Second Conference on Domain Specific Languages
Austin, Texas, October 3–5, 1999.

# Using Production Grammars in Software Testing

Emin Gün Sirer Brian N. Bershad

*Department of Computer Science*
*University of Washington*
*Box 352350, Seattle, WA 98195-2350*
*http://kimera.cs.washington.edu*
*{egs,bershad}@cs.washington.edu*

## Abstract

Extensible typesafe systems, such as Java, rely critically on a large and complex software base for their overall protection and integrity, and are therefore difficult to test and verify. Traditional testing techniques, such as manual test generation and formal verification, are too time consuming, expensive, and imprecise, or work only on abstract models of the implementation and are too simplistic. Consequently, commercial virtual machines deployed so far have exhibited numerous bugs and security holes.

In this paper, we discuss our experience with using production grammars in testing large, complex and safety-critical software systems. Specifically, we describe *lava*, a domain specific language we have developed for specifying production grammars, and relate our experience with using *lava* to generate effective test suites for the Java virtual machine. We demonstrate the effectiveness of production grammars in generating complex test cases that can, when combined with comparative and variant testing techniques, achieve high code and value coverage. We also describe an extension to production grammars that enables concurrent generation of certificates for test cases. A certificate is a behavioral description that specifies the intended outcome of the generated test case, and therefore acts as an oracle by which the correctness of the tested system can be evaluated in isolation. We report the results of applying these testing techniques to commercial Java implementations. We conclude that the use of production grammars in combination with other automated testing techniques is a powerful and effective method for testing software systems, and is enabled by a special purpose language for specifying extended production grammars.

## 1. Introduction

This paper describes *lava*, a special purpose language for specifying production grammars, and summarizes how production grammars can be used as part of a concerted software engineering effort to test large systems. In particular, we describe our experience with applying production grammars written in *lava* to the testing of Java virtual machines. We show that production grammars, expressed in a suitable language, can be used to automatically create, and reason about, complex test cases from concise, well-structured specifications.

Modern virtual machines [Lindholm&Yellin 99, Inferno, Adl-Tabatabai et al. 96], such as Java, have emerged in recent years as generic and ubiquitous components in extensible applications. Virtual machines can now be found in hypertext systems [Berners-Lee et al. 96], web servers [SunJWS], databases [Oracle], desktop applications, and consumer devices such as cellphones and smartcards. The primary appeal of virtual machines is that they enable users to safely run untrusted, and potentially malicious, code. The safety of modern virtual machines, Java in particular, depends critically on three large and complex software components: (1) a verifier for static inspection of untrusted code against a set of safety axioms, (2) an interpreter or a compiler to respect instruction semantics during execution, and (3) a runtime system to correctly provide services such as threading and garbage collection. A failure in any one of these components allows applications to violate system integrity, and may result in data theft or corruption [McGraw & Felten 96].

This capability to execute untrusted code places a large burden on the system implementers to ensure overall system safety and correctness. A typical modern virtual machine such as Java contains hundreds of fine-grain, subtle and diverse security axioms in its verifier, as well as many lines of similarly subtle code in its interpreter, compiler and system libraries. Consequently, verifying the correctness of a virtual machine is a difficult task.

## Testing Alternatives

The current state of the art in formal program verification is not yet sufficient to verify large, complex virtual machines. While the theory community has made substantial progress on provably correct bytecode verifiers [Stata&Abadi 98,Freund&Mitchell 98], current results exhibit three serious shortcomings. Namely, they operate on abstract models of the code instead of the actual implementation, cover only a subset of the constructs found in a typical virtual machine and require extensive human involvement to map the formal model onto the actual implementation. In addition, these type-system based formal methods for verifiers do not immediately apply to interpreters, compilers or the runtime system, whose functionality is hard to model using static type systems.

Subsequently, virtual machine developers have had to rely on manual verification and testing techniques to gain assurance in their implementations. Techniques such as independent code reviews [Dean et al. 97] and directed test case generation have been used extensively by various commercial virtual machine vendors. However, manual testing techniques, in general, are expensive and slow due to the amount of human effort they entail. In addition, independent code reviews require relinquishing source code access to reviewers, and lack a good progress metric for the reviewers' efforts. Manual test case generation, on the other hand, often requires massive amounts of tedious human effort to achieve high code coverage.

A common approach to automating testing is to simply use ad hoc scripts, written in a general-purpose language, to generate test inputs. While this approach may save some time in test creation, it exhibits a number of shortcomings. First, writing such test scripts is time consuming and difficult, as scripting languages often do not provide any convenient constructs for this programming domain beyond string manipulation. Second, managing many such scripts is operationally difficult once they have been written, especially as they evolve throughout the life cycle of the project. Third, the unstructured nature of general-purpose languages poses a steep learning curve for those who need to understand and modify the test scripts. Fundamentally, a general-purpose language is too general, and therefore too unstructured, for test generation.

## Production Grammars

In this paper, we describe *lava*, a special purpose language for specifying production grammars. A production grammar is a collection of non-terminal to terminal mappings that resembles a regular parsing grammar, but is used "in reverse." That is, instead of parsing a sequence of tokens into higher level constructs, a production grammar generates a stream of tokens from a set of non-terminals that specify the overall structure of the stream. We describe how *lava* grammars differ from traditional parsing grammars, and illustrate the different features we added into the language to support test generation. Our experience with *lava* demonstrates that a special purpose language for specifying production grammars can bring high coverage, simplicity, manageability and structure to the testing effort.

Production grammars are well-suited for test generation not only because they can create diverse test cases effectively, but also because they can provide guidance on how the test cases they generate ought to behave. It is often hard to determine the correct system behavior for automatically generated test cases -- a fundamental difficulty known as the oracle problem. In the worst case, automated test generation may require reverse engineering and manual examination to determine the expected behavior of the system on the given test input.

We address the oracle problem in two ways. We first show that test cases generated with production grammars can be used in conjunction with comparative testing to create effective test suites without human involvement. Second, we show how an extended production grammar language can generate self-describing test cases that obviate the need for comparative testing. In simple production systems, the code producer does not retain or carry forward any information through the generation process to be able to reason about the properties of the test case. We overcome this problem by extending the grammar specification language to concurrently generate certificates for test cases. Certificates are a concise description of the expected system behavior for the given test case. They act as oracles on the intended behavior of the generated test program, and thereby enable a single virtual machine to be tested without comparison against a reference implementation.

