Strongest Postcondition Calculus

– lecture notes –

Contents

The content of this document is not claimed to be original or to be published as original research. It is meant to serve as a learning resource for students.

1 Introduction

These notes provide an introduction to strongest postcondition calculus, a fundamental technique in program verification that complements the weakest precondition approach. While weakest precondition calculus works backward from a desired postcondition, strongest postcondition calculus works forward from a given precondition to compute what can be guaranteed about the program state after execution.

The concept of strongest postconditions is closely related to Robert W. Floyd's 1967 work on "Assigning Meanings to Programs," which precedes C. A. R. Hoare's more widely known formalization of program correctness in 1969. Floyd's approach to the assignment rule naturally leads to the notion of strongest postconditions.

2 Review of Hoare Logic and Floyd's Approach

2.1 Hoare's Assignment Rule

In standard Hoare logic, the assignment rule is:

$$\frac{}{\vdash \{Q[e/x]\}\ \texttt{x := e}\ \{Q\}}$$

This rule works *backward*: given a desired postcondition Q, we compute the precondition by substituting the expression e for variable x in Q.

2.2 Floyd's Assignment Rule

Floyd proposed an alternative formulation that works *forward*:

$$\frac{}{\vdash \{P\}\ \texttt{x := e}\ \{\exists v.(x = e[v/x]) \wedge P[v/x]\}}$$

In this rule:

- v is a fresh variable representing the *old value* of x before the assignment
- $e[v/x]$ is the expression e with v substituted for x
- $P[v/x]$ is the precondition P with v substituted for x

The existentially quantified postcondition captures both:

1. The new value of x (namely, $x = e[v/x]$)
2. The fact that the precondition P held for the old value v

This is exactly the strongest postcondition that can hold after the execution of the assignment because it is almost the same as the precondition, except that the variable x is different.

Example 2.1. *Consider the Hoare triple:*

$$\vdash \{s = 0\}\ \texttt{s := s + i}\ \{?\}$$

Using Floyd's rule, the postcondition is:

$$\exists v.(s = (s + i)[v/s]) \wedge (s = 0)[v/s]$$

After applying the substitutions:

$$\exists v.(s = v + i) \wedge (v = 0)$$

This can be simplified by substituting 0 *for* v *and eliminating the quantifier:*

$$s = 0 + i \equiv s = i$$

2.3 Comparison with Hoare's Rule

Let us verify that both approaches yield consistent results.

Example 2.2 (Continued)**.** *Using Hoare's rule with postcondition* $Q \equiv s = 0 + i$*:*

$$\begin{aligned} \textit{Precondition} &= Q[s + i/s] \\ &= (s = 0 + i)[s + i/s] \\ &= s + i = 0 + i \\ &\equiv s = 0 \end{aligned}$$

This matches our original precondition, confirming that $s = i$ *is indeed the strongest postcondition we can derive.*

3 From Floyd's Rule to Strongest Postconditions

3.1 Motivation

While Floyd discussed related notions in his 1967 paper (specifically, "strongest verifiable consequents"), the concept of strongest postconditions provides a systematic way to reason about programs in the forward direction.

Definition 3.1 (Strongest Postcondition Intuition)**.** *For a statement* S *and precondition* P*, the* strongest postcondition $sp(S, P)$ *is a logical formula such that:*

1. *If* P *holds before executing* S*, then* $sp(S, P)$ *holds after execution*
2. $sp(S, P)$ *is the* strongest *such formula: any other valid postcondition* Q *for* $\{P\}S\{Q\}$ *is implied by* $sp(S, P)$

Theorem 3.2. *A Hoare triple* $\{P\}S\{Q\}$ *is valid if and only if:*

$$sp(S, P) \rightarrow Q$$

This property reduces the verification problem to checking a logical implication, similar to how weakest precondition calculus works.

3.2 Relationship to Weakest Preconditions

Recall that for weakest preconditions:

$$\{P\}S\{Q\} \textit{ is valid} \text{ iff } P \rightarrow \text{wp}(S, Q)$$

The two approaches are dual:

- **Weakest Precondition:** Works backward from Q to find the weakest P
- **Strongest Postcondition:** Works forward from P to find the strongest Q

3.3 IMP Language - reminder

We recall here the syntax of IMP:

Arithmetic Expressions:

$$\text{AExp} ::= \text{Var} \mid \text{Int} \mid \text{AExp} + \text{AExp} \mid \text{AExp}/\text{AExp}$$

Boolean Expressions:

$$\text{BExp} ::= \texttt{true} \mid \texttt{false} \mid \text{AExp} < \text{AExp} \mid \texttt{not}\ \text{BExp} \mid \text{BExp}\ \texttt{and}\ \text{BExp}$$

