Efficient weakest preconditions

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Manuscript KRML 114a, 13 November 2003.
In memory of Edsger W. Dijkstra.

Abstract. Desired computer-program properties can be described by logical formulas called verification conditions. Different mathematically-equivalent forms of these verification conditions can have a great impact on the performance of an automatic theorem prover that tries to discharge them. This paper presents a simple weakest-precondition understanding of the ESC/Java technique for generating verification conditions. The new understanding of this technique spotlights the program property that makes the technique work.

Keywords: program correctness, formal semantics, automatic theorem proving

0 Introduction

Various computer-program checking tools and verification tools translate a given program into a *verification condition*, a logical formula whose validity reflects the correctness of the program. The verification condition is then passed to a mechanical theorem prover or some suite of automatic decision procedures. The Extended Static Checkers for Modula-3 and for Java are examples of program checkers built around this architecture [5, 8, 10].

There are many mathematically equivalent ways to formulate a verification condition, and which formulation one uses can have a dramatic impact on the performance of the program-checking system. The ESC/Modula-3 and ESC/Java projects have explored techniques for formulating verification conditions that substantially improve the way they are handled by the underlying automatic theorem prover. The variation of the technique used in ESC/Java is described by Flanagan and Saxe [9]. Their paper compares the ESC/Java technique with the well-known verification-condition technique of *weakest preconditions* [6]. In this paper, I show that the ESC/Java technique *is* in fact the technique of weakest preconditions with the additional use of a certain weakest-precondition property that holds only for a restricted class of programs.

1 Weakest preconditions

Let's start by reviewing weakest preconditions and the problem with their traditional application. We consider a simple language like the following, which is representative of the actual intermediate language used in ESC/Java [11]:

$$\begin{array}{rcl} S, T & ::= & Id := Expr \\ & | & \textbf{assert}\ Expr \\ & | & \textbf{assume}\ Expr \\ & | & S \ ; \ T \\ & | & S \ \square \ T \end{array}$$

The assignment statement $x := E$ sets program variable x to the value of E. The assert and assume statements are no-ops if the given expression evaluates to $true$. If the expression evaluates to $false$, the assert statement is an irrevocable error (the execution *goes wrong*) and the assume statement is a partial command that doesn't start (the execution *blocks*) [12]. Every execution in our simple language either blocks, goes wrong, or terminates. The statement $S \ ; \ T$ is the sequential composition of S and T, where T is executed only if S terminates, and $S \ \square \ T$ is the arbitrary choice between S and T. The statements of the simple language are rich enough to encode loops declared with loop invariants and procedure calls declared with procedure specifications (*cf.* [11, 1]). For this paper, it suffices to know that a common program statement like

$$\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{end}$$

is encoded as the choice statement

$$(\textbf{assume}\ B \ ; \ S) \ \square \ (\textbf{assume}\ \neg B \ ; \ T)$$

in the simple language.

The *weakest conservative precondition* of a statement S with respect to a predicate Q on the post-state of S, denoted $wp(S, Q)$, is a predicate on the pre-state of S, characterizing all pre-states from which every non-blocking execution of S does not go wrong and terminates in a state satisfying Q. Similarly, the *weakest liberal precondition* of S with respect to Q, denoted $wlp(S, Q)$, characterizes the pre-states from which every non-blocking execution of S either goes wrong or terminates in a state satisfying Q. The connection between wp and wlp is described by the following equation, which holds for every statement S [6]:

$$(\forall\, Q \bullet \ wp(S, Q) \ \equiv \ wp(S, true) \ \wedge \ wlp(S, Q)\ ) \tag{0}$$

The semantics of the statements in the simple language are defined by the following weakest preconditions, for any predicate Q [6, 12]:

$Stmt$	$wp(Stmt, Q)$	$wlp(Stmt, Q)$
$x := E$	$Q[x := E]$	$Q[x := E]$
$\textbf{assert}\ E$	$E \wedge Q$	$E \Rightarrow Q$
$\textbf{assume}\ E$	$E \Rightarrow Q$	$E \Rightarrow Q$
$S \ ; \ T$	$wp(S, wp(T, Q))$	$wlp(S, wlp(T, Q))$
$S \ \square \ T$	$wp(S, Q) \wedge wp(T, Q)$	$wlp(S, Q) \wedge wlp(T, Q)$

(1)

where $Q[x := E]$ says about E what Q says about x, that is:

$$Q[x := E] \ = \ \textbf{let}\ x = E\ \textbf{in}\ Q\ \textbf{end}$$

The verification condition for a given program S—which, recall, is a formula that is valid if and only if S is free of errors—is therefore the formula $wp(S, true)$. One way for a program checker to compute this verification condition from a program is to syntactically expand $wp(S, true)$ as suggested by the shapes of the formulas shown in (1). This computation is an instance of the general strategy for correctly computing a verification condition: syntactically transform the formulas according to valid mathematical properties.

