Towards a Transformational Approach to Program Verification

Myla Archer
Naval Research Laboratory, Code 5546
4555 Overlook Avenue, SW
Washington, DC 20375-5337 U.S.A.
archer@itd.nrl.navy.mil

Amy Lo* and Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 U.S.A.
amy@galaxy.nsc.com, olsson@cs.ucdavis.edu

Publisher: *Please send editorial correspondence regarding this paper to* Olsson

January 4, 1999

Abstract

Although most typically used in other contexts, program transformations can simplify program verification by transforming a program containing complex language features into a semantically equivalent program containing only simpler language features. The proof of the transformed program can then be performed using a set of proof rules for only the simpler features. There are tradeoffs between the transformational approach and the standard approach to program verification with regard to proof understandability and compactness of programs and assertions, establishing soundness of the program verification method, and providing mechanized support for the method. The transformational approach has clear advantages in some of these aspects.

This paper illustrates this transformational approach by considering proof rules for escape statements in iterative constructs, and discusses the tradeoffs with respect to its use. It also suggests how the approach can be applied to other language constructs, including some involving concurrency, and to solving some problems connected with the development of Hoare axiomatizations.

Keywords: program verification, program transformations, axiomatic semantics, proof rules, escape statements.

*Current address: c/o National Semiconductor Inc., 2900 Semiconductor Drive, P.O. Box 58090, M/S D3-677, Santa Clara, CA 95052-8090.

1 Introduction

Program transformations can simplify program verification by transforming a program containing complex language features into a semantically equivalent program containing only simpler language features. The proof of the transformed program can then be performed using a set of proof rules for only the simpler features.

Proofs of transformed programs that use simpler proof rules are simpler in the sense that the correctness of an asserted program is visually clearer. Rules that are referentially transparent, i.e., that do not depend on internal details of the program parts to which they are applied, are simpler than rules without this property. Their verification conditions are more easily checked visually in an asserted program, since they can be computed in a standard way from the assertions at specific, fixed structural points in the program, and do not require scanning for internal details. Moreover, the significance of the verification conditions for simpler proof rules tends to be simpler to understand, and thus the reasons a program is correct tend to be clearer when simpler rules are used. These advantages can be important, for example, in the context of the Cleanroom approach to program development (Mills et al., 1987), in which a team develops a consensus about code correctness using formal methods including specification and verification. Thus, simplifying the set of proof rules needed in a verification in the direction of referential transparency is a desirable goal.

In spite of their potential usefulness in the initial phase of program verification, program transformations have seldom been applied for this purpose. The transformational approach to program verification indeed has several tradeoffs with respect to the standard, direct approach. First, there are tradeoffs in the complexities of the proof and of the proof rules. Although the proof of the transformed program is simpler because it uses simpler proof rules, the transformed program is typically longer and not as clear as the original program, and requires somewhat more complex assertions in its proof. Second, there are one-time tradeoffs in regard to mechanizing verification and establishing soundness. For mechanization, the choice is between automating a larger set of complex proof rules and automating a smaller set of simpler proof rules together with a transformation scheme. The alternatives for establishing soundness are analogous: one must either prove the soundness of a large, complex set of rules, or the soundness of a translation scheme and a smaller, simpler set of rules, with respect to some form of semantics, typically operational, for the programming language. The tradeoff here arguably favors the transformational approach. A proof of transformation correctness is done based on only one formal semantics, that of the programming language.

Proofs of soundness of proof rules require relating the formal semantics of the programming language to that of a logical system, and for complex proof rules, are not easy to obtain. In fact, whether soundness proofs exist for the axiomatic semantics proposed for some languages is not always clear (see, for example, (London et al., 1978) and (Craigen, 1982)). On the other hand, soundness proofs for simple rules such as Hoare's rules for *while* programs are well known, and in fact, have been mechanized (Gordon, 1989).

This paper illustrates the transformational approach, and the first tradeoffs indicated above, by considering proof rules for escape statements in iterative constructs. An "escape statement" in this paper — generalizing the terminology introduced in (London et al., 1978) — is one that alters the normal execution flow of an enclosing construct. Typical of such statements are `continue`, `break`, and `return` in C (Kernighan and Ritchie, 1978); `exit` and `return` in Ada (Ada, 1983), Euclid (Lampson et al., 1977), and Turing (Holt and Cordy, 1988); and `next`, `exit`, and `return` in SR (Andrews et al., 1988; Andrews and Olsson, 1993). The examples in this paper are based on the language SR. SR's sequential constructs are simple in form and are representative of those found in the sequential parts of other languages.

The rest of this paper is organized as follows. Section 2 shows how to transform sequential SR programs that may contain escape statements into semantically equivalent SR programs that contain no escape statements, and establishes the soundness of the transformation scheme. It also compares alternative proof rules, proof outlines, and verification conditions for the (partial) correctness proof of an example program. Section 3 suggests how the transformational approach can be applied to other language constructs, including some involving concurrency (e.g., rendezvous), from SR and similar languages. Section 4 discusses related work. Finally, Section 5 discusses reasons for choosing the transformational approach, and presents some conclusions.

2 Transforming Sequential SR Programs with Escape Statements

2.1 Escape Statements

Escape statements (such as `continue`, `break`, `return`, `exit`, `next` in various languages, as noted in Section 1) arose as a means to provide a structured form of goto statement and to avoid the evils of gotos (Dijkstra, 1968), yet still provide reasonable expressiveness. Escape statements can be used in a number of situations to avoid cumbersome programming involving the use of flag variables.

2.2 The Sequential Subset SR_s of SR

This paper presents transformations for a sequential subset of the SR concurrent programming language (Andrews et al., 1988; Andrews and Olsson, 1993). This sequential subset, called SR_s, includes two escape statements: `exit` and `next`. Execution of an `exit` statement forces early termination of the smallest enclosing iterative statement. Execution of a `next` statement forces return to the loop control of the smallest enclosing iterative statement.

SR_s's constructs are representative of those found in the sequential parts of other languages. Aside from the two escape statements, SR_s has syntax and semantics that strongly resemble those of Dijkstra's language of guarded commands (Dijkstra, 1975; Dijkstra, 1976). SR_s includes variable declaration, increment, decrement, assignment, `do`, `if`, `exit`, and `next`. SR_s allows intermixing of statements and declarations within a block. `exit` and `next` statements are semantically proper only if they occur inside a `do` statement. SR_s programs have only proper occurrences of `exit` and `next`.

Below, Dijkstra's abbreviations are used for the `do` and `if` guarded command constructs:

$$DO\text{: } \mathtt{do}\ B1 \rightarrow S1\ []\ \ldots\ []\ BN \rightarrow SN\ \mathtt{od}$$

$$IF\text{: } \mathtt{if}\ B1 \rightarrow S1\ []\ \ldots\ []\ BN \rightarrow SN\ \mathtt{fi}$$

These constructs, like Dijkstra's, are non-deterministic: if multiple guards evaluate to true, then the one statement list to be executed is chosen non-deterministically from those statement lists corresponding to true guards. The expressions B_i are assumed to have no side effects. This paper uses the term "arm" interchangeably with Dijkstra's term "guarded command" to denote a particular $B_i \rightarrow S_i$ part of an IF or DO.

2.3 The Transformational Approach

The general transformational approach for SR_s programs with escape statements is:

- Apply transformations to an SR_s program, which might contain `exit` and `next` statements, to produce a semantically equivalent SR_s program that contains no `exit` or `next` statement.
- Prove the transformed program using standard proof rules.

