Weakest-Precondition of Unstructured Programs

Mike Barnett and K. Rustan M. Leino
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
{mbarnett,leino}@microsoft.com

Abstract

Program verification systems typically transform a program into a logical expression which is then fed to a theorem prover. The logical expression represents the weakest precondition of the program relative to its specification; when (and if!) the theorem prover is able to prove the expression, then the program is considered correct. Computing such a logical expression for an imperative, structured program is straightforward, although there are issues having to do with loops and the efficiency both of the computation and of the complexity of the formula with respect to the theorem prover. This paper presents a novel approach for computing the weakest precondition of an unstructured *program that is sound even in the presence of loops. The computation is efficient and the resulting logical expression provides more leeway for the theorem prover efficiently to attack the proof.*

0 Introduction

A technique for precisely checking that a computer program meets specified correctness criteria is *static program verification*. The typical architecture of a static program verifier takes as input a program and its specification, generates from these a *verification condition*—a first-order logical formula whose validity implies that the program meets the specification—and then passes the verification condition to a theorem prover. The engineering of the verification condition has a large impact on the proving task presented to the theorem prover [11]. The primary goal is to prevent redundancy in the verification condition, which lets the prover complete its task more efficiently. Although the exact nature of what constitutes redundancy may depend on the operation of the theorem prover, one general desideratum is that the formula not be dramatically larger than it needs to be.

In this paper, we describe the verification condition generation in the Spec# [2] static program verifier. It produces verification conditions that are decidedly smaller than those produced by ESC/Java [11, 13], the leading automatic program checker of its kind. Moreover, our verification condition generation is more general, because it applies to general control-flow graphs, not just to structured programs. Another little contribution of this paper is the data structure used when computing single-assignment incarnations, which can reduce the number of incarnations produced.

Like the verification condition generation in ESC/Java [10, 14, 11], we proceed in stages. Our starting point is a general control-flow graph. For us, this was a natural choice, because the Spec# static program verifier uses as its input language the intermediate language of the .NET virtual machine, whose branch instructions can give rise to any control flow. Using standard compilation techniques that duplicate instructions to eliminate multiple entry points to loops [0], we transform the general control-flow graph into a reducible one. (In fact, being a superset of C#, Spec# inherits **goto** statements that enable irreducible control flow already at the source level.) We then eliminate loops, producing an acyclic control-flow graph that is correct only if the original program is correct. We apply a *single-assignment* transformation to the acyclic program and then turn it into a *passive* program by changing assignment statements into **assume** statements. Finally, we apply *weakest preconditions* to the unstructured, acyclic, passive program to generate the verification condition.

In this paper, we describe the stages of this pipeline in reverse order. But before we do, we present the unstructured language under consideration and describe its executions and correctness criteria.

1 Programs and Correctness

Throughout this paper, we think of a *program* as a chunk of code that is to be verified. This may correspond to the implementation of a method in the source program, for example.

The language we consider in this paper follows this grammar:

$$\begin{array}{rcl} \mathit{Program} & ::= & \mathit{Block}^+ \\ \mathit{Block} & ::= & \mathit{BlockId} : \ \mathit{Stmt};\ \mathbf{goto}\ \mathit{BlockId}^* \\ \mathit{Stmt} & ::= & \mathit{VarId} := \mathit{Expr} \mid \mathbf{havoc}\ \mathit{VarId} \\ & \mid & \mathbf{assert}\ \mathit{Expr} \mid \mathbf{assume}\ \mathit{Expr} \\ & \mid & \mathit{Stmt}\ ;\ \mathit{Stmt} \mid \mathbf{skip} \end{array}$$

A program consists of a number of *basic blocks*. Each basic block has a *label*, a *body*, and a possibly empty set of *successors*. We assume the program's first block is labeled "Start".