Statements:

$$\begin{aligned}\text{Stmt} ::= &\ \text{Var} := \text{AExp}\\ \mid &\ \texttt{if}\ \text{BExp}\ \texttt{then}\ \text{Stmt}\ \texttt{else}\ \text{Stmt}\\ \mid &\ \texttt{while}\ \text{BExp}\varphi\ \texttt{do}\ \text{Stmt}\ \texttt{end}\\ \mid &\ \text{Stmt}; \text{Stmt}\\ \mid &\ \texttt{skip}\end{aligned}$$

4 Strongest Postcondition Calculus

We now present the complete strongest postcondition calculus for the IMP language.

4.1 Assignment

Definition 4.1 (SP for Assignment)**.**

$$sp(x := e, P) = \exists v.(x = e[v/x]) \wedge P[v/x]$$

The fresh variable v represents the old value of x, and the postcondition states that:

1. x has the new value obtained by evaluating e with the old value v
2. The precondition P held when x had value v

Example 4.2. *Consider verifying* $\{s = 0\}$ `s := s + i` $\{s \geq i\}$.

Step 1: *Compute* $sp($`s := s + i`$, s = 0)$*:*

$$\begin{aligned}sp(\texttt{s := s + i}, s = 0) &= \exists v.(s = (s + i)[v/s]) \wedge (s = 0)[v/s]\\ &= \exists v.(s = v + i) \wedge (v = 0)\\ &\equiv s = 0 + i\\ &\equiv s = i\end{aligned}$$

Step 2: *Check whether* $sp($`s := s + i`$, s = 0) \rightarrow s \geq i$*:*

$$s = i \rightarrow s \geq i$$

Since this implication is valid, the Hoare triple is valid.

4.2 Sequential Composition

Definition 4.3 (SP for Sequences)**.**

$$sp(S_1; S_2, P) = sp(S_2, sp(S_1, P))$$

This definition is natural: we first compute the strongest postcondition after S_1, then use that as the precondition for computing the strongest postcondition of S_2.

Remark 4.4. *Compare this with the weakest precondition for sequences:*

$$wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$$

The order of composition is reversed: wp works backward (first S_2, then S_1), while sp works forward (first S_1, then S_2).

Example 4.5. *Compute $sp(i := i + 1; s := s + i, i = 0 \wedge s = 0)$.*
Step 1: *Compute $sp(i := i + 1, i = 0 \wedge s = 0)$:*

$$\begin{aligned} &sp(i := i + 1, i = 0 \wedge s = 0) \\ &= \exists v.(i = (i + 1)[v/i]) \wedge (i = 0 \wedge s = 0)[v/i] \\ &= \exists v.(i = v + 1) \wedge (v = 0 \wedge s = 0) \end{aligned}$$

Step 2: *Compute $sp(s := s + i, \exists v.(i = v + 1) \wedge (v = 0 \wedge s = 0))$:*

$$\begin{aligned} &sp(s := s + i, \exists v.(i = v + 1) \wedge (v = 0 \wedge s = 0)) \\ &= \exists v'.(s = (s + i)[v'/s]) \wedge (\exists v.(i = v + 1) \wedge (v = 0 \wedge s = 0))[v'/s] \\ &= \exists v'.(s = v' + i) \wedge (\exists v.(i = v + 1) \wedge (v = 0 \wedge v' = 0)) \\ &\equiv \exists v.\exists v'.(s = v' + i) \wedge (i = v + 1) \wedge (v = 0 \wedge v' = 0) \\ &\equiv \exists v.\exists v'.(s = 0 + i) \wedge (i = 0 + 1) \wedge (v = 0 \wedge v' = 0) \\ &\equiv s = i \wedge i = 1 \\ &\equiv s = 1 \wedge i = 1 \end{aligned}$$

4.3 Skip

Definition 4.6 (SP for Skip)**.**

$$sp(\texttt{skip}, P) = P$$

Since the skip statement does nothing, the strongest postcondition is simply the precondition itself.

Remark 4.7. *This is identical to the weakest precondition: $wp(\texttt{skip}, Q) = Q$.*

4.4 Conditional Statement

Definition 4.8 (SP for If-Then-Else)**.**

$$sp(\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2, P) = sp(S_1, P \wedge C) \vee sp(S_2, P \wedge \neg C)$$

The postcondition is a disjunction:

- If the condition C was true, we execute S_1 starting from $P \wedge C$
- If the condition C was false, we execute S_2 starting from $P \wedge \neg C$

Remark 4.9. *Compare with weakest precondition:*

$$wlp(\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2, Q) = (C \to wlp(S_1, Q)) \land (\neg C \to wlp(S_2, Q))$$

The sp rule uses disjunction ($\lor$) because we don't know which branch was taken, while the wlp rule uses conjunction ($\land$) because we must satisfy Q regardless of which branch is taken.