The idea behind the specific transformations is simple. Each `exit` or `next` statement is replaced by an assignment to a corresponding flag variable. Code that follows the original `exit` or `next` statement must now be "protected" from being executed during the same iteration of the loop. Thus, the new code uses

IF constructs that test these flag variables. For an `exit`, subsequent iterations of the loop must now be prevented from being executed, so the guards controlling the loop now also test the `exit` flag.

The transformation uses the following symbols:

T	the top-level transformation
U	the general transformation
v	prefix of transformation-time formal parameters, whose values are the names of new flag variables that may appear in the transformed code
z	prefix of transformation-time variables, whose values are the names of new flag variables that may appear in the transformed code.

The general transformation, $U(A1; A2; ...; AN, ve, vn)$, takes three parameters: a code string to transform, the name of the current `exit` flag, and the name of the current `next` flag. The code string is a list of statements: $A1; A2; \ldots; AN$. The transformations assume that `exit` and `next` appear only within a DO (as noted in Section 2.2), and that only AN can be an `exit` or `next` statement. Except for these restrictions, every Ai can be any statement, and can in turn contain `exit` or `next` statements.

Since the top level of a program is not enclosed in a DO, the top-level transformation, T, simply applies the general transformation U with empty strings as the second and third parameters; i.e.:

$$T(program) = U(program, \text{“”}, \text{“”})$$

The general transformation U is defined in Figure 1 by cases based on values of U's parameters. The figure uses two typesetting conventions: code generated by the transformation is typeset in Roman and anything that requires some computation during the transformation is typeset in italics. The latter category includes parameter substitution and simple *let* expressions used to generate unique variable names. Case 1 handles a DO statement by generating two uniquely named variables, declaring and defining them, and modifying the guards on each arm. The variable ze controls `exit` from the DO; because its value is used in the guard on each arm, it is declared and defined outside the DO. In contrast, the variable zn records whether a `next` statement has been executed; because its value is used only within the loop, it is declared and defined within each arm of the DO.[1] This difference reflects the innate semantics of these two statements:

`exit` — execute no more of the current iteration and no more of the loop.

`next` — execute no more of the current iteration but possibly more of the loop.

[1]Note that the name of the `next` flag variable is generated once for each DO, so the same `next` flag variable name is declared and used within each arm. Alternatively, a new `next` flag variable name could be generated, and declared and used within each arm.

These flag variables — ze and zn — are passed as arguments to subsequent U transformations. Case 2 generates code that tests the flag variables of the most immediately enclosing DO to "protect" any code after an `exit` or `next` has executed. It can be generalized to handle any statement with a block that can contain an `exit` or a `next` for a containing DO;[2] in SR_s, IF is the only such statement. Case 3 handles a top-level IF by applying the general transformation on each of its arms. Cases 4 and 5 translate `exit` and `next` statements to assignments to the corresponding flag variables. By assumption, an `exit` or `next` appears only as the last statement in a block, so actually $N = 1$. Case 6 handles any other kind of statement, e.g., an assignment statement. Case 7 is the remaining of the three base cases of the recursive definition (Figure 1).

2.4 Example Transformed Programs

This section presents two typical examples of the use of `exit` and `next` statements, and the results of applying the transformations to them. These examples are meant to illustrate the effects of the transformations and the relative proof complexities, and have been chosen for simplicity and brevity rather than as illustrations of good code.

Figure 2(a) contains a program segment that searches for a value, v, in an $N \times N$ two-dimensional array, a. The code uses a nested DO that contains `exit` statements. Figure 2(b) shows the result of applying the transformations to the code in Figure 2(a). Figure 3(a) presents a program segment that prompts a user for input, and uses an `exit` statement to terminate the loop and a `next` statement to skip executing the main code of the loop if the input value is erroneous. Figure 3(b) shows the result of applying the transformations to the code in Figure 3(a). Both figures use underlining to highlight code introduced or modified by the transformation.

2.5 Standard Proof Rules for SR_s Programs

The proof rules for transformed programs — i.e., for SR_s programs without `exit` and `next` statements — are the standard ones for guarded command programs (Dijkstra, 1975; Dijkstra, 1976; Hoare, 1969). The two key proof rules for this paper are those for the IF and DO constructs[3]:

[2] Note that $ve = \text{""} \Leftrightarrow vn = \text{""}$.

[3] The rule for IF, like the rule in (Andrews, 1991) but unlike that in (Dijkstra, 1975; Dijkstra, 1976), indicates that an IF has no effect if none of its guards is true. The rule for DO, like the rule in (Hoare, 1969) but unlike that in (Dijkstra, 1975; Dijkstra, 1976), ignores termination.

IF Rule :

$$\frac{P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_N) \Rightarrow Q, \{P \wedge B_1\}S_1\{Q\}, \{P \wedge B_2\}S_2\{Q\}, \ldots, \{P \wedge B_N\}S_N\{Q\}}{\{P\}\ \texttt{if}\ B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_N \rightarrow S_N\ \texttt{fi}\ \{Q\}}$$

DO Rule :

$$\frac{\{I \wedge B_1\}S_1\{I\}, \{I \wedge B_2\}S_2\{I\}, \ldots, \{I \wedge B_N\}S_N\{I\}}{\{I\}\ \texttt{do}\ B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_N \rightarrow S_N\ \texttt{od}\ \{I \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_N)\}}$$

The predicate I in the DO Rule represents the loop's invariant.

2.6 Tagged Proof Rules for SR_s Programs

One approach to proof rules for escape statements assigns a *tag* to each statement in a given program (Olsson and Huang, 1991). Statements with which an escape statement can be associated (e.g., DO) have unique tags. Within the body of a DO, non-DO statements are assigned the same tag as the enclosing DO. Therefore, an `exit` statement is assigned the same tag as the smallest enclosing DO. Each statement has its tag enclosed in angle brackets — e.g., $\langle 3 \rangle$ — prepended to the actual program statement. The proof rules for tagged statements are similar to conventional proof rules, but use tags to formalize the association between DO and `exit`. The key proof rules for this paper are those for the IF, `exit`, `next`, and DO constructs:

IF Rule :

$$\frac{P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_N) \Rightarrow Q, \{P \wedge B_1\}\langle c_1 \rangle S_1\{Q\}, \{P \wedge B_2\}\langle c_2 \rangle S_2\{Q\}, \ldots, \{P \wedge B_N\}\langle c_N \rangle S_N\{Q\}}{\{P\}\ \langle c \rangle \texttt{if}\ B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_N \rightarrow S_N\ \texttt{fi}\ \{Q\}}$$

The tag on the top-level statements within an individual arm of the IF statement (e.g., $\langle c_1 \rangle$) will be the same as the tag on the entire IF statement (i.e., $\langle c \rangle$) except for DO statements, which as noted above are assigned unique tags.

Exit and Next Rules:

$$\{P\}\ \langle c \rangle \texttt{exit}\ \{false\}$$

$$\{P\}\ \langle c \rangle \texttt{next}\ \{false\}$$

The postcondition $\{false\}$ in the Exit and Next Rules intuitively means that control never reaches the point immediately after the `exit` or `next`.

