

Whole-Program Specifications Permit Better Abstraction and Concurrent Implementations

Bart Jacobs, Frank Piessens, Pieter Bekaert, and Eric Steegmans

Department of Computer Science
Katholieke Universiteit Leuven
Leuven, Belgium

{*Bart.Jacobs, Frank.Piessens, Pieter.Bekaert, Eric.Steegmans*}@cs.kuleuven.ac.be

1 Introduction

This article previews a new system for the formal specification and verification of object-oriented programs and modules. It differs from existing systems (see specifically [1], [4], and [3]) in that it is program-oriented, where existing systems are essentially method-oriented. Specifications in existing systems entail separate per-method proof obligations for each method, usually in the form of Horn triples. The specification formalism in these systems is designed to allow per-method verification. A specification in our system declares one abstract state space for the entire program. Pre- and postconditions of methods are stated in terms of the abstract pre- and post-state of the entire program. In our system, correctness of a program is expressed in terms of *tests*. Basically, a test is a sequence of method invocations. Roughly speaking, correctness of a program means that the results of all possible tests are consistent with the specification.

This approach has a number of implications. Firstly, it permits more freedom in the design of the state abstraction. In other words, there is a higher degree of data hiding. For example, abstract data structures that span multiple classes can be modelled more naturally. Secondly, it integrates more easily with a programming language semantics that supports concurrency. Existing systems are based on a denotational semantics, where statements and methods transform the object store. This does not accommodate concurrency easily, and indeed, existing systems restrict themselves to the sequential subset of Java. Our

system expresses the semantics of Java as a translation of Java programs to π -calculus [2] processes. This translation supports concurrency, and we attempt to comply with Java's memory model by modelling local copies of variables shared by multiple threads as state machines with empty, clean, and dirty states. So, although clients of a program specified using our system are restricted to a sequential use of the program (since correctness is in terms of tests, which are sequences of invocations), the program is free to use multiple threads internally.

2 Tests

A test is a compound Java statement, constructed in a particular way. All possible execution paths in a test end with a `return true;` or a `return false;` statement. Try-catch statements are inserted as necessary to ensure this. If a test, when executed in the context of a particular program, returns `true`, we say that it succeeds. Tests for a particular specification are constructed in such a way that it follows unambiguously from the specification if the test is allowed to succeed or not. Note that we say: "allowed to". This is because, as in most systems, non-deterministic specifications are supported. In our system, even non-deterministic implementations (due to concurrency) are supported. Thanks to our π -calculus-based semantics of Java, we can say, in the definition of correctness: if a test, when executed in the context of a program, has the capability of returning `true`, then the

specification must allow it to do so.

As said earlier, a test contains a sequence of method invocations. The argument expressions of the invocations must be either literals or local variable names, declared earlier in the test. The result of each method invocation is assigned to a fresh local variable. Without this provision, only purely imperative programs could be tested. The provision allows testing object behavior.

In addition to a global state space, a specification declares, for each reference type, a value space. Both are types in the underlying logic. These declarations determine the types of the names in pre- and postconditions that refer to arguments and results of reference type. So, contrary to existing systems, where the type of such names is determined by the system, in our system, this is at the discretion of the specification writer. For immutable reference types, one could choose the corresponding domain as the value space. For mutable reference types, one will usually introduce a fresh type in the underlying logic to model the object identity.

Besides method invocations, tests can also contain reference comparisons and type casts. Correspondingly, specifications can include reference comparison specification clauses and type cast specification clauses. The former provision allows including object identity guarantees, and, equally importantly, allows them to be omitted. The latter allows including guarantees as to the specific type of objects returned by method invocations.

3 Discussion

Above, we discussed correctness of a *program* with respect to a specification. A program, in our system, is any set of classes (and interfaces) without external references. That is, one that can be completely linked.

We define the concept of *module* as the combination of a set of classes and two specifications: the required specification and the provided specification. A module is correct if, for every program P that implements the required specification, the union of P and the classes of the module constitutes a program P' that implements the provided specification.

A major area of further work is the investigation of this concept of module. One issue is the composition of modules: if two modules M and M' provide specifications S

and S' , what specification does the union of M and M' provide? This investigation may unveil that we need to strengthen our definition of correctness if we want to support a useful notion of composition of modules.

Another area of further work is adding support for the specification of callbacks, cross-module inheritance, and I/O. In all of these cases, the issue comes down to this: free names in the π -calculus process, along which communication with the client or with the outside world takes place.

A third area of further work is the facilitation of verification. Although the logical basis for this is available, we have, as of yet, not attempted a formal verification exercise. We expect that we will need to do some work on the fine-tuning of our underlying theories, the construction of special-purpose proof rules, etc., in order to reach the same level of efficiency of verification as that which is being reported for the system in [1].

References

- [1] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in LNCS, pages 284–299. Springer-Verlag, 2001.
- [2] R. Milner. *Communicating and mobile systems: the π -calculus*. Cambridge University Press, 1999.
- [3] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer-Verlag, 1999.
- [4] Joachim van den Berg, Marieke Huisman, Bart Jacobs, and Erik Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in LNCS, pages 1–21. Springer-Verlag, 2000.