Inductive Proof Outlines for Multithreaded Java with Exceptions

—Extended Abstract—

30. April, 2004

Erika Ábrahám[1], Frank S. de Boer[2],
Willem-Paul de Roever[1], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
`{eab,wpr,ms}@informatik.uni-kiel.de`
[2] CWI Amsterdam, The Netherlands
`F.S.de.Boer@cwi.nl`

1 Introduction

In this extended abstract we briefly introduce an assertional proof system for a multithreaded sublanguage of *Java*. The language includes recursion, aliasing, object and thread creation, *Java*'s synchronization mechanism, and exception handling, but ignores the issues of inheritance and subtyping. The proof system is sound and relatively complete, and allows also to prove deadlock freedom [2].

Verification proceeds in three phases: First the program is *augmented* by fresh auxiliary variables and *annotated* with assertions used in the style of Floyd [4, 5] intended to hold during program execution when the flow of control reaches the annotated point. An augmented and annotated program is called a *proof outline* [12]. Afterwards, the proof system, applied to the proof outline, yields a number of *verification conditions* assuring that each program execution conforms to the annotation. Finally, the verification conditions must be *proven*. We use the theorem prover *PVS* [13] for this purpose.

With augmentation and annotation specified by the user, the *Verger tool* takes care of the second phase, i.e., for a proof outline it automatically generates the verification conditions in the syntax of *PVS*. The third phase, the actual verification within the theorem prover, is interactive. For the examples we did, however, most of the conditions could be discharged automatically using the built-in proof strategies of *PVS*. Human interaction was needed mostly for the proof of properties whose formulation required quantifiers.

This work extends earlier results [1] by including exception handling. This is the first sound and relatively complete assertion-based proof method for a concurrent *Java* sublanguage including both synchronization and exception handling. Research in the field of verification for object-oriented programs mostly focused on sequential languages. The Loop-project [6, 10], for instance, develops methods and tools for the verification of sequential object-oriented languages

$$\begin{array}{rcl}
e & ::= & x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(e,\ldots,e) \\
e_{ret} & ::= & \epsilon \mid e \\
stm & ::= & x := e \mid u := e \mid u := \mathsf{new}^{c} \mid u := e.m(e,\ldots,e) \mid e.m(e,\ldots,e) \\
 & \mid & \mathsf{throw}\ e \mid \mathsf{try}\ stm; \mathsf{catch}\ (c\,u)\ stm; \ldots \mathsf{catch}\ (c\,u)\ stm; \mathsf{finally}\ stm\ \mathsf{yrt} \\
 & \mid & \epsilon \mid stm; stm \mid \mathsf{if}\ e\ \mathsf{then}\ stm\ \mathsf{else}\ stm\ \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ stm\ \mathsf{od} \ldots \\
modif & ::= & \mathsf{nsync} \mid \mathsf{sync} \\
meth & ::= & modif\, m(u,\ldots,u)\{\ stm; \mathsf{return}\ e_{ret}\} \\
meth_{\mathsf{run}} & ::= & \mathsf{nsync}\ \mathsf{run}()\{\ stm; \mathsf{return}\ \} \\
meth_{predef} & ::= & meth_{\mathsf{start}}\ meth_{\mathsf{wait}}\ meth_{\mathsf{notify}}\ meth_{\mathsf{notifyAll}} \\
class & ::= & \mathsf{class}\ c\{meth\ldots meth\ meth_{\mathsf{run}}\ meth_{predef}\} \\
class_{\mathsf{main}} & ::= & class \\
prog & ::= & \langle class\ldots class\ class_{\mathsf{main}}\rangle
\end{array}$$

Table 1. Abstract syntax of the programming language

using *PVS* and *Isabelle/HOL*. Especially [8] and [7] formalize the exception mechanism of *Java*. Poetzsch-Heffter and Müller [15] present a Hoare-style programming logic for a sequential kernel of *Java*. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer-assisted soundness proof. In [16] a large subset of *JavaCard*, including exception handling, is formalized in *Isabelle/HOL*, and its soundness and completeness is shown within the theorem prover.

2 The programming language

The abstract syntax of our language is shown in Table 1. The language is strongly typed, where besides class types, we use booleans and integers as primitive types, and pairs and lists as composite types.