DO Rule:

$$\frac{\begin{array}{c}\{I \wedge B_1\}\langle c_1 \rangle S_1\{I\}, \{I \wedge B_2\}\langle c_2 \rangle S_2\{I\}, \ldots, \{I \wedge B_N\}\langle c_N \rangle S_N\{I\}, \\ (\forall R : \{R\}\langle d \rangle \texttt{exit}\ \{false\} : R \Rightarrow Q), \\ (\forall R : \{R\}\langle d \rangle \texttt{next}\ \{false\} : R \Rightarrow I)\end{array}}{\{I\}\ \langle d \rangle \texttt{do}\ B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_N \rightarrow S_N\ \texttt{od}\ \{(I \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_N)) \vee Q\}}$$

As in Section 2.5, the predicate I represents the loop's invariant. `exit` and `next` statements that associate with a particular DO have the same tag as the DO. The precondition R of any such `exit` statement must imply the predicate Q, part of the postcondition of the enclosing DO. The precondition R of any such `next` statement must imply the predicate I, the invariant of the enclosing DO.

2.7 Example Proof Outlines

Obtaining a proof outline, in the form of an interleaving of assertions with the statements, for a given sequential program based on a given set of proof rules is generally done, starting from the program's desired postcondition, by using a combination of weakest precondition computations analogous to those in (Dijkstra, 1976) for non-loop statements and—the heart of the proof—determination of appropriate loop invariants. Although there are increasingly sophisticated techniques that can automatically or semi-automatically generate adequate loop invariants for computations over sufficiently simple (e.g., numeric) data domains (Bjørner et al., 1995; Bensalem et al., 1996), in general, finding the appropriate loop invariants requires some human ingenuity. A typical approach for the respective DO rules in sections 2.5 and 2.6 is to first determine an I (respectively an I and Q), that is (are) as weak as possible, for which the postcondition of the DO rule implies the desired postcondition of the loop, and then iteratively strengthening I (respectively I and Q) and the loop's internal assertions using backwards reasoning through the arms of the DO statement, until a fixed point is found. When loops are nested, this requires repeating this process for the inner loop each time its postcondition is strengthened for the sake of the outer loop.

Figures 4, 5, and 6 present proof outlines of the two versions of the two-dimensional array search program from Section 2.4 that have been obtained in roughly this manner. Some care has been taken in the presentation to make the relationship between the assertions in the two proofs clear. Assertions common to the proof outlines appear in Figure 4. A proof outline — using the standard proof rules from Section 2.5 — of the transformed program from Figure 2(b) appears in Figure 5. A proof outline of the original program from Figure 2(a) — using the "tag method" from Section 2.6 — appears in Figure 6, which is taken from (Olsson and Huang, 1991) with minor changes in presentation. For later ease of reference, assertions in Figures 5 and 6 are labeled with numbers of the form $F.A$, where F represents the figure number and A represents the assertion number. Corresponding assertions in the two figures are given corresponding numbers, with the numbers in Figure 5 derived from those in Figure 6; extra assertions are given intermediate numbers (e.g., 5.12.1 and 5.12.2).

In Figure 5, the term $ze1$ reflects whether an `exit` statement would have been executed in the original program. Its negation appears in a disjunct of the invariant for the outer DO. (Similar comments apply to the term $ze2$ for the inner DO.)

In Figure 6, the tags on the two `exit` statements associate the `exit` statements with the correct DOs.

2.8 Comparisons: Proof Rules, Programs, Proofs

The set of proof rules for a language is simpler if the rules do not need to handle `exit` and `next` statements. This difference can be seen by comparing the two sets of proof rules described earlier. Section 2.5 presented proof rules for DO statements that contain no `exit` or `next` statements, as in the transformed programs. Section 2.6 presented proof rules for DO statements that do contain `exit` or `next` statements; it uses the "tag method". This latter set of proof rules is clearly more complex. Such complexity is also present in the proof rules developed for Euclid (London et al., 1978) and for Turing (Holt et al., 1987), which handle escape statements directly in their original form. Note that (Hoare and Wirth, 1973) omits giving any proof rule(s) to handle the `goto` escape statement of Pascal. The axiom in (London et al., 1978) for the `return` escape statement of Euclid is embedded in the rule for procedure calls, and can require some initial transformation of the program to make its use valid. The complexities of their proof rules for Euclid led the authors of (London et al., 1978) to express a lack of confidence in their correctness. The proof rules for Turing (Holt et al., 1987) handle the association between an escape statement and its associated block by program transformation and a non-standard extension to Hoare-like triples. A Turing program is transformed into an equivalent program that is built from only *basis* statements, for which semantics are defined (Holt et al., 1987). The effect of an exit statement is handled by including in the basis a BLOCK statement (which is not part of the regular language). An exit statement is defined as a jump to the end of the enclosing BLOCK. The proof rule of each basis statement is defined in terms of triples of predicates — *continuing*, *exiting*, and *returning* — reflecting the control flows that can result from the execution of a statement.

Transformed programs obtained from the transformation scheme in Figure 1 will be generally longer and less easy for a human reader to understand, as exemplified by the programs in Figures 2 and 3. However, the transformed code still resembles the original code: it adds only simple assignment statements and conjuncts in guards, and encloses some statements in simple IF statements.

Because the transformation scheme adds a few assignment statements and guards, the proof of the transformed program will also generally be longer than the proof of the original program using the "tag method". Because the proof of the transformed program must allow for the fact that execution continues after what was an escape statement in the original program, the assertions in the proof must carry along some extra information, and will also tend to be longer. However, the actual proof steps will be simpler.

These differences are exemplified by the proofs in Figures 5 and 6. As noted in Section 2.7, for convenience of comparison, the assertions in the proof of the original program (Figure 6) are numbered in order, and those in the proof of the transformed program (Figure 5) are numbered to correspond to those in the nearest analogous location in the original program. There is a clear relationship between the two proofs that exhibits the pattern generally expected when comparing the intermediate assertions in the proof of a transformed program with those in the proof of its original. Each loop invariant in the transformed program is obtained from the corresponding loop invariant in the original program by first adding the negation of the `exit` flag for the loop and the negations of both `next` and `exit` flags of any enclosing loops as additional conjuncts, and then adding on as disjuncts each `exit` assertion of the loop (the assertion prior to each `exit` statement in the corresponding loop of the original program) and of any enclosing loops. The addition of the `exit` assertions of the current loop assures that these assertions will be disjuncts in the assertion at the end of the loop. With the tag method, the invariant need not include the `exit` assertions; in this case, the DO proof rule directly makes them disjuncts of the DO's postcondition. Provided that the proof of the original program is correct, any `next` assertion of a loop implies the corresponding loop invariant in the proof; therefore, these assertions do not need to be accounted for in transforming the loop invariant. The remaining intermediate assertions at corresponding points in the two proofs are related analogously, with appropriate added conjuncts of flags or flag negations and disjuncts of `exit` assertions.

