Verification of loops and exceptions

Ryan Stansifer

Report Number:
86-631
VERIFICATION OF LOOPS AND EXCEPTIONS

Ryan Stansifer

CSD-TR-631
September 1986
Verification of loops and exceptions

Ryan Stansifer
Purdue University
West Lafayette, Indiana 47907
Telephone: (317) 494-7821
ryan@Purdue.edu

Abstract. We give a proof rule for a multiple-level exit construct not unlike the loop-exit statement in the ADA* programming language. We give a novel, yet simple, semantics for the loop-exit with which we can prove that the rule is both sound and (relatively) complete in the logic of Hoare triples. Hence, we can be satisfied that the proof rule is sufficient to prove all true Hoare triples using the multiple-level exit statement and is suitable for inclusion in a formal verification system. A verification condition generator using these rules is developed using a general method based on attribute grammars.

Introduction. The programming language ADA has a general loop statement encompassing three different forms (or iteration schemes as they are called in section 5.5 of the reference manual). One form subsumes the other two, that is, the other forms can be derived from it. It is this most general case that we consider here. The syntax of this loop construct looks like:

\[ l : \text{loop } S \text{ end loop } l; \]

where \( l \) is the label of the loop. Execution of the loop statement proceeds by repeatedly executing the statements in \( S \) until a statement of the following form is encountered:

\[ \text{exit } l \text{ when } B; \]

When the boolean condition \( B \) evaluates to true, the execution of the loop labeled \( l \) is ended and the next statement in sequence after the loop is executed. If the condition is false, execution of the loop continues with the next statement in sequence after the exit statement. This exit statement can be encountered while nested inside of more than one loop. Hence the execution of a program can jump out of loops nested arbitrarily deep. This complex flow of control is the cause of much travail in the denotational description of such constructs.

The approach that we will describe here for loops is also appropriate for the jumps in the flow of control caused by the raising and handling of exceptions. So we also examine an ADA-like raise construct. The raise statement:

\* ADA is a registered trademark of the U.S. Government, ADA Joint Program Office.
raise e;

halts the normal sequential execution of a program. If the statement is enclosed by a block with an exception handler for the named exception, then execution resumes with the handler. The ADA syntax for a block with an exception handler looks like this:

```
declare
    exception e
begin
    S_1
exception
    when e => S_2
end;
```

If during the course of the execution of the program segment $S_1$ the statement `raise e` is encountered, execution continues with the program segment $S_2$. The name of the exception is declared in the declaration section of the block. This is directly analogous to the implicit declaration of the label in a labeled loop statement. In this paper we do not consider procedure calls, so we do not treat the case of propagating an exception back to the calling routine, although this is an important part of the ADA exception mechanism.

In this paper we give a denotational semantics for a simple language containing an ADA-like loop-exit construct and an ADA-like raise-handle construct. For this language we present an axiom system for deriving assertions about the correctness of programs. From this axiom system it is possible to determine what verification conditions must be generated by a verification system for ADA. This is done in the last sections. Although the language focuses on a few constructs, certain generalizations are immediate (like the inclusion of the conditional construct). We show that this axiom system is sound using the denotational semantics given here. This is the least we can expect of the axiom system, and it insures that we can safely use it. We also show that this axiom system is (relatively) complete. Completeness guarantees that anything that is true about the loop language does have a proof in the axiom system we give. This is important because it means we can stop looking for a more comprehensive set of proof rules. The proof of completeness requires a type of definition for the semantics of the loop language slightly different than the traditional one, but the definition integrates easily into the traditional definition. This permits the incorporation of yet other generalizations from the literature of denotational semantics.

We assume the reader is familiar with Hoare triples [Hoare, 1969], and somewhat familiar with denotational semantics [Stoy, 1977] and the classical soundness and completeness results for Hoare logic, for example [Loeckx et al., 1984].

The while loop. We begin by considering the proof rule for the while loop, which is a special case of the loop statement. The proof rule shows how to derive Hoare triples from other Hoare triples. Hoare triples are statements of the form $\{ P \} S \{ Q \}$, where
\[ \{I \land B\} S \{I\} \]
\[ \{I\} \text{while } B \text{ loop } S \text{ end loop; } \{I \land \neg B\} \]

This rule permits one to derive Hoare triples concerning the while loop, if one can derive the premise about \( S \), the body of the loop. The assertion \( I \) is called the loop invariant. Cook proved that this rule was sound and (relatively) complete in a computational semantics for while programs [Cook, 1978]. A similar result was proved by de Bakker who used denotational semantics to assign meaning to the program segments [de Bakker, 1980]. In denotational semantics each statement of the language is denoted by a function that transforms states to states. We use the symbol \( C \) for the traditional mapping of programs to their denotations. For example, the denotation of the while loop with condition \( B \) and body \( S \) is the recursive function \( f_{\text{wh}} \) defined as follows (we use \( \sigma \) as a formal parameter for states):

\[
f_{\text{wh}}(\sigma) = \text{if \text{isTrue} } (E \{B\}\sigma) \text{ then } f_{\text{wh}}(C[S]\sigma) \text{ else } \sigma
\]

where \( E \{B\}\sigma \) is the value of the boolean expression \( B \) in state \( \sigma \), and \( C[S]\sigma \) is the denotation of program segment \( S \) applied to the state \( \sigma \) (in other words, the resulting state obtained after executing \( S \) beginning in state \( \sigma \)). If \( f_{\text{wh}}(\sigma) \) does not terminate, we will pretend its value is a specially designated state called \( \text{error} \). Using this function \( f_{\text{wh}} \) we write the case in the recursive definition of \( C \) for the while loop as:

\[
C[\text{while } B \text{ loop } S \text{ end loop; } \sigma = f_{\text{wh}}(\sigma) \]

Each construct of the language is given its denotation by a case in the definition of \( C \). For example, the meaning of the sequential execution of two statements is:

\[
C[S_1 S_2]\sigma = C[S_2](C[S_1]\sigma)
\]

(We assume that the definition of a semantic function maintains the strictness of the function, i.e., if \( \sigma = \text{error} \) then \( C[S]\sigma = \text{error} \), even if not specifically provided for in the definition.)

