Deductive Verification for Multithreaded Java

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

[1] Christian-Albrechts-University Kiel, Germany
[2] Utrecht University, The Netherlands

Zusammenfassung The semantical foundations of *Java* [9] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [2, 19, 6]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sublanguages (see e.g. [14, 21, 18]). This paper presents a proof system for *multithreaded Java* programs. Concentrating on the issues of concurrency, we introduce an abstract programming language $Java_{MT}$, a subset of *Java* featuring object creation, method invocation, object references with aliasing, and specifically concurrency.
The assertional proof system for verifying safety properties of $Java_{MT}$ is formulated in terms of *proof outlines* [17], i.e., of annotated programs where Hoare-style assertions [8, 12] are associated with every control point.

1 The programming language $Java_{MT}$

$Java_{MT}$ is a multithreaded well-typed sublanguage of *Java*. Programs, as in *Java*, are given by a collection of classes containing instance variable and method declarations. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation,* i.e., synchronous message passing. As we focus on a proof system for the concurrency aspects of *Java*, all classes in $Java_{MT}$ are thread classes in the sense of *Java*: Each class contains a start-method that can be invoked only once for each object, resulting in a new thread of execution. The new thread starts to execute the run-method of the given object while the initiating thread continues its own execution.

For variables, we notationally distinguish between *instance* and *temporary* variables, where instance variables are always private in $Java_{MT}$. Instance variables x hold the state of an object and exist throughout the object's lifetime. Temporary variables u play the role of formal parameters and local variables of method definitions and only exist during the execution of the method to which they belong. Therefore these temporary variables represent the local state of a thread of execution. Table 1 contains the abstract syntax of $Java_{MT}$.

For the semantics, we only highlight a few salient aspects. The formalization as structural operational semantics is given in [1].

The behaviour of a program results from the concurrent execution of threads, each described by the call-chain of its method invocations, given as a stack of

exp	$::= x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp,\ldots,exp)$	$e \in Exp_c^t$	expressions
$sexp$	$::= \mathsf{new}^{c'} \mid exp.m(exp,\ldots,exp) \mid exp.\mathsf{start}()$	$sexp \in SExp_c^t$	side-effect exp.
stm	$::= sexp \mid x := exp \mid u := exp \mid u := sexp$		
	$\mid \epsilon \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm$		
	$\mid \mathsf{while}\ exp\ \mathsf{do}\ stm \ldots$	$stm \in Stm_c$	statements
$modif$	$::= \mathsf{nsync} \mid \mathsf{sync}$		modifiers
$rexp$	$::= \mathsf{return} \mid \mathsf{return}\ exp$		
$meth$	$::= modif\ m(u,\ldots,u)\{\ stm; rexp\}$	$meth \in Meth_c$	methods
$meth_{\mathsf{run}}$	$::= modif\ \mathsf{run}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{run}} \in Meth_c$	run-method
$meth_{\mathsf{main}}$	$::= \mathsf{nsync\ main}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{main}} \in Meth_c$	main-method
$class$	$::= c\{meth\ \ldots\ meth\ meth_{\mathsf{run}}\}$	$class \in Class$	class defn's
$class_{\mathsf{main}}$	$::= c\{meth\ \ldots\ meth\ meth_{\mathsf{run}}\ meth_{\mathsf{main}}\}$	$class_{\mathsf{main}} \in Class$	main-class
$prog$	$::= \langle class\ \ldots\ class\ class_{\mathsf{main}}\rangle$		programs

Tabelle 1. $Java_{MT}$ abstract syntax

local configurations. Threads can be created via new and started by (the first) invocation of the start-method. The invocation of a method extends the call chain by creating a new local configuration. It is removed from the stack when returning from the method. *Java* offers a synchronization mechanism for the mutually exclusive execution of methods: *Synchronized* methods of an object can be invoked only if no other threads are currently executing any synchronized methods of the same object.

2 The proof system

This section sketches the assertional proof system formulated in terms of *proof outlines* [17, 7], i.e., where Hoare-style pre- and postconditions [8, 12] are associated with each program statement. The proof system has to accommodate for shared-variable concurrency, aliasing, method invocation, and dynamic object creation.