Overall, this paper makes three contributions. First, we describe the design of a domain specific language for specifying production grammars, and illustrate how it can be used on its own for carrying out performance analyses, for checking robustness, and for testing transformation components such as compilers. Further, we show that combining production grammars with other automated techniques, such as comparative evaluation, can achieve high code coverage. Finally, we show that the structured nature of production grammars can be used to reason about the behavior of test cases, and address the oracle problem. Testing, especially of commercial systems, is a field where success is fundamentally difficult to quantify because the number of undiscovered bugs cannot be known. We relate anecdotal descriptions of the types of errors uncovered by our approach, and use quantitative measures from software engineering, such as code coverage, whenever possible.

In the next section, we provide the necessary background on the Java virtual machine and define the scope and goals of our testing efforts. Readers with a background in the Java virtual machine can skip ahead to the end of that section. Section 3 describes the *lava* language, and illustrates how the language can be used to construct complex test cases from a grammar. Section 4 discusses the integration of production grammars with other automated testing techniques. Section 5 describes extensions to the *lava* language to concurrently generate certificates along with test cases. Section 6 discusses related work, and section 7 concludes.

## 2. Background & Goals

A Java virtual machine (JVM) is an extensive system combining operating system services and a language runtime around a typed, stack-based, object-oriented instruction set architecture. The integrity of a Java virtual machine relies critically on the three fundamental components that comprise the JVM; namely, on a verifier, interpreter/compiler and a set of standard system libraries. We focus our testing efforts on the first two components. The system libraries are written mostly in Java, and consequently benefit directly from increased assurance in the verifier and the execution engine. Verification and compilation form the bulk of the trusted computing base for a JVM, and embody a large amount of complex and subtle functionality.

The Java verifier ensures that untrusted code, presented to the virtual machine in the richly annotated bytecode format, conforms to an extensive suite of system security constraints. While these constraints are not formally specified, we extracted roughly 620 distinct security axioms through a close examination of the Java virtual machine specification and its errata [Lindholm&Yellin 99]. These safety axioms range in complexity from integer comparisons to sophisticated analyses that rely on type inference and data flow. Roughly, a third of the safety axioms are devoted to ensuring the consistency of top level structures within an object file. They deal with issues such as bound checks on structure lengths and range checks on table indices. Around 10% of the safety checks in the verifier are devoted to ensuring that the code section within the object file is well structured. For instance, these checks restrict control flow instructions from jumping outside the code segment or into the middle of instructions. Another 10% of the checks are devoted to making sure that the assumptions made in one class about another class are valid. These checks ensure, for example, that a field access made by one class refers to a valid exported field in another class of the right type. The remaining 300 or so checks are devoted to dataflow analysis, and form the crux of the verifier. They ensure that the operands for virtual machine instructions will be valid on all dynamic instruction paths. Since Java bytecode does not contain declarative type information, the type of each operand has to be inferred for all possible execution paths. This stage relies on standard dataflow analysis to exhaustively check the system state on all possible paths of execution, and ensures, for instance, that the operands to an integer addition operation always consist of two integers. Further, these checks also ensure that objects are properly initialized via parent constructors, and that every use of an object is preceded by an initialization of the same object. Based on various analyses [Drossopoulou+ 97,Drossopoulou+ 99,Syme 97], we assume that a correct implementation of these 620 axioms is sufficient to ensure typesafety and to protect the assumptions made in the virtual machine from being violated at runtime. The task of the system designer, then, is to ensure that the implementation of safety axioms in a verifier corresponds to the specification.

Similarly, the Java execution engine, whether it is an interpreter, a just-in-time compiler, a way-ahead-of-time compiler [Proebsting et al. 97] or a hybrid scheme [Muller et al. 97,Griswold], needs to implement a large amount of functionality to correctly execute Java bytecode instructions. Since the JVM instruction set architecture defines a CISC, the instruction semantics are high-level and complex. There are 202 distinct instructions, whose behavioral description takes up more than 160 pages in the JVM specification. In addition, the architecture allows non-uniform constructs such as arbitrary length instructions, instruction padding and variable opcode lengths, which introduce complications into the implementation.

The challenge for testing JVMs, then, is to ensure that the implementation of safety checks in a bytecode verifier, and that the implementation of instruction semantics in an interpreter, compiler and optimizer, are correct. We faced this challenge when we developed our own Java virtual machine [Sirer et al. 98]. Our attempts to create test cases manually were soon overwhelmed, and we sought a testing scheme that possessed the following properties:

- *Automatic*: Testing should proceed without human involvement, and therefore be relatively cheap. The technique should be easy to incorporate into nightly regression testing.
- *Complete*: Testing should generate numerous test cases that cover as much of the functionality of a virtual machine as possible. It should also admit a metric of progress that correlates with the amount of assurance in the virtual machine being tested.
- *Conservative*: Bad Java bytecodes should not be allowed to pass undetected through the bytecode verifier, and incorrectly executed instructions in the compiler or interpreter should be detected.
- *Well structured*: Examining, directing, check-pointing and resuming verification efforts should be simple. Error messages should be descriptive; that is, it should be easy for a programmer to track down and fix a problem.
- *Efficient*: Testing should result in a high-confidence Java virtual machine within a reasonable amount of time.

The rest of this paper describes our experience with *lava* aimed at achieving these goals.

## 3. *Lava* and Grammar-based Test Generation

Our approach to test generation is to use an automated, well-structured process driven by a production grammar. A production grammar is the opposite of a regular parsing grammar in that it produces a program (i.e. all terminals, or tokens) starting from a high-level description (i.e. a set of non-terminals). The composition of the generated program reflects the restrictions placed on it by the production grammar. Figure 1 illustrates the high-level structure of the test generation process. A generic code-generator-generator parses a Java bytecode grammar written in *lava* and emits a specialized code-generator. The code-generator is a state machine that in turn takes a seed as input and applies the grammar to it. The seed consists of the high-level description that guides the production process. Running the code-generator on a seed produces test cases in Java bytecode that can then be used for testing.

