Avoiding Exponential Explosion: Generating Compact Verification Conditions

Cormac Flanagan James B. Saxe
Compaq Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94301

Abstract

Current verification condition (VC) generation algorithms, such as weakest preconditions, yield a VC whose size may be exponential in the size of the code fragment being checked. This paper describes a two-stage VC generation algorithm that generates compact VCs whose size is worst-case quadratic in the size of the source fragment, and is close to linear in practice.

This two-stage VC generation algorithm has been implemented as part of the Extended Static Checker for Java. It has allowed us to check large and complex methods that would otherwise be impossible to check due to time and space constraints.

1 Introduction

Developing a reliable software system is a challenging task. A promising technique for ensuring software reliability is to verify statically the absence of as many errors as possible. Type systems have been successful at statically detecting certain kinds of errors, for example, applying a function to an incorrect number of arguments.

The goal of the Extended Static Checking project [DLNS98, LSS99, Ext] is to detect statically additional errors that are normally detected only at run-time, if ever. Such errors include null pointer dereferences, array bounds errors, division by zero, and the violation of programmer-specified properties such as method preconditions, method postconditions, and object invariants. Performing this kind of checking requires detailed reasoning about both the semantics of the program fragment being checked and the desired correctness property.

To appear in the Conference Record of POPL '01: The 28th ACM Symposium on the Principles of Programming Languages.

A standard approach for performing this kind of analysis is to split the problem into two stages. The first stage, *VC Generation*, translates a program fragment and its correctness property into logical formula, called a verification condition (VC). The VC has the property that if it is valid then the program fragment satisfies its correctness property. The second stage then uses an automatic decision procedure (see, *e.g.*, [Nel81, DNS01]) to determine the validity of the VC.

This approach works well for smaller or simpler program fragments. However, it is difficult to scale this approach to larger fragments that have more complex control flow, in part due to the extremely large size of the corresponding VCs produced by current VC generation algorithms. These algorithms can yield impractically large VCs for routines that contain only a few dozen lines of code.

For example, the weakest precondition translation [Dij76] produces a VC that may be exponential in the size of the original program fragment. Such large VCs are expensive to create, to store, and to verify. This exponential blow up is not only a theoretical concern; we have observed this behavior in practice on several more complex methods that we have tried to check using the Extended Static Checker for Java (ESC/Java). This situation is unfortunate since these complex methods would benefit most from automated techniques for software defect detection.

Other VC generation algorithms, such as strongest postconditions or symbolic execution [NL98], also suffer from the potential for exponential blow up.

This paper presents a new VC generation algorithm. The algorithm is based on an investigation into the cause of the exponential worst-case behavior of the weakest precondition translation. This investigation reveals that assignment statements are the primary cause of exponential behavior, both directly and through their interaction with control-flow merge points.

Based on this insight, we develop a two-stage VC generation algorithm. The first stage translates a source fragment into an assignment-free, or *passive*, intermediate form. The second stage uses a VC generation technique that is optimized to exploit the assignment-free nature of the passive form. This two-stage VC generation algorithm creates a VC whose size is worst-case quadratic in the size of the source fragment, and in practice appears to be close to linear.

The use of this two-stage VC generation algorithm has significantly improved the performance of ESC/Java. As expected, the smaller VCs are cheaper to create and store. In addition, our automatic decision procedure, Simplify [DNS01], requires less time to verify the validity of these smaller VCs. The combination of these performance improvements has enabled us to check a number of large and complex methods using ESC/Java that we were previously unable to check.

The presentation of our results proceeds as follows: Section 2 reviews ESC/Java, which provides the context for this work. Section 3 introduces the intermediate language that we use as the basis for our development. Section 4 defines the semantics of that language using the weakest precondition translation, and outlines the problems with this translation. Section 5 describes the translation of intermediate language statements into passive form, and Section 6 describes an optimized VC generation algorithm for passive statements. Section 7 extends our development to handle loops and exceptions. Section 8 describes the performance benefits of the two-stage VC generation algorithm in the context of ESC/Java. Section 9 describes related work, and we offer some conclusions in Section 10.

2 Review of ESC/Java

The Extended Static Checker for Java is a tool for finding, by static analysis, common defects in Java programs that are not normally detected until run-time, if ever [LSS99, LNS00]. ESC/Java is publicly available for research and educational purposes [Ext]. It takes as input a Java program, possibly including user annotations, and produces as output a list of possible defects in the program.

The annotations in the input program describe program properties such as method preconditions, method postconditions, and object invariants. These annotations allow ESC/Java to catch software defects using a modular, or method-local, analysis.

During this analysis, ESC/Java verifies that the annotations are consistent with the program, and it also uses the annotations to verify that each primitive operation (such as a dereference operation, an array access, or a type cast) will not raise a run-time exception (as might happen, for example, if a dereferenced pointer is null or if an array index is out-of-bounds). To perform this analysis, ESC/Java first translates each method and its correctness property into a verification condition, and then uses an automatic decision procedure to determine the validity of the verification condition.

Since the properties that it attempts to check are undecidable in the worst case, ESC/Java contains some degree of incompleteness and unsoundness by design. That is, it sometime issues spurious warnings and sometimes fails to detect genuine errors. However, the aspects of ESC/Java's processing that give rise to unsoundness and incompleteness are orthogonal to this paper. Except for the loop desugaring briefly discussed in section 7.1, none of the translations described below are unsound or incomplete.

3 Guarded Commands

Deriving verification conditions for a large and realistic language such as Java is quite complex. To help structure and modularize this translation,

$$
\begin{array}{rclll}
A, B, S & \in & \mathit{Stmt} & ::= & \textbf{assert } e \\
 & & & | & \textbf{assume } e \\
 & & & | & x := e \\
 & & & | & A \,;\, B \\
 & & & | & A \,\square\, B \\
e & \in & \mathit{Expr} & & \text{(expressions)} \\
x & \in & \mathit{Var} & & \text{(variables)}
\end{array}
$$

Figure 1: The guarded command language.

