Using Strongest Postconditions To Improve Software Quality

Si Pan, R. Geoff Dromey

Australian Software Quality Research Institute,
Griffith University, Nathan, Queensland, AUSTRALIA 4111

Abstract
The cost of developing and maintaining high quality software remains at a premium. In this paper we introduce a practical approach for enhancement of software quality, based on calculation of strongest postconditions of program components. The method provides a powerful means for detecting and removing by transformation various forms of redundancy and inconsistency in programs. The feasibility of this approach depends upon calculations of strongest postconditions for iterative constructs and assignments.

Keyword Codes: D.2.1; D.2.4, D.2.8; F.3.1, F.3.2
Keywords: Software Engineering, Reasoning about Programs, Formal Methods, Reengineering

1. INTRODUCTION

Many previous studies [1,3,5] have focused on calculation of Weakest Preconditions to assist with program derivation and program proof. Recently a number of authors [1,3,15] have studied the calculation of the strongest postcondition of a program statement S satisfying a precondition P. Calculation of postconditions has applications in program verification [15], quality improvement [1,15], deriving specifications from programs [15] and quality measurement [16]. For existing programs, without any accompanying formal specifications, improving program quality cannot be done effectively until the postcondition is calculated using the initial precondition and the program statements. In this paper, a major focus is upon finding the computable Strongest Postcondition Q of S under the precondition P.

Methods [7] to enhance program quality involve either Rule-Based approaches [6] or Computational approaches. The former are useful for detecting some classes of static quality defects, but they tend to lack generality and the necessary transformation machinery. Typical computational-based approaches, such as Symbolic Evaluation and Information/Data-Flow Analysis [2,8,9,11,12], focus on data dependence/independence and enable detection of some computational redundancy and inconsistency, but they are weak in enhancing the logical and structural quality of programs. The relations they use are localized and unable to detect many forms of defects and redundancy. Localized preconditions also prevent the method from achieving the ideal that has been recently suggested by Hoare, He & Sampaio [10].

The model we propose uses the strongest precondition at a given point to calculate the strongest postcondition for the current statement which is also the strongest precondition for the next statement. The process of strongest postcondition calculation is integrated with the transformations necessary to improve the quality of programs in line with the strongest postcondition results. This model, which includes a practical means for calculating strongest postconditions for loops has a number of potential applications.

2. BACKGROUND

In order to model practical programs, we have to introduce I/O statements, denoted by read(x) and write(E), and procedures/functions into Dijkstra’s Guarded Command Language [4] to support our theoretical development.

2.1. Problems With Strongest Postcondition Calculations

Previous studies [1,3] characterize the semantics of a small imperative language in terms of
strongest postconditions for given a precondition, P, as follows:

\[ DS_0: \text{sp}(P \land R, S) = \text{sp}(P, S) \lor \text{sp}(R, S) \]

\[ DS_1: \text{sp}(P, \emptyset) = P \]

\[ DS_2: \text{sp}(P, n := E) = (n = E) \land \exists n: P \]

\[ DS_3: \text{sp}(P, \text{if } C \text{ then } S_1 \text{ else } S_2) = \text{sp}(P \land C, S_1) \lor \text{sp}(P \land \neg C, S_2) \]

\[ DS_4: \text{sp}(P, \text{do } G \text{ while } S \text{ od}) = \neg \exists i: (i \leq d) \land \text{sp}(P, (i F G \land S f i)) \]

\[ DS_5: \text{sp}(P, S_1; S_2) = \text{sp}(\text{sp}(P, S_1); S_2) \]

Although these results give, in principal, a calculation method for strongest postconditions, they are limited for a number reasons. Firstly, the term \( \exists n: P \) in DS2a is not explicitly indicated since it only stipulates that the variable \( n \) is bounded by the precondition \( P \). This makes direct calculation difficult. Secondly, the inverse function \( E^{-1}(n) \) in DS2b may be undefined or difficult to calculate. Thirdly, the depth \( i \) of the calculation in DS4 is not fixed. This again makes direct calculation of strongest postconditions for loops difficult. A result of DS4, improved by Back [1] on loops involving a non-deterministic assignment statement, indicates that the calculation depth \( i \) may be bounded. However it is not fixed.

