SRC Technical Note
1999 - 002

21 May 1999

Checking Java programs via guarded commands

K. Rustan M. Leino, James B. Saxe, and Raymie Stata

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Abstract

This paper defines a simple guarded-command–like language and its semantics. The language is used as an intermediate language in generating verification conditions for Java. The paper discusses why it is a good idea to generate verification conditions via an intermediate language, rather than directly.

Publication history. This paper appears in *Formal Techniques for Java Programs*, workshop proceedings. Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors. Technical Report 251, Fernuniversität Hagen, 1999. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

Checking Java programs via guarded commands

K. Rustan M. Leino, James B. Saxe, and Raymie Stata
Compaq Systems Research Center
{rustan,saxe,stata}@pa.dec.com

0 Introduction

It is well-known that the later a software error is detected, the more expensive it is to correct. The Extended Static Checker for Java (ESC/Java) is a tool for finding, by static analysis, common programming errors normally not detected until run-time, if ever [3]. ESC/Java takes as input a Java program, possibly including user annotations, and produces as output a list of warnings of potential errors. It does so by deriving a *verification condition* for each routine (method or constructor), passing these verification conditions to an automatic theorem-prover, and post-processing the prover's output to produce warnings from failed proofs.

Deriving verification conditions for a practical language, rather than for a toy language, can be complex. Furthermore, in designing a tool for automatic checking, one faces trade-offs involving the frequency of spurious warnings, the frequency of missed errors, the efficiency of the tool, and the effort required to annotate programs. To explore and exploit these trade-offs flexibly, it must be easy to change the verification conditions generated by the tool.

To manage the complexity and achieve flexibility, we chose to derive verification conditions by first translating the source language into a simple intermediate *guarded-command* language, and then using the semantics of this guarded-command language to produce verification conditions. In this paper, we describe our intermediate language, give its semantics, and discuss how it is used in our tool.

1 Translation stages

Our translation from Java to verification conditions is broken into three stages. First, we translate from Java to a sugared form of our guarded-command language that includes high-level features such as iteration and method invocation. Second, we desugar the sugared guarded commands into primitive guarded commands. Finally, we compute verification conditions from these primitive guarded commands.

In the translation from Java into sugared guarded commands, we eliminate many of the complexities found in Java, such as `switch` statements and expressions with side effects. This part of the translation is bulky and tedious. We have designed the sugared guarded-command language to make the translation easy to understand and to implement. At the same time, this part of the translation is relatively stable. We find it nice to separate this bulky but stable part of the translation process from other parts that change during experimentation.

The desugaring into primitive guarded commands is where we need a lot of flexibility. This is the principal stage of the translation where we make the kinds of trade-offs mentioned in the introduction. In section 3, we give examples of how different desugarings, possibly chosen under user control, result in different kinds of checking.

The semantics of the primitive guarded-command language is quite simple. Indeed, a naive set of equations for deriving verification conditions from primitive guarded commands fills less than half a page. However, by experimenting with different, but semantically equivalent, equations, we have achieved significant performance gains. Because this stage of the translation begins with such a simple language, we have been able to perform these experiments easily.

2 Primitive guarded-command language

Our primitive guarded-command language is a form of Dijkstra's guarded commands [2], with several important distinguishing features: exceptions [0, 6], partial commands [8, 7], and going wrong (see, *e.g.*, section 6.2 of [4]). (By including partial commands, we no longer need the "guards" that originally gave the language its name.) The syntax of commands in our guarded-command language is as follows:

$$\begin{array}{l} \mathit{cmd} ::= \\ \quad \mathit{variable} = \mathit{expr} \mid \textbf{skip} \mid \textbf{raise} \mid \textbf{assert}\ \mathit{expr} \mid \textbf{assume}\ \mathit{expr} \\ \mid \textbf{var}\ \mathit{variable}^{+}\ \textbf{in}\ \mathit{cmd}\ \textbf{end} \mid \mathit{cmd}\ ;\ \mathit{cmd} \mid \mathit{cmd}\ !\ \mathit{cmd} \mid \mathit{cmd}\ \Box\ \mathit{cmd} \end{array}$$

where an expr is an expression in untyped first-order predicate calculus extended with *labels*. A labeled expression $(\textbf{label}\ L :\ e)$ is semantically equivalent to the expression e, but supplies the label L to the theorem-prover in order to facilitate the production of user-sensible warning messages (see section 6 of [1], which describes an extended static checker for Modula-3).