Using this definition of the meaning of the while loop we can give a precise meaning to the Hoare triple \( \{P\} \text{while } B \text{ loop } S \text{ end loop; } \{Q\} \). This meaning, or interpretation, reveals when the Hoare triple is "true" or not. We say that the Hoare triple above is "true" if the following formula of first-order predicate logic is true:

\[
\forall \sigma. (P \text{ is true in } \sigma) \land (\sigma' \neq \text{error}) \Rightarrow (Q \text{ is true in } \sigma')
\]
where $\sigma' = f_{\text{WH}}(\sigma)$. By taking the definition of $C$ for all of the constructs $S$ in a language, we can give the interpretation of any Hoare triple $\{P\} S \{Q\}$ by taking $\sigma' = C[S]\sigma$. We have defined here a logic for partial correctness, since termination is a hypothesis in the implication above.

The loop-exit construct. The proof rule for the loop statement is reminiscent of the rule for the while statement. Here is the informal presentation of the rule.

$$
\frac{\{I\} S \{I\}}{
\{I\} l : \text{loop } S \text{ end loop } l; \{Q_1\}
}$$

Like the while rule, the rule for the loop statement has an invariant assertion which we have called $I$ in the rule above. Execution of $S$, the body of loop, must maintain the assertion $I$. It is interesting to note that the loop rule requires just one invariant assertion despite the possibility of multiple exit statements in the body of the loop. For reasoning about the exit statement there is the following axiom:

$$
\{(Q_1 \& B) \mid (Q \& \neg B)\} \text{ exit } l \text{ when } B; \{Q\}
$$

The precondition says either the boolean condition is true and the postcondition for the loop is true, or the boolean condition is false and the postcondition is true. Here $Q_1$ is the postcondition of the loop labeled $l$.

Now we check to see if we can derive certain obviously true Hoare triples from these rules. These rules would be inadequate if we could not use them to derive even simple Hoare triples concerning the loop and exit statements. Our purpose at present is to strengthen the plausibility of these rules. Later we will prove that these rules derive only true Hoare triples, and that all true Hoare triples can be proved using these rules.

For example, if the loop does not terminate we expect to conclude any postcondition $Q$ (since this is a property of partial correctness logics). So we believe intuitively that the following Hoare triple is true and should be derivable:

$$
\{P\} l : \text{loop null; end loop } l; \{Q\}
$$

Indeed, this is derivable by a single application of the loop statement rule, if the Hoare triple $\{P\} \text{ null; } \{P\}$ is derivable. This last Hoare triple can be taken as the meaning of the null or skip statement.

Another way the loop may not terminate is if the guard on the exit is always false, as in the program segment of the next Hoare triple:

$$
\{P\} l : \text{loop exit } l \text{ when } false; \text{ end loop } l; \{Q\}
$$

The previous Hoare triple should be true, regardless of the assertions $P$ and $Q$. This is derivable using the loop statement rule, if the Hoare triple $\{P\} \text{ exit } l \text{ when } false; \{P\}$ is derivable. This follows from

$$
\{(Q \& false) \mid (P \& \neg false)\} \text{ exit } l \text{ when } false; \{P\}
$$
which is an instance of the exit rule. Finally we expect the Hoare triple

$$\{P\} l : \text{loop exit } l \text{ when } \text{true}; \text{ and loop } l; \{P\}$$

to be true, since this program segment acts like a no-op instruction. The Hoare triple above is derivable, since

$$\{(P \& \text{true}) \mid (P \& \neg \text{true})\} \text{ exit } l \text{ when true; } \{P\}$$
is an instance of the exit rule.

As further evidence of the plausibility of the loop-exit rules we show that, for the special case in which the loop construct is identical to a while loop, the loop-exit rules are equivalent with the well-known rule for the while construct. The following is an instance of the axiom given above for the exit statement.

$$\{(I \& \neg B) \mid (I \& B \& \neg \neg B)\} \text{ exit } l \text{ when } \neg B; \{I \& B\}$$

Since the precondition above is identical to $I$ we have:

$$\{I\} \text{ exit } l \text{ when } \neg B; \{I \& B\}$$

Given the premise of the rule for while loops, namely $\{I \& B\} S \{I\}$ we can derive

$$\{I\} \text{ exit } l \text{ when } \neg B; S; \{I\}$$

using the well-known law of composition. The last Hoare triple is exactly the premise needed in the rule for loop statements to give the following conclusion:

$$\{I\} l : \text{loop inv } I \text{ do exit } l \text{ when } \neg B; S; \text{ end loop } l; \{Q\}$$

The program segment in the Hoare triple above is equivalent to the while loop:

```
while B loop S end loop;
```

The raise-handle construct. The raise statement acts much like the exit statement with a constantly true guard. Therefore, it is not surprising that the rule for the raise statement looks like:

$$\{P_e\} \text{ raise } e \{\text{false}\}$$

Here the assertion $P_e$ is the precondition of the handler for the exception $e$. The assertion \text{false} as the postcondition of the statement means that one can assert anything at all to be true after a raise statement, because every assertion is vacuously true due to fact that control never reaches that point.

We abbreviate the ADA block:
declare
  exception e
begin
  $S_1$
exception
  when $e \Rightarrow S_2$
end:

by begin $S_1$ when $e \Rightarrow S_2$ end; so that programs will take up less space.

Now we can give the rule for the block:

\[
\frac{\{P\} S_1 \{Q\}, \{P_e\} S_2 \{Q\}}{\{P\} \text{begin } S_1 \text{ when } e \Rightarrow S_2 \text{ end; } \{Q\}}
\]

We have called the precondition of the exception handler $P_e$ to indicate that it is the same assertion that must be used in applications of the axiom for the raise statement. The system that formally enforces this constraint (and the similar constraint on the postcondition of loops) will be presented later.

In the next section we next turn to proving the soundness and (relative) completeness of this axiom system. To that end we define formally our ADA-like language and its semantics. We introduce assumptions and define the validity of Hoare triples relative to these assumptions. Finally we present the formal axiom system for the language and prove the soundness and completeness.