2.1 The assertion language

The underlying assertion language consists of two different levels: The local assertion language specifies the behaviour on the level of method execution, and is used to annotate programs. The global behaviour, including the communication topology of the objects, is expressed in the global language used in the cooperation test.

In the language of assertions, we introduce as usual a countably infinite set of *logical variables* with typical element z disjoint from the instance and the local variables occurring in programs. Logical variables are used as bound variables in quantifications and, on the global level, to represent the values of local variables.

Table 2 defines the syntax of the assertion language. *Local expressions* are expressions of the programming language possibly containing logical variables. *Local assertions* are standard logical formulas over local expressions, where unrestricted quantification is allowed for integer and boolean domains only. Quantification over objects is only allowed in a restricted form asserting the existence of an element or a subsequence of a given sequence. Restricted quantification involving objects ensures that the evaluation of a local assertion indeed only depends on the values of the instance and temporary variables. In deference to the local assertion language, quantification on the global level is allowed for all types. Quantifications over objects range over the set of *existing* objects only.

$exp_l ::= z \mid x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp_l, \ldots, exp_l)$	$e \in LExp_c^t$	local expressions
$ass_l ::= exp_l \mid \neg ass_l \mid ass_l \wedge ass_l$		
$\quad \mid \exists z(ass_l) \mid \exists z \in exp_l(ass_l) \mid \exists z \sqsubseteq exp_l(ass_l)$	$p \in LAss_c$	local assertions
$exp_g ::= z \mid \mathsf{nil} \mid \mathsf{f}(exp_g, \ldots, exp_g) \mid exp_g.x$	$E \in GExp^t$	global expressions
$ass_g ::= exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists z(ass_g)$	$P \in GAss$	global assertions

Tabelle 2. Syntax of assertions

2.2 Proof outlines

To be able to reason about the communication mechanism of method invocations, we split each method invocation statement into the sequential composition of an output and an input statement representing the invocation of the method and the reception of the return value.

Next, we augment the program by fresh *auxiliary* variables. Assignments can be extended to multiple assignments, and additional multiple assignments to auxiliary variables can be inserted at any point. We introduce three specific auxiliary variables id, lock, and $\mathsf{started}$ to represent information about the global configuration at the proof-theoretical level. The temporary variable id of type $\mathsf{Object} \times \mathsf{Int}$ stores the identity of the object in which the corresponding thread has begun its execution, together with the current depth of its stack. The auxiliary instance variable lock of the same type is used to reason about thread synchronization: The value $\perp$ states that no threads are currently executing any synchronized methods of the given object; otherwise, the value (α, n) identifies the thread which acquired the lock, together with the stack depth n, at which it has gotten the lock. The boolean instance variable $\mathsf{started}$ states whether the object's start-method has already been invoked.

Finally, we extend programs by *critical sections,* a conceptual notion, which is introduced for the purpose of proof and, therefore, does not influence the control flow. Semantically, a critical section expresses that the statements inside are executed without interleaving with other threads.

To specify invariant properties of the system, the transformed programs are *annotated* by attaching pre- and postconditions, formulated in the local assertion language, to all occurrences of statements. Besides that, for each class c, the annotation defines a local assertion I_c called *class invariant,* which refers only to instance variables, and expresses invariant properties of the instances of the class. Finally, the *global invariant* $GI \in GAss$ specifies properties of communication between objects. We require that for all qualified references $E.x$ in GI, all assignments to x in class c are enclosed in critical sections.

2.3 Proof system

The global behaviour of a *Java* program results from the concurrent execution of method bodies, that can interact by

- shared-variable concurrency,
- synchronous message passing for method calls, and
- object creation.

Apart from the *initial correctness*, meaning that the annotation is correct with respect to the initial configuration, the proof system is split into three parts. The execution of a single method body in isolation is captured by *local correctness* conditions that show the inductiveness of the annotated method bodies and which are standard.

Interaction via synchronous message passing and via object creation cannot be established locally but only relative to assumptions about the communicated values. These assumptions are verified in the *cooperation test.* The communication can take place within a single object or between different objects. As these two cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules used in [5] and in [15] for CSP.

Finally, the effect of shared-variable concurrency is handled, as usual, by the *interference freedom test,* which is modeled after the corresponding tests in the proof systems for shared-variable concurrency in [17] and in [15]. In the case of *Java* it additionally has to accommodate for reentrant code and the specific synchronization mechanism.

Local correctness A proof outline is *locally correct*, if the usual verification conditions [4] for standard sequential constructs hold: The precondition of a multiple assignment to instance and local variables must imply the postcondition after execution of the assignment. As output and return statements do not affect the state of the executing thread, their preconditions must directly imply their postconditions. Finally, the pre- and postconditions of all statements of a class are required to imply the class invariant.

The interference freedom test The conditions of the interference freedom test ensure the invariance of local properties of a thread under the activities of

other threads. Since we disallow public instance variables in $Java_{MT}$, we only have to deal with the invariance of properties under the execution of statements within the same object. Containing only temporary variables, communication and object creation statements do not change the state of the executing object. Thus we only have to take assignments $\vec{y} := \vec{e}$ into account.

Satisfaction of a local property of a thread may clearly be affected by the execution of assignments by a *different* thread in the same object. If, otherwise, the property describes the *same* thread that executes the assignment, the only control points endangered are those waiting for a return value earlier in the current execution stack, i.e., we have to show the invariance of preconditions of receive statements. Especially, the interference freedom test has to take care of *reentrant* method calls.

The cooperation test Whereas the verification conditions associated with local correctness and interference freedom cover the effects of assigning side-effect-free expressions to variables, the *cooperation test* deals with method invocation and object creation. Since different objects may be involved, it is formulated in the global assertion language. Besides defining verification conditions that ensure the invariance of the global invariant, it specifies conditions under which properties, whose evaluation depend on communicated values, are satisfied. Those properties are given by the preconditions of method bodies, and by the postconditions of receive and object creation statements.

3 Conclusion

In this extended abstract we sketched an assertional proof method for a multithreaded sublanguage of *Java*. The *soundness* of our method is shown by a standard albeit tedious induction on the length of the computation. Proving its *completeness* involves the introduction of appropriate assertions expressing reachability and auxiliary *history variables*. The details of the proofs can be found in [1].

Currently we are developing in the context of the European Fifth Framework RTD project Omega and the bilateral NWO/DFG project MobiJ a front-end tool for the computer-aided specification and verification of *Java* programs based on our proof method. Such a front-end tool consists of an editor and a parser for annotating *Java* programs, and of a compiler which translates these annotated *Java* programs into corresponding verification conditions. A theorem prover (HOL or PVS) is used for verifying the validity of these verifications conditions. Of particular interest in this context is an integration of our method with related approaches like the LOOP project [11, 16].

More in general, our future work focusses on the formalization of full-featured multithreading, inheritance, and polymorphic extensions involving behavioral subtyping [3].

Acknowledgements We thank Ulrich Hannemann for discussions and comments on an earlier version of the paper.

Literatur

1. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept: Soundness and completeness. Technical Report TR-ST-01-2, Lehrstuhl für Software-Technologie, Christian-Albrechts-Universität Kiel, 2001.
2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS State-of-the-Art-Survey. Springer, 1999.
3. P. America. A behavioural approach to subtyping in object-oriented programming languages. Technical report 443, Phillips Research Laboratories, 1989.
4. K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.
5. K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.
6. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [2].
7. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods.* Cambridge University Press, 2001.
8. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
9. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
10. C. Hankin, editor. *Programming Languages and Systems: Proceedings of the 7th European Symposium on Programming (ESOP '98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal, March/April 1998)*, LNCS 1381. Springer, 1998.
11. J. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Hankin [10].
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [13].
13. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science.* International Series in Computer Science. Prentice Hall, 1989.
14. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.
15. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
16. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.
17. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
18. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In Swierstra [20], pages 162–176.
19. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine.* Springer, 2001.
20. S. Swierstra, editor. *Proceedings of the 8th European Symposium on Programming (ESOP '99)*, LNCS 1576. Springer, 1999.
21. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. submitted for publication, 2002.