S	$wp.S.Q$
assert e	$e \wedge Q$
assume e	$e \Rightarrow Q$
$x := e$	$Q(x := e)$
$A \,;\, B$	$wp.A.(wp.B.Q)$
$A \,\square\, B$	$wp.A.Q \wedge wp.B.Q$

Figure 2: Weakest-precondition semantics.

ESC/Java first translates each method and its correctness property into an intermediate representation, known as a *guarded command*. This translation eliminates many of the complexities of the Java programming language and incorporates engineering compromises involving soundness and completeness; it is outlined elsewhere [LSS99].

In this paper, we focus on the subsequent translation of the guarded command representation into a verification condition, and, in particular, on how to generate a verification condition that is compact and can be efficiently verified using an automatic decision procedure.

We base our development on the intermediate guarded command language shown in Figure 1. The language is a variation of Dijkstra's guarded commands [Dij76], together with some more recent additions (see, *e.g.*, [Nel89, BvW98]). The language includes assert and assume statements, assignment statements, sequential composition, and demonic (indeterminate) choice. (By including assume statements, we no longer need the "guards" than originally gave the language its name.) The guarded command language used by ESC/Java is a small extension of this language with loops and exceptions; to clarify our presentation we defer discussion of these constructs to Section 7.

3.1 Informal Semantics

The execution of a guarded command statement has two possible outcomes: it may terminate normally (in some program state) or it may "go wrong" due to a failed assertion. The execution of the statement **assert** e terminates normally if the predicate e evaluates to *true* in the current program state, and goes wrong otherwise. The assume statement is partial: **assume** e terminates normally if the predicate e evaluates to *true*, and simply cannot be executed from a state where e evaluates to *false*. The assignment statement $x := e$ updates the program state so that x is bound to the current value of the expression e. The statement $A \,;\, B$ denotes the sequential composition of A and B. The execution of the choice statement $A \,\square\, B$ executes either A or B, but the choice between the two is made arbitrarily. Expressions in the language are pure (side-effect free) and include at least the boolean constants *true* and *false*. All variables have an arbitrary value in a program's initial state.

The indeterminism arising from choice statements and from the program's initial (arbitrary) state can be tamed by assume statements: the semantics of a program considers only those executions in which the predicate of each executed assume statement evaluates to *true*. For example, the statement

$$(\textbf{assume } 0 \le x \,;\, A) \;\;\square\;\; (\textbf{assume } \neg(0 \le x) \,;\, B)$$

is the deterministic statement commonly written as

$$\textbf{if } 0 \le x \textbf{ then } A \textbf{ else } B \textbf{ end}$$

Before going on, we introduce the notational conventions that we use in this paper. We use a left-associative infix "." (binding more strongly than any other operator) to denote function application. For any function f, we use $f[x \mapsto y]$ to denote the function that is identical to f except that it maps x to y.

4 Weakest Precondition Semantics

We define the semantics of the guarded command language using the weakest precondition translation

$$wp : \mathit{Stmt} \to \mathit{Formula} \to \mathit{Formula}$$

The set *Formula* of logical formulae is an extension of expressions that includes (at least) the usual boolean operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $=$) and quantification, and is closed under substitution of expressions for variables. We use the notation $Q(x := e)$ to denote the capture-free substitution of e for every free occurrence of x in a formula Q.

The translation *wp* is defined in Figure 2. For any statement S and predicate Q on the post-state of S, $wp.S.Q$ returns a formula characterizing the pre-states from which executions of S never go wrong and terminate only in states satisfying Q.

To illustrate the derivation of verification conditions using the weakest precondition translation, consider the following method `abs`. This method returns the absolute value of its argument and decrements a counter `c` provided `c` was initially positive.

```
static int abs(int x)
//@ ensures \result >= 0
{
    if (x < 0) {
        x = -x;
        //@ assert x > 0;
    }
    if (c > 0) {
      c--;
    }
    return x;
}
```

ESC/Java annotations are given as specially formatted Java comments beginning with an `@`-sign. This method contains two annotation: a postcondition `\result >= 0` and an assertion `x > 0`. To verify these properties, ESC/Java translates this method into the following guarded command S_{abs}:[1]

$$\begin{array}{l}
(\ \ (\textbf{assume}\ x < 0\ ;\ x = -x\ ;\ \textbf{assert}\ x > 0) \\
\ \ \Box (\textbf{assume}\ \neg(x < 0)))\ ; \\
(\ \ (\textbf{assume}\ c > 0\ ;\ c = c - 1) \\
\ \ \Box (\textbf{assume}\ \neg(c > 0)))\ ; \\
\textbf{assert}\ x \geq 0
\end{array}$$

This guarded command is subsequently translated into the VC $wp.S_{abs}.true$:[2]

$$\begin{array}{r}
x < 0\ \Rightarrow \left(\begin{array}{l} \quad -x > 0 \\ \wedge \left(\begin{array}{r} c > 0\ \Rightarrow\ -x \geq 0 \\ \wedge\ \neg(c > 0)\ \Rightarrow\ -x \geq 0 \end{array} \right) \end{array} \right) \\
\wedge\ \neg(x < 0)\ \Rightarrow \left(\begin{array}{r} c > 0\ \Rightarrow\ x \geq 0 \\ \wedge\ \neg(c > 0)\ \Rightarrow\ x \geq 0 \end{array} \right)
\end{array}$$

Since this VC is valid, the method `abs` satisfies its postcondition and assertion.