A program gives rise to a set of *execution traces*. An execution trace consists of a sequence of *program states*, each a valuation of the program variables. A trace is either infinite or it ends in *termination*, ends in *error*, or ends in *infeasibility*. Intuitively, each trace of a program consists of the execution of successive blocks starting from Start, at the end of each block arbitrarily choosing one of the declared successor blocks, if any; the trace ends in termination if there are no successors to choose from, ends in error if an **assert** statement evaluates to false, and ends in infeasibility if an **assume** statement evaluates to false. In the next two paragraphs, we make this definition more precise.

A statement gives rise to a set of finite execution traces. The assignment statement $x := E$ gives rise to the set of terminating traces $\sigma\ \tau$, where state τ is like state σ except that it evaluates x to $\sigma(E)$. The details of expressions are not important here, but we assume an expression always evaluates to some value in each state. The statement **havoc** x sets x to an arbitrary value, thus giving rise to the set of all terminating traces $\sigma\ \tau$, where σ and τ agree on their valuation of all variables except possibly x. The statement **assert** E gives rise to the terminating (single-state) traces σ where $\sigma(E)$, and to the erroneous traces σ where $\neg\sigma(E)$. The statement **assume** E gives rise to the terminating traces σ where $\sigma(E)$, and to the infeasible traces σ where $\neg\sigma(E)$. Sequential composition $S; T$ gives rise to the non-terminating traces of S, and to the terminating traces of S *continued* (via a matching intermediate state) by the traces of T. Finally, **skip** is just a shorthand for **assert** true.

The set of traces of a block A is the smallest set of traces that include (a) the set of non-terminating traces of the body of A, (b) the set of terminating traces of the body of A, if A has no successors, and (c) the set of terminating traces of the body of A continued by the traces of the successors of A, if A has successors. The set of traces of a program is the set of traces of block Start.

A program is *correct* if none of its traces ends in error. Note that this definition of correctness does not say anything about the final state of terminating executions, but one can encode given postconditions by putting an appropriate **assert** statement at the end of blocks with no successors. Note also that correct programs can have traces that end in infeasibility; such can be thought of as the execution making "mistakes" in the "arbitrary" choices inherent in **havoc** and **goto** statements and in the "arbitrary" choice of the initial state. Any given preconditions of the chunk of code to be checked can be encoded by putting an appropriate **assume** statement at the beginning of block Start. Finally, note that a correct program can include neverending executions.

Our little language may seem impoverished at first, but it suffices for verification purposes (*cf.* [14]). In fact, it closely resembles the statements in BoogiePL [5], the intermediate language used by the Spec# static program verifier. For example, conditional control flow, as in a common if statement

$$\mathbf{if}\ (E)\ \{\ S\ \}\ \mathbf{else}\ \{\ T\ \}$$

can be encoded in our little language as:

$$\begin{array}{ll} \mathit{Start}: & \mathbf{skip}\ ;\ \mathbf{goto}\ \mathit{Then}, \mathit{Else} \\ \mathit{Then}: & \mathbf{assume}\ E\ ;\ S\ ;\ \mathbf{goto}\ \mathit{End} \\ \mathit{Else}: & \mathbf{assume}\ \neg E\ ;\ T\ ;\ \mathbf{goto}\ \mathit{End} \\ \mathit{End}: & \ldots \end{array}$$

Iteration is supported via the **goto** statements. A procedure call is replaced by an encoding of the callee's pre/post specification, which can be done using **assert**, **havoc**, and **assume** [14]. Finally, exceptions are encoded using a couple of additional variables (*cf.* [14]) and conditional control flow that threads through exception handlers.

We are now ready to describe the details of the verification condition generation.

2 Weakest Preconditions

In this section, we define weakest preconditions of unstructured programs. This is the last stage in our verification-condition generation pipeline. We assume programs to be passive (there are no assignment statements). It is this computation of weakest preconditions that is at the heart of making the verification condition palatable to the theorem prover. In fact, our technique produces a verification condition that is linear in the size of the passive program.