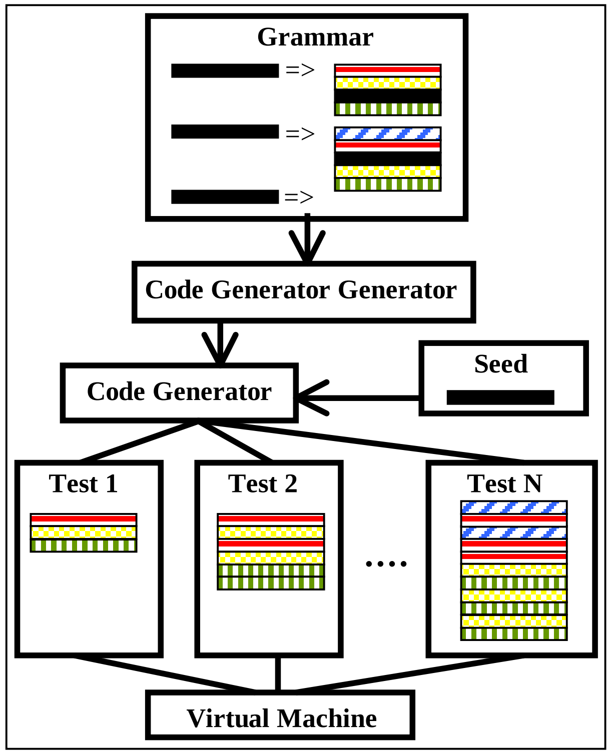


**Figure 1.** The structure of the test generation process. A code-generator-generator parses a production grammar, generates a code-generator, which in turn probabilistically generates test cases based on a seed.

The input to the *lava* code-generator-generator consists of a conventional grammar description that resembles the LALR grammar specification used in **yacc** (Figure 2). This context-free grammar consists of productions with a left-hand side (LHS) containing a single non-terminal that is matched against the input. If a match is found, the right-hand side (RHS) of the production replaces the LHS. As with traditional parser systems, we use a two-phase approach in *lava* for increased efficiency and ease of implementation. The code-generator-generator converts the grammar specification into a set of action tables, and generates a code-generator that performs the actual code production based on a given seed. In essence, the code-generator-generator performs code specialization on a generic production system for a particular grammar. We picked this two-phase approach to increase the efficiency of the code-generator through program specialization.

Within the code-generator, the main data structure is a newline-separated stream, initially set to correspond to the seed input. Being able to specify arbitrary seeds enables us to modify parts of existing programs in controlled ways. Each line in this stream is scanned, and occurrences of an LHS are replaced by the corresponding RHS. If there is more than one possible action for a given match, the code-generator picks an outcome probabilistically, based on weights that can be associated with each production. When all possible non-terminals are exhausted, the code-generator outputs the resulting stream. We then use the Jasmin bytecode assembler [Meyer & Downing 97] to go from the textual representation to the binary bytecode format.

```
Grammar-Rule := name [limit]
    identifier "=>" identifier*
                    [guard] [action]
Identifier := Token | Variable
Token := [a-zA-Z_][a-zA-Z0-9_]*
Variable := "$" Token
Guard := "guard{" code "}"
Action := "action{" code "}"
```

**Figure 2.** The grammar for the *lava* input language in EBNF form. Its simplicity makes the language easy to adopt, while variable substitution and arbitrary code sequences make it powerful.

*Lava* grammars can be annotated with three properties that are useful for test generation. First, each production rule can have an associated name. Along with each test case, the code-generator creates a summary file listing the names of the grammar rules that gave rise to that test case, thereby simplifying test selection and analysis. We used these summary files extensively during JVM development to pick (or to skip) test cases that exhibit certain instruction sequences, as the high-level descriptions were easier to act on than reverse engineering and pattern-matching instruction sequences. In addition to a name, each grammar rule in *lava* may have an associated limit on how many times it can be exercised. Java, unlike most language systems, enforces certain structural limits, such as limits on code length per method, in addition to any limits the tester may impose to guide the test generation process. The limits on production rules enable the grammar to accommodate such constraints. Finally, in order to enable the production of context-sensitive outputs, *lava* allows an optional code fragment, called an action, to be associated with each production. This code is executed when the production is selected during code-generation, in a manner analogous to context-sensitive parsing using **yacc** grammars. An escape sequence facilitates the substitution of variables defined in actions on the RHS of productions. Actions are widely used in this manner to generate labels, compute branch targets, and perform assignment of values to unique local variables. The code in action blocks, like the entire *lava* system, is written in AWK [Aho et al. 88].

In addition to these attributes, each production carries a weight that determines which productions fire in case more than one are eligible for a given LHS. This case occurs frequently in code production, as it is quite often possible to insert just about any instruction sequence at any point within a program. The probabilistic selection based on weights, then, introduces non-determinism and enables each code-generator run to produce a different test case. The weights, specified as part of the seed input, determine the relative occurrences of the constructs that appear in the grammar. Separating the weights from the rest of the grammar enabled us to generate different instruction mixes without having to rewrite or rearrange the grammar specification.

```
insts 5000 INSTS => INST INSTS
emptyinst  INSTS => /* empty */
ifeqstmt
INST => iconst_0; INSTS; ifeq L$1;
        jmp L$2; L$1: INSTS; L$2:
jsrstmt
INST => jsr L$1; jmp L$2;
        L$1: astore $reg; INSTS;
        ret $reg; L$2:
action{ ++reg; }
```

```
Weight jsrstmt 10
Weight ifeqstmt 10
Weight insts 1
Weight emptyinst 0

.class public testapplet$NUMBER

.method test()V
  INSTS; return
.end method
```

**Figure 3.** A simplified *lava* grammar and a corresponding seed for its use. Non-terminals are in upper case and dollar signs indicate that the value of the variable should be substituted at that location. Weights in the seed specify the relative mix of **jsr** and **if** statements. Overall, the grammar specification is concise and well-structured.

Figure 3 shows a sample grammar written in *lava*. This particular grammar is intended to exercise various control flow instructions in Java virtual machines, and is a simplified excerpt from a larger grammar that covers all possible outcomes for all branch instructions in bytecode. The **insts** production has a specified limit of 5000 invocations, which restricts the size of the generated test case. The two main productions, **jsrstmt** and **ifeqstmt**, generate instruction sequences that perform subroutine calls and integer equality tests. These concise descriptions, when exercised on the seed shown, generate a valid class file with complicated branching behavior. The seed itself is a skeletal class file whose code segment is to be filled in by the grammar. Equal weighting between the **if** and the **jsr** statements ensures that they are represented equally in the test cases. A weight of 0 for the **emptyinst** production effectively disables this production until the limit on the **insts** production is reached, ensuring that test generation does not terminate early. The grammar also illustrates the use of unique labels and the integration of actions with the grammar in order to create context-sensitive code. While the lava input language is quite general and admits arbitrarily complex grammar specification, we found that most of the grammars we wrote used common idioms which reflected the constructs we wished to test, as in the example above. Figure 4 contains a sample output generated by this toy grammar that was designed to generate "spaghetti code." Overall, the description language is concise, the test generation process is simple, and the generated tests are complex.

```
    iconst_0
    iconst_0
    jsr L18
    jmp L19
L18:astore 3
    ret 3
L19:ifeq L4
    jmp L5
L4:
L5: iconst_0
    ifeq L10
    jmp L11
L10:
L11:ifeq L0
    jmp L1
L0: jsr L6
    jmp L7
L6: astore 1
    ret 1
L7: jsr L12
    jmp L13
L12:astore 2
    ret 2
L13:
L1: jsr L2
    jmp L3
L2: astore 0
    iconst_0
    ifeq L14
    jmp L15
L14:
L15:ret 0
L3: iconst_0
    ifeq L8
    jmp L9
L8:
L9: iconst_0
    ifeq L16
    jmp L17
L16:
    return
```

**Figure 4.** A sample method body produced by the grammar shown in Figure 3. The resulting test cases are complex, take very little time to produce, and are more reliable than tests written by hand.

Initially, we considered *lava*'s limitation of left-hand sides to a single non-terminal as a serious restriction that would need to be addressed. We found, however, that cases that warrant multiple non-terminals on the LHS arise rarely in practice. In contrast, we found that there were numerous cases where a context free grammar would indicate that a given production is eligible for a particular location, but context-sensitive information prohibited such an instruction from appearing there. For example, the use of an object is legal only following proper allocation and initialization of that object. While it is possible to keep track of object initialization state by introducing new non-terminals and rules into the grammar, this overly complicates grammar specifications, and scales badly with the number of objects to be tracked. It is significantly simpler to keep track of such state in auxiliary data structures through actions, much like in **yacc** grammars. However, since these data structures are not visible to the code production engine, the code-generator may lack the information it needs to decide for which spots certain productions are not eligible. To address this problem, we introduced guards into the grammar specification. A guard is a Boolean function that is consulted to see if a given production can be used in a particular context. If the guard returns false, that replacement is not performed, and another production is selected probabilistically from the remaining set eligible for that substitution. Hence, guards enable the context-sensitive state that is tracked through actions to influence the choice of productions. While it is possible to use guards to associate probabilistic weights with productions, we decided to retain explicit support for weights in the language. Weights are such a common idiom that special-casing their specification both greatly simplifies the grammars, and increases the performance of the code generator. Further, using guards to implement weights would encode probabilities in the grammar specification, whereas in our implementation, they are specified on a per-seed basis, obviating the need for testers to modify the grammar specifications.

## Results

We have used *lava* in a number of different testing scenarios including performance analysis and fault detection of the virtual machine, and integrity of code transformation engines such as compilers and binary rewriters.

First, we used the test cases to test for easy-to-detect errors, such as system crashes. Typesafe systems such as virtual machines are never supposed to crash on any input. A crash indicates that typesafety was grossly violated, and that the error was luckily caught by the hardware protection mechanisms underneath the virtual machine. We used a simple grammar with three productions, and the very first test case that we generated found that the Sun JDK1.0.2 verifier on the DEC Alpha would dump core when faced with numerous deeply nested subroutine calls.

Second, we used stylized test cases generated by *lava* for characterizing the time complexity of our verifier as a function of basic block size and total code length. The parameterizable nature of the grammar facilitated test case construction, and only required a few runs of the code-generator with different weights and seeds. It took less than 10 minutes to write the grammar from scratch, and only several minutes to generate six test cases ranging in size from a few instructions to 60000. Each generated test case was unique, yet was comparable to the other tests on code metrics such as average basic block size and instruction usage. These tests uncovered that our verifier took time $O(N^2 \log N)$ in the number of basic blocks in a class file, and motivated us to find and fix the unnecessary use of ordered lists where unordered ones would have sufficed.

Finally, we used the generated tests to verify the correctness of Java components that perform transformations. Components such as Java compilers and binary rewriting engines [Sirer et al. 98] must retain program semantics through bytecode to native code or bytecode to bytecode transformations. Such components can be tested by submitting complex test cases to the transformations, and comparing the behavior of the transformed, e.g. compiled, program to the original. In our case, we wanted to ensure that our binary rewriter, which performs code movement and relocation, preserved the original program semantics through its basic block reordering routine. Our binary rewriter is a fairly mature piece of code, and we had not discovered any bugs in it for a while. To test the rewriter, we created 17 test cases which averaged 1900 instructions in length and exercised every control flow instruction in Java. The grammar was constructed to create test cases with both forward and backward branches for each type of jump instruction, and instrumented to print a unique number in each basic block. We then simply executed the original test cases, captured their output and compared it to the output of the test cases after they had been transformed through the binary rewriter. This testing technique caught a sign extension error that incorrectly calculated backward jumps in **tableswitch** and **lookupswitch** statements, and took a total of two days from start to finish.

Though production grammars facilitate test case generation, they do not by themselves provide a complete solution to software testing. The main problem in software testing has to do with determining when the tested system is behaving correctly on a given input, and when it is not. In the next section, we illustrate how this problem can be addressed by comparative evaluation.

## 4. Comparative Testing with Variations

Our first approach to addressing the oracle problem, that is, the problem of detecting when a virtual machine is misbehaving on a given input, is to rely on multiple implementations of a virtual machine to act as references for each other in a process commonly known as comparative, or differential, testing. The core idea in comparative testing is simply to direct the same test cases to two or more versions of a virtual machine, and to compare their outputs. A discrepancy indicates that at least one of the virtual machines differs from the others, and typically requires human involvement to determine the cause and severity of the discrepancy. Since the Java virtual machine interface is a de facto standard for which multiple implementations are available, we decided to use it to address the oracle problem for automatically generated test cases.

A separate reason compelled us to expand comparative testing by introducing variations into the test cases generated by production grammars. A variation is simply a random modification of the test case to generate a new test. The assembly language in which the test cases were generated hide some of the details of the Java bytecode format, making it impossible to exercise

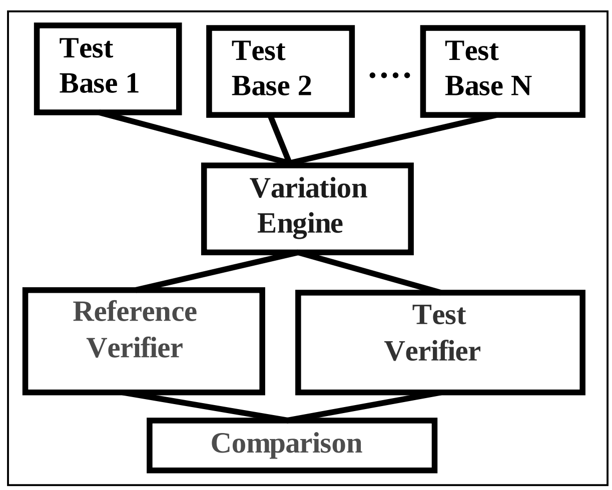


**Figure 5**. Comparative Evaluation. A variation engine injects errors into a set of test bases, which are fed to two different bytecode verifiers. A discrepancy indicates an error, a diversion from the specification, or an ambiguity in the specification.

certain conditions. For example, there is no way to generate a class file which exhibits internal consistency errors, such as out of bounds indices, using an assembly language such as Jasmin. To overcome this difficulty of manipulating low-level constructs from assembly language, we decided to introduce variations into the tests generated by *lava*. The overall process we followed is illustrated in Figure 5.

We compared our virtual machine implementation to Sun JDK 1.0.2 and the Microsoft Java virtual machine found in Internet Explorer 4.0. We formed the test inputs by taking the output of a production grammar and introducing one-byte pseudo-random modifications, or variations. We experimented with introducing multiple modifications into a single test base, but found that multiple changes made post-mortem investigation of discrepancies more difficult. Therefore, each test case was generated by introducing only a single one-byte value change at a random offset in a base class file.

In most cases, all consulted bytecode verifiers agreed that the mutated test inputs were safe or unsafe, yielding no information about their relative integrity. In some cases, however, one of the bytecode verifiers would disagree from the rest. Cases where the class file was accepted by our bytecode verifier but rejected by a commercial one usually pointed to an error in our implementation. We would fix the bug and continue. Conversely, cases where the class file was rejected by our bytecode verifier but accepted by a commercial bytecode verifier usually indicated an error in the commercial implementation. Occasionally, the differences were attributable to ambiguities in the specification or to benign diversions from the specification.

## Results

We found that the complex test cases generated by production grammars achieved as good as or better code coverage than the best hand-generated tests, had higher value coverage, and were in addition much easier to construct. Figure 6 examines the number of safety axioms in our verifier triggered by a hand-generated test base versus a test base generated by a *lava* grammar. The hand-generated test case was laboriously coded to exercise as many of the features of the Java virtual machine, in as diverse a manner as possible. The graph shows that automatically generated test cases exercise more check instances than hand-generated test cases. Indeed, after 30000 variations, the coverage attained by automatically generated test cases is a strict superset of the manually generated tests. We attribute this to the greater amount of complexity embodied in the test cases generated by the grammar. Further, the tests that are exercised only by grammar-based tests are those cases, such as polymorphic merges between various types at subroutine call sites, for which manual tests are especially hard and tedious to construct.

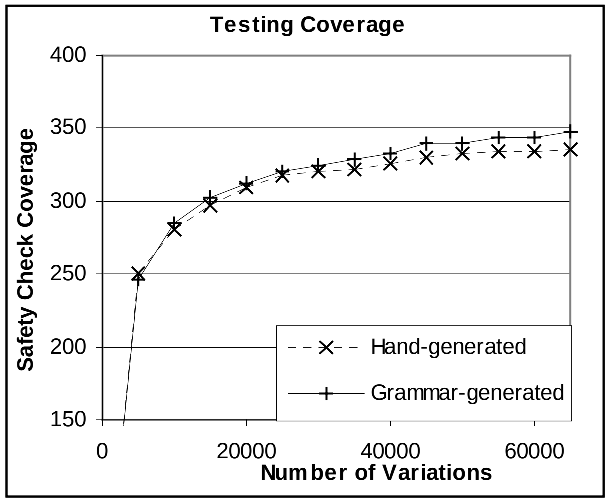


**Figure 6.** A plot of code coverage for hand generated and *lava* generated test cases shows that automatically generated test cases are as effective as carefully constructed hand-generated test cases at achieving breadth of coverage.

Further, we found that comparative testing with variations is fast. At a throughput of 250K bytes per second on a 300 MHz Alpha workstation with 128K of memory on five different base cases, comparative evaluation with variations exercised 75% of the checks in the bytecode verifier within an hour and 81% within a day. Since our test bases consisted of single classes, we did not expect to trigger any link-time checks, which accounted for 10% of the missing checks in the bytecode verifier. Further inspection revealed that another 7% were due to redundant checks in the bytecode verifier implementation. More specifically, there were some checks that would never be triggered if other checks were implemented correctly. For example, long and double operands in Java take up two machine words, and the halves of such an operand are intended to be indivisible. Although various checks in the bytecode verifier prohibit the creation of such operand fragments, our bytecode verifier redundantly checks for illegally split operand on every stack read for robustness.

We attribute this high rate of coverage to three factors. First, Java bytecode representation is reasonably compact, such that small changes often drastically alter the semantics of a program. In particular, constant values, types, field, method and class names are specified through a single index into a constant pool. Consequently, a small variation in the index can alter the meaning of the program in interesting ways, for instance, by substituting different types or values into an expression. For example, one of the first major security flaws we discovered in commercial JVM implementations stemmed from lack of type-checking on constant field initializers. A single variation, within the first 10000 iterations (15 minutes) of testing, uncovered this loophole by which integers could be turned into object references.

A second reason that one-byte variations achieve broad coverage is that the Java class file format is arranged around byte-boundaries. Since the variation granularity matches the granularity of fields in the class file format, a single change affects only a single data structure at a time, and thereby avoids wreaking havoc with the internal consistency of a class file. For example, one test uncovered a case in both commercial bytecode verifiers where the number of actual arguments in a method invocation could exceed the declared maximum. Had this field been packed, the chances of a random variation creating an interesting case would have been reduced. Similarly, random variations uncovered a case in the MS JVM where some branches could jump out of code boundaries, likely due to a sign extension error. Packing the destination offset would have made it harder for single byte variations to locate such flaws.

Finally, we attribute the effectiveness of this approach to the complexity of the test bases generated by *lava* grammars. In testing with one-byte variations, the total complexity of the test cases is limited primarily by the original complexity of the test bases. The grammar based test cases achieve high-complexity with very little human effort.

While comparative testing with variations partially addresses the oracle problem and enables the complex test cases generated by production grammars to be used effectively in testing, it requires another implementation of the same functionality to work. Even though finding alternative implementations was not at all difficult in the case of the Java virtual machine, comparative evaluation is not always the best testing approach. A duplicate may not be available or different implementations may exhibit too many benign differences. Most importantly, even when there is a second implementation to compare against, it may be in error in exactly the same way as the implementation of interest.

## 5. Producing Self-Describing Test Cases

We address the shortcomings of comparative evaluation by extending grammar testing to generate certificates concurrently with test cases. A certificate is a behavioral description that specifies the intended outcome of the generated test case, and therefore acts as an oracle by which the correctness of the tested system can be evaluated in isolation. The insight behind this technique is that certificates of code properties allow us to capture both static and dynamic properties of test programs, such as their safety, side effects or values they compute. The behavior of a virtual machine can then be compared against the certificate to check that the virtual machine is implemented correctly.

**A Sample *Lava* Grammar and Corresponding Behavioral Description**

```
1: STMTS => STMT STMTS
2: STMTS => nil
          : N = λt.t
3: STMT => iconst 0
          : A = λS.λt.(S (cons 0 t))
4: STMT => iconst 1
          : B = λS.λt.(S (cons 1 t))
5: STMT => iadd
          : C = λS.λt.(S (cons
                     (+ (car t) (cadr t)) (cddr t)))
6: STMT => ifeq L_{N+1;}
              L_N: STMTS; goto L_{N+2}
              L_{N+1}: STMTS
              L_{N+2}: STMTS
          : D = λS1.λS2.λS3.λt.(if
                      (equal (car t) 0)
                      (S3 (S1 (cdr t)))
                      (S3 (S2 (cdr t))))
```

**Figure 7.** A sample grammar with a corresponding behavioral description.

Two types of useful certificates may accompany synthetically generated code. The first form of a certificate is a proof over the grammar, which can accompany all test programs generated by that specification as a guarantee that they possess certain properties. For instance, it is easy to see, and inductively prove, that the sample grammar given in Figure 3 will execute all labeled blocks it generates. Using a third production, we can instrument every labeled basic block produced by that grammar, and check to ensure that a virtual machine enters all of the blocks during execution. We simply print the block label when the block is entered, and, at the end of the test run, check to make sure that all blocks appear exactly once in the trace. Using this technique, for instance, we found that some of the control flow instructions in an early version of our interpreter had the sense of their comparisons reversed.

Not all grammars lend themselves to such inductive proofs. We found a second form of certificates, describing the runtime behavior of a specific test, often more applicable. In essence, what the tester needs to overcome the oracle problem is a specification of what the test ought to compute, in a format other than Java bytecode that can be automatically evaluated. We chose to use lambda expressions for specifying such certificates because of their simplicity. Such certificates are difficult to construct by hand, however, as they require reasoning about the composition of the productions that generated that test case. Instead, we generate these certificates concurrently with the test program, based on annotations in the grammar. The annotations, in lambda expression form, are arranged by the programmer such that they capture interesting aspects of corresponding productions. The *lava* system merges these annotations together through the test production process, and generates a resulting lambda expression, which computes the same result as the test case when evaluated. The correctness of a compiler or interpreter can then be checked by executing the test case and comparing with the result of the lambda expression.

We have extended *lava* such that it takes two grammars, one for the intended test language and one for the behavioral description. Whenever a production in the first grammar is chosen, the corresponding production rule in the second grammar is also selected. Test generation thus creates two separate outputs corresponding to the test case and its behavioral description. We have used Scheme for our behavioral description language, and thus can compare the correctness of a Java virtual machine simply by evaluating a test program in Java and comparing its output to that of the Scheme program.

Figures 7 illustrates a behavioral specification for a limited grammar that supports stack push, arithmetic and control flow operations. Our behavioral descriptions use lambda expressions in continuation passing style. Each production is annotated with an expression that takes the current machine state and continuation as input, and performs the operations of the right hand side. Non-terminals appearing on the RHS constitute free variables, and each production step forms a substitution. The overall behavioral description is formed by substituting lambda expressions corresponding to productions for free variables. For instance, suppose that we start with a seed of "STMTS" and pick productions 1, 3, 1, 4, 1, 5, 1, 6, 1, 4, 2, 1, 3, 2, 2. The corresponding behavioral description would be (A (B (C (((D (B N)) (A N)) N)))). This expression, when evaluated on the initial state, yields (1), the result of executing the program. If execution of the generated bytecode disagrees with this result, there is a fault with either the virtual machine implementation or the lambda expression evaluator. Consequently, this approach enables checking the correctness of a JVM against a simple, formally-analyzed and time-tested implementation.

While behavioral descriptions in *lava* resemble operational semantics, they are vastly simpler to specify than a full formal semantics for the Java virtual machine. This is because the behavioral descriptions need to be provided not for every construct in the language, but only for idioms of interest, and need capture only those properties of the idioms in which the tester is interested. For example, providing the correct semantics for each of the Java virtual machine's control flow opcodes, which include conditional branches, jumps, method invocations, subroutine calls and exceptions, is a difficult task [Drossopoulou+ 98]. However, effects of an idiom, say one that consists of a jump, loop, exception raise, compute and return from exception, which uses these opcodes to push a predetermined value onto the stack, is easy to capture in a behavioral description. Essentially, *lava* enables the tester to provide partial operational descriptions at the production level, instead of having to provide complete and correct semantics at the instruction level.