Although this VC is of moderate size, in general a computation $wp.S.true$ may yield a verification condition whose size is exponential in the size of S. An examination of the definition of *wp* reveals that there are two situations where expressions or formulae are duplicated.

The first case arises from the translation of an assignment statement $x := e$. The weakest precondition of this statement with respect to a postcondition Q is $Q(x := e)$. Since the variable x may occur many times in Q, the substitution operation produces a precondition that contains that many copies of e. For example, the VC $wp.S_{abs}.true$ contains three copies of the expression $-x$.

This duplication is particularly pronounced when assignment statements are sequentially composed. To illustrate this problem, consider the statement:

$$\begin{array}{rl}
S \equiv (\ & x_1 := x_0 + x_0; \\
& x_2 := x_1 + x_1; \\
& \ldots; \\
& x_n := x_{n-1} + x_{n-1})
\end{array}$$

The weakest precondition $wp.S.(x_n > 0)$ has size exponential in n; in particular, it contains 2^n references to x_0. We note that this precondition could

[1] This guarded command elides some details introduced by the ESC/Java translation that are unrelated to the current discussion.

[2] For clarity, the VCs in this paper have been simplified by performing peephole optimizations, such as replacing $Q \wedge true$ by Q, as is also done by ESC/Java.

S	$\mathit{passify}.\langle \sigma, S \rangle$
assert e	$\begin{cases} \langle \bot,\ \textbf{assert}\ \mathit{false} \rangle & \text{if } e = \mathit{false} \\ \langle \sigma,\ \textbf{assert}\ \sigma.e \rangle & \text{if } e \neq \mathit{false} \end{cases}$
assume e	$\begin{cases} \langle \bot,\ \textbf{assume}\ \mathit{false} \rangle & \text{if } e = \mathit{false} \\ \langle \sigma,\ \textbf{assume}\ \sigma.e \rangle & \text{if } e \neq \mathit{false} \end{cases}$
$x := e$	$\langle \sigma[x \mapsto x'],\ \textbf{assume}\ x' = \sigma.e \rangle$ where x' is a fresh variable
$A\ ;\ B$	$\langle \sigma_b,\ (A'\ ;\ B') \rangle$ where $\begin{array}{rcl} \langle \sigma_a, A' \rangle & = & \mathit{passify}.\langle \sigma, A \rangle \\ \langle \sigma_b, B' \rangle & = & \mathit{passify}.\langle \sigma_a, B \rangle \end{array}$
$A \Box B$	$\langle \sigma',\ (A'\ ;\ R_a) \Box (B'\ ;\ R_b) \rangle$ where $\begin{array}{rcl} \langle \sigma_a, A' \rangle & = & \mathit{passify}.\langle \sigma, A \rangle \\ \langle \sigma_b, B' \rangle & = & \mathit{passify}.\langle \sigma, B \rangle \\ \langle \sigma', R_a, R_b \rangle & = & \mathit{merge}.\langle \sigma_a, \sigma_b \rangle \end{array}$

Figure 3: Translation into passive form.

$$
\begin{array}{rcl}
\mathit{merge}.\langle \sigma, \bot \rangle & = & \langle \sigma, \textbf{skip}, \textbf{skip} \rangle \\
\mathit{merge}.\langle \bot, \sigma \rangle & = & \langle \sigma, \textbf{skip}, \textbf{skip} \rangle \\
\mathit{merge}.\langle \sigma_a, \sigma_b \rangle & = & \langle \sigma', R_a, R_b \rangle \\
\text{where}\quad D & = & \{ x \in \mathit{Var} \mid \sigma_a.x \neq \sigma_b.x \} \\
\Theta & = & \text{map from } D \text{ to fresh variables} \\
R_a & = & ;\{ \textbf{assume}\ \sigma_a.x = \Theta.x \mid x \in D \} \\
R_b & = & ;\{ \textbf{assume}\ \sigma_b.x = \Theta.x \mid x \in D \} \\
\sigma' & = & x \mapsto \begin{cases} \Theta.x & \text{if } x \in D \\ \sigma_a.x & \text{if } x \notin D \end{cases}
\end{array}
$$

Figure 4: The function *merge*.

be represented in linear space as a directed acyclic graph (DAG).

The second cause of duplication occurs in the translation of a choice statement $A \Box B$, whose weakest precondition with respect to a postcondition Q is $wp.A.Q \wedge wp.B.Q$. Each conjunct contains a copy of Q (possibly modified by substitutions corresponding to assignment statements in A and B). Thus the postcondition Q (or some variant of it) is duplicated in both conjuncts. For example, in the VC $wp.S_{abs}.true$, there are four substitution instances of the assertion $x \geq 0$, one for each of the possible execution paths through the method `abs`.

Again, the duplication is more pronounced when choice statements are sequentially composed; the size of the resulting weakest precondition may be exponential in the size of the statement sequence. Unlike in the previous case, this precondition may require exponential space even when using a DAG representation, since the duplicated postconditions may not be exact copies of each other due to substitutions arising from assignments.

5 Translation into Passive Form

The preceding discussion indicates that assignment statements play a role in both sources of duplication in the weakest precondition translation. Based on this insight, we develop a two-stage VC generation algorithm. The first stage transforms a statement into an assignment-free, or *passive*, form. The second stage exploits the assignment-free nature of this passive form to create a compact verification condition.