Example 4.10. *Compute* $sp(\texttt{if } x < 0 \texttt{ then } m := -x \texttt{ else } m := x, \texttt{true})$.

Then branch: *Compute* $sp(m := -x, \texttt{true} \land (x < 0))$*:*

$$\begin{aligned} &sp(m := -x, \texttt{true} \land (x < 0)) \\ &= \exists v.(m = (-x)[v/m]) \land (\texttt{true} \land (x < 0))[v/m] \\ &= \exists v.(m = -x) \land (\texttt{true} \land (x < 0)) \\ &\equiv m = -x \land x < 0 \end{aligned}$$

Else branch: *Compute* $sp(m := x, \texttt{true} \land \neg(x < 0))$*:*

$$\begin{aligned} &sp(m := x, \texttt{true} \land \neg(x < 0)) \\ &= \exists v.(m = x[v/m]) \land (\texttt{true} \land \neg(x < 0))[v/m] \\ &= \exists v.(m = x) \land (\texttt{true} \land (x \geq 0)) \\ &\equiv m = x \land x \geq 0 \end{aligned}$$

Combined:

$$\begin{aligned} &sp(\texttt{if } x < 0 \texttt{ then } m := -x \texttt{ else } m := x, \texttt{true}) \\ &= (m = -x \land x < 0) \lor (m = x \land x \geq 0) \end{aligned}$$

This postcondition states that m contains the absolute value of x.

4.5 While Loop

Definition 4.11 (SP for While Loop)**.**

$$sp(\texttt{while } C \texttt{ do } S, P) = sp(\texttt{while } C \texttt{ do } S, sp(S, P \land C)) \lor (P \land \neg C)$$

This recursive definition captures the unrolling of the loop:

- $(P \land \neg C)$ represents the case where the loop never executes
- $\text{sp}(\texttt{while } C \texttt{ do } S, \text{sp}(S, P \land C))$ represents executing the loop body once and then recursively processing the remaining iterations

Remark 4.12. *The strongest postcondition calculus is only meaningful for* partial correctness*, as it captures the changes made to the program state after executing a statement, but does not reason about termination.*

Example 4.13. *Compute* $sp(\texttt{while } (i \leq n) \texttt{ do } i := i + 1, i = 0 \land n = 2)$.

We unroll the loop execution:

Iteration 0: *The loop exits immediately if $i > n$:*

$$i = 0 \land n = 2 \land i > n \equiv \texttt{false}$$

Iteration 1: *Execute the body once from* $i = 0 \land n = 2 \land (i \leq n)$*:*

$$\begin{aligned} &sp(i := i + 1, i = 0 \land n = 2 \land (i \leq n)) \\ &= \exists v.(i = v + 1) \land (v = 0 \land n = 2 \land (v \leq n)) \\ &\equiv i = 1 \land n = 2 \end{aligned}$$

Then check if the loop exits: $i = 1 \land n = 2 \land i > n \equiv \texttt{false}$

Iteration 2: *Execute the body from* $i = 1 \land n = 2 \land (i \leq n)$*:*

$$\begin{aligned} &sp(i := i + 1, i = 1 \land n = 2 \land (i \leq n)) \\ &\equiv i = 2 \land n = 2 \end{aligned}$$

Then check if the loop exits: $i = 2 \land n = 2 \land i > n \equiv \texttt{false}$

Iteration 3: *Execute the body from* $i = 2 \land n = 2 \land (i \leq n)$*:*

$$\begin{aligned} &sp(i := i + 1, i = 2 \land n = 2 \land (i \leq n)) \\ &\equiv i = 3 \land n = 2 \end{aligned}$$

Now check if the loop exits: $i = 3 \land n = 2 \land i > n \equiv \texttt{true}$

Final result:

$$sp(\texttt{while}\ (i \leq n)\ \texttt{do}\ i := i + 1, i = 0 \land n = 2) = i = 3 \land n = 2 \land i > n$$

This can be simplified to: $i = 3 \land n = 2$ *(since* $3 > 2$ *is already implied).*

Remark 4.14. *Each disjunct in the full expansion corresponds to a particular program execution path. In the example above, the conjuncts representing iterations 0, 1, and 2 are false, indicating impossible executions (the loop continues in those cases). Only the path corresponding to three iterations yields a satisfiable postcondition.*