For any statement S and predicate Q on the post-state of S, the *weakest precondition* of S with respect to Q, written $wp(S, Q)$, is a predicate that characterizes all pre-states of S from which no execution will go wrong and from which every terminating execution ends in a state satisfying Q [8]. The weakest preconditions of the passive statements are defined as follows, for any Q:

$$\begin{array}{lcl} wp(\mathbf{assert}\ P,\ Q) & = & P\ \wedge\ Q \\ wp(\mathbf{assume}\ P,\ Q) & = & P \Rightarrow Q \\ wp(S; T,\ Q) & = & wp(S, wp(T, Q)) \end{array}$$

Note that wp is monotonic in its second argument.

In a structured program, the central problem to be overcome in computing weakest preconditions is that of the choice statement, $S \ [] \ T$, which arbitrarily chooses one of S and T to execute. Its weakest precondition is defined by

$$wp(S \ [] \ T, \ Q) \quad = \quad wp(S, Q) \ \wedge \ wp(T, Q)$$

The problem is that the duplication of Q in the right-hand side of this equation introduces redundancy. Q represents proof obligations downstream of the choice statement, and this naive formulation suggests that the theorem prover would need to process Q twice. In general, Q may need to be processed twice, but in practice, large parts of Q are often independent of which choice is taken [11]. Luckily, passive programs satisfy a property that lets this wp equation be formulated in a way that significantly reduces redundancy [13]. The alternate form uses wp and so-called *weakest liberal preconditions* (wlp) and produces verification conditions whose size is quadratic in the size of the passive program [11]. This alternate form applies to structured programs only, so applying it to unstructured programs would require some preprocessing step.

Unstructured programs do not have the structured choice statement. Instead, they have **goto** statements, which at first seem even more disastrous—certainly, we would not like to explode the control-flow graph into a tree, which would lose all the sharing that a control-flow graph representation affords (not to mention that we don't actually assume acyclicity in this section of the paper, even though in our application the passive programs are all acyclic).

Here is our solution. For every block

$$A\text{:} \ \ S \ ; \ \textbf{goto} \ \ldots$$

we introduce an *auxiliary variable* A_{ok}. Intuitively, A_{ok} is $true$ if the program is in a state from which all executions beginning from block A are correct. Formally, we postulate the following *block equation*:

$$A_{ok} \quad \equiv \quad wp(S, \ \bigwedge_{B \in Succ(A)} B_{ok})$$

where $Succ(A)$ denotes the set of successors of A so that the second argument to wp is the conjunction of B_{ok} for each block B in that set. For example, the block equation for $Then$ in the previous section is:

$$Then_{ok} \quad \equiv \quad (E \Rightarrow wp(S, End_{ok}))$$

Each block contributes one block equation, call their conjunction R, and the program's verification condition is:

$$R \quad \Rightarrow \quad Start_{ok}$$

The verification condition and block equations are in terms of the program's variables and the auxiliary variables. In the rest of this section, it will be convenient to include the auxiliary variables in states and traces. When we do so, we'll refer to the states as *augmented states*.

Lemma 0. *For any program state σ, there is an augmented state α that agrees with σ on the values of the program variables and that satisfies all the program's block equations.*

Proof. The right-hand side of each block equation is a monotonic function of auxiliary variables (since wp is monotonic in its second argument). Thus, the conjunction of block equations can be put into the form $K = F(K)$, where K denotes the tuple of auxiliary variables and $F(K)$ is the tuple of block-equation right-hand sides. Since F is a monotonic function on a complete lattice, $K = F(K)$ has a solution in K (by Tarski's Theorem [17]). □

Lemma 1. *Let P be a passive program, A be a basic block in P, and α an augmented state that satisfies all block equations of P. If A_{ok} is $true$ in α, then every execution from α starting in A is either correct or has a correct prefix that returns to block A.*

Proof. By induction over the set of blocks not yet visited in an execution prefix. If B_{ok} holds at the beginning of the execution from a block B, then the fact that α satisfies the block equation for B means that the execution of B's body is correct and that, for every successor C of B, C_{ok} holds upon termination of the body. For any successor block that is already visited in the execution trace, we are done. Morever, since the program is passive, all block equations still hold, so the antecedent for applying the induction hypothesis on any successor C_{ok} holds, and applying the induction hypothesis leads to a well-founded induction because there is one fewer blocks still to visit. □