To translate a source statement to passive form we need to remove each assignment statement. The basic idea is to replace each assignment statement

$$x := e$$

by an assumption

$$\textbf{assume}\ x' = e$$

where x' is a fresh variable, and to change subsequent references to x to refer to x'. We call x' a *variant* of x, and use

$$\mathit{Subst} = (\mathit{Var} \rightarrow \mathit{Var}) \cup \{\bot\}$$

to denote the set of substitutions, or mappings from program variables to their current variants. The special substitution $\bot$ ("bottom") is used to indicate infeasible code paths as discussed below. We extend substitutions from variables to expressions and formulae in the usual manner, and for any expression or formula e, we define $\bot.e$ to be e.

The translation into passive form is performed by the function

$$\mathit{passify} : \mathit{Subst} \times \mathit{Stmt} \rightarrow \mathit{Subst} \times \mathit{PStmt}$$

defined in Figure 3, where $\mathit{PStmt} \subset \mathit{Stmt}$ denotes the set of assignment-free, or passive, statements.

The function *passify* takes two arguments: an initial substitution $\sigma \in \mathit{Subst}$ that specifies the current variant for each program variable, and a statement $S \in \mathit{Stmt}$ to be transformed, and returns a pair $\langle \sigma', S' \rangle$, where S' in a passive version of S, and where $\sigma' \in \mathit{Subst}$ describes the current variants after S' terminates. If S' is guaranteed never to terminate, then σ' may be $\perp$.

The translation of an assert statement **assert** e into passive form is straightforward: we simply apply σ to e. As an optimization, if e is *false*, we may indicate that the assertion will never terminate by returning the special substitution $\perp$. The translation of an assume statement proceeds in a similar manner. To translate an assignment statement $x := e$ into passive form, we choose a fresh variable x', introduce the assumption that $x' = \sigma.e$, and update σ to record that x' is the current variant of x. For a sequential composition, we simply translate each component in turn.

The translation of a choice statement $A \,\square\, B$ is more complicated, since the passive forms of A and B may introduce different variants, say x_a and x_b, of some program variable x. We handle this situation by introducing a third variant, say x', of x; appending to the passive form of A the assumption that $x' = x_a$; appending to the passive form of B the assumption that $x' = x_b$; and subsequently using x' as the current variant of x.

This translation of choice statements relies on a helper function

$$\mathit{merge} : \mathit{Subst} \times \mathit{Subst} \rightarrow \mathit{Subst} \times \mathit{PStmt} \times \mathit{PStmt}$$

shown in Figure 4. Given two substitutions σ_a and σ_b describing the current variants after the passive forms of A and B terminate, the function invocation $\mathit{merge}.\langle \sigma_a, \sigma_b \rangle$ returns a triple $\langle \sigma', R_a, R_b \rangle$, where R_a and R_b describe the additional assumptions, or renamings, to be appended to the passive forms of A and B, respectively, and σ' is resulting substitution. As an optimization, if one of the arguments to *merge* is $\perp$, then no fresh variables or additional assumptions are necessary, because either A or B will never terminate.

In the definition of *merge*, we use **skip** to abreviate **assume** *true*, and we use ";" as an operator applied to a set of statements to denote the sequential composition of those statements (in an arbitrary order).

For the method `abs` from Section 4, the corresponding passive statement S'_{abs} is:

$$\begin{array}{l}
(\;\; (\textbf{assume}\ x < 0\ ;\ \textbf{assume}\ x_1 = -x\ ; \\
\quad\;\; \textbf{assert}\ x_1 > 0\ ;\ \textbf{assume}\ x_2 = x_1) \\
\;\; \square (\textbf{assume}\ \neg(x < 0)\ ;\ \textbf{assume}\ x_2 = x))\ ; \\
(\;\; (\textbf{assume}\ c > 0\ ;\ \textbf{assume}\ c_1 = c - 1\ ; \\
\quad\;\; \textbf{assume}\ c_2 = c_1) \\
\;\; \square (\textbf{assume}\ \neg(c > 0)\ ;\ \textbf{assume}\ c_2 = c))\ ; \\
\textbf{assert}\ x \geq 0
\end{array}$$

The additional assumptions introduced by the *passify* translation may produce an increase in the code size. Although this increase may be quadratic in the worst case, in practice it appears to be closer to linear (see Section 8).

The translation of a guarded command statement into passive form preserves the semantics of the statement, as described by the following theorem.

Theorem 1 *Let $S \in \mathit{Stmt}$, let $\sigma \in \mathit{Subst}$, let*

$$\langle \sigma', S' \rangle = \mathit{passify}.\langle \sigma, S \rangle$$

and let $x_1, \ldots, x_n$ be the additional variables introduced by the passify translation. Then for any predicate Q containing no free occurrences of $x_1, \ldots, x_n$,

$$\sigma.(\mathit{wp}.S.Q) = \forall x_1, \ldots, x_n :\ \mathit{wp}.S'.(\sigma'.Q)$$

Proof: By structural induction on S.

6 VCs for Passive Statements

It remains to transform the passive representation into a verification condition. The assignment-free nature of the passive form allows us to perform this translation while avoiding the exponential blow up of the weakest precondition translation. The absence of assignment statements means that the execution of a passive statement cannot change the program state, and the only effect of such an execution is to choose among the two possible outcomes: normal termination and going wrong. Thus the semantics of a passive statement S can be completely captured by two *outcome predicates*: $N.S$, which

S	$N.S$	$W.S$
assert e	e	$\neg e$
assume e	e	*false*
$A \,;\, B$	$N.A \wedge N.B$	$W.A \vee (N.A \wedge W.B)$
$A \,\square\, B$	$N.A \vee N.B$	$W.A \vee W.B$

Figure 5: Outcome predicates for passive stmts.

describes the initial states from which the execution of S may terminate normally, and $W.S$, which describes states from which the execution of S may go wrong.