Currently, *lava*'s support for extended grammars is not as general as we would like. The behavioral descriptions are limited to grammars where the main production is right-recursive, as shown in production 1 in Figure 6.

Overall, extended code generation with certificates determines precise properties about test cases without recourse to a second implementation of the same functionality. This powerful technique is widely applicable in testing virtual machine components, including bytecode verifiers, compilers, and interpreters.

## 6. Related Work

Production grammars have been studied from a theoretical perspective, and adopted to a few domains besides testing. [Dershowitz+ 90,Dershowitz 93] give an overview of rewriting systems, of which production grammars are a subclass, and provide a theoretical foundation. In practice, production grammars have been used to describe fractals, and have been adopted in computer graphics to efficiently generate complex visual displays from simple and compact specifications [Prusinkiewicz et al. 88]. Our approach expands this work by applying production grammars to the testing of complex systems, and, within the testing field, overcomes the oracle problem by generating self-describing certificates concurrently with test cases to aid standalone testing.

There has been substantial recent work on formal methods for ensuring correctness of bytecode verifiers, especially on using type-systems as a foundation for bytecode verifier implementations. Stata and Abadi have formally analyzed Java subroutines for type-safety and postulated a type-system that can be used as the basis of a Java verifier [Stata&Abadi 98]. Freund and Mitchell have further extended this model to cover object allocation, initialization and use [Freund&Mitchell 98, Freund&Mitchell 99]. While these approaches are promising and address one of the most safety critical components in a virtual machine, to date, only a subset of the various constructs permitted by the Java virtual machine have been covered by these type system-based frameworks. Further, these approaches operate on abstract models of the code instead of the implementation, and require extensive human involvement to ensure that the model accurately captures the behavior of the bytecode verifier. The strong guarantees of formal reasoning are undermined by the manual mapping of a model onto the actual implementation.

An approach to verification that is common in industry is to perform source code audits, where a group of programmers examine the code and manually check for weaknesses. The Secure Internet Programming group at Princeton has found a number of security flaws in Java implementations using this technique [Dean et al. 97]. The primary shortcoming of manual audits is that they require intensive human labor, and are thus expensive. Further, providing source access to auditors is not always possible or desirable.

Production grammars have been used in conjunction with comparative evaluation to check compiler implementations for compatibility [McKeeman 98]. The author used a stochastic C grammar to generate test cases for C compilers. This work focused primarily on ensuring compatibility between different compilers, and relied on comparative evaluation. This approach resembles our second technique, and suffers from the same problem of requiring a second, preferably stronger implementation of the tested application. We extend this work by generating certificates for test cases that enable the testing of a program without reference to a second implementation. Similarly, [Celentano+ 80] outlines a scheme for generating minimal test cases that cover a BNF grammar, and in their experiences section, mention that the minimal test cases are at times too simple to test interesting inputs. Use of probabilistic productions in our scheme generate much more complex test cases, and provide a point of control for steering the testing effort.

Within the software engineering community, there are various efforts aimed at automatic test generation from formal specifications. [Weyuker et al. 94] describes a set of algorithms for generating tests from boolean specifications. [Mandrioli et al. 95] outline the TRIO system, which facilitates the semi-automatic creation of test cases from temporal logic specifications. [Chang et al. 95] describes a flexible, heuristic-driven, programmable scheme for generating tests from specifications in ADL, a language based on first-order predicate logic. [Gargantini+ 99] illustrates how a model checker can be used to create tests from state machine specifications, and [Bauer&Lamb 79] discusses how to derive test cases from functional requirements for state machines. These efforts all require a formal, modular and accurate specification of the system to be tested in a restrictive formal specification language. Such specifications are not always possible to write for large, complex, andin particular legacy systems, and require expertise to develop and maintain. Further, it is not clear how well the testing process can be steered using these automated testing techniques.

Proof carrying code [Necula 97] and certifying compilers [Necula & Lee 98] resemble extended grammars in that they associate certificates, in their case formal proofs, with code. Proof carrying code associates inherently tamper-proof typesafety proofs with native code, enabling it to be safely integrated into a base system. Certifying compilers are an extension of proof carrying code, where a compiler automatically generates safety proofs along with binary code. The safety proofs are expressed as statements in first-order logic, and are derived and carried forward from the typesafety properties of the source language. Our work shares the same insight that associating mechanically parseable certificates with code is a powerful technique, but differs from certifying compilers in two ways. First, and most fundamentally, we use extended grammars to ascertain the correctness of a virtual machine implementation, whereas certifying compilers ascertain the safety of a compiled application program. And second, certifying compilers require their input to be written in a typesafe language and construct proofs attesting to static properties such as typesafety, whereas we generate behavioral descriptions based on a programmer specified specification, and reason about dynamic program properties such as program results.

## 7. Conclusion

In this paper, we have described *lava*, a language for specifying production grammars, and demonstrated how production grammars can be used in software testing. *Lava* grammars enable a well-structured, manageable and simple testing effort to be undertaken using concise test descriptions. The language facilitates the construction of complex test cases, which can then serve as bases for comparative testing. An extended form of the *lava* language can not only generate interesting test cases, but also generate certificates about their behavior. Overall, these techniques enable the verification of large and complex software systems, such as virtual machines, cheaply and effectively.

Systems get safer only if they are designed, developed and refined with testing in mind, and only if the requisite testing infrastructure is in place to guide this effort. We hope that as virtual machines, which have promised safety but not yet delivered, become more pervasive, their growth is accompanied by the adoption of automated verification techniques.

## Acknowledgements

We would like to thank Eric Hoffman and the anonymous reviewers for their comments on earlier drafts of this paper. Arthur J. Gregory and Sean McDirmid implemented parts of our Java virtual machine. We would also like to thank Jim Roskind of Netscape, Charles Fitzgerald and Kory Srock of Microsoft, and Tim Lindholm, Sheng Liang, Marianne Mueller and Li Gong of Javasoft for working with us to address the Java security flaws found in commercial verifiers by this work.

## References

[Adl-Tabatabai et al. 96] Adl-Tabatabai, A., Langdale, G., Lucco, S. and Wahbe, R. “Efficient and Language-Independent Mobile Programs." In Conference on Programming Language Design and Implementation, May 1996, p. 127-136.