The validity of the proofs in Figures 5 and 6 must be determined by checking their verification conditions. It is here that the proof of the transformed program (i.e., no tags) has the advantage. For example, consider the verification conditions for the outer DO. For the transformed program, one applies the DO rule from Section 2.5 to obtain only the verification condition:

$$\{I \wedge B\}S\{I\}$$

For the original program (i.e., with tags), one applies the DO rule from Section 2.6 to obtain the verification condition:

$$\{I \wedge B\}\langle c_1\rangle S\{I\} \ \wedge \ (\forall R : \{R\}\langle 2\rangle\texttt{exit}\ \{false\} : R \Rightarrow Q(i,j)) \ \wedge \ (\forall R : \{R\}\langle 2\rangle\texttt{next}\ \{false\} : R \Rightarrow I)$$

In both verification conditions, B refers to the (single, for both programs) guard of the loop body, S to the loop body, and I to the loop invariant. In the transformed program, B is $\neg ze1 \wedge (i \leq N)$, and I is $(I1(i) \wedge \neg ze1) \vee Q(i,j)$. The assertion prior to S is 5.4: $(I1(i) \wedge i \leq N \wedge \neg ze1) \vee Q(i,j)$, which clearly is implied by $I \wedge B$. The assertion immediately after S is 5.19, I itself. Thus, the verification condition holds for the transformed program (provided, of course, that the proof of S is correct). For the first conjunct in the verification condition for the original program, B, I, and the assertion prior to S (6.4) are the same as their analogues for the tranformed program, minus all references to $\neg ze1$ and $Q(i,j)$. It is therefore established with similar ease. However, one must now scan the original program for instances of ⟨2⟩`exit` $\{false\}$ and ⟨2⟩`next` $\{false\}$. For this short program, it is not difficult to see that there is only one such instance, an occurrence of ⟨2⟩`exit` $\{false\}$, and to verify that its exit assertion implies $Q(i,j)$. For longer, more complex programs, this scanning task becomes more difficult and error prone when done visually.

In summary, proofs using the transformational method do not appear as "clean" because they need to carry along both the state information about `exit` or `next` flags and the `exit` assertions from the original program in nearly all their assertions; in essence, the effect of an `exit` or `next` has been moved from the original program to the proof of the transformed program. On the other hand, proofs using the "tag method" require assigning tags to statements to maintain knowledge of the nesting level of escapes; in addition, the hypotheses of the inference rules are more complex, and require delving into "hidden" parts of the statements to which they are being applied. Thus, understanding these rules and their application, and checking their associated verification conditions, is more complicated.

2.9 Verification Conditions

As noted in Section 2.8, although the asserted untransformed program in Figure 6 appears simpler than the corresponding asserted transformed program in Figure 5, the set of verification conditions of the untransformed program is more complex. Generating this set of verification conditions mechanically is likewise more complex: every body of a guarded command in a DO must be probed for occurrences of `exit` and `next` that are not inside a nested DO. As is pointed out below in Section 3, the transformational approach can be extended to languages that permit multilevel breaks from DOs. In generating the verification condition for a DO in such a language, its nested DOs must also be probed for corresponding breaks, and additional bookkeeping is required to allow one to match each break to its proper DO while avoiding multiple passes over parts of the code. As this illustrates, coding a verification condition generator becomes considerably more complicated for proof rules that lack referential transparency.

A (correctly implemented!) verification condition generator indeed does support visually checking the verification conditions of an asserted program. However, the transformational approach can permit visual checking, without mechanical support, of verification conditions whose significance is simpler to understand. Thus, it can make the reasons for a program's correctness easier to see.

2.10 Soundness of the Transformation Scheme

The soundness of the transformation scheme in Figure 1 can be proved based on an operational semantics for SR_s in which the notion of "control location" is appropriately defined, and the meaning of a program statement is a mapping from the set of program states to the set of program-state control-location pairs.

In an SR_s program, there are *control locations* just before and just after any statement. When a statement S is part of a sequence and is not the last statement of the sequence, the control location just after S is the same as the control location just before the statement that follows S. However, when S is the last statement in a sequence—that is, either the last statement in the program or the last statement in a guarded command—the control location just after S does not mark the beginning of any statement.

Some control locations are associated with implicit "`goto`"s. In particular:

- The control location at the beginning of any IF or DO has an implicit nondeterministic `goto` to an arm associated with a true guard, if one exists, and to the control location just after the IF or DO, if no guard is true.
- The control location at the end of each arm of a DO has an implicit `goto` to the control location at the beginning of the DO.
- The control location at the end of each arm of an IF has an implicit `goto` to the control location just after the IF.

Note that control locations with associated implicit `goto`s may or may not mark the beginning of a statement; in some cases, a control location only marks an implicit `goto`.

Given this notion of control location, every statement S has three special associated control locations: $nat\text{-}loc(S)$, $exit\text{-}loc(S)$, and $next\text{-}loc(S)$. The $nat\text{-}loc(S)$ is the ("natural") control location just after S; $exit\text{-}loc(S)$ is the control location just after its nearest enclosing DO; $next\text{-}loc(S)$ is the control location at the beginning of its nearest enclosing DO. When S is not inside a DO, $exit\text{-}loc(S)$ and $next\text{-}loc(S)$ are undefined. When S is inside a DO, its three associated control locations are all distinct. Every execution of the statement S starting in a state s results in a new state s' and a new control location that is one of $nat\text{-}loc(S)$, $exit\text{-}loc(S)$, or $next\text{-}loc(S)$. Because the DOs and IFs of SR_s may be nondeterministic, the execution of S may be nondeterministic. Thus, the operational meaning of S, a summary of the results of executions of S, maps a state s to the set of pairs (s', l) such that there is an execution of S starting in state s and ending in state s' at control location l. Note that in addition to having associated control locations $exit\text{-}loc(S)$ and $next\text{-}loc(S)$ derived from its nearest enclosing DO, every statement S inside a DO has associated exit and next flags, $exit\text{-}flag(S)$ and $next\text{-}flag(S)$, defined to be those of the nearest enclosing DO. If S is not inside a DO, the associated flags are undefined.

The possible executions of a statement S are derived by structural induction. If S is `exit` (respectively, `next`), then for any state s there is just one execution of S from s, resulting in the pair (s, l), where l is $exit\text{-}loc(S)$ (respectively, $next\text{-}loc(S)$). When S is any other primitive statement (i.e., an assignment, variable declaration, increment, or decrement), there is exactly one possible execution from any state s to a pair (s', l), where l is $nat\text{-}loc(S)$ and s' is the obvious modification of s. If S is a sequence $S_1; S_2$, then all executions of S are derived as follows:

- if E_1 is an execution of S_1 from s to (s', l), where l is not $nat\text{-}loc(S_1)$ (intuitively, if an `exit` or `next` is executed within S_1 in the execution E_1), then E_1 is also an execution of S from s to (s', l);
- if E_1 is an execution of S_1 from s to $(s', nat\text{-}loc(S_1))$, then, since $nat\text{-}loc(S_1)$ is the control location at the beginning of S_2, every composition of E_1 with some execution E_2 of S_2 is an execution of S.

The possible executions of a DO or an IF from a state s are derived by applying the above definitions and connecting partial executions through implicit `goto`s.

In this context, the equivalence of the transformed program to the original can also be established by structural induction. First, it is not difficult to establish by induction on the nesting depth of a program statement S that in the recursive call to the transformation U on S, the flag parameters will be the `next` and `exit` flags corresponding to the transformation $\hat{S}$ of S; i.e., $\hat{S} = U(S, exit\text{-}flag(\hat{S}), next\text{-}flag(\hat{S}))$. Then, one can use the following inductive assertion on an arbitrary statement S in the untransformed program with transformation $\hat{S}$:

> For every execution E of S from a state s to a state-location pair (s', l), and for every state $\hat{s}$ of the transformed program that agrees with s on all non-flag variables, there is an execution $\hat{E}$ of $\hat{S}$ from $\hat{s}$ resulting in a state-location pair $(\hat{s}', nat\text{-}loc(\hat{S}))$, such that:
>
> (a) $\hat{s}'$ agrees with s' on all non-flag variables;
>
> (b) $l = nat\text{-}loc(S)$ iff $exit\text{-}flag(\hat{S})$ and $next\text{-}flag(\hat{S})$ are both false in $\hat{s}'$.