The definition of the two outcome predicates

$$N, W : \mathit{PStmt} \rightarrow \mathit{Formula}$$

is shown in Figure 5. An assert or assume statement terminates normally if its predicate is true. An assert statement goes wrong if its predicate is false, whereas an assume statement never goes wrong. A sequential composition $A \,;\, B$ terminates normally only if both components terminate normally, and goes wrong if either A goes wrong, or if A terminates normally and B goes wrong. A choice statement $A \,\square\, B$ may terminate normally if either component terminates normally, and may go wrong if either component goes wrong.

The relationship between the outcome predicates and the *wp* translation is described by the following theorem.

Theorem 2 *If S is in passive form then*

$$wp.S.Q = \neg(W.S) \wedge (N.S \Rightarrow Q)$$

Proof: By structural induction on S.

Hence, for any passive statement S, the VCs $wp.S.true$ and $\neg(W.S)$ are equivalent. The VC $\neg(W.S'_{abs})$ for the passive statement S'_{abs} considered earlier is:

$$\neg\left(\begin{array}{l} \quad (x < 0 \wedge x_1 = -x \wedge \neg(x_1 > 0)) \\ \vee\ (Q_1 \wedge Q_2 \wedge \neg(x_2 \geq 0)) \end{array}\right)$$

where:

$$Q_1 = \left(\begin{array}{l} \quad (x < 0 \wedge x_1 = -x \wedge x_1 > 0 \wedge x_2 = x_1) \\ \vee\ (\neg(x < 0) \wedge x_2 = x) \end{array}\right)$$

$$Q_2 = \left(\begin{array}{l} \quad (c > 0 \wedge c_1 = c - 1 \wedge c_2 = c_1) \\ \vee\ (\neg(c > 0) \wedge c_2 = c) \end{array}\right)$$

Verifying the assertion expressed by $\neg(x_1 > 0)$ in this VC is straightforward. To verify the postcondition expressed by $\neg(x_2 \geq 0)$, a theorem prover would need to analyze separately the two disjuncts in Q_1 (corresponding to the two paths through the first choice statement in S'_{abs}). However, the theorem prover need not analyze the two disjuncts in Q_2 (corresponding to the two paths through the second choice statement). Thus, under the outcome predicate encoding of VCs, the analysis performed by a theorem prover when verifying a property (such as an assertion or postcondition) need not correspond to an all-paths analysis of the underlying method; the theorem prover can use various heuristics in an attempt to ignore disjunctions arising from choice statements that do not affect the correctness of the property being verified.

The VCs generated by this approach are typically significantly smaller than that generated by *wp*. In particular, for a passive statement S, the VC $\neg(W.S)$ is at most quadratic in the size of S.

Theorem 3 *If S is in passive form then*

1. $|N.S|$ *is* $O(|S|)$, *and*
2. $|W.S|$ *is* $O(|S|^2)$.

Proof: By structural induction on S.

The VC $\neg(W.S)$ still contains a significant amount of duplication (and hence the quadratic as opposed to linear bound). For example, the VC $\neg(W.S'_{abs})$ contains two copies of the subformula $x < 0 \wedge x_1 = -x$.

Since the duplicated subformulae in a VC $\neg(W.S)$ are exact copies (and not substitution instances) of each other, it is straightforward to introduce a name for each duplicated subformula Q, for example, via $(\exists x :\ (x = Q) \Rightarrow \ldots)$, and to replace each occurrence of Q by a reference to the variable x. In the context of the Simplify theorem prover, for efficiency purposes we name each duplicated subformula Q via the nullary predicate definition (DEFPRED (x) Q), and then replace each occurrence of Q by the predicate application (x).

Of course, if Q is small, it may not be worth naming it, due to the overhead of processing the resulting indirection in the theorem prover. Our implementation uses a cutoff K, and only names a duplicated subformula if the formula is larger than K (where the size of a formula is the number of nodes in its representation). For any finite value of this cutoff, the size of the resulting VC is linear in the size of the passive statement, and hence at most quadratic in the size of the source program. Section 8 describes experimental results concerning the effect of K on both the size of VCs and on the time required to prove them.

7 Handling Loops and Exceptions

ESC/Java uses an extended guarded command language that also incorporates loops and exceptions. This section outlines how to extend our development to handle these constructs.

7.1 Loops

ESC/Java incorporates a variety of translations for loops that provide different trade-offs between soundness, completeness, and annotation overhead. These translations are outlined in a related paper [LSS99]. In this section, we sketch a sound but incomplete translation for **while** loops.

This translation relies on the programmer to annotate each loop with a loop invariant Q; this invariant is required to hold at the beginning of each iteration of the loop. Using this loop invariant, we desugar each while statement into a collection of more primitive guarded commands. Under this desugaring, a while loop

$$\textbf{while}\ \{Q\}\ e\ \textbf{do}\ S\ \textbf{end}$$

becomes

$$\begin{array}{l}
\textbf{assert}\ Q\ ;\\
x_1 := y_1\ \ ;\ \ldots\ ;\ x_n := y_n\ \ ;\\
\textbf{assume}\ Q\ ;\\
(\quad (\textbf{assume}\ e\ ;\ S\ ;\ \textbf{assert}\ Q\ ;\ \textbf{assume}\ \mathit{false})\\
\quad \Box\ \ \textbf{assume}\ \neg e)
\end{array}$$

where $x_1, \ldots, x_n$ are the variables assigned in S, and $y_1, \ldots, y_n$ are fresh variables that hold arbitrary values (from the program's initial state).

The desugared code ensures the loop invariant holds initially, and then sets the variables $x_1, \ldots, x_n$ to arbitrary values that satisfy the loop invariant. The code then checks that if e is true, then the loop invariant still holds after executing S; if e is false, then the desugared loop terminates.