2.2. Functions

We define the following function (Reassignment Rules) to preserve the integrity of strongest postcondition calculations when a variable is initialized (either by input or assignment) or reassigned:

Given any \( P \), the function takes \( P \) and a variable \( x \) as arguments and returns \( \text{pV}_x \{ P \} \) which is independent of \( x \) has the following properties:

- when \( P \land (x = e) \):
  \[ \text{pV}_x \{ P \} = P[e/x] \]
  \[ R1 \]
- otherwise:
  \[ \text{pV}_x \{ P \} = P[y/x] \]
  \[ R2 \]

where \( P[e/x] \) represents the expression in which all occurrences of \( x \) in \( P \) are replaced by \( e \). More importantly, these reassignment rules provide a renaming method to detect and correct quality defects where the variable \( x \) is used for two purposes \( R1 \). When reassignment of a variable takes place, it must be replaced by its value throughout the precondition. This effectively removes redundancy or renames a two-purpose variable in the program.

Another useful function is the Split Function:

For any (conjunctive) guard expression \( C \) and a (conjunctive) precondition \( P \), the Split Function, which is denoted by \( \text{Sp}(P, C) \), takes \( C \) and \( P \) as its arguments and returns a (conjunctive) redundancy-free guard expression \( R \) such that

\[ \exists R((P \land C) \land R) \land \forall r((P \land C) \land r) \Rightarrow (r \Rightarrow R)) \]

To illustrate this, let us consider a guard \( i \leq a[i] < X \) under the precondition \( X < a[i] \). If we use only the precondition \( X < a[i] \) to evaluate \( i \leq a[i] < X \) then we have \( X < a[i] \land (i \land a[i] < X) \) which is a redundant component of the guard. This example provides the justification for the definition we have introduced. This definition provides not only mathematical rigour and consistency but also the basis for calculating the level of redundancy in any guard expression [15].

3. STRONGEST POSTCONDITIONS FOR NON-ITERATIVE STATEMENTS

Non-Iterative Statements include assignments, I/Os, branch statements and procedures. We will review the calculation of strongest postconditions for assignments, I/Os and procedures.

3.1 Strongest Postconditions for I/O Statements

When \( \text{read}(x) \) is successfully executed (independent of any previous value of \( x \)), \( x \) will always be assigned a value in the type domain \( DT(x) \) which is independent of its precondition. The statement \( \text{write}(E) \) is successfully executed if and only if the variables of the expression \( E \) are assigned and their types-match. What we are interested in is its strongest postcondition under \( P \) that \( \text{write}(E) \) does not change the variable state. So that we simply define:

\[ \text{sp}(P, \text{read}(x)) = P[V \setminus x] \land x \in DT(x) \]

\[ \text{sp}(P, \text{write}(E)) = P \]
3.2 Strongest Postconditions for Assignments

To overcome the difficulty with strongest postcondition for assignments, we prove [15]

\[ \text{sp}(P, x:=E) \equiv P \land \forall y. y \not= x \rightarrow \text{sp}(P', x:=y), \]

where \( P' \) is the result of replacing all occurrences of \( x \) in \( P \) by \( y \).

In practice, these theoretical formulas may make the calculations difficult. Firstly, \( P(x) \land \forall y. y \not= x \rightarrow \text{sp}(P', x:=y) \) may not be easily obtained from \( P(x) \). Secondly, the inverse functions of \( x=E(x) \) may be undefined (e.g. \( x=x/0 \)) or difficult to capture (e.g. \( x=Ax^2+5x \)) or an infinite number of inverse functions (e.g. \( x=x \mod 2 \)). However, from a theoretical viewpoint, for any assignment \( x:=e \), \( \text{sp}(P, x:=e) \) can always be calculated. Since the following calculation/evaluation never uses such an auxiliary variable \( t \) again, we need not remove the auxiliary variable \( t \) from \( \text{sp}(P, t:=x; x:=e(t/x)) \), because it does not affect any subsequent calculation/evaluation. This auxiliary variable gives us a really practical but also simple approach to solve the problem of calculating strongest postconditions involving assignments. More importantly, these calculation difficulties reflect the presence of quality defects in a program: that is, variables are being used for two purposes. The treatment indicates a consistent structure of the program. For example, the variable \( x \) in

\[ \text{read}(x); y:=Ax^2+5x; x:=3; \]