Lemma 2. *Let P be a passive program, A be a basic block in P, and α an augmented state that satisfies all block equations of P. If A_{ok} is $true$ in α, then every execution from α starting in A is correct.*

Proof. Take any execution trace and chop it up into the longest possible segments that do not repeat any blocks. Since this is a passive program, the first and last states of any terminating segment are the same. Then each of these segments is correct, by Lemma 1, which implies that the whole execution is correct. □

Theorem 3. *For any passive program P, if the verification condition for P is a valid formula, then P is correct.*

Proof. By Lemma 0, we can augment the initial state with values for the auxiliary variables to form an augmented state α that satisfies the conjunction of block equations. From the validity of the verification condition, we then conclude that $Start_{ok}$ holds in α. By Lemma 2, every execution of the program is correct. □

3 Passification

We convert a loop-free program into a *passive* program by first rewriting it in a single-assigment form and then removing all of the assignment statements.

3.0 Single Assignment

Dynamic single-assignment (DSA) [9] is similar to the standard static single-assignment (SSA) [4] where even statically in the program text there is at most one definition for each variable. In DSA form, there may be more than one definition, but in any program execution, at most one of them will be executed.

We convert the loop-free program into DSA form by noting that after each update to a variable, its value must be understood relative to the newly updated state by identifying each updated value as a new *incarnation* of the variable. For instance, we replace the assignment statement:

$$x \quad := \quad x + 1$$

with the assignment statement:

$$x_{i+k} \quad := \quad x_i + 1$$

where x_{i+k} is a fresh incarnation. In general, all variables read by the statement are replaced by their *current incarnations*. After a variable update (assignment or **havoc** statement), a fresh incarnation becomes the new current incarnation for the updated variable. At the beginning of the program, an initial incarnation is created for each program variable. We call the last incarnation of a variable in a block the *block's incarnation* for that variable. The algorithm for performing these replacements processes the graph in a topologically sorted order.

For straight-line code, it is simple to iterate over the sequence of statements, replacing all of the variables with their current incarnations. But at join points (nodes in the control-flow graph with more than one predecessor), a node may be "inheriting" conflicting current incarnations from its predecessors. For instance, in the program in Section 1, let $Start$'s incarnation for x be x_0, $Then$'s incarnation be x_1, and $Else$'s incarnation be x_2. Consider block End: which incarnation should a reference to x (on the right-hand side of an assignment statement) be taken to be, x_1 or x_2? To model the joining of the values, we introduce a fresh incarnation, x_3, and introduce new assignment statements at the end of blocks B and C: $x_3 := x_1$ and $x_3 := x_2$, respectively. We also update each block's incarnation (for x) to be x_3. (This reflects a choice; we could leave their incarnations to be x_1 and x_2 respectively, but we next discuss how either choice leads to the excessive creation of incarnations.) This has the effect that during any particular execution, each incarnation is assigned to at most once. In the current example, either block $Then$ or block $Else$ will execute and x_3 will be equal to the corresponding incarnation from that block.

This procedure for converting the program to DSA form means that a new incarnation is potentially created for each variable at every join point. However, this may lead to the introduction of more incarnations than strictly necessary. Consider the program in Fig. 0. The algorithm sketched

A	$\ldots := x$;	**goto** B, C, D
B	$x := f(x)$;	**goto** E
C	$\ldots := x$;	**goto** E, F
D	$\ldots := x$;	**goto** F
E	$\ldots := x$;	**goto**
F	$\ldots := x$;	**goto**

Figure 0. A program that does not need a new incarnation at every join point.

above would create a fresh incarnation at the join points E and F, resulting in the program in Fig. 1. But it is clear

A	$\ldots := x_0$;	**goto** B, C, D
B	$x_1 := f(x_0)$; $x_2 := x_1$;	**goto** E
C	$\ldots := x_0$; $x_2 := x_0$; $x_3 := x_2$;	**goto** E, F
D	$\ldots := x_0$; $x_3 := x_0$;	**goto** F
E	$\ldots := x_2$;	**goto**
F	$\ldots := x_3$;	**goto**

Figure 1. The program from Fig. 0 in DSA form. Assuming the processing order is: A, B, C, E, D, F, the incarnation x_2 replaces x_0 as C's incarnation when processing E and the incarnation x_3 is then generated when processing F since D's incarnation is x_0.

that a minimal renaming would result in the DSA shown in Fig. 2. We achieve this reduction by keeping a *set* of incarnations as each block's incarnation. All of the incarnations have the same value, so when a join point is reached, any one of the incarnations in its predecessors's incarnation set can be used.

3.1 Passive Programs

Once the program has been converted to DSA form, we replace all assignment statements by **assume** statements.

A	$\ldots := x_0$;	**goto** B, C, D
B	$x_1 := f(x_0)$; $x_2 := x_1$;	**goto** E
C	$\ldots := x_0$; $x_2 := x_0$;	**goto** E, F
D	$\ldots := x_0$;	**goto** F
E	$\ldots := x_2$;	**goto**
F	$\ldots := x_0$;	**goto**

Figure 2. The program from Fig. 0 with minimal renaming. Note that there was no need to create the incarnation x_3 when processing F since C's incarnation is the set $\{x_0, x_2\}$ instead of having to choose either one of them.

We replace the assignment statement:

$$x_i \quad := \quad E$$

with the statement:

$$\mathbf{assume}\ x_i \quad = \quad E$$

We are able to replace the assignment with an **assume** statement since the value of x_i is not used prior to its definition—in effect, we thus assume that x_i had the desired value all along. Using an **assume** statement in this way expresses what some language use a *let binding* for: giving a name to a particular value.

4 Loops

In this section, we describe the transformation from a reducible control-flow graph into an acyclic control-flow graph. (We use the standard techniques for converting an irreducible graph into an equivalent, although possibly far larger, reducible graph. We are looking into ways to deal with irreducible graphs that avoid this problem, but so far it has not been an issue.) A reducible control-flow graph is one where it is possible to identify a unique loop head for each loop (throughout this section, we use standard terminology from compilers [0]).

In order to identify the loops, we begin by finding all of the *back edges*. It is the existence of a back edge that uniquely identifies a loop. A back edge is an edge in the control-flow graph whose tail (target of the edge) *dominates* its head (source of the edge). One node dominates another node when all paths to the latter pass through the former. The *loop header* for a back edge, B, is the target of the edge. A loop header, L, may have more than one loop associated with it: each *natural loop* is identified by the pair (L, B).

We remove all back edges to cut the loops, thus transforming the graph into an acyclic one. But in order for the loop body to represent an arbitrary loop iteration, we must make sure that the values of any variables modified within the loop have a value that they might hold on any iteration of the loop.

For each natural loop (L, B), we collect into a set $H(L, B)$ the variables that are updated by any statement in any block in the loop. These variables are called *loop targets*. For each loop target v in $H(L, B)$, we introduce a **havoc** statement and insert it at the beginning of L, before any of the existing statements in that block.

Wiping out all knowledge of the value a variable might hold may cause the theorem prover to be unable to prove the verification condition. That is, it induces an over-approximation of the original program and loses too much precision. To this end, we allow for each loop to have an *invariant*: a condition that must be met on each iteration of the loop. A loop invariant may be written by a user or it may be one inferred by another component of the Spec# static program verifier. (Inferring invariants is important to spare the programmer from an undue annotation burden.)

Loop invariants are encoded as a prefix of **assert** statements at the beginning of the loop header's code block. These assert statements cannot be validated if any of the variables they mention are in $H(L, B)$. Instead, we introduce a copy of this sequence of statements into each predecessor node of L (including the node that is the source of the back edge). Since the assertions are now checked just before the jump to the loop header, we change the statements into **assume** statements in L itself. We process loop invariants in this way before adding the **havoc** statements and cutting the back edges. The resulting **havoc** followed by the **assume** statements have the effect of retaining, about the loop targets, the information in the loop invariant.

We claim that this transformation does not affect the correctness of the program. It may however increase the *size* of the code since it introduces a copy of some code at the source of each edge instead of having a single copy at its target. When a loop head's predecessor has additional edges to other nodes than the header, this adds an assertion to control-flow paths that it had not been on previously. However this is a conservative approximation: if the transformed program executes correctly, then so would the original program.

Note that even after removing all back edges, the source node of the back edge is still reached from the loop header along forward edges.

5 Example

We illustrate our technique with a simple example. Consider the following Spec# source program:

```
int M(int x)
  requires 100 <= x;  // precondition
  ensures result == 0;  // postcondition
{
  while (0 < x)
    invariant 0 <= x;  // loop invariant
  {
    x = x - 1;
  }
  return x;
}
```

The control-flow graph corresponding to this method is encoded as follows, where we have used a variable r to denote the result value:

Start:	**assume** $100 \leq x$; // precondition
	goto *LoopHead* ;
LoopHead:	**assert** $0 \leq x$; // loop invariant
	goto *Body*, *After* ;
Body:	**assume** $0 < x$; // loop guard
	$x := x - 1$;
	goto *LoopHead* ;
After:	**assume** $\neg(0 < x)$; // negation of guard
	$r := x$; // **return** statement
	assert $r = 0$; // postcondition
	goto ;

After cutting back edges, the loop-free program is:

Start:	**assume** $100 \leq x$;
	assert $0 \leq x$; // check inv.
	goto *LoopHead* ;
LoopHead:	**havoc** x ; // havoc loop targets
	assume $0 \leq x$; // assume inv.
	goto *Body*, *After* ;
Body:	**assume** $0 < x$;
	$x := x - 1$;
	assert $0 \leq x$; // check inv.
	goto ; // removed back edge
After:	**assume** $\neg(0 < x)$;
	$r := x$;
	assert $r = 0$;
	goto ;

The passive form of the program is then:

Start:	**assume** $100 \leq x_0$;
	assert $0 \leq x_0$;
	goto *LoopHead* ;
LoopHead:	**skip** ;
	assume $0 \leq x_1$;
	goto *Body*, *After* ;
Body:	**assume** $0 < x_1$;
	assume $x_2 = x_1 - 1$;
	assert $0 \leq x_2$;
	goto ;
After:	**assume** $\neg(0 < x_1)$;
	assume $r_1 = x_1$;
	assert $r_1 = 0$;
	goto ;

After computing the wp, the set of block equations are then:

$Start_{be}$	$Start_{ok}$	$\equiv$	$100 \leq x_0 \Rightarrow$ $0 \leq x_0 \ \wedge$ $LoopHead_{ok}$
$LoopHead_{be}$	$LoopHead_{ok}$	$\equiv$	$0 \leq x_1 \Rightarrow$ $Body_{ok} \ \wedge \ After_{ok}$
$Body_{be}$	$Body_{ok}$	$\equiv$	$0 < x_1 \Rightarrow$ $x_2 = x_1 - 1 \Rightarrow$ $0 \leq x_2 \ \wedge \ true$
$After_{be}$	$After_{ok}$	$\equiv$	$\neg(0 < x_1) \Rightarrow$ $r_1 = x_1 \Rightarrow$ $r_1 = 0 \ \wedge \ true$

where we use $\Rightarrow$ as a right-associative operator whose binding power lies between that of $\equiv$ and $\wedge$. Finally, the verification condition is:

$$Start_{be} \wedge LoopHead_{be} \wedge Body_{be} \wedge After_{be} \ \Rightarrow \ Start_{ok}$$

6 Related Work

The use of single-assignment form for program analysis has a long history; the canonical reference is Cytron *et al.* [4]. Feautrier [9] introduced *dynamic* single-assignment, using it in the analysis of nested-loop programs. Since then, it has been used extensively in the context of nested-loop programs, *e.g.*, [1, 15, 16].