S	$wp.S.\langle Q, R\rangle$
assert e	$e \wedge Q$
assume e	$e \Rightarrow Q$
$x := e$	$Q(x := e)$
$A\ ;\ B$	$wp.A.\langle wp.B.\langle Q, R\rangle, R\rangle$
$A \ \Box\ B$	$wp.A.\langle Q, R\rangle \wedge wp.B.\langle Q, R\rangle$
raise	R
$A\ !\ B$	$wp.A.\langle Q, wp.B.\langle Q, R\rangle\rangle$

Figure 6: Weakest preconditions with exceptions.

7.2 Exceptions

ESC/Java's guarded command language includes two additional constructs for raising and catching exceptions:

$$S ::= \ldots \mid \textbf{raise} \mid A\ !\ B$$

The statement **raise** simply raises an exception. The guarded command language does not distinguish different types of exceptions; this is accomplished in ESC/Java by using an auxiliary program variable. ESC/Java uses guarded command exceptions not only to model the behavior of Java exceptions, but also to model other forms of "escaping" control flow, such as `break` and `return` statements. In the catch statement $S_1\ !\ S_2$, the statement S_2 is the exception handler for any exception raised (and not caught) in S_1. If S_1 terminates normally, then S_2 is not executed.

The weakest precondition translation for the extended language

$$wp : \mathit{Stmt} \to (\mathit{Formula} \times \mathit{Formula}) \to \mathit{Formula}$$

now takes as arguments a statement S and two postconditions, Q and R, which describe properties that should hold whenever S terminates normally or exceptionally, respectively. If the precondition $wp.S.\langle Q, R\rangle$ holds a particular state, then any execution of S from the state can never go wrong,

S	$\mathit{passify}.\langle \sigma, S\rangle$
assert e	$\begin{cases} \langle \bot, \bot, \ \ \textbf{assert}\ \mathit{false}\rangle & \text{if } e = \mathit{false} \\ \langle \sigma, \bot, \ \ \textbf{assert}\ \sigma.e\rangle & \text{if } e \neq \mathit{false} \end{cases}$
assume e	$\begin{cases} \langle \bot, \bot, \ \ \textbf{assume}\ \mathit{false}\rangle & \text{if } e = \mathit{false} \\ \langle \sigma, \bot, \ \ \textbf{assume}\ \sigma.e\rangle & \text{if } e \neq \mathit{false} \end{cases}$
$x := e$	$\langle \sigma[x \mapsto x'], \bot, \ \ \textbf{assume}\ x' = \sigma.e\rangle$ where x' is a fresh variable
$A\ ;\ B$	$\langle \sigma_{bn}, \sigma_x, \ \ (A'\ !\ (R_{ax}\ ;\ \textbf{raise}))\ ;\ (B'\ !\ (R_{bx}\ ;\ \textbf{raise}))\rangle$ where $\begin{array}{rcl} \langle \sigma_{an}, \sigma_{ax}, A'\rangle &=& \mathit{passify}.\langle A, \sigma\rangle \\ \langle \sigma_{bn}, \sigma_{bx}, B'\rangle &=& \mathit{passify}.\langle B, \sigma_{an}\rangle \\ \langle R_{ax}, R_{bx}, \sigma_x\rangle &=& \mathit{merge}.\langle \sigma_{ax}, \sigma_{bx}\rangle \end{array}$
$A \Box B$	$\langle \sigma_n, \sigma_x, \ \ ((A'\ ;\ R_{an})\ !\ (R_{ax}\ ;\ \textbf{raise})) \Box ((B'\ ;\ R_{bn})\ !\ (R_{bx}\ ;\ \textbf{raise}))\rangle$ where $\begin{array}{rcl} \langle \sigma_{an}, \sigma_{ax}, A'\rangle &=& \mathit{passify}.\langle A, \sigma\rangle \\ \langle \sigma_{bn}, \sigma_{bx}, B'\rangle &=& \mathit{passify}.\langle B, \sigma\rangle \\ \langle R_{an}, R_{bn}, \sigma_n\rangle &=& \mathit{merge}.\langle \sigma_{an}, \sigma_{bn}\rangle \\ \langle R_{ax}, R_{bx}, \sigma_x\rangle &=& \mathit{merge}.\langle \sigma_{ax}, \sigma_{bx}\rangle \end{array}$
raise	$\langle \bot, \sigma, \ \ \textbf{raise}\rangle$
$A\ !\ B$	$\langle \sigma_n, \sigma_{bx}, \ \ (A'\ ;\ R_{an})\ !\ (B'\ ;\ R_{bn})\rangle$ where $\begin{array}{rcl} \langle \sigma_{an}, \sigma_{ax}, A'\rangle &=& \mathit{passify}.\langle A, \sigma\rangle \\ \langle \sigma_{bn}, \sigma_{bx}, B'\rangle &=& \mathit{passify}.\langle B, \sigma_{ax}\rangle \\ \langle R_{an}, R_{bn}, \sigma_n\rangle &=& \mathit{merge}.\langle \sigma_{an}, \sigma_{bn}\rangle \end{array}$

Figure 7: Translation into passive form for statements with exceptions.

S	$N.S$	$X.S$	$W.S$
assert e	e	false	$\neg e$
assume e	e	false	false
$A\ ;\ B$	$N.A \wedge N.B$	$X.A \vee (N.A \wedge X.B)$	$W.A \vee (N.A \wedge W.B)$
$A \Box B$	$N.A \vee N.B$	$X.A \vee X.B$	$W.A \vee W.B$
raise	false	true	false
$A\ !\ B$	$N.A \vee (X.A \wedge N.B)$	$X.A \wedge X.B$	$W.A \vee (X.A \wedge W.B)$

Figure 8: Outcome predicates for passive statements with exceptions.

can terminate normally only in states satisfying Q, and can terminate exceptionally only in states satisfying R. The new translation is described in Figure 6.