... is used for two purposes (one to record the value of the input and another the value 3). The ideal code should be

\[ \text{read}(t); y:=At^2+5t; x:=3; \]

Examples:

1. \( \text{sp}(a[i]=a[j], x:=a[i]; a[j]:=a[i]); a[j]:=x) \equiv \text{sp}(\text{sp}(a[i]:=a[i] \land x=a[i], a[i]:=a[j]), a[j]:=x) \) (DS5)

2. \( \text{sp}(p(y) \land \neg p(a) \land p(f(a)) \lor x=a, x:=f(x)) \equiv \pm p(y) \land \neg p(a) \land p(f(a)) \lor f^{-1}(x)=a \) (for the abstracted function \( f \))

3. \( \text{sp}(x>-1 \land y>0, x:=x^2) \equiv (\forall x>-1 \land \neg \forall x<1) \land y>0 = (\forall x \geq 0 \lor \forall x<1) \land y>0 = \forall x \geq 0 \land y>0 = x \geq 0 \land y>0 \)

3.3 Strongest Postcondition for Procedures/Functions

Since a procedure/function without side-effects can be treated as a normal statement/expression (where the local variables should be removed from their strongest postconditions), we will therefore focus upon defining strongest postconditions for procedures/function with side-effects.

Given a precondition \( P \) for a procedure \( Q \) with side-effects, the strongest postcondition should be \( \text{sp}(P, Q) \). During calculation, the state-changing statements (i.e., assignments and \( \text{read} \)-statements) in \( Q \) change the initial states of variables in \( P \). As a result, the final variable states in \( \text{sp}(P, Q) \) may be different from the states in \( P \).

When a function has a side-effect, it may change some variable states in the expression. For example, consider an expression \( x+f(x)+x^2 \), where the side-effect function \( f(x) \) increases \( x \) by 1 then returns the current value of \( x \). When the initial value of \( x \) is 1, the expression may correspond to either \( 1+2+2*2 \) if \( x^2 \) is evaluated after \( f(x) \), or \( 1+2+1*1 \) if \( x^2 \) is evaluated before \( f(x) \). The value of the expression depends on the evaluation-order that the selected compiler chooses. In order to model the evaluation-order problem, we denote an expression \( E(x, f(...), y) \) as \( E(x, f(...), y) \), where the variable sets \( x \) and \( y \) are evaluated before and after execution of \( f(...) \) respectively. Given any precondition \( P \) for such an \( E(x, f(...), y) \), the strongest postcondition is defined as

\[ \text{sp}(P, E(x, f(...), y)) \equiv \text{sp}(P, x:=x; f':=f(...); E(x', f', y)) \land \forall x'. \]

Where \( x' \) and \( f' \) are fresh variables (set). This formula corresponds to the following steps:

1. storing \( x \) using a fresh variable set \( x' \), i.e., \( \text{sp}(P, x:=x) \);
2. evaluating function \( f(\ldots) \), i.e., \( \text{sp}(\text{sp}(P, x':=x), f':=f(\ldots)) \);
3. evaluating 'normal' expression \( E'(x', f', y) \), i.e., \( \text{sp}(P, \ldots; E'(x', f', y)) \);
4. removing the auxiliary variables, i.e., \( \text{sp}(P, x':=x; f':=f(\ldots); E'(x', f', y))V-(x', f') \)