[Aho et al. 88] Aho, A.V., Kernighan, B.W., Weinberger P. J. The Awk Programming Language. Addison-Wesley, June 1988.

[Bauer&Lamb 79] Bauer, J.A. and Finger, A.G. “Test Plan Generation Using Formal Grammars.” In Proceedings of the 4th International Conference on Software Engineering. IEEE, New York, NY, USA, 1979, pp.425-432.

[Berners-Lee et al. 96] Berners-Lee, T., Fielding, R., Frystyk, H. "Informational RFC 1945 - Hypertext Transfer Protocol -- HTTP/1.0," Request for Comments, Internet Engineering Task Force, May 1996.

[Celentano+ 80] Celentano, A., Crespi-Reghizzi, S., Vigna, P. D., Ghezzi, C., Granata, G., and Savoretti, F. “Compiler Testing Using a Sentence Generator.” In Software: Practice & Experience. 10(11), 1980, pp. 897-918.

[Chang et al. 95] Chang, J., Richardson, D. J., and Sankar, S. “Automated Test Selection from ADL Specifications." In the First California Software Symposium (CSS'95), March 1995.

[Dean et al. 97] Dean, D., Felten, E. W., Wallach, D. S. and Belfanz, D. “Java Security: Web Browers and Beyond.” In Internet Beseiged: Countering Cyberspace Scofflaws, D. E. Denning and P. J. Denning, eds. ACM Press, October 1997.

[Dershowitz+ 90] Dershowitz, N. and Jouannaud, J. Rewrite Systems. In Handbook of Theoretical Computer Science.Volume B: Formal Methods and Semantics. Van Leeuwen, ed. Amsterdam 1990.

[Dershowitz 93] Dershowitz, N. A Taste of Rewrite Systems. In Lecture Notes in Computer Science 693, 199-228 (1993).

[Drossopoulou+ 97] Drossopoulou, S., Eisenbach, S. “Java Is Typesafe – Probably.” In 11th European Conference on Object Oriented Programming, June 1997.

[Drossopoulou+ 98] Drossopoulou, S., Eisenbach, S. “Towards an Operational Semantics and Proof of Type Soundness for Java.” March 1998, to be published. http://www-dse.doc.ic.ac.uk/projects/slurp/pubs/chapter.ps.

[Drossopoulou+ 99] Drossopoulou, S., Eisenbach, S. and Khurshid, S. “Is the Java Typesystem Sound ?” In Theory and Practice of Object Systems, Volume 5(1), p. 3-24, 1999.

[Freund&Mitchell 98] Freund, S. N. and Mitchell, J. C. “A type system for object initialization in the Java Bytecode Language.” In ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, 1998.

[Freund&Mitchell 99] Freund, S. N. and Mitchell, J. C. “A Type System for Java Bytecode Subroutines and Exceptions.” To be published.

[Gargantini+ 99] Gargantini, A. and Heitmeyer, C. "Using Model Checking to Generate Tests from Requirements Specifications." In Proceedings of the Joint 7th Eur. Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Toulouse, France, September 1999.

[Griswold] Griswold, D. “The Java HotSpot Virtual Machine Architecture.” http://www.javasoft.com/products/hotspot/whitepaper.html

[Inferno] Lucent Technologies. Inferno. http://inferno.bell-labs.com/inferno/

[Lindholm&Yellin 99] Lindholm, T. and Yellin, F. The Java Virtual Machine Specification, 2nd Ed. Addison-Wesley, 1999.

[Mandrioli et al. 95] Mandrioli, D., Morasca, S., and Morzenti, A. “Generating Test Cases for Real-Time Systems from Logic Specifications.” ACM Transactions on Computer Systems. Vol.13, no. 4, November 1995, pp. 365-398.

[McGraw & Felten 96] McGraw, G. and Felten, E. W. Java Security: Hostile Applets, Holes and Antidotes. John Wiley and Sons, New York, 1996.

[McKeeman 98] McKeeman, W. M. “Differential Testing for Software.” Digital Technical Journal Vol 10, No 1, December 1998.

[Meyer & Downing 97] Meyer, J. and Downing, T. Java Virtual Machine. O’Reilly, March 1997.

[Muller et al. 97] Muller, G., Moura, B., Bellard, F., Consel, C. “Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code.” In Proceedings of the Third Conference on Object Oriented Technologies, 1997.

[Necula 97] Necula, G. “Proof Carrying Code.” In Principles of Programming Languages. Paris, France, January 1997.

[Necula & Lee 98] Necula, G. and Lee, P. “The Design and Implementation of a Certifying Compiler.” In Programming Languages, Design and Implementation. Montreal, Canada, June1998.

[Oracle] Oracle Corporation. Oracle Application Server Release 4.0 Documentation. http://technet.oracle.com/doc/was.htm

[Proebsting et al. 97] Proebsting, T. A., Townsend, G., Bridges, P., Hartman, J.H., Newsham, T. and Watterson S. A. “Toba: Java for Applications: A Way Ahead of Time (WAT) Compiler.” In Proceedings of the Third Conference on Object Oriented Technologies, 1997.

[Prusinkiewicz et al. 88] Prusinkiewicz, P., Lindenmayer, A., Hanan, J. “Developmental Models of Herbaceous Plants for Computer Imagery Purposes.” In Computer Graphics, 22 (4), August 1988.

[Sirer et al. 98] Sirer, E. G., Grimm, R., Bershad, B. N., Gregory, A. J. and McDirmid, S. “Distributed Virtual Machines: A System Architecture for Network Computing.” European SIGOPS, September 1998.

[Stata&Abadi 98] Stata, R. and Abadi, M. “A type system for Java bytecode subroutines.” In Proceedings of the 25th ACM Principles of Programming Languages. San Diego, California, January 1998, p. 149--160.

[SunJWS] Sun. Java Web Server. http://www.sun.com/software/jwebserver/index.html

[Syme 97] Syme, D. “Proving Java Type-Soundness.” University of Cambridge Computer Laboratory Technical Report 427, June 1997.

[Weyuker et al. 94] Weyuker, E., Goradia, T., and Singh, A. “Automatically Generating Test Data From a Boolean Specification.” In IEEE Transactions on Software Engineering, Vol. 20, no. 5 (May 1994), 353-363.