5 Summary of Strongest Postcondition Rules

Statement	**Strongest Postcondition**
Assignment	$\text{sp}(x := e, P) = \exists v.(x = e[v/x]) \land P[v/x]$
Sequence	$\text{sp}(S_1; S_2, P) = \text{sp}(S_2, \text{sp}(S_1, P))$
Skip	$\text{sp}(\texttt{skip}, P) = P$
Conditional	$\text{sp}(\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2, P)$ $= \text{sp}(S_1, P \land C) \lor \text{sp}(S_2, P \land \neg C)$
While Loop	$\text{sp}(\texttt{while}\ C\ \texttt{do}\ S, P)$ $= \text{sp}(\texttt{while}\ C\ \texttt{do}\ S, \text{sp}(S, P \land C)) \lor (P \land \neg C)$

6 Properties of Strongest Postconditions

6.1 Main Theorems

Theorem 6.1 (Soundness and Completeness of SP)**.** *A Hoare triple* $\{P\}S\{Q\}$ *is valid if and only if:*

$$\models sp(S, P) \to Q$$

Proof Sketch. ($\Rightarrow$) If $\{P\}S\{Q\}$ is valid, then by definition of strongest postcondition, any valid postcondition must be implied by $\text{sp}(S, P)$, hence $\text{sp}(S, P) \to Q$.

($\Leftarrow$) If $\text{sp}(S, P) \to Q$, then by the soundness of sp, we have $\{P\}S\{\text{sp}(S, P)\}$, and by the consequence rule, $\{P\}S\{Q\}$. □

6.2 Relationship to Weakest Preconditions

Theorem 6.2 (Duality of WP and SP)**.** *For any statement S, precondition P, and postcondition Q:*

$$\models sp(S, P) \to Q \text{ iff } \models P \to wp(S, Q)$$

This theorem establishes that sp and wp are dual approaches to the same verification problem.

6.3 Comparison: SP vs. WP

Strongest Postcondition	**Weakest Precondition**
Works forward from precondition	Works backward from postcondition
Computes what is guaranteed after execution	Computes what is required before execution
Aligned with symbolic execution	Aligned with automated verification
Natural for testing and analysis	Natural for verification and proof
Handles loops less elegantly (no invariants)	Handles loops with invariants and variants
Better for exploring program behavior	Better for verifying specifications

7 Strongest Postconditions and Symbolic Execution

7.1 Connection to Symbolic Execution

Definition 7.1 (Symbolic Execution)**.** Symbolic execution *is a program analysis technique that executes a program with* symbolic inputs *(representing sets of possible values) rather than concrete inputs. The execution maintains:*

1. *A* symbolic state *that maps variables to symbolic expressions*
2. *A* path condition *that constrains the symbolic values along the current execution path*

Remark 7.2. *Strongest postconditions provide a formal foundation for symbolic execution:*

- *The symbolic state after executing S from precondition P is captured by $sp(S, P)$*
- *Path conditions correspond to the conjuncts in the sp formula*
- *Each disjunct in an sp formula represents a different execution path*

7.2 Symbolic Execution Example

Consider the following program:

read n
$s := 0$

while $n > 0$ **do**
  $s := s + n$
  $n := n - 1$
end while

Symbolic execution with symbolic input x (representing the initial value of n) produces multiple execution paths:

1. **Path 1:** $x \leq 0$
   - Loop never executes
   - Final state: $s = 0, n = x$
2. **Path 2:** $x > 0 \wedge (x - 1) \leq 0$
   - Loop executes once
   - Final state: $s = x, n = x - 1$
3. **Path 3:** $x > 0 \wedge (x - 1) > 0 \wedge (x - 2) \leq 0$
   - Loop executes twice
   - Final state: $s = x + (x - 1), n = x - 2$
4. **Path k:** $x > k - 1 \wedge (x - k) \leq 0$
   - Loop executes k times
   - Final state: $s = x + (x - 1) + \cdots + (x - k + 1), n = x - k$

7.3 Key Concepts in Symbolic Execution

Symbolic Values: Variables like x that represent arbitrary (unknown) concrete values.

Path Conditions: Logical formulas like $x > 0 \wedge (x - 1) \leq 0$ that constrain the symbolic values along a specific execution path.