**Example:**
\[
\text{sp}(x=1\land y>x, t:=x+f(x)+x*x) \]
\[
= \text{sp}(x=1\land y>x, x':=x; f':=f(x); t:=x'+f'+x*x)V-(x', f') \] where \( f(x): x:=x+1; \text{return}(x) \)
\[
= \text{sp}(x=2\land y>x-1\land x'=1\land f'=x+1\land t=x'+f'+x*x)V-(x', f') \]
\[
= \text{sp}(x=2\land y>x-1\land t=1+2+x*x) \]

### 3.4 Non-Iterative Statement Optimization

Using strongest postcondition calculations, we can define and detect directly various quality defects, such as uninitialized variables, mystery numbers, two-purpose variables and so on. In addition, a statement \( S \) is called **logically redundant** under a given precondition \( P \) if and only if \( \text{sp}(P, S) \Rightarrow P \). A guard \( C \) is called **logically redundant** under a given precondition \( P \) if and only if \( C \) is pre-determined under \( P \) (i.e., \( P \Rightarrow C \) or \( P \Rightarrow \neg C \)). Any redundant statement \( S \), although it executes successfully, can be removed because although it executes successfully, does not change the domain of the initial state. Any redundant \( C \) has its value (true or false) determined before execution. Instead, we can use its value directly in a program which will not change any state domains over the execution path. All (logical) redundancy can be removed while still preserving correctness [15].

During the process of calculation of the strongest postcondition, the result for each statement (including procedures) can be used to remove redundant components using a set of transformation rules, such as:

\[
\{P\} x:=E \quad \Rightarrow \quad \{P\} \emptyset \quad \text{when } P \Rightarrow (x=E), \text{Pfalse} \\
\{P\} \text{ if } C \rightarrow S_1 \: \neg C \rightarrow S_2 \: \text{fi} \quad \Rightarrow \quad \{P\} \text{ if } C \not\Rightarrow P \rightarrow S_1 \: \neg (C \not\Rightarrow P) \rightarrow S_2 \: \text{fi} \quad \text{where Pfalse} 
\]

The first rule removes self-assignment \( x:=x \) and any ineffective assignment \( x:=E \) that establishes an already established condition in \( P \). The second rule guarantees that each branch guard is not pre-determined under the precondition. The definition for any redundant procedure \( Q \) under a precondition \( P \) still follows \( \text{sp}(P, Q) \Rightarrow P \). In addition, another type of redundant procedure occurs where only output variables other than global variables (i.e., it is a side-effect free procedure) are employed, if all output variables are never used before they are reassigned.

The strongest postcondition calculation can allow an arbitrary pair of two non-iterative statements to change their order and preserve logical equivalence. This results in a small number (10) of equivalence transformation rules being required for removing redundancy from non-iterative statement sequences [15].

### 4. STRONGEST POSTCONDITION FOR LOOPS

Before proceeding with the strongest postcondition for loops, we briefly review some issues relating to use of strongest postcondition calculations to improve loop structures.

#### 4.1. Loop Normalization

Major quality improvement tasks are detecting and removing redundancy and detecting defects (including non-termination). To assist in handling these problems, a procedure, called Loop Normalization [15], has been developed, based on strongest postcondition calculations, graph theory and recursion. It transforms any loop into an equivalent segment containing only Well-Structured loops, from which redundancy has been removed and each guard (including branch-guards and nested subloop guards) is not pre-determined under its precondition during iteration.

Here we use an example -- List Merge [14], which contains considerable inefficiencies:

\[
\text{out}:=\text{Nil; do } \neg C_1 \lor \neg C_2 \iff \text{if } C_2 \lor (\neg C_1 \lor \neg C_2 \lor \neg C_3) \rightarrow S_1 \rightarrow \neg C_2 \lor (C_1 \lor C_2 \lor C_3) \rightarrow S_2 \text{ fi od} \\
\text{where } C_1=(\text{in1=Nil}), S_1=(\text{out}=\text{out}++\text{in1.head}; \text{in1}=\text{in1.tail}) \text{ for } i=1,2, \text{ and } C_3=(\text{in1.head}<\text{in2.head}).
\]
Using strongest postconditions to improve software quality

to illustrate briefly how loop normalization reconstructs loops. Firstly, this loop body indicates
four branches, that is

\[ \text{if } C_1 \land \neg C_2 \rightarrow S_2 \land C_2 \land C_1 \rightarrow S_1 \land \neg C_1 \land \neg C_2 \land C_3 \rightarrow S_3 \land C_1 \land \neg C_2 \land C_3 \rightarrow S_2. \]