(c) $l = exit\text{-}loc(S)$ iff $exit\text{-}flag(\hat{S})$ is true and $next\text{-}flag(\hat{S})$ is false in $\hat{s}'$.

(d) $l = next\text{-}loc(S)$ iff $next\text{-}flag(\hat{S})$ is true and $exit\text{-}flag(\hat{S})$ is false in $\hat{s}'$.

Conversely, for every execution $\hat{E}$ of $\hat{S}$, there is an execution E of S such that $\hat{E}$ corresponds to E as described above.

Note that because of the restrictions on the use of `exit` and `next` in SR_s (Section 2.3), it will never happen that both $exit\text{-}flag(\hat{S})$ and $next\text{-}flag(\hat{S})$ are true.

2.11 A Note on Efficiency

As a matter of curiosity, a comparison was made between the running times of original and transformed versions for a small collection of programs, including those in Section 2.4. The running time for the transformed program was generally longer, reflecting the testing of flags that appear in the guards of the *DO*s and in the guards of the "protecting" *IF*s. The extra time ranged between 0% and 200%, although generally between 25% and 150%. Accordingly, these transformations can be termed "de-optimizing"!

3 Extensions

3.1 Extending the Transformations

The transformations presented in Section 2.3 can be extended to handle other sequential SR language features.

- The *DO* statement's last guard can be an `else`; the associated statement list is executed if none of the other guards is true. The transformation can replace the `else` by $\neg ze \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_N)$. (Recall from Section 2.2 that the expressions B_i are assumed to have no side effects.)

- A `return` statement within a procedure can be handled similarly to `exit` and `next` within a *DO*. A single flag can be associated with each procedure. Each `return` statement would be replaced by setting of this flag. The code following any statement that might contain a `return` — i.e., any *IF* or *DO* in the procedure body — would be "protected" with an *IF* statement. The flag would be added to the guards in any enclosing *DO* statement.

- The iterative forall statement in SR:

$$FA\text{: } \texttt{fa } I \texttt{ := } E1 \texttt{ to } E2 \rightarrow S \texttt{ af}$$

can easily be transformed to an equivalent DO (Andrews and Olsson, 1993), to which the transformations presented in Section 2.3 can be applied. Note how this use of transformation is another example of how transformations can be used to simplify the proof rules: a separate FA proof rule is not needed.

On the other hand, it is not entirely clear how to write a direct transformation — i.e., one in a style similar to the other transformations — for a FA. The difficulty is needing to cause control to break out of the loop early for an `exit`: a boolean flag variable cannot be used as part of the termination test of the loop. Three possible approaches to effect termination are:

- Execute the loop for all iterations but do nothing on those iterations after the `exit` flag has been set. The body of the loop can be "protected" by testing the `exit` flag. (Of course, this transformation would be particularly "de-optimizing".)
- Assign to the loop's index variable. The code following the `exit` statement would be "protected" as in the current transformations. For example, if the index variable is to range from 1 to 10, then setting it to 11 will terminate the loop.
- Modify the loop's final expression. For example, if the index variable i is to range from 1 to 10, then the loop can be transformed to $term$`:= 10; fa` i `:= 1 to` $term \rightarrow S$ `af`, and $term$ can be assigned -1 within the body. The code following the `exit` statement would be "protected" as in the current transformations.

Whether these transformations are semantics-preserving (as well as legal) for analogous iterative loops in languages other than SR depends on the semantics of loops in those particular languages: whether index variables are local to loops, whether they can be modified, and whether final expressions are evaluated on each iteration of the loop.

The transformations generate code in which statements and declarations are intermixed: the declaration of a DO's `exit` flag is generated immediately before the DO. Conveniently, SR allows such intermixing. For a language that does not, similar transformations might still generate intermixed statements and declarations, but then an extra transformation phase could move all such declarations to the beginning in blocks or procedures, as required.

The transformations defined in Figure 1 always add extra code for an IF within a DO. They could instead be designed to generate the extra code only if the IF contains an `exit` or `next` statement for the

DO. However, determining the presence of an `exit` or `next` statement would require looking ahead for an `exit` or `next` through the entire body of a DO and matching the `exit` or `next` to the correct DO, thus complicating the transformations and their application. The transformations in Figure 1 require minimal lookahead. For programs having a low ratio of `exit` and `next` statements to DOs, especially if they are long programs, these transformations may add enough unnecessary extra code to justify performing the extra lookahead.

The transformations can be extended to handle a multiple-level `exit` or `next` (not present in SR). For example, `exit 2` means to continue execution after the DO enclosing the DO in which this `exit` appears; this would have been useful in the original program in Figure 2. (The number of levels is assumed to be specified as an integer constant.) Rather than pass only the name of the current `exit` flag and the name of the current `next` flag as parameters in the transformations, the transformations would now need to pass a list of all active `exit` flags and a list of all active `next` flags. An `exit` N statement would be transformed to a series of N assignments to set the N most recently declared `exit` flags in the list. A `next` N statement would be transformed to a series of N assignments to set the $N-1$ most recently declared `exit` flags in the list and one assignment to set the N-th most recently declared `next` flag in the list. IF statements protecting against `exit` or `next` would still only need to refer to the most recent `exit` and `next` flags. As a specific example, these extended transformations can be applied to Ada's `exit` statement, which can specify the label of an enclosing loop from which to exit. First, though, each such `exit` statement must be transformed into an `exit` statement with a corresponding number of levels, which is easily accomplished.

3.2 Extending the Approach to Other Constructs

To provide axiomatic semantics for all of SR, the proof rules for SR_s would need to be extended with additional proof rules to handle SR's concurrency mechanisms, including processes, asynchronous message passing, synchronous message passing (rendezvous and remote procedure call), early reply, deferred reply, and concurrent invocation (Andrews et al., 1988; Andrews and Olsson, 1993).

The transformational approach might help to reduce the complexity of the proof rules dealing with concurrency. Of the above mechanisms, asynchronous message passing is the most fundamental, and the other mechanisms can be expressed in terms of it. Then, a proof rule for asynchronous message passing (AMP) (Andrews, 1991) would suffice for proving the transformed program, obviating the need for proof rules for the other mechanisms themselves. Understanding a proof of such a program would require the reader to know only a single proof rule (that for AMP) rather than separate rules for AMP, rendezvous, etc.

For example, a rendezvous can be transformed to a send/receive pair in the invoking process (client) and a receive/send pair in the servicing process (server). These messages make explicit the nature of a rendezvous: the client sends its request for service to the server; the server performs some computation and sends back the results to the client. The proof of the transformed program would employ the proof rule for asynchronous message passing.

One potential drawback of the transformational approach in this rendezvous example is that the transformation process loses information, and this loss can complicate the proof of the transformed program. For example, the receipt of the results of the rendezvous is implicit in the original program, and is represented in the proof rule for rendezvous. In contrast, in the transformed program, the fact that the receive that now appears in the client is intended for the results of the rendezvous is lost. The proof of the transformed program would use the *general* proof rule for asynchronous message passing, which is potentially more complex than what is needed.

A related issue is how far to take the transformational approach. Some mechanisms can be transformed to simpler ones, but the complexity of the resulting code might make doing so undesirable; e.g., the transformations and the transformed programs might be difficult to understand and lengthy. For example, deferred reply, early reply, and concurrent invocation in SR all can be transformed to asynchronous message passing, but the result is very complex (Andrews and Olsson, 1993).

