the main part of the thesis dealt with a unit testing framework for sequential object-oriented programs. The framework was introduced in four steps. Firstly, we presented a formally defined component-based object-oriented

programming language *Japl* that captured the basic features of *Java*, C♯, and similar languages. Second, we introduced the test specification language as an extension of *Japl*. As a third step, we developed a *test code generation algorithm* which allows to automatically generate a *Japl* test program from a test specification. A central contribution is the *correctness proof* of the code generation algorithm. We concluded the main part with a discussion about possible extensions of both, the programming language and the test specification language.

Even though the programming language *Japl*, as mentioned, captures only a small fraction of the features of today's commonly used object-oriented languages, its formal representation is already quite complex. In particular, the discussion about the language extension with subtyping and inheritance in Section 5.3 gives a foretaste of the complexity one would be confronted with, if aiming at a formal representation for the whole *Java* or C♯ language. One may conclude that implementations of these object-oriented languages are likewise so complex that compilers, virtual machines, and the like themselves should be thoroughly tested. But due to the lack of complete formal specifications regarding these languages, which would be essential to derive useful test cases, it is doubtful that these tests exist either.

Similarly, the development of the code generation algorithm and its correctness proof demonstrated the amount of considerations necessary for writing proper test code in general. Against this background, one specifically can get an impression on the effort that has to be made for writing interaction-based test code without the support of a tool that abstracts away some of the entailed intricacies.

The second part of the thesis introduced *concurrency* into the programming language by means of *thread classes*. Afterwards, we discussed a corresponding extension of the test specification language which also included a sketch regarding the necessary modifications of the code generation algorithm. Clearly, the above remarks about the difficulties of writing proper test code is even more true if concurrency comes into play [60].

As a conclusion, the testing framework proposed in this thesis, indeed, may facilitate writing interaction-based unit tests. Thus, regarding our testing approach, it suggests itself that implementing the framework in *Java* or C♯ is one of the next steps in the near future. The fact that parts of code generation algorithm is given in terms of simple functional programming language code may expedite the implementation. Moreover, as the specification language is defined as an extension of the programming language, it could be implemented with the help of an *extensible compiler* framework like *Polyglot* [51] or *The Dryad Compiler* [43], for instance. As indicated already, however, probably more effort will be necessary for embedding further features of real word programming languages into the formal framework. Besides subtyping and inheritance, synchronization mechanisms like synchronized methods and monitors represent interesting candidates for the

multi-threaded setting. Although it takes quite an effort to define a formal framework for interaction-based testing, it clearly has the benefit that it could form a basis for further testing-related research, in general. For instance, it could be useful in the field of *compositional testing* as it investigates how we can reuse test results about smaller components for the integration test of their composition [14].