2 The problem of redundancy

The problem with the verification conditions computed as suggested by (1) becomes clear when we consider an if statement: in the computation of $wp(S \;\square\; T, Q)$, which expands to $wp(S, Q) \;\wedge\; wp(T, Q)$, we duplicate Q. This results in a verification condition whose size is exponential in the size of the program. If size were the only problem, we could easily provide a fix by naming the common subexpression:

$$wp(S \;\square\; T, Q) \;\;\equiv\;\; \mathbf{let}\; q = Q \;\mathbf{in}\; wp(S, q) \;\wedge\; wp(T, q) \;\mathbf{end} \tag{2}$$

This equation expresses the same mathematical property about $wp(S \;\square\; T, Q)$ as in (1), but computing $wp(S \;\square\; T, Q)$ by syntactically expanding it as suggested by equation (2) results in a verification condition that is just linear in the size of the program.[0]

Unfortunately, size does not matter, or, more accurately, size is not all that matters. Depending on its structure, even a syntactically small verification condition can push an automatic theorem prover beyond the practical limit of an exponential cliff. What matters is how the theorem prover will go about processing the given formula.

Given a formula whose top-level operator is a conjunction, as in

$$A \;\wedge\; B$$

ESC/Modula-3 and ESC/Java's theorem prover, Simplify [4], first attempts to prove A and then attempts to prove B, and it is easy to imagine other theorem provers that would follow the same strategy.[1] Consequently, by syntactically expanding $wp(S \;\square\; T, Q)$ as suggested by (1), a proof obligation in a program will lead to twice as many copies of the same (or similar) proof obligations for every preceding if statement, even when which branch is taken in an if statement is inconsequential to the proof obligation downstream of the if. By introducing a name, like q in (2), for the common subexpression Q, we do not change the fundamental way in which the theorem prover will attempt to prove the given formula: the theorem prover would still have to consider q as many times as it had to consider Q. We'll have to try something else.

[0] The formulation with "$\mathbf{let}\; q$" is correct only if q attains an appropriate higher-level status, so that, for example, $q[x := E]$ still means the right thing. One way to encode that is to explicate the dependence of q on the program variables, as in $q(x, y, z)$ where x, y, z is the list of program variables. With this encoding, the $\mathbf{let}$ formulation yields a formula that is quadratic in the size of the program.

[1] Actually, Simplify works by negating the given formula and trying to find a satisfying assignment for the negation. So, instead of attempting to prove A and then B, Simplify actually first attempts to satisfy $\neg A$ and then $\neg B$. But in either case, any common proof obligations in A and B end up being considered twice.

3 Reducing redundancy

Another way to avoid duplicating the second argument in the expansion of $wp(S, Q)$ is to change the formula into something that replaces the second argument with something that's independent of Q, like a constant. Lo and behold, we know a formula for doing just that, namely the connection between wp and wlp: formula (0) allows us to compute $wp(S, true)$ instead of $wp(S, Q)$, provided we also compute $wlp(S, Q)$. But here we encounter the same problem, because computing $wlp(S \;\square\; T, Q)$ as suggested by (1) suffers from the same kind of duplication as does $wp(S \;\square\; T, Q)$.

Encouraged by the trick of replacing Q with a constant in $wp(S, Q)$, let's try the same for $wlp(S, Q)$. The constant $true$ won't work, however, because $wlp(S, true)$ is $true$ for every statement S [6]. Instead, consider the following "dream property": [2]

$$(\forall\, Q \bullet\ wlp(S, Q) \ \equiv\ wlp(S, false) \ \vee\ Q\ ) \tag{3}$$

Using this dream property, we'd be done, because then, to compute $wp(S, Q)$, we can simply compute

$$wp(S, true) \ \wedge\ (wlp(S, false) \ \vee\ Q)$$

in which Q is not duplicated. To illustrate further, if the statement is a choice statement, we can compute $wp(S \;\square\; T, Q)$ as

$$wp(S, true) \ \wedge\ wp(T, true) \ \wedge\ ((wlp(S, false) \ \wedge\ wlp(T, false)) \ \vee\ Q)$$

But there's a wrinkle: our dream property does not hold for every statement S. To more convincingly describe which statements S have the dream property, let us rewrite the dream property into the form

$$(\forall\, Q \bullet\ Q \ \Rightarrow\ wlp(S, Q)\ ) \tag{4}$$

which is equivalent to the previous formulation:

Proof ((3) $\equiv$ (4)). First, we show that (4) follows from (3):

$$\begin{array}{ll}
 & Q \\
\Rightarrow & \qquad \{\ \text{logic}\ \} \\
 & wlp(S, false) \ \vee\ Q \\
= & \qquad \{\ (3)\ \} \\
 & wlp(S, Q)
\end{array}$$