A simple language. Here is the syntax of the language we wish to consider. We start with two essential constructs.

\[
S ::= \text{assign}; | S \ S
\]

To the assignment statement and composition we add the loop-exit statements.

\[
S ::= l : \text{loop } S \text{ end loop } l; | \text{exit } l \text{ when } B;
\]

To the language thus far we add a block with an exception handler and the raise statement.

\[
S ::= \text{begin } S \text{ when } e \Rightarrow S \text{ end; | raise } e;
\]

We shall assume that loops and exceptions are labeled uniquely in all program segments in this language, and we assume that loop labels are distinct from exception names. We call the program segment closed if all statements of the form exit $l_0$ when $B$ are nested inside a loop statement labeled $l_0$ and if all statements of the form raise $e$; are nested inside a block with an exception handler for $e$:

\[
\text{begin } S_1 \text{ when } e \Rightarrow S_2 \text{ end;}
\]

An ADA compiler must check that all exit statements are properly nested and that raise statements appear in the static scope of an exception declaration. On the other
hand, ADA does not require some exception handler to catch each exception (as we have
done here in this paper). Indeed, it is expected and useful for exceptions to propagate
up the dynamic chain. This kind of behavior is intimately linked to the procedure calling
mechanism. An axiomatic treatment can be found in [Luckham and Polak, 1980]. A
complete axiom system for exception handling in this case is not likely due to results of
[Clarke, 1979].

Next we give a formal semantics for the language just introduced. The denotations
we give to program segments differ slightly from the typical denotations. Instead of trans­
formations from states to states (States → States), we use transformations from states
to pairs of labels and states (States → (Con × States)), where Con is the set of possible
identifiers for loops and exceptions. We must add a special designator ne (for normal exit)
to the set of labels. This designator indicates that the normal sequential execution has
been followed. We assume that the set of loop labels and exception names are disjoint
and that ne occurs in neither set. The denotations we give to program segments of the
simple loop language are functions from states to pairs of labels and states. The semantic
function that maps program segments to their denotations is denoted \(X\), and it has the
functionality:

\[
X : S \rightarrow \text{States} \rightarrow ((ne + Con) \times \text{States})
\]

Next we give the six cases in the recursive definition of the semantic function \(X\). The
first case is that of assignment. We assume the assignment statement modifies the state
in some manner.

\[
X[\text{assign;} \; j = e] \sigma = (ne, \sigma')
\]

The state \(\sigma'\) is the resulting, modified state after the assignment. The details of the
modification are of no importance to the present discussion.

The next case is for the sequential execution of two program segments. If the execution
of \(S_1\) proceeds normally, then the program segment \(S_2\) is executed in the resulting state.
Otherwise, the execution of \(S_2\) is skipped.

\[
X[S_1; S_2] \sigma = \begin{cases} 
(X[S_1] \sigma \text{ in } & \text{if } j = ne \text{ then } X[S_2] \sigma' \text{ else } (j, \sigma') 
end}
\]

The next two rules are for the two types of statements in the language which together
constitute the loop-exit construct. The exit statement is a statement whose execution can
result in initiating a path of execution that is not the "normal" sequential path represented
by the pair \((ne, \sigma)\). But this occurs only if the guard is true, in which case the result is
\((l, \sigma)\) where \(l\) is the label of the loop being exited. Notice that the exit statement does not,
in either case, change the state.
\( X[exit \; l \; \text{when} \; B; \; \sigma = \)
\[
\quad \text{if} \ \text{IsTrue} \ (\mathcal{E} [B] \; \sigma) \ \text{then} \ (l, \sigma) \ \text{else} \ (ne, \sigma) \\
\]

As in the case of the while statement, the denotation of the loop statement is a recursively defined function. We have called the function \( \text{flp} \) below.

\[
X[\ell : \; \text{loop} \; S \; \text{end} \; \text{loop} \; \ell]\sigma = \text{flp}(\sigma) \\
\quad \text{where rec} \ \text{flp}(\sigma) = \\
\quad \quad \text{let} \ (c, \sigma') = X[S]\sigma \ \text{in} \\
\quad \quad \quad \text{if} \ c = ne \ \text{then} \ \text{flp}(\sigma') \\
\quad \quad \quad \text{else if} \ c = l \ \text{then} \ (ne, \sigma') \\
\quad \quad \quad \text{else} \ (c, \sigma') \\
\quad \text{end} \\
\]

The function \( \text{flp} \) keeps calling itself recursively until \( c \neq ne \). If \( c \) is the label of the current loop then the loop statement exits normally. This is an instance in the definition of \( X \) that a subcomponent, in this case the body of the loop, exits with \( c \neq ne \) and the language construct transforms it to a normal termination \( c = ne \). The final case in the definition of \( \text{flp} \) is when \( c \neq ne \) and \( c \neq l \). In this case, the exit of another loop or an exception is propagated, presumably to be caught by the appropriate construct. If the loop does not terminate, we set the value of \( \text{flp} \) arbitrarily to \( (ne, error) \). (Making the set of control designators into a lattice by adding a bottom element is not necessary.)

The remaining two rules are for the raise-handle construct.

\[
X[\text{raise} \; e; \; \sigma = (e, \sigma) \\
\]

The raise statement does not change the state, but always sets the control designator to \( e \) to indicate that exception \( e \) has been raise. The block statement catches the declared exception, if it has been raised in the body of the block, and executes the handler.

\[
X[\text{begin} \; S_1 \; \text{when} \; e \Rightarrow S_2 \; \text{end} ; \; \sigma = \\
\quad \text{let} \ (c, \sigma') = X[S_1]\sigma \ \text{in} \\
\quad \quad \text{if} \ c = e \ \text{then} \ X[S_2]\sigma' \\
\quad \quad \text{else} \ (c, \sigma') \\
\quad \text{end} \\
\]

If the declared exception is not raised, the semantics of the block statement is no different than the semantics of the body of the block.
For the purposes of defining which Hoare triples are true, we first define what we mean by an assumption. An assumption is a pair consisting of a control designator and an assertion. It is intended that the assumption \((c, R_c)\) represent the fact that \(R_c\) is the postcondition of the loop labeled \(c\), if \(c\) is a loop label. If \(c\) is an exception name, then \((c, R_c)\) represents that fact that \(R_c\) is the precondition of the exception handler for \(c\). We shall be interested in sets of assumptions in which a control designator occurs at most once. We will call these sets proper. If \(\Phi\) is a proper set of assumptions, then the set of all control designators \(c\) such that \((e, R_e)\) is in \(\Phi\) is called the domain of \(\Phi\) and is written \(\text{Dom}(\Phi)\). This technique is inspired by a similar construction for goto statements [de Bruin, 1980].

We say that the Hoare triple \(\{P\} S \{Q\}\) is valid with respect to a proper set of assumptions \(\Phi\) (we will write this as \(\Phi \vdash \{P\} S \{Q\}\)) whenever \(\forall \sigma. j \neq ne \Rightarrow j \in \text{Dom}(\Phi)\) and

\[
\forall \sigma. (P \text{ is true in } \sigma) \& (\sigma' \neq \text{error}) \Rightarrow (Q_j \text{ is true in } \sigma')
\]

where \((j, \sigma') = X[S]\sigma\) and the assertion \(Q_j\) is defined as follows:

\[
Q_j = \begin{cases} 
Q_i, & \text{if } j = ne; \\
Q_{it}, & \text{if } j = c \text{ for some } (l, R_c) \in \Phi.
\end{cases}
\]

Intuitively the notion of validity means that if \(P\) is true in \(\sigma\) and \(S\) exits normally, then \(Q\) is true in the resulting state, and if \(S\) exits via some "abnormal" thread of control \(c\), then \(R_c\) is true in the resulting state. If \(c\) is not in the domain of \(\Phi\), the Hoare triple is automatically invalid.

Whenever \(S\) is closed (i.e., when \(j = ne\) for all \(\sigma\)), the definition of validity corresponds to the usual one, because \(Q_j\) is always \(Q\). Thus the semantics for the loop-exit construct presented here can be easily integrated into the usual semantics with the following definition:

\[
C[S]\sigma =
\begin{align*}
\text{let } (j, \sigma') &= X[S]\sigma \text{ in} \\
&\quad \text{if } j = ne \text{ then } \sigma' \text{ else error}
\end{align*}
\]

Thus, the denotations of closed statements can be viewed as state transformations just like the classical approach. This is important in fitting together these rules for the loop-exit construct with results on other constructs like procedure call rules.

The details. Next we give the precise rules for the loop-exit construct and raise-handle construct. This requires making the previous rules relative to proper sets of assumptions. Here is the rule for the loop statement:

\[
\Phi \cup \{l, Q_l\} \vdash \{I\} S \{I\}
\]

\[
\Phi \vdash \{I\} l : \text{loop } S \text{ end loop } l; \{Q_l\}
\]

We have discharged the assumption \((l, Q_l)\) by enclosing \(S\) in the loop labeled \(l\). The exit rule, given below, introduces the assumption \((l, Q_l)\).

\[
\left(\{Q_l \& B\} | (Q \& \neg B)\right) \text{ exit } l \text{ when } B; \{Q\}
\]
The rules for the raise-handle construct also introduce and discharge the appropriate assumption.

\[
(e, P_e) \vdash \{ P_z \} \text{raise } e \{ \text{false} \}
\]

\[
\Phi \cup (e, P_e) \vdash \{ P \} S_1 \{ Q \}, \quad \Phi \vdash \{ P_z \} S_2 \{ Q \}
\]

\[
\Phi \vdash \{ P \} \text{begin } S_1 \text{ when } e \Rightarrow S_2 \text{ end } \{ Q \}
\]

We list the remainder of the axiom system to show the effect of relativising the usual rules. The assignment axiom modifies the post condition in some way which we leave unspecified.

\[
\emptyset \vdash \{ P' \} \text{assign; } \{ P \}
\]

The rule for the juxtaposition of two statements does not deviate substantially from the original rule proposed by Hoare. Notice that by the definition of validity each program segment \( S_1 \) and \( S_2 \) must "capture" its own exceptions and loop exits in order for the premises to be valid. It is not possible for the other list of assumptions to incorrectly handle those cases.

\[
\Phi \vdash \{ P \} S_1 \{ Q \}, \quad \Psi \vdash \{ Q \} S_2 \{ R \}
\]

\[
\Phi \cup \Psi \vdash \{ P \} S_1 S_2 \{ R \}
\]

The last rule is the rule of consequence:

\[
P_1 \Rightarrow P_2, \quad \Phi \vdash \{ P_2 \} S \{ Q_1 \}, \quad Q_1 \Rightarrow Q_2
\]

\[
\Phi \vdash (c, R_c) \vdash \{ P_1 \} S \{ Q_2 \}
\]

This rule allows a Hoare triple to be modified in any one of three independent ways: strengthening the precondition, weakening the postcondition, or expanding the assumptions.

With the precise statement of the rules of inference of the simple programming language, it is now possible to give the proof of soundness and (relative) completeness. We do not give the whole proof as that would not be illuminating, but instead sketch the relevant cases.

**Soundness of the exit statement.** We are to show that the Hoare triple \( \{(Q_1 \& B) | (Q \& \neg B)\} S \{ Q \} \) is valid with respect to \((l, Q_1)\), where \( S \) is the program segment exit \( l \) when \( B \). Let \( \sigma \) be such that \( (Q_1 \& B) | (Q \& \neg B) \) is true in \( \sigma \). The proof breaks into two cases. Either \( B \) is true in \( \sigma \), or it is not. If \( B \) is true, then \( Q_1 \) must be true. Therefore, since the exit statement does not change the state we have the desired conclusion, namely, \( Q_1 \) is true in \( \sigma \). In the case that \( B \) is false in \( \sigma \), \( P \) must be true. Either way, the Hoare triple is valid.

**Completeness of the exit statement.** We are given that the Hoare triple \( \{ P \} S \{ Q \} \) is valid with respect to \((l, Q_1)\), where \( S \) is the program segment exit \( l \) when \( B \). We must show we can derive this Hoare triple. We can derive this triple in two steps using the exit rule and the rule of consequence, if we can show that \( (Q_1 \& B) | (Q \& \neg B) \) \( \Rightarrow P \).

\[
((Q_1 \& B) | (Q \& \neg B)) \Rightarrow P, \quad (l, Q_1) \vdash ((Q_1 \& B) | (Q \& \neg B)) \text{exit } l \text{ when } B; \{ Q \}
\]

\[
(l, Q_1) \vdash \{ P \} \text{exit } l \text{ when } B; \{ Q \}
\]
Assume first that \( P \) and \( \neg B \) are true in \( q \). Then \( X[S] \sigma = (ne, \sigma) \). Since \( \{ P \} S \{ Q \} \) is valid, \( Q \) is true in \( \sigma \). In other words, \( (P \land \neg B) \Rightarrow Q \). Now assume \( P \) and \( B \) are true in \( \sigma \). In this case \( X[S] \sigma = (l, \sigma) \). Since \( \{ P \} S \{ Q \} \) is valid with respect to \( (l, Q_l) \), \( Q_l \) is true in \( \sigma \). Hence both assertions hold.

**Soundness of the loop statement.** Suppose the Hoare triple \( \{ I \} S \{ I \} \) is valid with respect to \( \Phi \cup (l, Q_l) \). Let \( \sigma \) be some arbitrary state such that \( I \) is true in \( \sigma \) (and \( f_(l\sigma) \) terminates). Now define the (finite) sequence

\[
\sigma_0, (ne, \sigma_1), (ne, \sigma_2), \ldots, (ne, \sigma_{n-1}), (j, \sigma_n)
\]

where \( \sigma_0 = \sigma \) and \( (j_i, \sigma_i) = X[S] \sigma_{i-1} \). The sequence stops when \( j_i \neq ne \), corresponding to when the execution of the loop halts and \( f_(l\sigma) \) terminates. By induction it holds that \( I \) is true in \( \sigma_{n-1} \). And thus, if \( j = l \), then \( Q_l \) holds. If \( j \) is some other label in \( \Phi \), then the appropriate assertion holds as well. Hence,

\[
\{ I \} L : \text{loop } S \text{ end loop } l ; \{ Q_l \}
\]

is valid with respect to \( \Phi \). This concludes the proof of soundness.

In the proof of completeness for the loop statement we will need the definition of the weakest precondition. The **weakest precondition** of \( S \) and \( Q \) (relative to a proper set of assumption \( \Phi \)), denoted \( wp(S; Q) \), is that assertion such that

\[
\Phi \vdash \{ wp(S; Q) \} S \{ Q \}, \text{ and } P \Rightarrow wp(S; Q)
\]

for all \( P \) such that \( \Phi \vdash \{ P \} S \{ Q \} \). We assume that the weakest precondition is always expressible in the language of the assertions.

**Completeness of the loop statement.** Suppose the Hoare triple \( \{ P \} L \{ Q \} \) is valid with respect to \( \Phi \), where \( L \) is the program segment \( l : \text{loop } S \text{ end loop } l ; \). We pick \( Q_l \) to be \( Q \) and \( I \) to be \( wp(L, Q) \). So by definition \( \{ I \} L \{ Q \} \) is valid with respect to \( \Phi \). By definition of the weakest precondition \( P \Rightarrow I \), so from \( \Phi \vdash \{ I \} L \{ Q \} \) we can derive \( \Phi \vdash \{ P \} L \{ Q \} \) using the rule of consequence. Therefore, it remains to be proved that \( \Phi \vdash \{ I \} L \{ Q \} \) is derivable. By the induction hypothesis we know we can derive this Hoare triple, if \( \{ I \} S \{ I \} \) is valid with respect to \( \Phi \cup (l, Q_l) \).

Suppose \( I \) is true in \( \sigma \). Now execute the loop \( L \) beginning in this state. If the execution of the body does not complete even once, then we must have \( X[L] \sigma = (j, \sigma') \) and \( j \neq ne \). Since \( \Phi \vdash \{ I \} L \{ Q \} \), \( j = l \) or \( j = l' \) for some \( (l', Q_{l'}) \in \Phi \). If \( X[L] \sigma = (l, \sigma') \) then \( X[L] \sigma = (ne, \sigma') \), and since \( \{ I \} L \{ Q \} \) we have \( Q \) is true in \( \sigma' \). Hence \( (l, Q_l) \vdash \{ I \} S \{ I \} \).

If \( j \neq l \), then

\[
\Phi \cup (l, Q) \vdash \{ I \} S \{ I \}
\]

follows because \( (j, Q_j) \in \Phi \).

On the other hand, executing the loop may execute the body completely at least once. In other words:

\[
\Phi \cup (l, Q) \vdash \{ I \} L \{ Q \} \equiv \Phi \cup (l, Q) \vdash \{ I \} L \{ Q \}
\]

Since \( I \) is the weakest precondition of \( L \) we have \( \{ I \} S \{ I \} \). (We leave the justification to the following lemma.) This completes the proof.
Lemma. Suppose \( \Phi \vdash \{ P \} S_1 S_2 \{ Q \} \) then \( \Phi \vdash \{ P \} S_1 \{ R \} \) where \( R \) is \( \text{wp}(S_2, Q) \). The proof is by contradiction. Suppose \( \Phi \vdash \{ P \} S_1 S_2 \{ Q \} \), but \( \{ P \} S_1 \{ R \} \) is not valid with respect to \( \Phi \). Then there is some state \( \sigma \) for which \( P \) is true in \( \sigma \), but \( R \) is not true in \( \sigma' \), where \( (j, \sigma') \) is \( \mathcal{X}[S_1]\sigma \). (If \( j \neq ne \), then there is an immediate contradiction.) Then in no case is it possible to arrive at a state \( \sigma'' \) in which \( Q \) is true, as that contradicts the assumption that \( R \) is the weakest precondition.

**Soundness of the raise statement.** We want to prove that the Hoare triple

\[
\{ P \} \text{raise } e; \{ \text{false} \}
\]

is valid with respect to \((e, P_e)\). First note that \( \mathcal{X}[\text{raise } e;]\sigma = (e, \sigma) \), for all states \( \sigma \). Consequently the control designator is always in the domain of the assumptions. That is the first condition of validity. Furthermore, if we start in a state \( \sigma \) such that \( P_e \) is true in \( \sigma \), the state does not change and hence \( P_e \) is true after executing the raise statement, hence, the Hoare triple is valid.