A *Java* program specifies a set of classes, where each class declares its own methods and instance variables. The behavior of a program results from the concurrent execution of methods. To support a clean interface between internal and external object behavior, we *exclude qualified references* to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries.

3 The assertion language

To support a clean interface between internal and external object behavior, we disallow qualified references to instance variables. To mirror this modularity, the assertion logic consists of a *local* and a *global* sublanguage. *Local assertions* are used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong. *Global assertions* describe a whole system of objects and their communication structure and will be used in the cooperation test.

4 The proof system

For a complete proof system it is necessary that the transition semantics can be encoded in the assertion language. As the assertion language reasons about the local and global states, we have to *augment* the program with assignments to *fresh auxiliary variables* which we call *observations* to represent information about the control points and stack structures within the local and global states. Auxiliary variables are variables not occurring in the program and may be added to the program to observe certain aspects of the flow of control without affecting it. In general, the observations record information about the intra-object interleaving of threads, which gives rise to shared-variable concurrency, or the inter-object communication via method calls and returns.

Besides user-definable auxiliary variables, our proof system is formulated in terms of *built-in* auxiliary variables, automatically included into all augmentations and used in the verification conditions. These auxiliary variables record information needed to identify threads and local configurations, and to describe monitor synchronization.

Invariant program properties are specified by an *annotation,* associating assertions with the control points of the augmented program. In contrast to the pre- and post-specifications for methods in a sequential context, in our concurrent setting *each* control point needs to be annotated to capture the effect of interleaving.

Besides pre- and postconditions, the annotation defines for each class a local assertion called the *class invariant*, specifying invariant properties of instances of the class in terms of its instance variables. Finally, a global assertion called the *global invariant* specifies properties of communication between objects.

An augmented and annotated program is called a *proof outline* or an *asserted program.*

Invariance of the program properties as specified by the annotation is guaranteed by the verification conditions of the proof system, which are grouped as follows. *Initial correctness* covers satisfaction of the properties in the initial program configuration. The execution of a single method body in isolation is captured by *local correctness* conditions, using the local assertion language. Interference between concurrent method executions is covered by the *interference freedom test* [9, 12], formulated also in the local language. It has especially to accommodate for reentrant code and the specific synchronization mechanism. Possibly affecting more than one object, communication and object creation is covered by the *cooperation test*, using the global language. Communication can take place within a single object or between different objects. As these cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules in [3] and in [9] for CSP.

5 Conclusion

In this work we briefly introduced a sound and complete tool-supported assertional proof system for a *Java* sublanguage including concurrency, synchronization, and exception handling, but no inheritance.

There are a lot of challenging and interesting research topics in the field, which still need further analysis. For future work, we plan to extend the programming language by further constructs, like inheritance and subtyping. In fact, given a general theory of inheritance, e.g., along the lines of [14], we can straightforwardly extend the proof system. We are also interested in the development of a compositional proof system. Also further development of the *Verger* tool is of interest.

References

1. Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof-outlines for monitors in Java. In Najm et al. [11], pages 155–169. (http://www.informatik.uni-freiburg.de/ eab/fmoods03.ps). A longer version appeared as technical report TR-ST-03-1, April 2003 (http://www.informatik.uni-kiel.de/inf/deRoever/techreports/03/tr-st-03-1.pdf).
2. Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof outlines for multithreaded Java with exceptions. Technical Report 0313, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, December 2003. Available at http://www.informatik.uni-kiel.de/reports/2003/0313.html .
3. Krzysztof R. Apt, Nissim Francez, and Willem-Paul de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.
4. Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
5. Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
6. Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
7. Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Proceedings of FASE'00*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer-Verlag, 2000.
8. Bart Jacobs. A formalisation of Java's exception mechanism. In David Sands, editor, *Proceedings of ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 284–301. Springer-Verlag, 2001.
9. Gary Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
10. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/ bart/LOOP/ , 2001.
11. Elie Najm, Uwe Nestmann, and Perdita Stevens, editors. *Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '03), Paris*, volume 2884 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2003.

12. Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
13. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.
14. Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In Najm et al. [11], pages 64–78. A extended version appeared as University of Utrecht Technical Report UU-CS-2003-010.
15. Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. Doaitse Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.
16. David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.