Since statements can now terminate exceptionally, we update the *passify* translation to return a triple $\langle \sigma_n, \sigma_x, S'\rangle$, where S' is a passive statement, σ_n describes the current variants after S' terminates normally, and σ_x describes the current variants after S' terminates exceptionally. When translating a sequential composition $A\ ;\ B$, we need to merge the substitutions corresponding to the exceptional terminations of A and B. Hence, we append to the passive forms of A and B code that catches any thrown exception, performs the appropriate renaming, and then re-raises the exception. Conversely, for a catch statement $A\ !\ B$, we need to merge the substitutions corresponding to the normal terminations of A and B. For a choice statement, we need to merge substitutions corresponding to both normal and exceptional terminations.

The definition of the new *passify* translation is shown in Figure 7. The infeasible path optimization (using $\perp$) is particularly important in the presence of exceptions (see Section 7.2) because many statements will never terminate exceptionally.

Finally, we introduce a third outcome predicate, $X.S$, which describes the pre-states of S from which exceptional termination is possible. The appropriate definition of the three outcome predicates

$$N, X, W : \mathit{PStmt} \rightarrow \mathit{Formula}$$

for passive statements with exceptions is shown in Figure 8.

Note that because of additional control flow paths introduced by exceptions, the size of a VC $\neg(W.S)$ may now be exponential in the size of S. To illustrate this, consider the application of the outcome predicate N to the code fragment $(A\ ;\ B)\ !\ C$, which yields the formula

$$\begin{array}{ll} & N.((A\ ;\ B)\ !\ C) \\ \equiv & N.(A\ ;\ B) \vee (X.(A\ ;\ B) \wedge N.C) \\ \equiv & (N.A \wedge N.B) \\ & \vee\ ((X.A \vee (N.A \wedge X.B)) \wedge N.C) \end{array}$$

containing two occurrences of the subformula $N.A$.

However, by naming shared subformulae as described in Section 6, we can still represent the VC $\neg(W.S)$ as a formula whose size is linear in the size of S.

8 Experimental Results

We have implemented, as part of ESC/Java, both VC generation algorithms described in this paper. This section presents experimental results comparing five VC-generation options: the standard *wp*-based translation, and the two-stage translation based on with four different values (0, 10, 30, ∞) for the cutoff size K above which to introduce names for duplicated outcome predicates.

These experiments were performed on a 667Mhz ES40 Alpha workstation containing 4Gb of memory running Tru64 UNIX. ESC/Java is written in Java, and was run on the Compaq Fast VM. The Simplify theorem prover is written in Modula-3, and runs as a separate process. The two processes communicate via Unix pipes.

The benchmark we used for these experiments is ESC/Java's front-end, which we have annotated with appropriate specifications. This front-end consists of 2292 routines (methods and constructors) totaling over 20,000 lines of code.

We have divided the routines in the benchmark into three categories according to their worst performance under any of the five options. The first category contains the ten "hardest" routines. The verification of each of these routines either exhausted a 1Gb heap or required more than five minutes under at least one of the options. The second category of routines contains the 17 routines that required at least 100 seconds under some option, but no more than 300 seconds under any option. The third category contains the 2184 routines in the benchmark that were successfully verified in under 100 seconds regardless of the VC generation option chosen. The remaining 81 routines in the front end are routines for which ESC/Java performs no VC generation (for example, methods declared in interfaces); these routines are not included in the benchmark.

Figure 9 describes the performance of the five VC generation options on the routines in the benchmark, with results for the ten "hardest" routines given individually, and summed results for the other two categories.

The columns of the table identify:

- the routine name (or summed category);
- the size of the abstract syntax tree for the routine (number of nodes);
- the size of the guarded command (number of nodes);
- the size of the VC under the *wp* translation (number of nodes);
- the time required to check this routine under the *wp* translation (seconds);
- the size of the passive version of the guarded command (number of nodes and percentage of original guarded command);
- the cutoff K for naming duplicated outcome predicates (number of nodes);

			wp translation		New translation						
Routine name	AST	GC	VC	time	PGC	%	K	VC	%	time	%
`BinaryExpr.postCheck`	420	1805	too big		1545	86	0	5758		1.4	
							10	4840		0.6	
							30	4700		0.6	
							∞	5513		1.0	
`LiteralExpr.postCheck`	423	1417	too big		1608	113	0	5735		1.5	
							10	5023		1.3	
							30	5141		1.3	
							∞	5765		1.3	
`finishFloatingPointLiteral`	653	3464	too big		7927	229	0	13134		69.5	
							10	10616		8.9	
							30	10583		10.7	
							∞	20416		73.9	
`scanCharOrString`	812	3896	too big		23904	614	0	33502		49.8	
							10	29851		18.1	
							30	29924		17.1	
							∞	51357		16.3	
`scanNumber`	1030	4170	too big		12383	297	0	19080		34.6	
							10	15334		23.0	
							30	15204		14.6	
							∞	29093		10.7	
`scanPunctuation`	509	3326	4751446	350.0	10457	314	0	15525	0	26.0	7
							10	13748	0	12.9	4
							30	13816	0	12.0	3
							∞	17972	0	6.8	2
`parseNewExpression`	794	7052	102780	77.0	27170	385	0	38186	37	530.1	688
							10	36116	35	432.0	561
							30	35749	35	419.0	544
							∞	83659	81	339.2	440
`checkExpr`	3945	17448	2798672	> 2000.0	35491	203	0	61813	2	750.4	< 30
							10	53779	2	401.5	< 16
							30	51874	2	347.8	< 14
							∞	90646	3	169.4	< 7
`checkStmt`	2883	15746	1041210	309.0	43417	276	0	67915	7	457.5	148
							10	61382	6	251.1	81
							30	57726	6	145.4	47
							∞	105297	10	151.9	49
`visitMethodDecl`	479	4331	2022351	381.5	5270	122	0	12423	1	10.2	3
							10	11430	1	8.7	2
							30	11581	1	8.7	2
							∞	12544	1	8.8	2
sum of 17 other routines needing > 100 seconds in some run	14735	124511	11782617	1574.5	312822	251	0	489529	4	1777.8	113
							10	461027	4	1048.0	67
							30	450090	4	900.7	57
							∞	955276	8	676.8	43
sum of remaining 2184 routines	148168	1851166	11668371	1613.4	2379973	129	0	7471786	64	2473.1	153
							10	7095724	61	1662.8	103
							30	6998342	60	1508.5	93
							∞	7530778	65	1564.1	97