We model Java instance fields as maps from objects to values. Thus, we translate the Java expression `o.f` into $\mathit{select}(f, o)$, where the select function extracts from map f the component indexed by o.

Unlike Dijkstra's guarded commands, our commands can terminate not only *normally*, but also *exceptionally* and *erroneously*. We use exceptional termination to model Java's exceptions and also Java's control-transfer statements `break`, `continue`, and `return`. We use erroneous termination ("going wrong") to model violations of the programming discipline that ESC/Java checks.

The semantics of our primitive guarded commands is given by their *weakest liberal preconditions*. For any command C and predicates (on the post-state of C) N, X, and W, the predicate $wlp.C.(N, X, W)$ holds in exactly those initial states from which each execution of C either terminates normally in a state satisfying N, terminates exceptionally in a state satisfying X, or terminates erroneously in a state satisfying W (or doesn't terminate at all, but all of our primitive commands do terminate). We define wlp by the following equations:

$$\begin{array}{rcl}
wlp.(v = e).(N, X, W) & \equiv & N[v \leftarrow e] \\
wlp.\mathbf{skip}.(N, X, W) & \equiv & N \\
wlp.\mathbf{raise}.(N, X, W) & \equiv & X \\
wlp.(\mathbf{assert}\ e).(N, X, W) & \equiv & (e \wedge N) \vee (\neg e \wedge W) \\
wlp.(\mathbf{assume}\ e).(N, X, W) & \equiv & e \Rightarrow N \\
wlp.(\mathbf{var}\ v_1 \ldots v_n\ \mathbf{in}\ C\ \mathbf{end}).(N, X, W) & \equiv & \langle \forall v_1 \ldots v_n \triangleright wlp.C.(N, X, W) \rangle \\
wlp.(C0\ ;\ C1).(N, X, W) & \equiv & wlp.C0.(wlp.C1.(N, X, W), X, W) \\
wlp.(C0\ !\ C1).(N, X, W) & \equiv & wlp.C0.(N, wlp.C1.(N, X, W), W) \\
wlp.(C0\ \Box\ C1).(N, X, W) & \equiv & wlp.C0.(N, X, W) \wedge wlp.C1.(N, X, W)
\end{array}$$

where in the equation for the **var** command, $v_1 \ldots v_n$ are distinct variables not occurring free in N, X, or W.

The verification condition for a routine r has the form

$$BP \Rightarrow wlp.C.(true, true, false)$$

where C is the translation of r and BP is the *background predicate*. The background predicate is a set of axioms, derived in part from declarations in the user's program, that encode various properties guaranteed by Java, such as properties of the type system (see [5] for the background predicate of a simple object-oriented language).

3 Sugared guarded-command language

At the outset of our project, we considered it fairly obvious that trying to expand Java directly into verification conditions would result in a software engineering disaster. Introducing a desugaring stage was a less obvious design decision, but one that has turned out to be valuable in managing complexity and maximizing flexibility. In this section, we give examples of constructs in our sugared language.

Checks. To achieve a flexible treatment of conditions such a null dereferences, we use a command called **check**. For example, a Java statement `v = o.f;` on line 27 translates into the sugared commands

> **check** *Null*, 27, o != *null* ;
> $v =$ *select*(f, o)

We have several choices in the desugaring of the **check** command. If we want treat null dereferences as errors, then we desugar the **check** command into

> **assert** (**label** *Null*@27 : o != *null*)

ESC/Java lets users suppress null dereference warnings, either selectively or globally. If null dereference warnings are suppressed on line 27, then the **check** command desugars into

> **assume** o != *null*

Introducing this assumption (instead of, say, desugaring the **check** command into **skip**) prevents ESC/Java from, for example, generating a warning on line 28 if that line contains the dereference `o.g`. (But there are other cases where we do desugar a **check** into **skip**.)