4 Related Work

There has been considerable work on providing axiomatic semantics for languages with a variety of features, e.g., (Hoare, 1969; Clint and Hoare, 1972; Hoare and Wirth, 1973; Owicki and Gries, 1976; London et al., 1978; Arbib and Alagic, 1979; Meldal, 1986; Holt et al., 1987; Huang and Olsson, 1990; Olsson and Huang, 1991); see also the surveys (Apt, 1981; Apt, 1984). Some of these have dealt with languages having escape statements (London et al., 1978; Holt et al., 1987; Huang and Olsson, 1990; Olsson and Huang, 1991) or jumps (Arbib and Alagic, 1979; Clint and Hoare, 1972). With minor exceptions (e.g., in (London et al., 1978) and (Holt et al., 1987)), all of these attempt to treat the language structures directly, with no transformations, generally resulting in complex proof rules with context-dependent parts.

Some of the problems that can arise when one tries to construct an axiomatic semantics for a complex programming language are pointed out in (Clarke, 1977) and (Craigen, 1982). For example, it is demonstrated in (Clarke, 1977) that an axiomatic semantics that is both sound and complete in the sense of (Cook, 1978) is impossible to obtain for a block-structured language with static scope, global variables, recursion, internal procedures, and procedures as parameters of procedure calls. However, Clarke points out that weakening any of these features in certain ways—e.g., using dynamic instead of static scope, or disallowing global variables—permits a "good" axiomatization to be obtained. Certain soundness problems with the Euclid proof rules in (London et al., 1978) are pointed out in (Craigen, 1982). Repairing these problems would require at least adding extra tests (and hence extra complexity) to the proof rules.

Some practical verification tools, such as those described in (Boyer and Moore, 1981) (for FORTRAN) and (Barnes, 1997) (for Ada), avoid using complex proof rules, and, rather, adopt the approach of (Floyd, 1967), in which the flow graph of a program is divided into linear segments, each with a verification condition that can be computed from assertions associated with its end points. Both (Boyer and Moore, 1981) and (Barnes, 1997) enforce certain language restrictions to ensure that their methods are sound.

Most work on program transformations focusses on transformations for efficiency (e.g., (Janicki et al., 1986)), implementation from specification (e.g., (Balzer, 1981)), derivation of correct programs via refinement (e.g., (Back, 1981; Back, 1988; Back and Sere, 1989)), or programming convenience (e.g., (Dershowitz and Manna, 1977)), or in the context of partial evaluation (see, e.g., (Consel and Danvy, 1993)). In contrast, the transformations in this paper can be viewed as a kind of "inverse refinement" of programs for verification purposes.

There has been a certain amount of previous work on transformations as an aid in verification. Meldal (Meldal, 1986) takes the transformational approach, as this paper does, but applies it to changing concurrent CSP programs (Hoare, 1978) into sequential, nondeterministic programs. Owicki and Gries (Owicki and Gries, 1976) use the transformation idea to some extent, in particular, showing how concurrent programs with semaphores can be first modified into their language and then verified, and showing how some programs need to be transformed by the introduction of new variables to be proved by their methods (as is the case with this paper's method, also). (London et al., 1978) requires the use of transformations prior to application of certain rules in the axiomatic semantics for Euclid, as does (Holt et al., 1987) for Turing. No use of transformations to eliminate embedded escape statements seems to appear in the literature.

Two other transformation schemes for programs containing escape statements were developed by this paper's authors. One scheme (Hung, 1991) reduces SR_s programs to programs very similar to Hoare's *while* programs, except for having a non-deterministic if statement with arbitrarily many branches (i.e., IF). Correspondingly, the proof rules are as simple as Hoare's (Hoare, 1969). This scheme has also been implemented in *yacc*. (The scheme in Section 2.3 could also be implemented using *yacc*.) The other transformation scheme (Archer, 1993) is similar to the scheme in (Hung, 1991). Its transformations are simpler, but they lead to a greater expansion in program length.

5 Discussion and Conclusions

In program verification, both the axiomatic approach of Hoare and the Floyd approach taken in (Boyer and Moore, 1981) and (Barnes, 1997) can pose difficulties to a human trying to visually relate the assertions associated with a program text to the correctness of the program. With the Floyd approach, visually checking the proof of an asserted program requires either construction of a visual representation of the flow graph or very close attention to the details of the program's flow of control to ensure that all of the linear segments have been examined. With the usual versions of the Hoare approach, the complexity—in particular, lack of referential transparency—of the proof rules require scanning of internal and contextual aspects of a programming construct to determine its verification conditions.

This paper has introduced a transformational verification method aimed at making the connection between a program's assertions and its correctness simpler to see when the Hoare approach is taken. The method has been illustrated for programs in a sequential subset of the language SR containing escape statements.

Tradeoffs do exist with respect to the nature of program correctness proofs, establishing soundness, and implementation of automated verification support. With respect to these tradeoffs, there are three reasons why the transformational approach might be preferred. First, although it was noted in Section 2.8 that transformed programs are generally longer and slightly less easy for a human reader to understand, and their proofs are longer and slightly more complicated, the transformational approach permits use of simpler and clearer proof rules, and the relative simplicity of the proof rules permit a human more easily to eyeball the correctness of an asserted program. Second, the soundness of a simpler set of proof rules and a straightforward transformation scheme is likely to be easier to establish than the soundness of complex rules whose interpretations are often not transparent. Third, the relative simplicity of the proof rules in the transformational approach simplifies implementing a verification condition generator. From the authors' experience, it is straightforward to automate a low-cost transformation scheme and thus eliminate the human overhead of applying the transformations. Hence, the entire transformational method is likely to be more easily automated than a direct method using proof rules that involve tags, looking inside statements, or other complexities.

A fourth reason why the transformational approach is worth considering is that it potentially can solve the problem of a language that does not have a "good" Hoare-like axiomatization. For example, in a language having the combination of five features indicated as problematic in (Clarke, 1977) (see Section 4), transformations could be used to eliminate global variables in favor of "VAR" (i.e., reference) procedure parameters (which Clarke in effect allows). The transformations used in the axiomatic semantics of Euclid in (London et al., 1978) and of Turing in (Holt et al., 1987) are clearly intended to permit at least a more straightforward Hoare-like axiomatization of these languages.

The next steps in this investigation will be to extend the transformation scheme and proof comparisons to programs that include other features of SR, particularly those for concurrency described in Section 3.2, and study other possible uses for transformation prior to verification in solving the problem of obtaining "good" axiomatizations.

Acknowledgments

The authors wish to thank the anonymous reviewers for insightful comments that have led to many improvements in this paper.

References

Ada (1983). *Reference Manual for the Ada Programming Language*. American National Standards Institute. ANSI/MIL-STD-1815A.

Andrews, G., Olsson, R., Coffin, M., Elshoff, I., Nilsen, K., Purdin, T., and Townsend, G. (1988). An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86.

Andrews, G. R. (1991). *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA.

Andrews, G. R. and Olsson, R. A. (1993). *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA.

Apt, K. R. (1981). Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483.

Apt, K. R. (1984). Ten years of Hoare's logic: A survey — Part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109.

Arbib, M. and Alagic, S. (1979). Proof rules for gotos. *Acta Informatica*, 11(2):139–148.

Archer, M. (1993). Simpler transformations for escape statements. Unpublished notes.

Back, R. J. R. (1981). On correct refinment of programs. *Journal of Computer and System Science*, 23(1):49–68.

Back, R. J. R. (1988). A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624.

Back, R. J. R. and Sere, K. (1989). Stepwise refinement of action systems. In *Mathematics of Program Construction. Proceedings of the 375th Anniversary of the Groningen University International Conference*, pages 115–138.

