Simplifying Loop Invariant Generation Using Splitter Predicates[⋆]

Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken

Department of Computer Science
Stanford University
{sharmar,isil,tdillig,aiken}@cs.stanford.edu

Abstract. We present a novel static analysis technique that substantially improves the quality of invariants inferred by standard loop invariant generation techniques. Our technique decomposes *multi-phase* loops, which require disjunctive invariants, into a semantically equivalent sequence of single-phase loops, each of which requires simple, conjunctive invariants. We define *splitter predicates* which are used to identify phase transitions in loops, and we present an algorithm to find useful splitter predicates that enable the phase-reducing transformation. We show experimentally on a set of representative benchmarks from the literature and real code examples that our technique substantially increases the quality of invariants inferred by standard invariant generation techniques. Our technique is conceptually simple, easy to implement, and can be integrated into any automatic loop invariant generator.

Keywords: Static analysis, invariant generation, decomposition of multi-phase loops

1 Introduction

A key problem in any automatic software verification system is the inference of loop invariants. A consistent theme in the literature is that most loops found in practice require only simple quantifier-free *conjunctive* invariants (i.e., invariants that are conjunctions of elementary facts) and that such invariants are relatively easy to infer using standard techniques such as [1,2,3].

However, some loops in real programs fundamentally require *disjunctive* invariants (i.e., an invariant with at least one disjunction). While relatively rare, obtaining accurate invariants for such loops is still important for successful verification, as imprecision in the analysis of even a small part of a program tends to spread, often reducing analysis precision for much of, or even the entire, program. As a result, a number of previous efforts have proposed techniques for inferring disjunctive invariants [4,5,6,7,8]. While there is considerable diversity in the approaches taken, the proposed disjunctive invariant generation techniques are considerably more involved than the more straightforward conjunctive case.

[⋆] This work was supported by NSF grants CNS-050955 and CCF-0702681.

To illustrate the problem of disjunctive invariant generation, consider Fig. 1(a), which is the motivating example of [4]. To prove the validity of the assertion, the following disjunctive invariant is required:

$$(x \leq 50 \wedge y = 50) \vee (50 \leq x \leq 100 \wedge y = x) \ . \tag{1}$$

Abstract interpretation-based techniques that generate only conjunctive invariants fail on this example. For instance, the widely-used abstract interpretation-based tool Interproc works over convex abstract domains [9] and computes invariants that are conjunctions of linear inequalities. For this loop, Interproc computes the post-condition $50y \geq 2599$ and cannot verify the assertion $y = 100$ tested in the last line of Fig. 1(a). Techniques such as [10] can infer disjunctive invariants, but for this example do so in a brute-force manner, performing 50 refinement iterations. Similarly, predicate abstraction techniques such as Slam [11] and Blast [12] generate a sequence of predicates of the form $x = 1, x = 2, \ldots$ during abstraction refinement and require 100 refinement iterations. Some much more elaborate techniques using interpolants [13] and probabilistic inference [4] can verify the correctness of this program without counting to the loop bound, but it is difficult to give an intuitive characterization of the class of loops for which these techniques can infer useful disjunctive invariants.

```
x=0;y=50;
while(x<100)
{
  x=x+1;
  if(x>50)
    y=y+1;
}
assert(y==100);
```

(a) Example from [4].

```
x=0;y=50;
while(x<=49)
{
  x=x+1;
}
while(x<100 && x>49)
{
  x=x+1;
  y=y+1;
}
assert(y==100);
```

(b) The example after splitting.

Fig. 1. Loop (a) requires a disjunctive invariant, but the equivalent program (b) requires only conjunctive invariants.

While Fig. 1(a) is a synthetic example, it is representative of the loops found in practice that require disjunctive invariants. Specifically, this example has two important properties:

(a) The need for a disjunctive invariant arises from a particular conditional or conditionals in the loop body; in this case, the statement `if (x > 50)`. Not all conditionals imply that a disjunctive invariant is needed, but conditionals whose predicate is related to how many iterations the loop has executed, as in Fig. 1(a), usually do. For example, one of the more common patterns in practice is that the conditional causes the loop to do something different in its base (the first or first few iterations) and inductive cases (all subsequent iterations).

(b) The conditionals in question exhibit a fixed number of *phase transitions* during execution. A *phase* is a sequence of iterations in which the conditional, if it is evaluated, always evaluates to the same value, either *true* or *false*. A *phase transition* occurs when the conditional evaluates to b in one iteration, and the next time it is evaluated, it evaluates to $\neg b$. In Fig. 1, the conditional test `x > 50` has two phases and one phase transition: it is *false* for iterations 1-50, and *true* for iterations 51-100.

In principle, there are many loops requiring disjunctive invariants that do not satisfy conditions (a) and (b). However, it is our experience that the vast majority of loops arising in practice that require a disjunctive invariant do so because of conditionals with a fixed number of phases. For example, we have manually inspected the 95 loops found in OpenSSH, and found that exactly 9 of these loops require disjunctive invariants. Furthermore, of these 9 loops, all but one[1] satisfies conditions (a) and (b) above. Throughout this paper, we refer to loops satisfying conditions (a) and (b) as *multi-phase loops*.

The observation that multi-phase loops constitute a large majority of the loops that are not amenable to reasoning by standard invariant generation techniques motivates our approach: Rather than developing techniques to directly infer disjunctive invariants, we employ static analysis to identify phase transitions of multi-phase loops by computing *splitter predicates*. We then perform a program transformation that converts such multi-phase loops requiring disjunctive invariants to a semantically equivalent sequence of single-phase loops, each of which requires only conjunctive invariants. In general, if a loop has a conditional with k phases, it can be split into k successive loops without the conditional test.