**Completeness of the raise statement.** Suppose the Hoare triple \( \{ P \} \text{raise } e; \{ Q \} \) is valid with respect to \( \Phi \). Since \( \mathcal{X}[\text{raise } e;]\sigma = (e, \sigma) \), we must have \((e, P_e) \in \Phi \) for some assertion \( P_e \). Furthermore, we must have that \( P \) implies \( P_e \). We must show that we can derive \( \{ P \} \text{raise } e; \{ Q \} \). This can be done in two steps using the axiom for the raise statement and rule of consequence as follows:

\[
\frac{P \Rightarrow P_e, \quad (e, P_e) \vdash \{ P_e \} \text{raise } e; \{ \text{false} \}, \quad \text{false} \Rightarrow Q}{(e, P_e) \vdash \{ P \} \text{raise } e; \{ Q \}}
\]

**Soundness of the block statement.** Assume that (1) the Hoare triple \( \{ P \} S_1 \{ Q \} \) is valid with respect to \( \Phi \cup (e, P_e) \) and (2) the Hoare triple \( \{ P_e \} S_2 \{ Q \} \) is valid with respect to \( \Phi \). We are to show that \( \{ P \} S \{ Q \} \) is valid with respect to \( \Phi \), where \( S \) is the program segment:

\[
\text{begin } S_1 \text{ when } e \Rightarrow S_2 \text{ end};
\]

Let \( \sigma \) be a state such that the assert \( P \) is true in \( \sigma \). Now consider \( \mathcal{X}[S_1]\sigma = (c, \sigma') \). If \( c \neq e \), then \( \mathcal{X}[S]\sigma = (e, \sigma') \). We must show \( c \) is in the domain of \( \Phi \) (if \( c \neq ne \)) and that the appropriate assertion is true in \( \sigma' \). Both follow immediately from the first assumption. If \( c = e \), then \( \mathcal{X}[S]\sigma = \mathcal{X}[S_2]\sigma' \). By the first assumption we know that \( P_e \) is true in \( \sigma' \). So \( \{ P \} S \{ Q \} \) is valid because of the second assumption.

**Completeness of the block statement.** Suppose that \( \{ P \} S \{ Q \} \) is valid with respect to \( \Phi \), where \( S \) is the program segment:

\[
\text{begin } S_1 \text{ when } e \Rightarrow S_2 \text{ end};
\]
We must show how to derive that Hoare triple. We begin by choosing the assertion $P_e$ to be the weakest precondition of $S_2$ and $Q$. Now if we can prove that

$$\Phi \cup (e, P_e) \vdash \{P\} S_1 \{Q\} \quad \text{and} \quad \Phi \vdash \{P_e\} S_2 \{Q\}$$

are valid (and hence derivable), then we use the block rule to derive the desired Hoare triple. The second Hoare triple above is true by choice of $P_e$.

The proof of validity of the first Hoare triple follows. Let $\sigma$ be a state such that $P$ is true in $\sigma$. If $\llbracket S_1 \rrbracket \sigma = (e, \sigma')$, then the appropriate assertion holds in $\sigma'$ by virtue of the fact that $\{P\} S \{Q\}$ is valid with respect to $\Phi$. On the other hand, if $\llbracket e \rrbracket = e$ why should $P_e$ be true in $\sigma'$? Were that not the case, then since $\{P\} S \{Q\}$ is assumed valid, $P_e$ would not be the weakest precondition.

Generation of verification conditions. A verification system can be built for a programming language using the axioms that describe the statements of the language. This has been done for PASCAL in the Stanford PASCAL verifier. An axiomatic description of PASCAL was published in [Hoare and Wirth, 1973]. This verification system operates by generating a number of formulas in first-order logic (called verification conditions) from a given program and a given specification (in first-order logic) of what the program is to do. If all the formulas are valid, as checked by some theorem prover, the program is verified, i.e., it is possible to make a mathematically precise statement about the behavior of the program. In this section we describe how to generate verification conditions for the ADA-like language presented earlier using the rules we have developed.

The choice of the rules of a language are to some degree arbitrary, even though they may be sound and complete. For instance, the rule for the while loop:

$$\frac{\{I \& B\} S \{I\}}{\{I\} \text{ while } B \text{ loop } S \text{ end loop; } \{I \& \neg B\}}$$

could just as well be replaced by:

$$\frac{(I \& B) \Rightarrow R, \quad \{R\} S \{I\}, \quad (I \& \neg B) \Rightarrow Q}{\{I\} \text{ while } B \text{ loop } S \text{ end loop; } \{Q\}}$$

The interchangeability of these two formulations can be proved using the rule of consequence. The exit statement also has another sound and complete alternative:

$$\frac{(Q \& B) \Rightarrow Q_i}{(l, Q_i) \vdash \{Q\} \text{ exit } l \text{ when } B; \{Q \& \neg B\}}$$

yet we chose the axiom:

$$\frac{(l, Q_i) \vdash \{(Q_i \& B) \mid (Q \& \neg B)\} \text{ exit } l \text{ when } B; \{Q\}}$$

For purposes of generating verification conditions we will prefer the latter forms. Roughly speaking, we wish to craft the rules so that there are no constraints on the postcondition of the Hoare triples in the conclusion of these rules, so that no constraints on the preconditions
of any Hoare triples in the premises, and so that all other formulas occurring in the rule can be computed as a combination of this postcondition, these preconditions, and the syntax of the program segment. This will insure the success of the method used below in designing a program verification system.

The resulting character of the verification system we are going to describe has the "backward" flow familiar in weakest precondition systems. For our verification condition generator to work, all the rules must "flow" in the same direction or chaos would result in statement composition. We prefer to flow "backward" for two reasons. In terms of verifying a program it is convenient to specify what the program is to do and worry less about the conditions under which the program will accomplish it, than the other way around. Secondly, the assignment statement (which we have been ignoring) works better flowing "backward" than "forward."

We describe the verification condition generator using an attribute grammar. This particularly perspicuous method of presentation was suggested by [Reps and Alpern, 1984]. The advantages of using this approach are threefold. One, it is easy to describe the verification condition generator. Two, generators exist that take attribute grammars as input and produce the desired program. Thus, prototyping is easy. And three, this approach integrates well with compiler technology and syntax directed editors.

An attribute grammar is a context free grammar augmented by semantic attributes and semantic rules. Attributes are divided into two classes: inherited and synthesized. In order to introduce the approach, we will first describe a verification condition generator for a simple language. Then we will show how to modify it for the ADA-like language we have been discussing.

First we look at a context free grammar for the language of while programs. The language is not quite what one would expect, because it has been altered slightly to suit the purpose of verification. Here is the grammar:

\[ P ::= S\{Q\} \]
\[ S ::= \text{assign;} \]
\[ S ::= \text{assert } Q; \]
\[ S ::= S \ S \]
\[ S ::= \text{if } B \text{ then } S \text{ else } S \text{ end if;} \]
\[ S ::= \text{inv } Q \text{ while } B \text{ loop } S \text{ end loop;} \]

The customary language of while programs has been changed in three ways. First, we added a syntactic class for programs which syntactically includes an output assertion. This corresponds to the wish to have a program segment verified with respect to some output condition. The output condition is how the specification gets injected in the verification of the program segment.