Using strongest postcondition calculations, we know that after execution of the first or second
branch, the control-flow may terminate or enter the same branch again (as \( \text{sp}(C_1 \land \neg C_2, S_2) \Rightarrow C_1 \)
and \( \text{sp}(C_2 \land \neg C_1, S_1) \Rightarrow C_2 \)). And, after execution of the last branch the control-flow may
terminate or enter one of the last three branches. The third branch has a similar property to the
last branch. From these results we can construct the \textit{Branch Successor Graph} (BSG) for this loop:

The BSG is constructed as follows: we first calculate the strongest postcondition that applies
when each branch has been executed. By comparing this condition (using implication) with the
condition for executing each possible branch we can identify which branches are directly
reachable (i.e., successors) after executing a given branch. Obviously, a quality loop structure
should exhibit a \textit{strongly connected} BSG. This means, any node should be connected to all nodes
(i.e., after execution of any branch, the control-flow should be able to access all branches if the
loop guard holds). In carrying out the \textit{structural refinement} of loops the ultimate goal is to
transform every BSG into a single or a set of strongly connected components. For example, with
List Merge, there are three strongly connected components in its BSG (corresponding to three
loops). So the normalized loop is:

\[
\text{out}:=\text{Nil};
\text{do } C_1 \land C_2 \rightarrow \text{if } C_3 \rightarrow S_1 \land \neg C_3 \rightarrow S_2 \land C_3 \rightarrow S_1 \land \neg C_1 \rightarrow S_2 \land C_3 \rightarrow S_3 \land C_2 \rightarrow S_2. \]

4.2. Strongest Accessible Conjunctive Postcondition

We use a computational model to calculate the \textit{Strongest Accessible Conjunctive Postconditions} \[15\] for well-structured loops. The formal definition is:

Given a conjunctive precondition \( P \) of a loop \( \text{do } G \rightarrow S \text{ od} \), the \textit{Strongest Accessible Conjunctive Postcondition}, denoted by \( \text{sp}_{ac}(P, \text{do } G \rightarrow S \text{ od}) \), is the conjunctive condition such that:

\[
\text{sp}_{ac}(P, \text{do } G \rightarrow S \text{ od}) \subseteq V(P) \cup V(S) \quad (\text{Accessibility})
\]

\[
\text{sp}(P, \text{do } G \rightarrow S \text{ od}) \Rightarrow \text{sp}_{ac}(P, \text{do } G \rightarrow S \text{ od}) \quad (\text{Strongest Conjunctive})
\]

\[
\forall q \left( \text{sp}(P, \text{do } G \rightarrow S \text{ od}) \Rightarrow q \right) \Rightarrow \left( \text{sp}_{ac}(P, \text{do } G \rightarrow S \text{ od}) \Rightarrow q \right)
\]

where \( V(S) \) represents the variable set which is assigned by \( S \), and \( q \) is a conjunctive formula
and \( q \neq \text{true} \).

The desired relationship \( \text{sp}_{ac}(P, \text{do } G \rightarrow S \text{ od}) \equiv \text{sp}(P, \text{do } G \rightarrow S \text{ od}) \) may hold. When it does
not hold, the former is the strongest (accessible) conjunctive condition that the latter implies.
Even in those cases, the former is still strong enough to be useful for improving program quality
and deriving specifications from programs \[15\] because of their conjunctive form and
accessibility. In contrast to DS4 and Back's result \[1\], the calculation depth of our process \[15\]
to obtain \( \text{sp}_{ac}(P, \text{do } G \rightarrow S \text{ od}) \) is fixed.