Balzer, R. (1981). Transformational implementation: an example. *IEEE Transactions on Software Engineering*, SE-7(1):3–14.

Barnes, J. (1997). *High Integrity Ada: The SPARK Approach*. Addison-Wesley.

Bensalem, S., Lakhnech, Y., and Saidi, H. (1996). Powerful techniques for the automatic generation of invariants. In Alur, R. and Henzinger, T. A., editors, *Computer Aided Verification (Proceedings of CAV '96)*, pages 323–335.

Bjørner, N., Browne, A., and Manna, Z. (1995). Automatic generation of invariants and intermediate asssertions. In Montanari, U., editor, *1st Int. Conf. on Principles and Practice of Constraint Programming*.

Boyer, R. S. and Moore, J. S. (1981). *The Correctness Problem in Computer Science*. Academic Press.

Clarke, Jr., E. M. (1977). Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 10–20, New York. ACM.

Clint, M. and Hoare, C. A. R. (1972). Program proving: jumps and functions. *Acta Informatica*, 1:214–224.

Consel, C. and Danvy, O. (1993). Tutorial notes on partial evaluation. In *Proceedings of the Twentieth ACM Symposium on the Principles of Programming Languages (POPL'93)*, pages 493–501, Charleston, SC.

Cook, S. A. (1978). Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90.

Craigen, D. (1982). The Euclid proof rules: A critique. Technical Report TR-5091-82-1, I.P. Sharp Associates, Ottowa, Ontario.

Dershowitz, N. and Manna, Z. (1977). The evolution of programs: automatic program modification. *IEEE Transactions on Software Engineering*, SE-3(6):377–385.

Dijkstra, E. W. (1968). Goto statement considered harmful. *Communications of the ACM*, 11(3):147–148.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J.

Floyd, R. W. (1967). Assigning meanings to programs. *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, 19:19–32.

Gordon, M. J. C. (1989). Mechanizing programming logics in higher order logic. In Birtwistle, G. and Subrahmanyam, P. A., editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583.

Hoare, C. A. R. (1978). Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–667.

Hoare, C. A. R. and Wirth, N. (1973). An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2(4):335–355.

Holt, R. C. and Cordy, J. R. (1988). The Turing programming language. *Communications of the ACM*, 31(12):1410–1423.

Holt, R. C., Matthews, P. A., Rosselet, J. A., and Cordy, J. R. (1987). *The Turing Programming Language: Design and Definition.* Prentice-Hall, Englewood Cliffs, N.J.

Huang, D. T. and Olsson, R. A. (1990). Axiomatic semantics for SR. unpublished notes.

Hung, A. (1991). Towards a simpler axiomatic semantics for SR. Master's thesis, Dept. of Computer Science, University of California, Davis.

Janicki, R., Lauer, P. E., Koutny, M., and Devillers, R. (1986). Concurrent and maximally concurrent evolution of nonsequential systems. *Theoretical Computer Science*, 43:213–238.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J.

Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. W., and Popek, G. J. (1977). Report on the programming language Euclid. *SIGPLAN Notices*, 12(2):1–79.

London, R. L., Guttag, J. V., Horning, J. J., Lampson, B. W., Mitchell, J. W., and Popek, G. J. (1978). Proof rules for the programming language Euclid. *Acta Informatica*, 10(1):1–26.

Meldal, S. (1986). An axiomatic semantics for nested concurrency. *BIT*, 26:164–174.

Mills, H. D., Dyer, M., and Linger, R. C. (1987). Cleanroom software engineering. *IEEE Software*, pages 19–25.

Olsson, R. A. and Huang, D. T. (1991). Axiomatic semantics for "escape" statements. *Information Processing Letters*, 39:27–33.

Owicki, S. and Gries, D. (1976). Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285.

Case 1: $N > 0$ and $A1$ is a DO.

$U(A1; A2; ...; AN, ve, vn) =$ $\quad let\ ze\ =\ newexitflagname$
$\quad let\ zn\ =\ newnextflagname$
$\quad$ var ze := false
$\quad$ do not ze & $(B1) \rightarrow$ var zn := false; $U(S1, ze, zn)$
$\quad$ [] not ze & $(B2) \rightarrow$ var zn := false; $U(S2, ze, zn)$
$\quad$ ...
$\quad$ od
$\quad U(A2; ...; AN, ve, vn)$

Case 2: $N > 0$, $A1$ is an IF, and ve is not "", i.e., $A1$ is an IF statement within a DO.

$U(A1; A2; ...; AN, ve, vn) =$ $\quad$ if $B1 \rightarrow U(S1, ve, vn)$
$\quad$ [] $B2 \rightarrow U(S2, ve, vn)$
$\quad$ ...
$\quad$ fi
$\quad$ if not ve & not $vn \rightarrow$ # "protection" from `exit/next`.
$\quad\quad U(A2; ...; AN, ve, vn)$
$\quad$ fi

Case 3: $N > 0$, $A1$ is an IF, and ve is "", i.e., $A1$ is an IF not enclosed within any DO.

$U(A1; A2; ...; AN, ve, vn) =$ $\quad$ if $B1 \rightarrow U(S1, ve, vn)$
$\quad$ [] $B2 \rightarrow U(S2, ve, vn)$
$\quad$ ...
$\quad$ fi
$\quad U(A2; ...; AN, ve, vn)$

Case 4: $N > 0$ and $A1$ is an `exit`.

$U(A1, ve, vn) =$ $\quad ve$:= true

Case 5: $N > 0$ and $A1$ is a `next`.

$U(A1, ve, vn) =$ $\quad vn$:= true

Case 6: $N > 0$ and $A1$ is none of above.

$U(A1; A2; ...; AN, ve, vn) =$ $\quad A1$; $U(A2; ...; AN, ve, vn)$

Case 7: $N = 0$.

$U($"", $ve, vn) =$ $\quad$ ""

Figure 1: The general transformation $U(A1; A2; ...AN, ve, vn)$.
(Recall that guarded commands in DO and IF are $B_i \rightarrow S_i$.)

```
var a[N,N]: int
 ...
var found: bool; var i,j: int
 ...
found := false
i := 1
do i <= N ->
  j := 1
  do j <= N ->
    if a[i,j] = v ->
      found := true
      exit
    fi
    j++
  od
  if found ->
    exit
  fi
  i++
od
```

(a) Original program.

```
var a[N,N]: int
 ...
var found: bool; var i,j: int
 ...
found := false
i := 1
var ze1 := false
do not ze1 & ( i <= N ) ->
  var zn1 := false
  j := 1
  var ze2 := false
  do not ze2 & ( j <= N ) ->
    var zn2 := false
    if a[i,j] = v ->
      found := true
      ze2 := true
    fi
    if not ze2 & not zn2 ->
      j++
    fi
  od
  if found ->
    ze1 := true
  fi
  if not ze1 & not zn1 ->
    i++
  fi
od
```

(b) Transformed program.

Figure 2: Two-dimensional array search program and its transformation.

```
var x: int
do true ->
  writes("enter a number > 0 "
  writes("(or = 0 when done): ")
  read(x)
  if x = 0 ->
    exit
  [] x < 0 ->
    write("number must be >= 0")
    next
  fi
  var fact := 1
  var i := 1
  do i <= x ->
    fact *:= i; i++
  od
  write(x, "factorial is", fact)
od
```

(a) Original program.

```
var x: int
var ze1 := false
do not ze1 & ( true ) ->
  var zn1 := false
  writes("enter a number > 0 ")
  writes("(or = 0 when done): ")
  read(x)
  if x = 0 ->
     ze1 := true
  [] x < 0 ->
     write("number must be >= 0")
     zn1 := true
  fi
  if not ze1 & not zn1 ->
    var fact := 1
    var i := 1
    var ze2 := false
    do not ze2 & ( i <= x ) ->
      var zn2 := false; fact *:= i; i++
    od
    write(x, "factorial is", fact)
  fi
od
```

(b) Transformed program.