Next, we show that (3) follows from (4), which we do by assuming (4) and establishing three properties whose conjunction is equivalent to (3):

$$wlp(S, false) \ \Rightarrow\ wlp(S, Q) \tag{5}$$

$$Q \ \Rightarrow\ wlp(S, Q) \tag{6}$$

$$wlp(S, Q) \ \Rightarrow\ wlp(S, false) \vee Q \tag{7}$$

[2] A different way to write dream property (3) is:

$$(\forall\, P, Q \bullet\ wlp(S, P \ \vee\ Q) \ \equiv\ wlp(S, P) \ \vee\ Q\ )$$

A simple mutual-implication argument shows that this formula is equivalent to (3).

Property (5) follows directly from the monotonicity of $wlp(S, \cdot)$ (which is a consequence of $wlp(S, \cdot)$ being conjunctive [6, 7]). Property (6) is exactly the assumed (4). Finally, property (7) is equivalent to

$$wlp(S, Q) \ \wedge \ \neg Q \ \Rightarrow \ wlp(S, \mathit{false})$$

which we establish by the following calculation:

$$\begin{array}{ll}
 & wlp(S, Q) \ \wedge \ \neg Q \\
\Rightarrow & \qquad \{ \text{ (4) with } Q := \neg Q \ \} \\
 & wlp(S, Q) \ \wedge \ wlp(S, \neg Q) \\
= & \qquad \{ \ wlp(S, \cdot) \text{ is conjunctive } \} \\
 & wlp(S, Q \ \wedge \ \neg Q) \\
= & \qquad \{ \text{ logic } \} \\
 & wlp(S, \mathit{false})
\end{array}$$

∎

We have now established that the dream property can be expressed as formula (4). This formula lets us think about what the dream property means in terms of program statements. It says that every non-blocking execution of S that starts in a state satisfying some predicate Q either goes wrong or terminates in a state that also satisfies Q. So the dream property apparently characterizes those statements S that terminate only without any net effect on the program state. These statements are the *passive commands*, which exclude assignment statements [9].

To summarize, we have now arrived at a technique by which we can compute weakest preconditions of statements with reduced redundancy. To compute $wp(S, Q)$ for a Q that is not the literal true, compute $wp(S, \mathit{true})$ and $wlp(S, Q)$ and take the conjunction of the two; to compute $wlp(S, Q)$ for a Q that is not the literal false, compute $wlp(S, \mathit{false})$ and take the disjunction of it and Q; and to compute $wp(S, \mathit{true})$ and $wlp(S, \mathit{false})$, apply the syntactic transformations suggested by (1). The resulting formula for $wp(S, Q)$ is quadratic in the size of S: the expansion will have a $wp(\cdot, \mathit{true})$ term for every atomic substatement of S and a $wlp(\cdot, \mathit{false})$ term for every substatement of S, each such $wlp(\cdot, \mathit{false})$ term being linear in the size of the substatement.

Since the technique works only for passive commands, one first has to eliminate assignment statements from the program under consideration. This elimination can be done by transforming the program into something like an SSA form [0,3] and then changing the assignment statements into assume statements [9]. So, instead of translating the source language into the intermediate language in section 1 and then computing weakest preconditions as suggested by (1), a program checker would translate the source language into passive commands (quite possibly by using the language of section 1 as an intermediate stepping stone) and then computing weakest preconditions as described in the previous paragraph.

4 Programs with exceptions

In some programming languages, statements can terminate not just normally but also *exceptionally*. To think about such language features, we extend the simple language above as follows:

$$\begin{array}{rcl} S, T & ::= & \ldots \\ & | & \mathbf{raise} \\ & | & S \;!\; T \end{array}$$

where $\mathbf{raise}$ raises the exception and $S \;!\; T$ prescribes T as the handler for any exception that escapes S. More precisely, statement $\mathbf{raise}$ always terminates exceptionally without changing the program state. Sequential composition $S \;;\; T$ executes S and then, if S terminates normally, executes T. Dually, $S \;!\; T$ executes S and then, if S terminates exceptionally, executes T.

For programs with exceptions, we define wp and wlp with three arguments, one statement and two predicates on the post-state [2]: $wp(S, Q, R)$ characterizes all pre-states from which every non-blocking execution does not go wrong and either terminates normally in Q or terminates exceptionally in R, and $wlp(S, Q, R)$ characterizes all pre-states from which every non-blocking execution goes wrong, terminates normally in Q, or terminates exceptionally in R. In the presence of exceptions, the connection between wp and wlp reads

$$(\forall\, Q, R \bullet\; wp(S, Q, R) \;\equiv\; wp(S, true, true) \;\wedge\; wlp(S, Q, R)\,) \tag{8}$$

The dream property for programs with exceptions is

$$(\forall\, A, B, Q \bullet\; wlp(S, A \vee Q, B \vee Q) \;\equiv\; wlp(S, A, B) \;\vee\; Q\,) \tag{9}$$

whose similarity with the dream property of the previous section is best seen when compared with the formulation in footnote 2. Using these properties, we can compute $wp(S, Q, R)$ by computing the last line in the following calculation:

$$\begin{array}{ll}
 & wp(S, Q, R) \\
= & \qquad \{\; wp\text{-}wlp \text{ connection (8) } \} \\
 & wp(S, true, true) \;\wedge\; wlp(S, Q, R) \\
= & \qquad \{\; \text{conjunctivity of } wlp \text{ in its second and third arguments } \} \\
 & wp(S, true, true) \;\wedge\; wlp(S, Q, true) \;\wedge\; wlp(S, true, R) \\
= & \qquad \{\; \text{logic } \} \\
 & wp(S, true, true) \;\wedge \\
 & \qquad wlp(S, false \vee Q, true \vee Q) \;\wedge\; wlp(S, true \vee R, false \vee R) \\
= & \qquad \{\; \text{dream property (9) } \} \\
 & wp(S, true, true) \;\wedge \\
 & \qquad (wlp(S, false, true) \vee Q) \;\wedge\; (wlp(S, true, false) \vee R)
\end{array}$$

For exceptional programs, the dream property again holds for exactly those statements that do not update the state.

In this rewriting, the resulting verification condition can still be exponential in the size of the program, see [9] for details.

5 Conclusion

Let's compare this weakest-precondition understanding of the ESC/Java technique for generating verification conditions with its previous description [9]. Flanagan and Saxe define two functions on statements, N and W. For any statement S, $N(S)$ characterizes those initial states from which execution of S *may* terminate normally, and $W(S)$ characterizes those initial states from which execution of S *may* go wrong. That is, $\neg W(S)$ characterizes those states from which S is guaranteed *not* to go wrong, and $\neg N(S)$ characterizes those states from which S is guaranteed *not* to terminate normally. In other words, we have

$$\begin{array}{rcl} \neg W(S) & = & wp(S, true) \\ \neg N(S) & = & wlp(S, false) \end{array}$$

These equations were previously not pointed out.

For programs with exceptions, Flanagan and Saxe additionally define a function X such that for any statement S, $X(S)$ characterizes those initial states from which execution of S *may* terminate exceptionally. Thus, in terms of the wp and wlp for exceptional programs, we have

$$\begin{array}{rcl} \neg W(S) & = & wp(S, true, true) \\ \neg N(S) & = & wlp(S, false, true) \\ \neg X(S) & = & wlp(S, true, false) \end{array}$$

There are two key advantages of the weakest-precondition understanding of the ESC/Java technique for generating verification conditions. One advantage is that one can use the standard wp - wlp semantics of the program statements, which focuses on what's necessary to *guarantee* particular post-states, as opposed to having to define what it means that a statement *may* have some particular outcome. The other, larger, advantage is that it draws out the very property that needs to hold of the statements in order to apply the technique. This property, which can be seen as a wlp distribution property (in formula (3) and in footnote 2) or as an invariant-preserving property (in formula (4)), holds exactly of statements that don't alter the program state, the passive commands.

Acknowledgments The ESC/Java technique for generating verification conditions was developed by Jim Saxe, Greg Nelson, and Cormac Flanagan as a variation of the technique used in ESC/Modula-3 and invented by Saxe, Nelson, and Dave Detlefs. The general technique, which can also be used with *strongest postconditions*, as it was in ESC/Modula-3, is described in a patent [13].

I'm grateful to Rajeev Joshi for suggesting that I prove the equivalence between the dream property (3) and property (4), which more readily reveals the property as saying "S has no effect on the program state". Thanks also to Cormac Flanagan for providing comments on this paper.

References

0. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11. ACM, January 1988.

1. R. J. R. Back and J. von Wright. Combining angles, demons and miracles in program specifications. *Theoretical Computer Science*, 100(2):365–383, 1992.
2. Flaviu Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163–174, 1984.
3. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35. ACM, January 1989.
4. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
5. 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.
6. Edsger W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1976.
7. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
8. 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.
9. 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.
10. K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*. Springer, 2000.
11. 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.
12. Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.
13. James Benjamin Saxe, Charles Gregory Nelson, and David Luke Detlefs. Case-reduced verification condition generation system and method using weakest precondition operator expressed using strongest postcondition operators. United States Patent #6,553,362, filed 16 July 2001, issued 22 April 2003.