Formal JVM Code Analysis in JavaFAN

Azadeh Farzan, José Meseguer, and Grigore Roşu

Department of Computer Science,
University of Illinois at Urbana-Champaign
{afarzan,meseguer,grosu}@cs.uiuc.edu

Abstract. JavaFAN uses a Maude rewriting logic specification of the JVM semantics as the basis of a software analysis tool with competitive performance. It supports formal analysis of concurrent JVM programs by means of symbolic simulation, breadth-first search, and LTL model checking. We discuss JavaFAN's executable formal specification of the JVM, illustrate its formal analysis capabilities using several case studies, and compare its performance with similar Java analysis tools.

1 Introduction

There is a general belief in the algebraic specification community that all traditional programming language features can be described with equational specifications [2, 9, 29]. What is less known, or tends to be ignored, is that *concurrency*, which is a feature of almost any current programming language, cannot be naturally handled by equational specifications, unless one makes deterministic restrictions on how the different processes or threads are interleaved. While some of these restrictions may be acceptable, as most programming languages also provide thread or process scheduling algorithms, most of them are unacceptable in practice because concurrent execution typically depends upon the external environment, which is unpredictable. Rewriting logic [17] extends equational logic with rewriting rules and has been mainly introduced as a *unified model of concurrency*; indeed, many formal theories of concurrency have been naturally mapped into rewriting logic during the last decade.

A next natural challenge is to define mainstream concurrent programming languages in rewriting logic and then use those definitions to build formal analysis tools for such languages. There is already a substantial body of case studies, of which we only mention [25, 24, 28], backing up one of the key claims of this paper, namely that *rewriting logic can be fruitfully used as a unifying framework for defining programming languages*. Further evidence on this claim includes modeling of a wide range of programming language features that has been developed and tested as part of a recent course taught at the University of Illinois [22]. In this paper we give detailed evidence for a second key claim, namely that rewriting logic specifications can be used *in practice* to build simulators and formal analysis tools for mainstream programming languages such as Java with competitive performance. Here, we focus on Java's bytecode, but our methodology is general and can be applied also to the Java source code level and to many other languages.

C. Rattray et al. (Eds.): AMAST 2004, LNCS 3116, pp. 132–147, 2004.

The JavaFAN (Java Formal Analyzer) tool specifies the semantics of the most commonly used JVM bytecode instructions (150 out of the 250 total) as a Maude module specifying a rewrite theory $T_{\mathrm{JVM}} = (\Sigma_{\mathrm{JVM}}, E_{\mathrm{JVM}}, R_{\mathrm{JVM}})$, where $(\Sigma_{\mathrm{JVM}}, E_{\mathrm{JVM}})$ is an equational theory giving an algebraic semantics with semantic equations E_{JVM} to the *deterministic* JVM instructions, whereas R_{JVM} is a set of *rewrite rules*, with concurrent transition semantics, specifying the behavior of all *concurrent* JVM instructions. The three kinds of formal analysis currently supported in JavaFAN are: (1) *symbolic simulation*, where the theory T_{JVM} is executed in Maude as a JVM intepreter supporting fair execution and allowing some input values to be symbolic; (2) *breadth-first search*, where the entire, possibly infinite, state space of a program is explored starting from its initial state using Maude's `search` command to find safety property violations; and (3) *model checking*, where if a program's set of reachable states is finite, linear time temporal logic (LTL) properties are verified using Maude's LTL model checker.

A remarkable fact is that, as we explain in Section 4, even though T_{JVM} gives indeed a *mathematical semantics* to the JVM, it becomes the basis of a formal analysis tool whose performance is *competitive* and in some cases surpasses that of other Java analysis tools. The reasons for this are twofold. On the one hand, Maude [3] is a high-performance logical engine, achieving millions of rewrites per second on real applications, efficiently supporting search, and performing model checking with performance similar to that of SPIN [13]. On the other, the algebraic specification of system states, as well as the equations E_{JVM} and rules R_{JVM}, have been *optimized* for performance through several techniques explained in Section 3.5, including keeping only the dynamic parts of the state explicitly in the state representation, and making most equations and rules *unconditional*. In this regard, rewriting logic's distinction between the equations E_{JVM} and the rules R_{JVM} has a crucial performance impact in drastically reducing the sate space size. The point is that rewriting with the rules R_{JVM} takes place *modulo* the equations E_{JVM}, and therefore only the rules R_{JVM} affect state space size. Our experience in specifying the JVM in rewriting logic is that we gain the best benefits from algebraic (equations) and SOS [20] (Rules) paradigms in a combined way, while being able to distinguish between deterministic and concurrent features in a way not possible in either SOS or algebraic semantics.

Related Work. The different approaches to formal analysis for Java can be classified as focusing on either *sequential* or *concurrent* programs. Our work falls in the second category. More specifically, it belongs to a family of approaches that use a *formal executable specification of the concurrent semantics* of the JVM as a basis for formal reasoning. Two other approaches in precisely this category are one based on the ACL2 logic and theorem prover [15], and another based on a formal JVM semantics and reasoning based on Abstract State Machines (ASM) [23]. Our approach seems complementary to both of these, in the sense that it provides new formal analysis capabilities, namely search and LTL model checking. The ACL2 work is in a sense more powerful, since it uses an inductive theorem prover, but this greater power requires greater expertise and effort.