Symbolic Execution Tree: A tree structure where:

- Each node represents a program point
- Each edge represents a transition (either sequential or conditional)
- Each path from root to leaf represents a possible execution

7.4 Formal Connection

Property 7.3. *The final symbolic state and path condition after symbolically executing statement S from an initial symbolic state satisfying P exactly corresponds to $sp(S, P)$.*

More precisely:

- Each disjunct in $\text{sp}(S, P)$ represents a feasible execution path
- The path condition for each path appears as conjuncts within that disjunct
- The symbolic state is captured by the equalities in the formula

7.5 Desired Properties

For sound and complete symbolic execution, we require:

1. **Coverage (Soundness):** Every concrete execution is covered by at least one symbolic execution path
2. **Precision (Completeness):** Every symbolic execution path corresponds to at least one concrete execution

8 Practical Challenges and Applications

8.1 Challenges in Symbolic Execution

Path Explosion: The number of execution paths grows exponentially with program size, particularly for programs with many conditional branches.

Loop Handling: Loops create infinitely many paths (or very many paths for bounded loops). Strategies include:

- Loop unrolling (bounded depth)
- Abstraction and summarization
- Integration with abstract interpretation

Constraint Solving: Path conditions can become very complex, requiring powerful SMT solvers and constraint simplification techniques.

Scalability: Real-world programs often exceed the capabilities of pure symbolic execution, necessitating selective or compositional approaches.

8.2 Mitigation Strategies

1. **Concolic Testing:** Combine concrete and symbolic execution (e.g., start with concrete values, switch to symbolic when needed)
2. **Path Pruning:** Use heuristics to eliminate unlikely or redundant paths
3. **State Merging:** Combine similar symbolic states to reduce path explosion
4. **Abstract Interpretation:** Use abstract domains to approximate program behavior
5. **Compositional Analysis:** Analyze program components separately and compose results

8.3 Applications

Strongest postconditions and symbolic execution are used in:

- **Automated Test Generation:** Generate test inputs by solving path conditions
- **Bug Finding:** Explore execution paths to find crashes, assertion violations, and undefined behavior
- **Vulnerability Detection:** Identify security vulnerabilities like buffer overflows
- **Program Understanding:** Analyze program behavior and document possible states
- **Verification:** Prove properties about program behavior

9 Tools Based on Symbolic Execution

Several mature tools implement symbolic execution:

- **KLEE:** A symbolic execution engine built on top of LLVM, primarily for C programs
  - URL: `https://klee.github.io/`
  - Focus: Automated test generation and bug finding
- **S2E (Selective Symbolic Execution):** A platform for analyzing entire system stacks
  - URL: `https://s2e.systems/`
  - Focus: Whole-system analysis combining concrete and symbolic execution
- **Angr:** A Python framework for binary analysis with symbolic execution
  - URL: `https://angr.io/`
  - Focus: Reverse engineering and vulnerability analysis
- **SymCC:** A compiler-based symbolic execution tool
  - URL: `https://github.com/eurecom-s3/symcc`
  - Focus: Fast symbolic execution through compilation
- **Manticore:** A symbolic execution tool for analyzing binaries and smart contracts
  - URL: `https://github.com/trailofbits/manticore`
  - Focus: Security analysis of native code and Ethereum smart contracts
- **Driller:** A tool combining fuzzing with symbolic execution
  - URL: `https://github.com/shellphish/driller`
  - Focus: Augmenting fuzzing to overcome complex branch conditions

10 Summary and Conclusions

1. **Forward Reasoning:** Strongest postcondition calculus provides a systematic way to reason about programs in the forward direction, starting from a precondition.
2. **Duality with WP:** Strongest postconditions are dual to weakest preconditions: both can be used to verify program correctness, but they work in opposite directions.
3. **Connection to Symbolic Execution:** The sp calculus provides the formal foundation for symbolic execution, with each disjunct representing an execution path.
4. **Practical Challenges:** Path explosion, loop handling, and constraint complexity remain significant challenges in practice.
5. **Complementary Approaches:** While wp is preferred for verification (due to invariants), sp is natural for program exploration and testing.

10.1 Limitations

Both strongest postconditions and weakest preconditions have limitations:

- Loop invariants (for wp) or path explosion (for sp) remain challenges
- Expressiveness is limited by the underlying logic
- Automation is incomplete: some verification tasks require human insight
- Scalability to large real-world programs remains difficult

11 Further Reading

- Robert W. Floyd, "Assigning Meanings to Programs," *Proceedings of Symposia in Applied Mathematics*, Vol. 19, 1967
  - URL: `https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf`
- C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, 1969
  - URL: `https://dl.acm.org/doi/pdf/10.1145/363235.363259`
- James C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, Vol. 19, No. 7, 1976
- Patrice Godefroid, Michael Y. Levin, and David Molnar, "Automated Whitebox Fuzz Testing," *NDSS*, 2008
- Cristian Cadar and Koushik Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communications of the ACM*, Vol. 56, No. 2, 2013