The ESC/Java checker [10] used DSA in its generation of verification conditions [11]. ESC/Java also converts programs to be loop free in order to compute verification conditions, either by unrolling the loop a certain number of times (which misses some execution traces) or by a sound treatment [14].

We have not seen descriptions of single-assignment that map variables to *sets* of incarnations, like we do to reduce the number of incarnations needed.

Despite their significant advantages, many other verification tools, including LOOP [18] and JACK [3], do not make use of redundancy-reducing techniques when generating verification conditions, thus producing voluminous verification conditions.

Weakest preconditions for unstructured programs have been defined in a similar way before [12]. However, in that work, the definitions were applied directly to programs that were neither passive nor loop-free, so the block equations used auxiliary functions instead of auxiliary variables, and the program semantics (that is, the antecedent of the verification condition, R) was defined to be a fixpoint of these functions. The rewriting of these formulas into formulas without quantifiers, functions, and fixpoints can give rise to a doubly exponential increase in size.

7 Conclusions

We have presented a detailed account of our procedure for computing a verification condition from a program (and its specification) in order to use an automatic theorem prover for program verification. Our input does not need to be a structured program; we deal efficiently with unstructured control-flow graphs. As a special case, our technique can be applied to structured programs, which will yield formulas with less redundancy than previously reported (formulas linear in the size of the passive program compared to the previous quadratic).

In the end, it is the time and space needed to generate verification conditions and the resulting theorem prover performance that matter. We had first implemented a transformation of the unstructured program into a structured one, from which we could then use previous techniques. We found that the transformation, which is exponential in the general case, caused our machines to run out of memory for some methods in the programs we applied our tool to. Good heuristics could probably have improved the situation, but since our new technique can be applied directly to unstructured programs, we abandoned the transformation in favor of it.

The theorem prover we currently use, Simplify [6], was developed along with redundancy-reduction techniques for the ESC/Modula-3 checker [7, 11], similar to those of the later ESC/Java checker. We were uncertain that Simplify would perform well on our new verification conditions, since their flat structure does not provide any guidance about a good order in which to do case splits and Simplify performs case splits only as a last resort. So far, we have not detected any such problems, though. We are in the process of switching to a theorem prover whose case splits are performed by a SAT solver, and we are hopeful that our verification conditions will be an especially good match for such a theorem prover.

We are also investigating whether our use of sets of incarnations achieves minimality, because there may be blocks considered even later in the algorithm that force new incarnations to be created.

Acknowledgments

We'd like to thank the Spec# team for various discussions about this design. Manuel Fähndrich contributed to the design of a previous scheme to generate verification conditions by first transforming unstructured programs into structured ones and to the design of where to place declared loop invariants in BoogiePL programs. Bart Jacobs implemented loop invariants in Spec#, taking measures to make sure these end up in the right place in the BoogiePL programs. Simon Ou coded up the block equations. We also thank Dave Naumann and the referees for their careful readings of this paper.

References

[0] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1987.

[1] Zena M. Ariola, Barton C. Massey, M. Sami, and Evan Tick. A common intermediate language and its use in partitioning concurrent declarative programs. *New Generation Computing*, 14(3):281–315, 1996.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[3] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439. Springer, September 2003.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-

oriented programs. Technical Report 2005-70, Microsoft Research, May 2005.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.

[7] 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.

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

[9] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[10] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[11] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[12] K. Rustan M. Leino. A SAT characterization of boolean-program correctness. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software: SPIN 2003*, volume 2648 of *LNCS*, pages 104–120. Springer, May 2003.

[13] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, March 2005.

[14] 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 Technical Note 1999-002, Compaq Systems Research Center.

[15] Carl Offner and Kathleen Knobe. Weak dynamic single assignment form. Technical Report HPL-2003-169, HP Laboratories, 2003.

[16] K. C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Geometric model checking: An automatic verification technique for loop and data reuse transformations. *Electronic Notes Theoretical Computer Science*, 65(2), 2002.

[17] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[18] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001*, volume 2031 of *LNCS*, pages 299–312. Springer, April 2001.