Second, we added a new statement in the language which is useful to control the process of verification. This is the assert statement. It allows the verification to "forget" all extra details and concentrate on the asserted assertion. Also we can now express the important property of the verification system. The program assert \( P; S \{Q\} \) is verified.
by the system if, and only if, the Hoare triple \( \{ P \} S \{ Q \} \) is valid. It is easy to add this "statement" to the language complete with axiomatization:

\[
\begin{align*}
P & \Rightarrow Q \\
\{ P \} \text{ assert } P; \{ Q \}
\end{align*}
\]

The semantics is that of the null statement.

Finally, we modified the loop statements to contain the invariant assertion explicitly. This is because it is not always possible to derive the invariant from the program.

From the underlying context-free grammar given above, we build an attribute grammar by adding three attributes and the requisite semantic rules. We use a synthesized attribute \( VCs \) for the set of all the verification conditions. We use a synthesized attribute \( pre \) for the assertion characterizing the precondition of a program segment. We have an inherited attribute \( post \) for the postcondition of a program segment.

Now we consider the first production of the grammar. To this production are added two semantic actions. The first semantic action starts the verification process off by setting the post condition of the program segment \( S \) to the formula \( Q \). The other action gathers together the verification condition generated by the segment \( S \).

\[
P ::= S\{Q\} \\
S.post = Q \\
P.VCs = S.VCs
\]

If all the formulas in the list of conditions \( P.VCs \) are true, then the program \( S \) is verified with respect to the output assertion \( Q \).

The next production in the grammar of while programs is the assignment construct. The assignment statement generates no verification conditions.

\[
S ::= \text{assign;} \\
S.pre = \text{Modify}(S.post) \\
S.VCs = \emptyset
\]

The precondition is computed using a function \( \text{Modify} \) from the post condition. Once again, we leave the details concerning the assignment statement out.

The assert statement asserts that the formula \( Q \) is the "current" precondition, but this requires that \( Q \) imply the precondition of the next statement.

\[
S ::= \text{assert } Q; \\
S.pre = Q \\
S.VCs = (Q \Rightarrow S.post)
\]

The juxtaposition of two statements requires some routine manipulation of the attributes. The precondition of the pair statements is taken to be the assertion computed to be the precondition of the first statement. This precondition must be computed under the assumption that the post condition of the first statement is the precondition of the second
statement. The postcondition of the pair is inherited. Finally, the verification conditions are just those generated by the each of the statements.

\[
S_1 ::= S_2 S_3 \\
S_1 \cdot pre = S_2 \cdot pre \\
S_2 \cdot post = S_3 \cdot pre \\
S_3 \cdot post = S_1 \cdot post \\
S_1 \cdot VCs = S_2 \cdot VCs \cup S_3 \cdot VCs
\]

The last statement is the while loop. All the difficult work is spared by insisting that the invariant be given along with the syntax of the loop.

\[
S_1 ::= \text{inv } I \text{ while } B \text{ loop } S_2 \text{ end loop;} \\
S_1 \cdot pre = I \\
S_2 \cdot post = I \\
S_1 \cdot VCs = S_2 \cdot VCs \cup (I \& B \Rightarrow S_2 \cdot pre) \& (I \& \neg B \Rightarrow S_1 \cdot post)
\]

Notice that verification conditions added by the while loop are obtained from the premises of the second while-loop rule given above.

Having completed the description of the verification generator for the simple while programming language, we discuss the modifications necessary to include the loop-exit and raise-handle constructs. We give first the underlying context-free grammar for the ADA-like language.

\[
P ::= \ S (Q) \\
S ::= \text{assign} \\
S ::= \text{assert } Q; \\
S ::= \ S \ S \\
S ::= \text{exit } l \text{ when } B; \\
S ::= \text{loop } l; \\
S ::= \text{raise } e; \\
S ::= \text{begin } S \text{ when } e \Rightarrow S \text{ end;}
\]

The attribute grammar description of the verification condition generator for the ADA-like language is similar to that of the while language. In fact, the rules for the assignment statement and assert statement are the same. But in order to handle the bookkeeping required by the list of assumptions used in the rules for the loop-exit and raise-handle constructs, we add a new inherited attribute \textit{asmp}. The values of this attribute are lists of pairs consisting of a control designator and an assertion. These lists will be used to keep the postcondition of all surrounding blocks and to keep the precondition of all surround exception handlers.

The production for the syntactic category for programs is as before, except that we add an initializing rule which sets the list of assumptions to empty.
P := S\{Q\}
S.post = Q
S.asmp = \emptyset
P.VCs = S.VCs

The juxtaposition of statements is also much as before. The only difference is that each statement inherits the list of assumptions.

\[ S_1 := S_2 S_3 \]
\[ S_1.pre = S_2.pre \]
\[ S_2.post = S_3.pre \]
\[ S_2.asmp = S_1.asmp \]
\[ S_3.post = S_1.post \]
\[ S_3.asmp = S_1.asmp \]
\[ S_1.VCs = S_2.VCs \cup S_3.VCs \]

The exit statement depends critically on the list of assumptions. The postcondition of the loop must be found in the list in order to compute the precondition.

\[ S := \text{exit } I \text{ when } B; \]
\[ S.pre = (\text{LookUp}(l, S.asmp) \& B) \mid (S.post \& \neg B) \]
\[ S.VCs = \emptyset \]
\[ S_1 := I\text{:inv } I \text{ loop } S_2 \text{ end loop } I; \]
\[ S_1.pre = I \]
\[ S_2.post = I \]
\[ S_2.asmp = (l, S_1.post) + S_1.asmp \]
\[ S_1.VCs = S_2.VCs \cup (I \Rightarrow S_2.pre) \]

The loop statement adds its postcondition to the list of assumptions before passing them on to the statements in the body of the loop.

\[ S := \text{raise } e; \]
\[ S.pre = \text{LookUp}(e, S.excep) \]
\[ S.VCs = \emptyset \]
\[ S_1 := \text{begin } S_2 \text{ when } e \Rightarrow S_3 \text{ end; } \]
\[ S_1.pre = S_2.pre \]
\[ S_1.VCs = S_2.VCs \cup S_3.VCs \]
\[ S_2.post = S_1.post \]
\[ S_2.asmp = (e, S_3.pre) + S_1.asmp \]
\[ S_3.post = S_1.post \]
\[ S_3.asmp = S_1.asmp \]

Note that the list of assumptions in the block is augmented by the precondition of the exception \(e\), but the original list is passed to the handler \(S_3\). Note also that there is no need to have an assertion associated with the exception; it can be inferred from the handler.
Correctness of the verification condition generator. We summarize the description of the verification condition generator for our ADA-like language by stating the important correctness properties of the verification condition generator.