ESC/Java enforces a programming discipline in which null dereferences are considered to be errors. If we wanted to support a programming style in which the programmer might intentionally dereference null and then handle the resulting Java exception, we would desugar the **check** command into something like

> (**assume** $o ==$ *null* ; ... ; **raise**) $\Box$ **assume** o != *null*

where the "..." elides the commands that make the subsequent **raise** model the raising of a new *NullPointerException*.

Loops. The translation of Java `while`, `do`, and `for` loops produces commands that contain a sugared command of the form

> **loop** { **invariant** J } C **end**

In contrast to exiting the loop when C can no longer be executed [8], control exits this loop when C raises an exception. The usual way of defining *wlp* for loops involves a strongest fixed point. We approximate this fixed point by considering only executions that iterate at most once. That is, we desugar the **loop** command into

> **check** *LoopInvInit*, *loc*, J ;
> C ;
> **check** *LoopInvMaintained*, *loc*, J ;
> **assume** *false*

where *loc* is the source code location of the Java loop statement. While this approximation is coarse, we have found that it still allows the checker to find many program errors, even when J is the trivial invariant *true* (see section 9 of [1]). By translating Java loops into commands that contain **loop** commands, we retain the flexibility to try different desugarings. For example, we could unroll a **loop** two or more times. Or, we could produce a conservative desugaring of the form

> **check** *LoopInvInit*, *loc*, J ; **assume** *false*
> $\square$
> . . . ; **assume** J ; C ; **check** *LoopInvMaintained*, *loc*, J ; **assume** *false*

where the "…" assigns arbitrary values to the assignment targets of the loop. Lastly, note that our translation retains the flexibility of strengthening any programmer-declared invariant with any kind of inferred invariants, for which the literature offers numerous techniques.

Calls. Our sugared language also contains a **call** command, whose desugaring depends on the specification of the routine being called. Roughly speaking, **call** $r(\, e0, e1 \,)$, where routine r is allowed to modify x, desugars into a command of the form

> **var** $p0\; p1$ **in**
> $p0 = e0$; $p1 = e1$; **check** . . . *preconditions* . . . ;
>   **var** x_0 **in** $x_0 = x$; **modify** x ; **assume** . . . *postconditions* . . . **end** ; . . .
> **end**

where **modify** x is a sugared command that desugars into

> **var** x' **in** $x = x'$ **end**

The actual desugaring of **call** is more complicated. For example, result values and exceptions must be treated, and postconditions include both user-declared conditions and conditions guaranteed by Java.

The **modify** command uses the nondeterminism inherent in the primitive **var** command. In the desugaring of **call** (and also elsewhere in our translation), we use **assume** commands to restrict that nondeterminism. (Our translation uses the nondeterminism only of the **var** command, never of the $\Box$ command. Whenever our translation generates a $\Box$ command, the enabling conditions of the subcommands are mutually exclusive.)

4 Conclusions

Generating verification conditions for a real-world language like Java is a significant engineering challenge. Such languages provide many programmer conveniences that make the derivation bulky and tedious. Also, finding the right derivation is as much an art as a science, an art involving much trial-and-error. Thus, it is important to appropriately separate concerns both to manage complexity and to maximize flexibility. In building the ESC/Java verification condition generator, we have applied this principle in decomposing the verification condition generation into a three-stage process that seems to have served us well.

History and acknowledgements. ESC/Java was built by Cormac Flanagan, Mark Lillibridge, Greg Nelson, and the authors. Greg Nelson first suggested verification condition generation via guarded commands, almost a decade ago. Subsequently, this became the basis for the ESC/Modula-3 verification condition generator, written initially by Damien Doligez and then mainly by Dave Detlefs.

References

[0] Flaviu Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163–174, 1984.

[1] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, December 1998. Available from `www.research.digital.com/SRC/publications/src-rr.html`.

[2] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[3] Extended Static Checking home page, Compaq Systems Research Center. On the Web at `www.research.digital.com/SRC/esc/Esc.html`.

[4] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, Pasadena, CA 91125, January 1995. Technical Report Caltech-CS-TR-95-03.

[5] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `www.cs.williams.edu/~kim/FOOL/FOOL4.html`.

[6] M. S. Manasse and C. G. Nelson. Correct compilation of control structures. Technical report, AT&T Bell Laboratories, September 1984.

[7] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[8] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.