JVer: A Java Verifier

Ajay Chander[1], David Espinosa[1], Nayeem Islam[1],
Peter Lee[2], and George Necula[3]

[1] DoCoMo Labs USA, San Jose, CA
{chander, espinosa, islam}@docomolabs-usa.com
[2] Carnegie Mellon University, Pittsburgh, PA
Peter.Lee@cs.cmu.edu
[3] University of California, Berkeley, CA
necula@eecs.berkeley.edu
fax 408-573-1090

1 Introduction

We describe JVer, a tool for verifying Java bytecode programs annotated with pre and post conditions in the style of Hoare and Dijkstra. JVer is similar to ESC/Java [1], except that: (1) it produces verification conditions for Java bytecode, not Java source; (2) it is sound, because it makes conservative assumptions about aliasing and heap modification; (3) it produces verification conditions directly using symbolic simulation, without an intermediate guarded-command language; (4) by restricting predicates to conjunctions of relations between integers, it produces verification conditions that are more efficient to verify than general first-order formulae; (5) it generates independently verifiable proofs using the Kettle proof-generating theorem prover [2].

We initially designed JVer as a tool for verifying that downloaded Java bytecode programs do not abuse the computational resources available on a cell phone [3]. These resources include physical resources such as CPU, memory, storage, and network bandwidth, and virtual resources such as handles and threads. However, since JVer uses standard pre and post conditions, it has many uses not limited to resource certification, such as bug finding and security hole detection. We describe JVer's implementation, as well as an experiment using it to limit the resources consumed by a cell phone version of `tetris`.

2 Verifier

Figure 1 shows our annotation language, which is a subset of JML. It includes the usual Hoare-style pre and post conditions, global invariants, loop invariants, and side-effect annotations. The `exsures` annotation means that the method terminates with an exception of the given class.

Predicates are conjunctions of literals. Literals are of the form $e_0 \geq e_1$ or $e_0 = e_1$. Expressions include only the usual Java operators, without method calls. Expressions can refer to class fields and instance fields. Expressions in post conditions can include the keyword `result` to refer to the method's return value (in `ensures`) or thrown exception (in `exsures`).

JVER uses `true` as the default loop invariant. If necessary for verification, the user must supply a stronger invariant, which is located by program counter value. In this respect, JVer differs from ESC/Java, which unrolls loops a fixed number of times and is therefore unsound.

```
annotation ::=
   invariant    pred        |
   ghost        class.field |
   static ghost class.field |
   type class.method (type argument, ...)
     method-annotation*

method-annotation ::=
   requires       pred       |
   ensures        pred       |
   exsures        class pred |
   loop_invariant pc, pred

pred ::= literal ∧ · · · ∧ literal

literal ::= exp relop exp

exp ::= int | argument | class.field | exp.field |
    exp binop exp | \old(exp) | \result

relop ::= = | != | < | <= | > | >=

binop ::= + | - | * | / | % | << | >> | >>> | & | | | ^
```

Fig. 1. Annotation definition

To verify Java bytecode, we use a standard verification condition generator (VCG) based on weakest pre conditions. The verifier begins at the start of a method and at each loop invariant and traces all paths through the code. Each path must terminate either at the end of the method, or at a loop invariant. If a path loops back on itself without encountering a loop invariant, the verifier raises an error and fails to verify the program.

Along each path, the verifier begins with a abstract symbolic state containing logical variables for the method's arguments and for all class fields. It simulates the bytecode using a stack of expressions. At the end of the path, it produces the VC that if the pre condition (or initial loop invariant) holds of the initial state, and all of the conditionals hold at their respective intermediate states, then the post condition (or final loop invariant) holds of the final state.

The VC for the program is the conjunction of the VCs for the methods. The VC for the method is the conjunction of the VCs for the execution paths. The VC for each path is an implication between conjunctions of literals, of the form

$$a_1 \wedge \cdots \wedge a_m \Rightarrow b_1 \wedge \cdots \wedge b_n$$

where the a_i and b_i are literals. This implication is valid if and only if

$$a_1 \wedge \cdots \wedge a_m \wedge \neg b_i$$

is unsatisfiable for each b_i, which we check with a decision procedure for satisfiability of conjuncts of literals. In essence, we check the original formula for validity by converting it to CNF [4].