Lemma. The attribute grammar fails to evaluate correctly if, and only if, the program segment is not closed. We simply note that the attribute grammar is an absolutely non-circular grammar and the evaluation can only fail if LookUp does not find the control designator in the list of assumptions. This occurs only if the program segment is not closed.

Theorem 1. If all the formulas generated by the attribute grammar are valid for the program assert \( P; S \{Q\} \) where \( S \) is closed, then \( \emptyset \vdash \{P\} S \{Q\} \).

The proof is by induction on the structure of \( S \) and follows the proof of soundness exactly.

The converse of Theorem 1 is also true. Theorem 2. If \( \emptyset \vdash \{P\} S \{Q\} \), then all the formulas generated by the attribute grammar for the program assert \( P; S \{Q\} \) are valid.

The proof is by induction on the structure of \( S \) and follows the proof of completeness exactly.

Implementing the verification condition generator. So far we have only described the verification condition generator, but not indicated how the verification conditions would be generated by the evaluation of the attribute grammar. Of course, we could give the description directly to an attribute grammar evaluator [Reps and Teitelbaum, 1984]. In fact, however, it is possible to build a prototype verification condition generator quit easily. The implementation of the verification condition generator has been done by means of simulating the attribute evaluation using a technique described in [Katayama, 1984].

The simulation technique has three steps.

1. The underlying context free grammar is rendered a type definition.
2. Mutually recursive functions are written for each synthesized attribute.
3. Inherited attributes are passed as arguments to the functions.

Step (1) suggests that the language we use to simulate the attribute evaluation be one that has recursive type definitions. We used the language ML [Cardelli, 1983] for this reason.

Here is the underlying grammar as a recursive type definition in ML. This definition uses other type definitions: Formula, Cond, Exc, and Label. We leave their definitions to the imagination since they are not needed to understand for the verification conditions are generated.

```
type rec Stmt =
    assign
    assert of Formula
    CondStmt of Cond * Stmt * Stmt
    Comp of Stmt * Stmt
    Raise of Exc
    Handle of Stmt * Exc * Stmt
```
Loop of Formula * Label * Stmt | Exit of Label * Cond;

```
type Prog = Prog of Stmt * Formula;
```

This type definition corresponds to the abstract syntax of our ADA-like programming language.

We need to define an auxiliary function to search for a control designator in list of assumptions. This function LookUp is for control designators which have been given the type Control in ML.

```
type Control = Exception of Exc | Label of Label | ne;

val rec LookUp (c: Control, asmp: (Control*Formula) list): Formula =
  case asmp of
  nil          . escape "control designator undefined" |
  (c',phi)::rest . if c=c' then phi else LookUp (c, rest);
```

Corresponding to the two synthesized attributes we have two recursive functions Pre and VCS. They take as arguments the program segment of type Stmt and the two inherited attributes post and asmp.

```
val rec Pre (s: Stmt, post: Formula, asmp): Formula =
  case a of
  assign. Modify (post) |
  assert (phi). phi |
  CondStmt (Cond phi, s1.s2) .
    And (Implies (phi, Pre (s1.post,asmp)),
         Implies (Not phi, Pre (s2.post,asmp))) |
  Loop (inv, lbl, bdy) . inv |
  Exit (lbl, Cond phi) .
  Or (And (LookUp (Label lbl, asmp), phi), And (post, Not phi)) |
  Raise e . LookUp (Exception e, asmp) |
  Handle (s1, e, s2) . Pre (s1, post, (Exception e.pre)::asmp)
    where val pre = Pre (s2, post, asmp) end |
  Comp (s1,s2) . Pre (s1, Pre (s2, post, asmp), asmp);
```

The next function computes the list of verification conditions for a given program segment. It is defined recursively on the structure of program segments.

```
val rec VCS (s: Stmt, post: Formula, asmp): Formula list =
  case s of
  assign . [] |
```
assert (phi) . [Implies (phi, post)] |
CondStmt (Cond phi, s1,s2) .
    VCS (s1, post, asmp) @ VCS (s2, post, asmp) |
Loop (inv, lbl, bdy) .
    Implies (inv, Pre(bdy, inv, asmp)) ::
        VCS (bdy, inv, (Label lbl,post)::asmp) |
Exit (lbl, Cond phi) . [] |
Raise e . [] |
Handle (s1,e,s2) .
    VCS(s1,post,(Exception e,pre)::asmp) @ VCS(s2,post,asmp)
    where val pre = Pre (s2, post, asmp) end |
Comp (s1,s2) .
    VCS (s1, pre, asmp) @ VCS (s2,post,asmp)
    where val pre = Pre (s2, post, asmp) end;

Conclusion. Using a somewhat different semantics we have given a proof rule for an ADA-like loop-exit construct which is sound and complete. This rule can be safely included in a system to formally verify the correctness of ADA programs. The loop rule is no harder to use and understand than the rule for the while statement. The bookkeeping necessary for associating loop labels and the appropriate postconditions is straightforward. The only detail that prevents the rule from being applied mechanically is the discovery of the loop invariant. This, of course, is not surprising. What is surprising is that only one invariant must be found regardless of the number of exit statements.

The proof rules have been used to build a prototype verification condition generator for a simple language with loops and exceptions. The verification condition generator can be described conveniently in the form of an attribute grammar. This is useful in presenting and experimenting with the rules for the verification conditions. From the attribute grammar description and prototype verification condition generator is easy to build using recursive function to simulated the attribute evaluation.

References.


Clarke, Edmund Melson, Jr. “Programming language constructs for which it is impossible to obtain good Hoare axiom systems.” J. of the ACM, volume 26, number 1, January 1979, pages 129-147.