\textbf{Example:}

The Finding-First-Maximum-Position program \[5\]:

\[
\{ 1 \leq N \} \ q := i \times i ;
\text{do } i \leq N \rightarrow \text{if } A[i+1] \leq A[p] \rightarrow i := i+1; \quad \text{if } A[p] < A[i+1] \rightarrow i := i+1; \quad p := i \times i \text{ od};
\]

Our calculation method yields the strongest accessible conjunctive postcondition
\( \exists j \in \{ 1 \ldots N \}, \exists j^* \in \{ 1 \ldots p-1 \}, \text{A} \leq [N, J^*] \land \forall j \in \{ N, J^* + 1 \} A \leq \text{A} \leq [N, J^*] \), which is also its strongest
postcondition. More importantly, after removal of the local variable (i.e., using \( R_1 \) to remove \( i \)),
we end up with its \textit{(post-)specification} \( \exists j \in \{ 1 \ldots N \}, \exists j^* \in \{ 1 \ldots p-1 \} A \leq [N, J^*] \land \forall j \in \{ 1 \ldots p, J^* \} A \leq [N, J^*] \).

4.3 Using Strongest Postcondition To Improve Software Quality

We now use the example of the Abstract Program \( AP \) suggested by Manna \[13\]:

\[
\text{if } p(y) \rightarrow x := y; \text{ do } \neg p(x) \rightarrow x := f(x) \text{ od}
\]

\[
\text{if } p(y) \rightarrow x := a;
\]
if \( \text{p(x)} \rightarrow \text{do} \neg\text{p(x)} \rightarrow x:=\text{f(x)} \) od \( \neg\text{p(x)} \rightarrow x:=\text{f(x)} \)

\[
\text{fl}
\]

to show the strongest postcondition calculation for \( \text{AP} \) under the initial precondition true. A summary of the calculations is as follows:

\[
\text{sp(true, } \text{AP}) \Rightarrow \text{sp(true} \wedge \neg\text{p(y)}, \ x:=y; \ \text{if} \ \text{p(a)} \ \text{then} \ x:=a; \ \text{else} \ x:=f(a); \ \text{fi}
\]

\[
(DS3)
\]

\[
\text{sp(p(y)} \wedge x=y, \ do \ \neg\text{p(x)} \rightarrow \ldots \) \ \text{od} \ \vee \ \text{sp(}\neg\text{p(y)} \wedge x=a \wedge \text{p(x)}, \ do \ \ldots \) \ \text{od} \ \vee \ \text{sp(}\neg\text{p(y)} \wedge x=a \wedge \neg\text{p(x)}, \ x:=f(x); \ \text{if} \ \ldots \) \ \text{fi}
\]

\[
(\text{since( ... \wedge \neg\text{p(a)} \wedge x=a \wedge \text{p(f(a)}), \ x:=a; \ \text{do} \ \ldots \) \ \text{fi})}
\]

\[
(\text{p(y)} \wedge x=y) \ \vee \ (\neg\text{p(y)} \wedge x=a \wedge \text{p(x)}) \ \vee \ (\neg\text{p(y)} \wedge \text{p(a)} \wedge x=f(a) \wedge \neg\text{p(x)}) \ \vee
\]

\[
\text{sp(}\neg\text{p(y)} \wedge \neg\text{p(a)} \wedge x=f(a) \wedge \text{p(f(a)}), x:=a; \ x:=f(x); \ \text{do} \ \ldots \) \ \text{fi}
\]

5. CONCLUSION

In this paper we have shown how the concept of strongest postconditions provides the basis for a formal and powerful computational model that may be used to enhance the quality of imperative programs. When these calculations are combined with optimizing transformations a very effective method is obtained for removing various forms of redundancy and inconsistency from programs. The closed form for calculation of the strongest postcondition (including loops) makes the whole approach of significant practical interest for monitoring, controlling and improving the quality of new and existing software.

References: