# From Executable Formal Specification to Java Property Verification

*Extended Abstract*

I. Attali, D. Caromel, H. Nilsson and M. Russo
INRIA Sophia Antipolis,
CNRS - I3S - Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
tel: 33 4 92 38 79 10
fax: 33 4 92 38 76 33
First.Last@sophia.inria.fr
http://www.inria.fr/oasis/java

## 1 Position Statement

To be sure of the meaning of a programming language, we need to have its formal semantics. But semantic specifications are hard to write, and it is difficult to be convinced that they are correct. Having an executable semantics helps, since this allows the semantics to be tested on real programs which tends to expose many mistakes. But ultimately the correctness of the semantics has to be proved and then preferably by means of a computerized proof assistant to avoid mistakes in the proof. This is why we are working towards machine proofs with executable formal specifications. The extended abstract that follows is a step towards that goal.

## 2 Introduction

In this article, we are defining a dynamic semantics of a large, concurrent subset of Java. We are not concerned with typing (we assume our programs are typed correctly). This specification is executable, but it is also used as the basis for formal verification of Java semantical properties. This executable specification enables us to derive an environment which includes visualization of programs execution and tests our semantics.

In Section 3 we present related work in the domain of formal semantics and property verification. Section 4 describes our formal specification of Java (syntax and semantics), and the graphical interactive execution environment for Java we derive from this executable semantics. Section 5 presents Java property verification in the context of multithreaded applications.. Section 6 concludes the paper.

## 3 Related Work

The formal specification of the Java semantics is an active research area. There are two main approaches. The first possibility is to directly specify the semantics of Java source code. This has been done by e.g. Drossopoulou and Eisenbach [7], Syme [17], Nipkow and Oheimb [14]. Another approach is to work at the JVM byte-code level. Qian [16] has specified a subset of the JVM instructions for objects, methods and subroutines. Börger and Schulte [2, 1] define the JVM in order to prove the correctness of Java compilation. Jensen, Le Metayer and Thorn [10] formalize dynamic class loading mechanisms in the JVM and study some security properties of Java.

One of the most common approaches to verification is to translate the program source code into an analysis formalism by using some abstraction techniques. Correctness properties are expressed in the same formalism and a specific mechanism is applied on the program translation to check this correctness property. This approach has been applied to prove the absence of deadlocks in multi-threaded Java programs using model-checking (as in [4], and in [9, 5] with the Spin system), and the absence of livelocks thanks to data and control-flow analysis [12].

Another important research topic is to prove properties of the language itself. Type-soundness (*Well typed programs cannot go wrong*) is an important language property studied in [7, 13, 17, 15]. In [15], Müller and Poetzsch-Heffter present a Hoare-style [11] programming logic for a sequential kernel of Java. They also specify a formal operational semantics of this subset and prove the soundness of this semantics by formalizing it and the typing rules in higher-order logic.the

Our work is original in that the same formal specifications are used both to derive an environment and as a basis for proving properties.

## 4 Formal Specifications For Debugging

This section presents our formal specification of a large, concurrent subset of Java. We also explain how we derive a visualizing execution environment from the formal specification.

These specifications have been written within the Centaur system [3], which is a generic programming environment: from the syntax and the semantics specifications of a given language, one can automatically produce a structure editor and semantics-based tools (such as type checkers and interpreters) for this language.

## 4.1 Java formal specification

Every structured object of our semantics is represented within the system as an abstract syntax tree. The abstract syntax so defined includes 128 operators and 56 types related to Java constructors (as defined in [8]). In our semantics specification (about 800 Typol rules), objects, threads, and configurations (a pair of objects and static variables) are modeled as semantic structures. Typol [6] (a formalism which is an implementation of the Natural Semantics) rules are organized in modules which enhances the design, improves the readability, and facilitates debugging of the specification. Some modules are expressed in a Natural Semantics style (especially object-oriented features). Some others are expressed in a Structural Operational Semantics style (especially concurrent features). Our operational semantics simulates concurrency with a deterministic interleaving of threads.

In Java, the only way for a thread to acquire a lock on a given object is to either enter a *synchronized* block or to call a *synchronized* method. A *synchronized* method is equivalent to a "normal" method whose body is enclosed in a *synchronized* block. The behavior of the *synchronized* statement is the following:

1. Compute the reference to the concerned object .
2. Perform a lock action on that object: the thread may become (temporarily) *blocked* if the object is already locked by another thread).
3. Execute the *synchronized* block body.
4. Perform an unlock action on the object.

Our semantics reflects this behavior through five Typol rules which correspond to each step of the above informal specification. Two Typol rules are corresponding to the first step of the behavior: one concerns the evaluation of the expression in order to get the reference of the object to lock, whereas the second one enables the transition between the first step and the second one of the synchronization specification. Figure 1 presents the Typol rule which correspond to the second step of the *synchronized* statement behavior.

## 4.2 Visualization environment

A visualization environment (see *http://www.inria/oasis/java/* for snap shots of the environment) is derived from these formal specifications. It includes both textual and graphical visualization of Java programs interpretation. This environment shows the list of objects using two visualization engines, both of them based on the semantic structure modeling the object list. The textual

```
SynchronizedStatement_GetTheLockOrBeBlocked:
   --CS: synchronized(#8) Statement
   getObjectLockInformation(
      ObjId2 |- ObjL1 -> OThId2, nbLocks )  &
   getTheLockOrBeBlocked(
      OThId1, ObjId2, Statement, ObjL1, ClVarL |- OThId2, nbLocks -> ObjL1_1, Inst, ThStatus )
   --------------------------------
   ObjL1, ClVarL, OThId1 |-
    clr(ObjId1, Mode, MethName, Env, insts[synchroLock(ObjId2, Statement).InstL]) ->
       ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, insts[Inst.InstL]), ThStatus  ;
```

Figure 1: Lock the object or block the currently *executing* thread.

view is a direct pretty-printing of the list of objects and threads. The graphical view shows the complete topology of the object graph.

In order to provide graphical animation *during* program execution, the semantics interpreter notifies the two textual and graphical visualization engines of important events (state change). This is done by annotating the semantics. The annotations are effectively procedure calls for sending information to and synchronize with the visualization engines.

## 5 Formal Specifications For Proofs

This section presents work in progress: the verification of important properties of our semantics. We are particularly interested in synchronization properties as they are quite complex and difficult to test.

If two distinct threads can access some specific object, this object is shared by these two threads. This can potentially lead to a protection problem of the shared data;

Here we aim to present the skeleton of the proof of the following property: `Two threads cannot hold a lock on a same object at the same time` (as stated in [8], page 399). To prove this, we first observe the following:

1. When the program interpretation begins, there is only one thread: the *main* thread, that is to say a thread which has not been created explicitly by the programmer but implicitly, to execute the *main* method.

2. Synchronization needs appear only when there are at least two threads.

From point 1, we can deduce that at the execution beginning, the property holds.

The main point is to prove that if this property holds at a given step of the execution, then it still holds at the next step. Indeed, the property holds as long as no thread wants to acquire a lock on a given object. The step interesting to study is in fact when an object is locked by a given thread, and when a second thread wants to acquire a lock on this object (if the object is not already locked, there is no problem). Our semantics simulates concurrency through the introduction of interleaving between the different threads which have been created by a given program. At each execution step there is therefore only one *executing* thread. Moreover, a Typol rule is atomic, which means that nothing can happen between the execution of the two premises of the rule presented in Figure 1. The information acquired by the first premise about the absence or presence of lock on the considered object is therefore always right when the second premise is executed.

There is one special case to study. A thread (thread 1) which calls its *wait* method on a specific object, releases all the locks it owned on this object and becomes *dormant*. When another thread (thread 2) calls the *notify* or *notifyAll* method on this object, thread 1 becomes *blocked*, waiting for a lock on this object. Once thread 2 has released and thread 1 acquired the lock, thread 1 becomes executing again.

# 6 Conclusion

In this extended abstract, we briefly presented our semantic definition of a large subset of Java. The main point of our work is that the same formal specifications are used to derive an environment and also to prove properties.

The semantic specification, using both a small-step and a big-step style (thanks to the Typol logical framework which makes it possible to mix these two styles), includes primitives types, classes, inheritance, instance variables and methods, class variables and methods, interfaces, overloading, shadowing, dynamic method binding, object and thread creation, concurrency, arrays and exceptions. From this specification we derive a graphical programming environment. This environment is animated and interactive, it includes visualization of the object topology during program execution.

The last section presented in this abstract is a work in progress. We have to formalize the proof skeleton given in Section 5. We also intend to study other thread-specific properties. For example, if the execution of a *synchronized* method is ever completed, either normally or abruptly (with an exception), an *unlock* action is always performed on that same lock ([8], page 416).

# References

[1] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer-Verlag, 1998. to appear.

[2] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In *Formal Syntax and Semantics of Java*. Springer-Verlag, LNCS 1523, 1998.

[3] P. Borras and et al. Centaur: the System. In *SIGSOFT'88 Third Annual Symposium on Software Development Environments*, Boston, 1988.

[4] J. C. Corbett. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. Technical report, October 1998. submitted for publication.

[5] C. Demartini, R. Iosif, and R. Sisto. Modeling and Validation of Java Multithreaded Applications using Spin. In *Proc. of the 4th SPIN Workshop, Paris, France*, November 1998.

[6] T. Despeyroux. Typol: A Formalism to Implement Natural Semantics. Research Report 94, INRIA, 1988.

[7] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. *Lecture Notes in Computer Science*, 1523, 1999.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. In *International Journal on Software Tools for Technology Transfer (STTT)*, 1998. To appear.

[10] T. Jensen, D. Le Métayer, and T. Thorn. Security and Dynamic Class Loading in Java: a Formal isation. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 4–15, May 1998.

[11] L. Lamport. The 'hoare logic' of concurrent programs. In Springer-Verlag, editor, *Acta Informatica 14*, pages 21–37, 1980.

[12] G. Naumovich, S. Avrunin, and L. A. Clarke. Data Flow Analysis for Checking Properties of Concurrent Java Programs. In *Proc. of the International Conference on Software Engineering*, May 1999.

[13] T. Nipkow. Invited talk: Embedding programming languages in theorem provers. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, page 398, Berlin, July 7–10, 1999. Springer-Verlag.

[14] T. Nipkow and D. von Oheimb. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119+37. SV, 1998.

[15] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[16] Z. Qian. A formal specification of Java[TM] virtual machine instructions for objects, methods and subroutines. *Lecture Notes in Computer Science*, 1523, 1999.

[17] D. Syme. Proving Java type soundness. *Lecture Notes in Computer Science*, 1523, 1999.