Figure 3: Error-checking program and its transformation.

Let $I1(i)$ be: $\{1 \leq i \leq N+1 \wedge P(i)\}$
Let $I2(i,j)$ be: $\{\neg found \wedge 1 \leq j \leq N+1 \wedge (\forall m : 1 \leq m \leq j-1 : a[i,m] \neq v)\}$
$I1(i)$ and $I2(i,j)$ are parts of the loop invariants.

Let $P(i)$ be: $\{\neg found \wedge (\forall k, m : 1 \leq k \leq i-1, 1 \leq m \leq N : a[k,m] \neq v)\}$
$P(i)$ represents an unsuccessful search of the first $i-1$ rows.

Let $Q(i,j)$ be: $\{found \wedge 1 \leq i \leq N \wedge 1 \leq j \leq N \wedge a[i,j] = v\}$
$Q(i,j)$ represents a successful search.

Figure 4: Assertions common to the proof outlines in Figures 5 and 6.

(Definitions of $I1(i)$, $I2(i,j)$, $P(i)$, and $Q(i,j)$ appear in Figure 4.)

$(I1(i) \wedge \neg ze1) \vee Q(i,j)$ is the invariant for the outer loop.
$(I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge zn1 \wedge \neg ze2) \vee Q(i,j)$ is the invariant for the inner loop.

5.1	$\{true\}$
	`found := false`
	`i := 1;   var ze1 := false`
5.2	$\{\neg found \wedge i = 1 \wedge \neg ze1\}$
5.3	$\{(I1(i) \wedge \neg ze1) \vee Q(i,j)\}$
	`do not ze1 & ( i <= N ) ->`
5.4	$\{(I1(i) \wedge i \leq N \wedge \neg ze1) \vee Q(i,j)\}$
	`var zn1 := false;   j := 1;   var ze2 := false`
5.5	$\{((I1(i) \wedge i \leq N \wedge \neg ze1) \vee Q(i,j)) \wedge \neg zn1 \wedge \neg ze2 \wedge j = 1\}$
5.6	$\{(I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2) \vee Q(i,j)\}$
	`do not ze2 & ( j <= N ) ->`
	`var zn2 := false`
5.7	$\{(I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2 \wedge \neg zn2 \wedge j \leq N) \vee Q(i,j)\}$
	`if a[i,j] = v ->`
5.8	$\{((I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2 \wedge \neg zn2 \wedge j \leq N) \vee Q(i,j)) \wedge a[i,j] = v\}$
	`found := true;   ze2 := true`
5.10	$\{Q(i,j) \wedge ze2\}$
	`fi`
5.11	$\{((I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2 \wedge \neg zn2 \wedge j \leq N \wedge a[i,j] \neq v) \vee (Q(i,j) \wedge ze2))\}$
5.12	$\{((I1(i) \wedge i \leq N \wedge I2(i,j+1) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2 \wedge \neg zn2) \vee (Q(i,j) \wedge ze2))\}$
	`if not ze2 & not zn2 ->`
5.12.1	$\{I1(i) \wedge i \leq N \wedge I2(i,j+1) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2 \wedge \neg zn2\}$
	`j++`
5.12.2	$\{I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2 \wedge \neg zn2\}$
	`fi`
5.13	$\{(I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2) \vee (Q(i,j) \wedge ze2)\}$
	`od`
5.14	$\{((I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge \neg ze2) \vee Q(i,j)) \wedge (ze2 \vee j > N)\}$
5.14.1	$\{(I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge j > N) \vee Q(i,j)\}$
	`if found ->`
5.15	$\{Q(i,j)\}$
	`ze1 := true`
5.16	$\{Q(i,j) \wedge ze1\}$
	`fi`
5.17	$\{(I1(i) \wedge i \leq N \wedge I2(i,j) \wedge \neg ze1 \wedge \neg zn1 \wedge j > N) \vee (Q(i,j) \wedge ze1)\}$
	`if not ze1 & not zn1 ->`
5.18.1	$\{I1(i+1) \wedge \neg ze1 \wedge \neg zn1\}$
	`i++`
5.18.2	$\{I1(i) \wedge \neg ze1\}$
	`fi`
5.19	$\{(I1(i) \wedge \neg ze1) \vee Q(i,j)\}$
	`od`
5.20	$\{((I1(i) \wedge \neg ze1) \vee Q(i,j)) \wedge (ze1 \vee i > N)\}$
5.21	$\{P(N+1) \vee Q(i,j)\}$

Figure 5: Proof of transformed program.

(Definitions of $I1(i)$, $I2(i,j)$, $P(i)$, and $Q(i,j)$ appear in Figure 4.)

$I1(i)$ is the invariant for the outer loop.
$I1(i) \wedge i \leq N \wedge I2(i,j)$ is the invariant for the inner loop.

6.1	$\{true\}$
	$\langle 1 \rangle$`found := false`
	$\langle 1 \rangle$`i := 1`
6.2	$\{\neg found \wedge i = 1\}$
6.3	$\{I1(i)\}$
	$\langle 2 \rangle$`do i <= N ->`
6.4	$\{I1(i) \wedge i \leq N\}$
	$\langle 2 \rangle$`j := 1`
6.5	$\{I1(i) \wedge i \leq N \wedge j = 1\}$
6.6	$\{I1(i) \wedge i \leq N \wedge I2(i,j)\}$
	$\langle 3 \rangle$`do j <= N ->`
6.7	$\{I1(i) \wedge i \leq N \wedge I2(i,j) \wedge j \leq N\}$
	$\langle 3 \rangle$`if a[i,j] = v ->`
6.8	$\{I1(i) \wedge i \leq N \wedge I2(i,j) \wedge j \leq N \wedge a[i,j] = v\}$
	$\langle 3 \rangle$`found := true`
6.9	$\{Q(i,j)\}$
	$\langle 3 \rangle$`exit`
6.10	$\{false\}$
	`fi`
6.11	$\{I1(i) \wedge i \leq N \wedge I2(i,j) \wedge j \leq N \wedge a[i,j] \neq v\}$
6.12	$\{I1(i) \wedge i \leq N \wedge I2(i,j+1)\}$
	$\langle 3 \rangle$`j++`
6.13	$\{I1(i) \wedge i \leq N \wedge I2(i,j)\}$
	`od`
6.14	$\{(I1(i) \wedge i \leq N \wedge I2(i,j) \wedge j > N) \vee Q(i,j)\}$
	$\langle 2 \rangle$`if found ->`
6.15	$\{Q(i,j)\}$
	$\langle 2 \rangle$`exit`
6.16	$\{false\}$
	`fi`
6.17	$\{I1(i) \wedge i \leq N \wedge I2(i,j) \wedge j > N\}$
6.18	$\{I1(i+1)\}$
	$\langle 2 \rangle$`i++`
6.19	$\{I1(i)\}$
	`od`
6.20	$\{(I1(i) \wedge i > N) \vee Q(i,j)\}$
6.21	$\{P(N+1) \vee Q(i,j)\}$

Figure 6: Proof, using the "tag method", of original program.