2.1 Java Features

Java includes several features that make it more difficult to verify than a hypothetical "simple imperative language": concurrency, exceptions, inheritance, and the object heap. We address these issues in turn.

Concurrency. Since most cell phone applets are single-threaded, JVER does not handle concurrency. In particular, we assume that each method has exclusive access to shared data for the duration of its execution. In contrast, ESC/Java detects unprotected shared variable access and discovers race conditions using a user-declared partial order.

Exceptions. Java has three sources of exceptions: explicit `throw` instructions, instructions that raise various exceptional conditions, such as `NullPointer` or `ArrayIndexOutOfBounds`, and calling methods that themselves raise exceptions. Since our control flow analyzer produces a *set* of possible next instructions, it handles exceptions without difficulty. In essence, an exception is a form of multi-way branch, like the usual conditionals, or the JVM instructions `tableswitch` and `lookupswitch`. ESC/Java provides essentially the same support for exceptions.

Inheritance. If class `B` extends class `A`, then when invoking method `m` on an object of class `A`, we may actually execute `B.m` instead of `A.m`. Thus, the pre condition for `B.m` must be weaker than the pre condition for `A.m`, while the post condition for `B.m` must be *stronger* than the post condition for `A.m`. Thus, when we compute the post condition for `B.m`, we conjoin the post condition for `A.m`. And when we compute the pre condition for `A.m`, we conjoin the pre condition for `B.m`. Thus, we inherit post conditions *downwards* and pre conditions *upwards*. On the other hand, if class `B` defines method `m`, but class `A` does not, then we do not inherit pre or post conditions in either direction.

Inheritance of pre and post conditions is convenient, because we can state them just once, and JVer propagates them as necessary. However, to determine the specification for a method, we need the specification for the methods related to it by inheritance, both up *and* down. However, once we know its specification, we can verify each method in isolation. ESC/Java inherits pre conditions *downwards*, which is unsound in the presence of multiple inheritance via interfaces.

Object Heap. At the moment, we use Java's type system to automatically over-estimate the set of heap locations that each method modifies. That is, we assume that the assignment `a.x = e` modifies the `x` field of all objects whose type is compatible with `a`. We also determine automatically which static class variables each method modifies. In the future, we plan to experiment with more precise alias analysis algorithms. If the user requires more precise modification information, he can declare explicitly in the post condition that `x = \old(x)`. ESC/Java requires the user to state explicitly which heap locations each method modifies, but since it does not verify this information, its heap model is unsound.

3 Applications

We are using JVer to enforce resource bounds on downloaded cell phone applets using proof-carrying code [3]. Thus, we need a prover that can generate proofs and a small, fast verifier that can check them on the handset.

Our resource-verification technique uses a static ghost variable `pool` to ensure that the applet dynamically allocates the resources that it uses. Allocations increment `pool`, while uses decrease it. We use JVer to check the invariant that `pool` remains non-negative.

In an experiment, we verified the security of a Tetris game / News display cell phone applet running on DoCoMo's DoJa Java library. The security policy limited the applet's use of the network, persistent storage, and backlight. The 1850-line applet required 111 lines of annotation and verified in less than one second. By checking network use once per download of the news feed rather than once per byte, we reduced the number of dynamic checks by a factor of roughly 5000.

4 Conclusion

Unlike ESC/Java, JVer is sound, simple, efficient, and produces independently-verifiable proofs from Java bytecode, not source. It accomplishes these goals by restricting the properties that it checks and by requiring more user-supplied annotations. We have found in practice that JVer is a useful and efficient tool for verifying properties of cell phone applets.

References

1. Flanagan, C., Leino, R., Lilibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. In: Programming Language Design and Implementation, Berlin, Germany (2002)
2. Necula, G.C., Lee, P.: Efficient representation and validation of proofs. In: Logic in Computer Science, Indianapolis, Indiana (1998)
3. Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.: Enforcing resource bounds via static verification of dynamic checks. In: European Symposium on Programming, Edinburgh, Scotland (2005)
4. Paulson, L.: ML for the Working Programmer. Cambridge University Press (1996)