Figure 9: Experimental results.

- the size of the VC under the two-stage translation (number of nodes and percentage of the *wp* VC);
- the time required to verify this routine under the two-stage translation (seconds and percentage of time using *wp*)

The times in this table include the time required to translate the Java abstract syntax tree representation into a guarded command, the time required to translate the guarded command into a VC (including translation into the intermediate passive representation, if necessary), and time required to check the VC. For the summed categories, the entries in the percentage columns tell how big one summed quantity is as a percentage of another.

The results indicate that all the VC generation algorithms work well on the simpler routines in the third category. The two-stage translation produces smaller VCs that are slightly easier to prove for $K = 30$ or $K = \infty$. Choosing $K = 0$ results in the theorem prover performing significant extra work to process the resulting indirections.

The *wp* translation has difficulty scaling to the larger or more complex routines; for five of the routines in this benchmark the *wp* translation exhausted a 1Gb heap limit.

The two-stage translation performs much better on these complex routines; the resulting VCs are significantly smaller, and can easily fit in the heap. Again, choosing $K = 0$ results in larger proof times. For $K > 0$, the two-stage translation yields VCs that require significantly less time to prove (sometimes by an order of magnitude) than the VCs produced by the *wp* translation.

The routine `parseNewExpression` requires significantly more time to verify under the two-stage translation than under *wp*. In general, the time required by Simplify to verify a formula is highly dependent on the order in which Simplify chooses to perform various case-splits, and the case-split order is determined by a variety of heuristics. For this routine, we suspect that these heuristics are misled in some manner by the VC generated by the two-stage translation.

Overall, the two-stage translation perform significantly better than the *wp* translation. It enables ESC/Java to check all of the routines in this benchmark due to the smaller space requirements, and is significantly faster. Excluding the six routines that could not be checked using the *wp* translation, checking the entire benchmark using *wp* required 4305 seconds, whereas using the two-stage translation requires only 2994 seconds (for $K = 30$), or 2748 seconds (for $K = \infty$).

9 Related Work

The VC-generation technique described in this paper is the result of joint work by the authors with Greg Nelson as part of the ESC/Java project [Ext]. It is related to an earlier (unpublished) VC-generation technique developed by one of the authors (Saxe), Nelson, and David Detlefs as part of an extended static checker for Modula-3 [DLNS98]. The earlier technique produced similarly compact and efficiently provable VCs, but was based on intermixing weakest-precondition computations with strongest-postcondition computations rather than on translation of statements to passive form.

An alternative approach for VC generation is based on all-paths symbolic forward execution. This approach is well-suited for handling unstructured or assembly-level code. It has been used in proof-carrying code systems [NL98] and in compiler validation techniques [Nec00]. However, this approach also suffers from the same causes of potential exponential blow up as the *wp* translation, namely, the handling of assignments and control-flow merge points.

The passive intermediate representation described in this paper is similar to the static single assignment (SSA) representation used in optimizing compilers [AWZ88, CFR+89]. Both intermediate representations are designed to facilitate later analysis stages. The renamings introduced by the *passify* translation at control flow merge points are analogous to the phi-functions used in the SSA representation.

10 Conclusions

Verification conditions provide a clean way to reason about program behavior and correctness. We have shown that the size of VCs need not be exponential in the size on the program fragment being checked, and we have presented a two-stage VC generation algorithm, based on a passive intermediate representation, that generates compact VCs. The size of these VCs is worst-case quadratic in the size of the source fragment, and is close to linear in practice.

The new algorithm has resulted in significant performance benefits in ESC/Java; we can now check many large and complex methods that we previously could not check due to time or space constraints.

Some other possible applications for our VC generation algorithm may include proof-carrying code [NL98], software model checking [BR00], compiler validation [Nec00], and full program verification.

Acknowledgments: The authors thank our current and previous colleagues on the extended static checking project, including David Detlefs, Rustan Leino, Mark Lillibridge, Greg Nelson, and Raymie Stata for their part in building the framework that allowed us to carry out the research reported here. Greg Nelson and David Detlefs made particularly relevant contributions, as described in section 9.

References

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs. In *15th ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[BR00] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, 2000.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, January 1989.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998. Available from `research.compaq.com/SRC/publications`.

[DNS01] David L. Detlefs, Greg Nelson, and James B. Saxe. An automatic theorem-prover for program checking, to appear 2001.

[Ext] Extended Static Checking web page, Compaq Systems Research Center. On the Web at `research.compaq.com/SRC/esc/`.

[LNS00] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Compaq Systems Research Center Technical Note 2000-002, October 2000. Available from `research.compaq.com/SRC/publications`.

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Compaq Systems Research Center Technical Note 1999-002, from `research.compaq.com/SRC/publications`.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 83–94, June 2000.

[Nel81] Greg Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 333–344, June 1998.