As an example, consider again the loop from Fig. 1. Here, we can eliminate the phase transition by *splitting* the loop into two loops, one for each phase, as shown in Fig. 1(b). The resulting two loops have no conditionals and require only simple conjunctive invariants. Recall that the invariant for Fig. 1(a) is $(x \leq 50 \land y = 50) \lor (50 \leq x \leq 100 \land y = x)$. Here, the first disjunct, $x \leq 50 \land y = 50$, corresponds to the invariant of the first loop from Fig. 1(b), and the second disjunct, $50 \leq x \leq 100 \land y = x$, is the invariant of the second loop in Fig. 1(b). Furthermore, INTERPROC, which fails to verify the assertion for Fig. 1(a), easily discovers the loop invariants needed to prove the assertion for Fig. 1(b).

As this example illustrates, our approach effectively reduces the problem of inferring disjunctive invariants for a complex, multi-phase loop to the better understood problem of inferring conjunctive invariants for a sequence of single-phase loops. This strategy explicitly separates the task of identifying phase transitions from the inference of loop invariants, and allows standard invariant generation techniques to be successful for programs which previously might only be verified using much more sophisticated methods. Our technique is conceptually

[1] This loop alternates its behavior from iteration to iteration.

simple, easy to implement, and improves the quality of invariants discovered by a large class of invariant generation techniques.

1.1 An Overview of the Technique

Consider a loop $\texttt{while}(P)\{E[C]\}$ where E is an expression with one *hole* $[\cdot]$ for the predicate of an `if` statement, and C is the predicate plugged into the hole. For example, in Fig. 1(a), $P = \texttt{x} < \texttt{100}$, $E = \texttt{x++; if(}[\cdot]\texttt{) y++;}$, and $C = \texttt{x} > \texttt{50}$. We are interested in finding a predicate Q with two properties:

(a) Q should be a *splitter predicate*, which informally means that it can be used to divide the loop into two loops that execute one after the other.
(b) When Q (resp. $\neg Q$) is true on entry into the loop body, a particular conditional test C in the loop should be guaranteed to be *true* (resp. *false*).

If Q has both of these properties, which we formalize in Sect. 3, then the following semantic equivalence holds:

$$\texttt{while}(P)\{E[C]\} \equiv \texttt{while}(P \wedge \neg Q)\{E[\mathit{false}]\}; \texttt{while}(P \wedge Q)\{E[\mathit{true}]\} \ . \quad (2)$$

If we can find such a splitter predicate Q, then we can decompose the original loop into two loops, one in which the conditional's predicate is always *false* and the other in which it is always *true*. Constant folding then eliminates the conditionals, resulting in simpler loops.

Note that a predicate satisfying conditions (a) and (b) above identifies the loop iteration in which a phase transition occurs for C: When Q becomes *true*, the first of the split loops terminates, and this corresponds to the first iteration in which C evaluates to *true* in the original loop. Furthermore, observe that the the splitter predicate is, in general, different from the conditional test C. For example, $Q = \texttt{x} > \texttt{49}$ satisfies (2) for the program of Fig. 1(a).

It is straightforward to generalize (2) to transform loops with `if` conditions having any fixed number k of phase transitions into the composition of k loops. In this paper, we discuss only the case where the predicate of an `if` statement has at most one phase transition; besides being simpler to present, this case is also the only one we have thus far encountered in practice.

This paper makes the following contributions:

- We present a static analysis technique to decompose multi-phase loops requiring disjunctive invariants into a sequence of simpler single-phase loops, whose invariants can be inferred using standard techniques.
- We define phase splitter predicates, which are key to identifying phase transitions of multi-phase loops, and we present an algorithm for computing them.
- The proposed technique is simple to implement and relies only on SMT solvers already used in many verification systems.
- Our evaluation on a combination of representative examples from the literature and loops taken from real programs shows that our technique allows

standard conjunctive invariant generation techniques to produce results comparable to some of the recently proposed advanced techniques for disjunctive invariant generation.

The rest of the paper is organized as follows: Section 2 presents a simple language for the formal development; Section 3 defines *splitter* and *phase splitter* predicates. Section 4 gives an algorithm for computing phase splitter predicates. Section 5 describes our prototype implementation, and Section 6 describes our experimental results. Section 7 surveys related work, and Section 8 concludes.

2 Language

Figure 2 gives the syntax of a simple imperative language we use for the formal development. We assume a family of integer-valued program variables x, y, z, ..., a set of primitive relational operators (RelOp), and binary arithmetic operators (BinOp). We distinguish between a normal statement s and a *statement with one hole* h; the unique hole $[\cdot]$ in h indicates where a predicate can be inserted to complete the statement. If h is a statement with one hole and C is a predicate, then $h[C]$ is the statement (with no holes) formed by replacing the $[\cdot]$ in h by C. We omit the formal definition of the replacement operation, which is standard.

$$\begin{array}{rcl}
s \in \mathit{Statement} & ::= & \texttt{skip} \mid x := e \mid \texttt{assert}(P) \\
 & & \mid s; s \\
 & & \mid \texttt{while}(P)\{s\} \\
 & & \mid \texttt{if}(P)\ \{s\}\ \texttt{else}\ \{s\} \\
 & & \\
P \in \mathit{Predicate} & ::= & \mathit{true} \mid \mathit{false} \mid e\ \mathit{RelOp}\ e \mid \neg P \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \\
 & & \\
e \in \mathit{Iexpr} & ::= & \mathit{int} \mid x \mid x\ \mathit{BinOp}\ y \\
 & & \\
h \in \mathit{StatementWithOneHole} & ::= & \mid h; s \mid s; h \\
 & & \mid \texttt{if}([\cdot])\ \{s\}\ \texttt{else}\ \{s\} \\
 & & \mid \texttt{if}(P)\ \{h\}\ \texttt{else}\ \{s\} \\
 & & \mid \texttt{if}(P)\ \{s\}\ \texttt{else}\ \{h\}
\end{array}$$

Fig. 2. The syntax of the language we use for the formal development

Figure 3 gives the small-step operational semantics for the language of Fig. 2. An environment E is a function from program variables to integers. Integer values are denoted by v. In each step we take a reducible expression, execute one step of computation, and possibly update E. Each transition maps an environment, statement pair $\langle E, s\rangle$ to a new pair $\langle E', s'\rangle$. The operational rules are standard; we note only that the rules for statement sequences always reduce the first statement $\langle E, s_1; s_2\rangle \to \langle E', s_1'; s_2\rangle$ until the first statement evaluates to `skip`, at which point the rule $\langle E, \texttt{skip}; s\rangle \to \langle E, s\rangle$ is applied to transfer control to the remainder of the statement sequence.

Definition 1. (Semantic Equivalence) *Two statements s_1 and s_2 are semantically equivalent, denoted $s_1 \equiv s_2$, if*

$$\forall E_1, E_2 \quad (\langle E_1, s_1\rangle \to^* \langle E_2, \texttt{skip}\rangle) \Leftrightarrow (\langle E_1, s_2\rangle \to^* \langle E_2, \texttt{skip}\rangle)$$

$$\frac{}{\langle E, \texttt{skip}; s\rangle \rightarrow \langle E, s\rangle}$$

$$\frac{}{\langle E, x := v\rangle \rightarrow \langle E[x \mapsto v], \texttt{skip}\rangle}$$

$$\frac{}{\langle E, x := y\rangle \rightarrow \langle E[x \mapsto E(y)], \texttt{skip}\rangle}$$

$$\frac{v = E(x)\ \mathit{BinOp}\ E(y)}{\langle E, z := x\ \mathit{BinOp}\ y\rangle \rightarrow \langle E[z \mapsto v], \texttt{skip}\rangle}$$

$$\frac{E(P) = \mathit{false}}{\langle E, \texttt{assert}(P)\rangle \rightarrow \text{ABORT}}$$

$$\frac{E(P) = \mathit{true}}{\langle E, \texttt{assert}(P)\rangle \rightarrow \langle E, \texttt{skip}\rangle}$$

$$\frac{\langle E, s_1\rangle \rightarrow \langle E_1, s_1'\rangle}{\langle E, s_1; s_2\rangle \rightarrow \langle E_1, s_1'; s_2\rangle}$$

$$\frac{E(P) = \mathit{true}}{\langle E, \texttt{if}(P)\ \{s_t\}\ \texttt{else}\ \{s_f\}\rangle \rightarrow \langle E, s_t\rangle}$$

$$\frac{E(P) = \mathit{false}}{\langle E, \texttt{if}(P)\ \{s_t\}\ \texttt{else}\ \{s_f\}\rangle \rightarrow \langle E, s_f\rangle}$$

$$\frac{E(P) = \mathit{true}}{\langle E, \texttt{while}(P)\{s\}\rangle \rightarrow \langle E, s; \texttt{while}(P)\{s\}\rangle}$$

$$\frac{E(P) = \mathit{false}}{\langle E, \texttt{while}(P)\{s\}\rangle \rightarrow \langle E, \texttt{skip}\rangle}$$

Fig. 3. The small-step operational semantics of the language from Fig. 2

3 Splitter Predicates

As discussed in Sect. 1, a key idea that allows our technique to identify phase transitions in loops is the concept of *splitter predicates*, which we define next.

Definition 2. (Splitter Predicate) A predicate Q is a *splitter predicate* for a loop $\texttt{while}(P)\{B\}$ if

$$\texttt{while}(P)\{B\} \equiv \texttt{while}(P \wedge \neg Q)\{B\}; \texttt{while}(P \wedge Q)\{B\}$$

According to this definition, a predicate Q is a splitter predicate for a loop L if L can be decomposed as the sequence of two loops $L_1; L_2$ where $\neg Q$ always holds at the head of L_1 and Q always holds at the head of L_2. Thus, if Q is a splitter predicate, then Q's observed truth value changes at most once during the execution of L.

The following theorem describes how to verify whether a given predicate Q is a splitter predicate for a loop L by issuing a single query to a constraint solver:

Theorem 1. For a loop $L = \texttt{while}(P)\{B\}$, if Q satisfies the Hoare triple

$$\{P \wedge Q\}\ B\ \{Q \vee \neg P\}$$

then Q is a splitter predicate for L.

Proof. We claim the following:

$$\begin{aligned} &\texttt{while}(P)\{B\} && (a) \\ \equiv\ &\texttt{while}(P)\{\texttt{while}(P \wedge \neg Q)\{B\}; \texttt{while}(P \wedge Q)\{B\}\} && (b) \\ \equiv\ &\texttt{while}(P \wedge \neg Q)\{B\}; \texttt{while}(P \wedge Q)\{B\} && (c) \end{aligned}$$

For any predicate Q (whether Q is a splitter predicate or not), it is easily verified that loop (a) is equivalent to loop (b). Intuitively, loop (b) expresses that the truth-value of Q may in general change any number of times in the original loop. For the second step, if the outer loop executes its body either 0 or 1 times (i.e., Q's truth value changes at most once) then it is easy to check that (b) is equivalent to (c). Thus, it suffices to prove that the outer loop of (b) executes either 0 or 1 times if $\{P \wedge Q\}\ B\ \{Q \vee \neg P\}$. There are two cases:

- If $\neg P$ holds on entry to the outer loop, the outer loop executes its body 0 times.
- If P holds on entry to the outer loop, the outer loop's body is executed at least once. Clearly $\neg P \vee Q$ is a post-condition of the first inner loop. There are two cases on entry to the second inner loop:
  - If $\neg P$ holds, the second inner loop terminates without executing its body and then the outer loop terminates after one iteration.
  - Otherwise P holds, and therefore from the post-condition of the first inner loop we know Q also holds on entry to the second inner loop. Applying the assumption $\{P \wedge Q\}\ B\ \{Q \vee \neg P\}$, we conclude that Q is an invariant of the second inner loop whenever the second inner loop executes at least once. Since Q cannot become *false*, the second inner loop terminates only when $\neg P$ holds, and therefore the outer loop exits after completing one iteration.

Since the outer loop executes its body 0 or 1 times in all cases, (b) $\equiv$ (c), and therefore (a) $\equiv$ (c). Therefore, Q is a splitter predicate. □

Observe that not every splitter predicate is useful for the purpose of decomposing a loop into phases, because not every splitter identifies the phase transition associated with a conditional in the loop body. For example, in Fig. 1(a), $x > 60$ is a splitter predicate, as $x > 60$ is initially false, but stays true once it becomes true. On the other hand, $x > 60$ is not a useful splitter because it does not exactly split the loop into the two phases of the conditional test $x > 50$. We require a class of splitter predicates that satisfy a stronger condition:

Definition 3. (Phase Splitter Predicate) A splitter predicate Q that satisfies the additional requirement

$$\texttt{while}(P)\{E[C]\} \equiv \texttt{while}(P \wedge \neg Q)\{E[\mathit{false}]\}; \texttt{while}(P \wedge Q)\{E[\mathit{true}]\}$$

is called a *phase splitter predicate.*

According to this definition, a phase splitter predicate Q for a loop L decomposes the loop into two loops $L_1; L_2$, where both L_1 and L_2 have fewer branches in the loop body. Since a phase splitter predicate eliminates a conditional C in the original loop body, there is a relationship between the phase splitter for loop L and the conditional C. We now make this relationship precise.

Definition 4. Consider a loop $\texttt{while}(P)\{B[C]\}$. We define $\overline{B}$, the code that executes before the hole in B, by structural induction on B.

$$\begin{aligned}
\overline{\texttt{if}([\cdot])\,\{s\}\ \texttt{else}\ \{s\}} &= \texttt{skip} \\
\overline{\texttt{if}(P)\,\{h\}\ \texttt{else}\ \{s\}} &= \texttt{assert}(P);\overline{h} \\
\overline{\texttt{if}(P)\,\{s\}\ \texttt{else}\ \{h\}} &= \texttt{assert}(\neg P);\overline{h} \\
\overline{h;s} &= \overline{h} \\
\overline{s;h} &= s;\overline{h}
\end{aligned}$$

Note that if we allow holes inside `if` statements of nested loops, then the notion of code that executes before the hole is no longer straightforward. This is the primary reason for disallowing holes inside nested loops in the definition of *StatementWithOneHole* in Fig. 2.

Recall that our goal is to find a splitter predicate Q such that (i) if Q holds at the loop head, then the conditional C inside one `if` statement always evaluates to *true*, and (ii) if $\neg Q$ holds at the loop head, then the conditional C inside the same `if` statement always evaluates to *false*. The following lemma states the relationship between a predicate Q at the loop head and the conditional C in an `if` statement:

Lemma 1. Let Q be any predicate. Then,

$$\begin{aligned}
&\text{If } \{Q\}\,\overline{B}\,\{C\}, \text{ then } \texttt{while}(P \wedge Q)\{B[C]\} \equiv \texttt{while}(P \wedge Q)\{B[\mathit{true}]\} \\
&\text{If } \{Q\}\,\overline{B}\,\{\neg C\}, \text{ then } \texttt{while}(P \wedge Q)\{B[C]\} \equiv \texttt{while}(P \wedge Q)\{B[\mathit{false}]\}
\end{aligned}$$

Proof. The proof of this lemma is given in the Appendix. □

Just as Thm. 1 showed that we could use constraint solving techniques to determine whether Q is a splitter predicate, Lemma 1 shows that solving another constraint problem determines whether Q causes a conditional to have only one phase within the loop. Because we split the original loop into two loops, we must solve two constraint problems, one for each of the split loops, to ensure that the conditional in both loops can be eliminated. This leads us to the following theorem, which reduces the problem of checking phase splitter predicates to a constraint solving problem:

Theorem 2. Consider a loop $L = \texttt{while}(P)\{B[C]\}$ and let Q be a predicate such that

$$\{Q\} \quad \overline{B} \quad \{C\} \tag{3}$$

$$\{\neg Q\} \quad \overline{B} \quad \{\neg C\} \tag{4}$$

$$\{P \wedge Q\}\ B[C]\ \{Q \vee \neg P\} \tag{5}$$

Then Q is a phase splitter predicate.

Proof. By (5) and Thm. 1, Q is a splitter predicate for L. Hence $L \equiv L_1; L_2$ where $L_1 = \texttt{while}(P \wedge \neg Q)\{B[C]\}$ and $L_2 = \texttt{while}(P \wedge Q)\{B[C]\}$. By (4) and Lemma 1, $L_1 \equiv \texttt{while}(P \wedge \neg Q)\{B[\mathit{false}]\}$. By (3) and Lemma 1, $L_2 \equiv \texttt{while}(P \wedge Q)\{B[\mathit{true}]\}$. Hence Q is a phase splitter predicate. □

4 Algorithm for Splitting

In Thm. 2 of the previous section, we showed how to check whether a predicate is a phase splitter, but we have not yet answered the question of how to find candidate splitter predicates. In this section, we discuss an algorithm for finding candidate phase splitter predicates and transforming a multi-phase loop into a sequence of simpler loops.

```
phase_split($L$)
 1:   foreach conditional test $C$ in $L = \texttt{while}(P)\{B[C]\}$ do
 2:     $Q = WP(\overline{B}, C)$
 3:     if $(\{\neg Q\}\ \overline{B}\ \{\neg C\}) \wedge (\{P \wedge Q\}\ B[C]\ \{Q \vee \neg P\})$ then
 4:       $L_1 = \texttt{while}(P \wedge \neg Q)\{B[\mathit{false}]\}$
 5:       $L_2 = \texttt{while}(P \wedge Q)\{B[\mathit{true}]\}$
 6:       $B_1 = \texttt{phase\_split}(L_1)$
 7:       $B_2 = \texttt{phase\_split}(L_2)$
 8:       return $B_1; B_2$
 9:     endif
10:  done
11:  return $L$
```

Fig. 4. Phase splitting algorithm

Our algorithm considers loops in an inside-out fashion, starting with the innermost nested loops first. The pseudo-code for splitting a single loop is given in Fig. 4; in the figure, *WP* denotes a standard precondition computation. This precondition should be as weak as possible, and ideally it is the weakest precondition (hence *WP*) although in practice we must settle for a decidable approximation. Here, we repeatedly consider each `if` statement in the body of the outermost loop and attempt to find a splitter predicate for the `if`'s conditional test. Given the conditional C of some `if` statement, we first compute the precondition Q of C with respect to $\overline{B}$. Since Q is a precondition, it must satisfy condition (3) of Thm. 2. In our implementation we use a constraint solver to compute a precondition that is as weak as possible (see Sect. 5). We then explicitly check the other two conditions (4) and (5) of Thm. 2 to guarantee that we do not split the loop unless Q is a phase splitter predicate. If Q is indeed a predicate identifying phase transitions, then L is split into two loops L_1 and L_2 in lines 5 and 6 of Fig. 4. Since it may be possible to further decompose L_1 or L_2 into even simpler loops with fewer phases, we recursively invoke the `phase_split` algorithm, which transforms L_1 (resp. L_2) into a sequence of loops B_1 (resp. B_2). The final result of splitting L is then given by the sequence $B_1; B_2$.

Observe that the algorithm in Fig. 4 considers conditionals in the loop body in an arbitrary order. The reader might wonder whether the order in which conditionals are considered matters, as one order might yield a better decomposition than another. Fortunately, it turns out that the order in which we test the conditionals in the algorithm is irrelevant, because if C is a splitter predicate of the original loop, then it is guaranteed to remain a splitter predicate of the transformed loop. The following theorem makes this statement precise:

Theorem 3. If Q is a splitter predicate of $\texttt{while}(P)\{B\}$ satisfying the hypothesis of Thm. 1 and $P' \Rightarrow P$, then Q is a splitter predicate of $\texttt{while}(P')\{B\}$.

Proof. We show $\{P' \wedge Q\}\, B\, \{\neg P' \vee Q\}$. It then follows from Thm. 1 that Q is a splitter predicate for $\texttt{while}(P')\{B\}$. We reason as follows:

$$\begin{array}{l}(P' \Rightarrow P) \Rightarrow (Q \vee \neg P \Rightarrow Q \vee \neg P')\\ (P' \Rightarrow P) \Rightarrow (P' \wedge Q \Rightarrow P \wedge Q)\\ \ \{P \wedge Q\}\ B\ \{Q \vee \neg P\}\end{array}$$

Using Hoare's consequence rule we obtain $\{P' \wedge Q\}\, B\, \{Q \vee \neg P'\}$. Hence Q is a splitter predicate for $\texttt{while}(P')\{B\}$. □

Because the loop predicates in split loops are only stronger than the loop predicate of the original loop, Theorem 3 shows that if we have two splitters Q_1 and Q_2, then if we split on Q_1, Q_2 remains a splitter predicate for each of the new loops. Furthermore, it is easy to see from Thm. 2, that Q_2 still causes the same conditional(s) to be eliminated whether we split on Q_1 first or not. Thus, choosing one phase splitter predicate Q over another phase splitter predicate Q' cannot make further splitting by Q' illegal or vice versa.

As mentioned above, loops are split beginning with innermost loops and proceeding to outermost loops. The reason for selecting this order is that the weakest precondition of a code fragment containing a loop requires computing loop invariants. Thus, when splitting an outer loop, the weakest precondition computation may be required to compute loop invariants for any inner loops. By splitting the innermost loops first, the weakest precondition computation deals only with loops that have already been simplified as much as possible, making it easier to infer better invariants using standard techniques.

4.1 Revisiting the Running Example

We now illustrate the execution of our algorithm on the example from Fig. 1(a).

1. For this loop, we have:

$$\begin{array}{l}P = \texttt{x} \leq \texttt{99}\\ B = \texttt{x++;if(}[\cdot]\texttt{)y++}\\ C = \texttt{x} > \texttt{50}\\ \overline{B} = \texttt{x++}\end{array}$$

2. A candidate phase splitter predicate Q is computed as $Q = \mathit{WP}(\texttt{x} > \texttt{50}, \texttt{x++})$. Using a weakest precondition computation engine, we obtain $Q = x > 49$ as a candidate phase splitter predicate.
3. Now, we check whether the candidate predicate Q satisfies condition (4) of Thm. 2 by querying the validity of the formula $(\neg x > 49 \wedge x' = x + 1) \Rightarrow \neg x' > 50$, which is indeed valid.
4. Finally, we verify that candidate Q is a phase splitter predicate by checking the validity of the following constraint:

$$(x > 49 \wedge x \leq 99 \wedge x' = x + 1 \wedge (x' > 50 \Rightarrow y' = y + 1)) \Rightarrow x' > 49 \vee \neg x' \leq 99$$

   This formula is valid, allowing us to perform the phase splitting transformation, which yields the decomposition shown in Fig. 1(b).

5 Implementation

We have implemented a prototype version of the algorithm described in this paper using the SAIL program analysis front-end [14] and the Mistral SMT solver [15,16] for a subset of the C programming language. The weakest precondition computation step in the algorithm is implemented by using the quantifier elimination capabilities of Mistral. More specifically, to compute the weakest precondition of C with respect to code fragment $\overline{B}$, we first convert $\overline{B}$ to single static assignment (SSA) form [17]. We then generate a constraint ϕ_s for any statement s in the following way: For each basic statement s (e.g., an assignment or assertion), we generate the corresponding atomic constraint in the constraint language, and a sequence $s_1; s_2$ is converted to the constraint $\phi_{s_1} \wedge \phi_{s_2}$ where ϕ_{s_1} and ϕ_{s_2} are the constraints derived from statements s_1 and s_2 respectively. For an `if` statement, `if(C) then s1 else s2`, we generate the constraint $(C \wedge \phi_{s_1}) \vee (\neg C \wedge \phi_{s_2})$.[2] To generate weakest preconditions for nested loops, we use a constraint obtained with the help of an invariant generation tool. Finally, we compute the weakest precondition of C with respect to $\overline{B}$ by computing the constraint $\phi_{\overline{B}}$ and then by existentially quantifying and eliminating all intermediate variables (i.e., variables with version number greater than one in SSA form).

6 Experiments

We evaluate our technique by comparing the quality of the loop invariants obtained from two publicly available invariant generation tools, Interproc [19] and InvGen [6], before and after decomposing multi-phase loops into a sequence of single-phase loops. Interproc is an abstract interpretation-based tool that implements the interval, octagon, and polyhedra abstract domains using the Apron [9] and Fixpoint[20] libraries. In contrast to Interproc, InvGen is a template-based invariant generator (see Sect. 7), which employs non-linear constraint solving to find valid instantiations for the unknown parameters of user-specified template invariants. Table 1 summarizes the results of our experiments on a set of challenging benchmarks, consisting of representative examples from the literature. All of our experimental benchmarks are available from `http://www.stanford.edu/~isil/ invariant-benchmarks.txt`. For generating invariants on each of these benchmarks, we used the polyhedra abstract domain of Interproc and the default templates provided by InvGen.

We now briefly describe the benchmark programs from Table 1. All of the benchmarks contain one or more assertions. The benchmark `popl07` is the program in Fig. 1 and the motivating example of [4]; `cav06` and `tacas08` are the motivating examples from [21] and [22] respectively. The next four benchmarks are programs from the test suite of InvGen. The program `spam` also occurs as

[2] For soundness it is important that the negation here result in an overapproximation; for example, bracketing constraints [18] can be used.

Table 1. Comparison of the invariants generated by Interproc and InvGen on some benchmark programs before and after applying our technique.

File	LOC	Interproc					InvGen				
		Before split		After split		Q	Before split		After split		Q
		time(s)	Proof?	time(s)	Proof?		time(s)	Proof?	time(s)	Proof?	
popl07	13	0.014	*N*	0.014	*Y*	+	0.425	*N*	0.215	*Y*	+
cav06	22	0.020	*N*	0.030	*Y*	+	0.318	*N*	0.28	*Y*	+
tacas08	30	0.018	*N*	0.021	*Y*	+	0.344	*Y*	0.298	*Y*	=
svd*	48	0.016	*Y*	0.014	*Y*	+	0.784	*Y*	0.794	*Y*	+
heapsort*	45	0.022	*Y*	0.036	*Y*	+	0.976	*Y*	1.55	*Y*	+
mergesort*	73	0.048	*N*	0.09	*N*	||	4.813	*Y*	12.138	*Y*	+
spam*	55	0.024	*Y*	0.029	*Y*	+	0.0521	*Y*	0.0397	*Y*	+
ex1	23	0.090	*N*	0.027	*Y*	+	416.985	*N*	0.621	*Y*	+
ex2	21	0.011	*N*	0.011	*Y*	+	123.945	*N*	0.553	*Y*	+
svd1	49	0.016	*N*	0.014	*Y*	+	0.456	*N*	0.784	*Y*	+
heapsort1	46	0.022	*N*	0.036	*Y*	+	2.291	*Y*	1.278	*Y*	+
mergesort1	74	0.048	*N*	0.090	*Y*	||	4.924	*Y*	11.431	*Y*	+
spam1	56	0.024	*N*	0.029	*Y*	+	0.46	*Y*	0.759	*Y*	+

`SpamAssassin-loop` in [23]. The next benchmark, `ex1`, is an interesting variation of `cav06`, and `ex2` is an example illustrating that splitting can be carried out in any order to obtain equivalent results in the presence of multiple splitter predicates. The programs `svd1`, `heapsort1`, and `spam1` have code similar to `svd`, `heapsort`, and `spam` but require stronger assertions to be proved; similarly, `mergesort1` differs only in having weaker assertions than `mergesort`.

The entries in the table marked *Y* indicate that a given tool was able to prove the assertions correct for the benchmark and *N* indicates that the tool could not prove at least one assertion. The column labeled "Before Split" shows whether a given tool was able to prove the assertions without using our technique, and the column labeled "After Split" describes whether the same tool could prove the same assertions on the loops decomposed by our technique. The column labeled "Q" compares the quality of the invariants obtained before and after using our technique. An entry labeled + in this column means that the tool generates better, i.e., logically stronger, invariants on the loops transformed by our technique, a || indicates that the invariants are incomparable (there are several invariants and some are stronger and some are weaker), and an = indicates that the inferred invariants were logically equivalent. Benchmarks marked with * indicate that the original benchmark code used a feature which is not part of the input language of Interproc; we manually modified these benchmarks before using them as input to Interproc. The time taken for computing phase splitter predicates was negligible; our algorithm took no longer than 90 milliseconds on any benchmark from Table 1.

The results shown in Table 1 demonstrate that our technique substantially improves the quality of the invariants generated by both Interproc and InvGen and allows them to verify assertions they could not verify previously.

Consider only the first 9 programs (those above the double line); we discuss the 4 variations separately below. The invariants discovered by INTERPROC improve (i.e., are logically strengthened) in 8 out the 9 benchmarks. On `mergesort` the invariants discovered by INTERPROC are incomparable before and after splitting due to the non-monotonicity of the widening operator [24,1]. On the same set of 9 benchmarks, INVGEN discovers logically stronger invariants on 8 benchmarks after splitting, and obtains a logically equivalent invariant on one benchmark.

Table 1 shows not only that there is an improvement in the quality of invariants after splitting, but also that INTERPROC and INVGEN can prove many assertions after splitting that they could not previously verify. More specifically, in 6 of the first 9 benchmarks, INTERPROC fails to prove at least one assertion in the original program, but can verify all assertions in these programs after splitting. Similarly, INVGEN cannot verify 4 of the 9 original benchmarks, but it can prove all the assertions in these programs after our transformation. Recall that five of the benchmarks (`tacas08` through `spam`) are included in INVGEN's test suite; thus, it is not surprising that INVGEN can verify the assertions in programs on which it was developed. We note that INVGEN was unable to verify the four programs that were not taken from its test suite, but was able to verify all four of them after performing our transformation.

Observe that there are some benchmarks in Table 1 where the tools are able to prove the assertions both before and after splitting, but yield strictly stronger invariants after splitting. To demonstrate that this extra precision is useful, we created variants `svd1`, `heapsort1`, and `spam1` of `svd`, `heapsort`, and `spam` with stronger assertions (shown below the double line). Notice that INTERPROC can verify these three benchmarks after splitting, but is unable to do so before.

Also observe that for `mergesort1` INTERPROC is unable to prove the assertions both before and after splitting, despite yielding new facts after splitting. We demonstrate that these new invariants are again useful by designing `mergesort1` with weaker assertions. As shown below the double line in Table 1, INTERPROC can take advantage of these new facts obtained through splitting.

7 Related Work

Techniques for Multi-Phase Loops There is an existing body of work whose goal is to improve the quality of invariants for multi-phase loops [25,20,21,8,7]. For example, Mauborgne and Rival address this problem in [25]. In contrast to our technique, their method is not fully automatic; it critically depends on user input for partitioning directives. The techniques described in [20,21] also attempt to improve the quality of invariants for multi-phase loops by guiding a static analysis to compute a fix-point on one phase of the loop before considering the next phase. Since our algorithm for identifying phases is independent of the particular abstract domain used for inferring invariants, our approach can recover precision irrespective of the abstract domain used for invariant generation.

Two recent works [7,8] take a similar approach to the one we present: splitting a loop to produce multiple loops with simpler invariants. These approaches first enumerate all the paths through a loop body and then they either search for all the possible sequences in which these paths can execute [7] or they eliminate the infeasible path sequences [8]. Since the number of paths is, in general, best case exponential in the number of conditionals in the loop body, both techniques have to rely on heuristics to keep the number of paths under consideration tractable. In contrast to both [7] and [8], our approach is less eager: we delay the worst-case exponential search to an SMT solver; if the solver can prove the properties of splitter predicates without reasoning about all the paths, we take advantage of that fact. Our approach is also much simpler and easier to implement, does not use heuristics, and yields some new insight into the nature of the problem (e.g., the independence of splitter predicates).

Direct Techniques for Inferring Disjunctive Invariants Many different approaches have been developed for directly inferring disjunctive invariants, and some of these techniques are capable of discovering precise invariants for some of our benchmarks without decomposing the loop into phases. These approaches include (i) template-based techniques, such as [26,6,27], (ii) techniques based on predicate abstraction such as [11,12,13,22,5], and (iii) techniques based on probabilistic inference [4]. While some of these approaches are, in principle, capable of discovering precise invariants in loops exhibiting multiple phases, they are significantly more complicated, less efficient, and less widely-used than standard abstract interpretation-based techniques for generating conjunctive numeric invariants such as [1,2,28]. We discuss each of these three classes in more detail below.

Template-Based Techniques Given an input template (i.e., parametrized form of invariant) provided by the user, template-based techniques find values for the parameters such that these instantiated templates correspond to inductive invariants [26,6,27]. While these techniques can, in principle, find precise invariants for multi-phase loops if the user provides appropriate disjunctive templates, they suffer from two drawbacks: First, they are not fully automatic since they require the user to specify the shape of the desired invariant. Second, since many template-based techniques require solving non-linear constraints, their applicability is limited by the lack of efficient algorithms for solving such constraints.

Predicate Abstraction Techniques Techniques based on the basic form of counterexample guided abstraction refinement such as [11,12], are, in principle, capable of inferring disjunctive invariants. However, these techniques often diverge or take a very large number of refinement steps. More sophisticated invariant generation techniques based on predicate abstraction are considered in [13,22,5]. The basic idea underlying [13] is to use interpolants to generate counterexamples. To guarantee convergence, this technique restricts the language of the interpolants to some finite language L, and can therefore only find invariants in this restricted language. Hence, a poor choice of language degrades its performance. The technique described in [22] uses counterexample guided abstraction refinement to tune widening strategies in an abstract interpretation framework. While this technique can sometimes be helpful for generating more precise invariants for multi-phase loops, it is difficult to characterize the class of loops for which this technique will generate useful invariants. The technique presented in [5] combines counterexample-guided abstraction refinement with template-based invariant generation techniques. More specifically, the counterexamples produced in this technique are not finite program paths, but full-fledged programs called *path programs*. The algorithm in [5] then employs template-based techniques to infer invariants of the path program, which are used to refine the analysis. While this technique is capable of finding disjunctive invariants, it is not directly helpful for multi-phase loops.

Probabilistic Techniques Gulwani and Jojic formulate the problem of invariant generation as probabilistic inference, and use machine learning techniques to infer invariants [4]. Their technique is capable of inferring the disjunctive invariant from Fig. 1. However, this technique is not guaranteed to converge, and it is difficult to characterize the class of loops for which it succeeds. Furthermore, this approach is significantly more involved than our algorithm for splitting loops with multiple phases.

8 Conclusion

We have proposed a static analysis technique to identify phase transitions in loops and decompose multi-phase loops into a sequence of simpler loops with fewer phases. We have demonstrated that standard invariant generation tools benefit substantially from the technique proposed in this paper, raising their level of precision to that of recently proposed methods for disjunctive invariant generation. Our technique is conceptually simple, easy to implement, and can be integrated into any invariant generation technique.

Acknowledgments

We would like to thank Denis Gopan, Francesco Logozzo, and Sumit Gulwani for their helpful pointers to benchmarks and invariant generation tools. We also thank Bertrand Jeannet for his guidance with the usage of INTERPROC and Ashutosh Gupta for his help with INVGEN. Finally we would like to thank Yannick Moy for pointing out an error in the definition of $\overline{B}$ in an earlier version of this paper.

References

1. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. (1978) 84–96
2. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1) (2006) 31–100
3. Karr, M.: Affine relationships among variables of a program. Acta Inf. **6** (1976) 133–151
4. Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: POPL. (2007) 277–289

5. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI. (2007) 300–309
6. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: CAV. (2009) 634–640
7. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI. (2009) 375–385
8. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: EMSOFT. (2009) 49–58
9. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV. (2009) 661–667
10. Gulavani, B.S., Rajamani, S.K.: Counterexample driven refinement for abstract interpretation. In: TACAS. (2006) 474–488
11. Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. In: POPL. (2002) 1–3
12. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. STTT **9**(5-6) (2007) 505–525
13. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS. (2006) 459–473
14. Dillig, I., Dillig, T., Aiken, A.: SAIL: Static Analysis Intermediate Language with a Two-Level Representation. Stanford University Technical Report (2009)
15. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: CAV. (2009) 233–247
16. Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: On-line constraint simplification in scalable static analysis. In: SAS. (2010) 236–252
17. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS **13**(4) (1991) 451–490
18. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: ESOP. (2010) 246–266
19. Lalire, G., Argoud, M., Jeannet, B.: The Interproc Analyzer. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html
20. Gopan, D., Reps, T.W.: Guided static analysis. In: SAS. (2007) 349–365
21. Gopan, D., Reps, T.W.: Lookahead widening. In: CAV. (2006) 452–466
22. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: TACAS. (2008) 443–458
23. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE. (2007) 389–392
24. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
25. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP. (2005) 5–20
26. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: CAV. (2003) 420–432
27. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: VMCAI. (2007) 378–394
28. Laviron, V., Logozzo, F.: Subpolyhedra: A (more) scalable approach to infer linear inequalities. In: VMCAI. (2009) 229–244

Appendix: Proof of Lemma 1

The proof of this lemma uses the following helper lemma which we prove first:

Lemma 2. *If* $\{Q\}\,\overline{h}\,\{C\}$*, then*

$$\texttt{assert}(Q); h[C] \equiv \texttt{assert}(Q); h[\mathit{true}]$$

Proof. The proof is by structural induction on the derivation of h.

Case 1 (Base Case): $h = \texttt{if}([\cdot])\{s_1\}\ \texttt{else}\ \{s_2\};\ \overline{h} = \texttt{skip}$
Given $\{Q\}\ \texttt{skip}\ \{C\}$, we need to show

$$\texttt{assert}(Q); \texttt{if}(C)\{s_1\}\ \texttt{else}\ \{s_2\} \equiv \texttt{assert}(Q); s_1$$

Since $\{Q\}\ \texttt{skip}\ \{C\}$, $Q \Rightarrow C$. Thus, if Q is *true*, $\texttt{if}(C)\{s_1\}\ \texttt{else}\ \{s_2\} \equiv s_1$; otherwise, both statements abort.

Case 2: $h = \texttt{if}(P)\{h_1\}\ \texttt{else}\ \{s\};\ \overline{h} = \texttt{assert}(P); \overline{h_1}$
Given $\{Q\}\ \texttt{assert}(P); \overline{h_1}\ \{C\}$, we need to show:

$$\texttt{assert}(Q); \texttt{if}(P)\{h_1[C]\}\ \texttt{else}\ \{s\} \equiv \texttt{assert}(Q); \texttt{if}(P)\{h_1[\mathit{true}]\}\ \texttt{else}\ \{s\}$$

First, $\{Q\}\,\texttt{assert}(P); \overline{h_1}\,\{C\}$ implies $\{Q \wedge P\}\,\texttt{assert}(P); \overline{h_1}\,\{C\}$, which implies $\{Q \wedge P\}\,\overline{h_1}\,\{C\}$. By the inductive hypothesis, we have:

$$\text{If } \{Q \wedge P\}\,\overline{h_1}\,\{C\},\ \text{then } \texttt{assert}(Q \wedge P); h_1[C] \equiv \texttt{assert}(Q \wedge P); h_1[\mathit{true}]$$

Thus, using $\{Q \wedge P\}\,\overline{h_1}\,\{C\}$, we have

$$\texttt{assert}(Q \wedge P); h_1[C] \equiv \texttt{assert}(Q \wedge P); h_1[\mathit{true}] \quad (*)$$

Now, $\texttt{assert}(Q); \texttt{if}(P)\{h_1[C]\}\ \texttt{else}\ \{s\}$ can be rewritten as:

$$\texttt{assert}(Q); \texttt{if}(P)\{\texttt{assert}(Q \wedge P); h_1[C]\}\ \texttt{else}\ \{s\}$$

which, using (*), is equivalent to:

$$\begin{aligned}&\texttt{assert}(Q); \texttt{if}(P)\{\texttt{assert}(Q \wedge P); h_1[\mathit{true}]\}\ \texttt{else}\ \{s\}\\ \equiv\ &\texttt{assert}(Q); \texttt{if}(P)\{h_1[\mathit{true}]\}\ \texttt{else}\ \{s\}\end{aligned}$$

Case 3: $h = \texttt{if}(P)\{s\}\ \texttt{else}\ \{h_1\}$. This case is symmetric to Case 2.

Case 4: $h = h_1; s;\ \overline{h} = \overline{h_1}$
Given $\{Q\}\overline{h_1}\{C\}$, we need to show $\texttt{assert}(Q); h_1[C]; s \equiv \texttt{assert}(Q); h_1[\mathit{true}]; s$. This follows immediately from the inductive hypothesis.

Case 5: $h = s; h_1;\ \overline{h} = s; \overline{h_1}$

Given $\{Q\}s;\overline{h_1}\{C\}$, we need to show $\texttt{assert}(Q); s; h_1[C] \;\equiv\; \texttt{assert}(Q); s; h_1[\mathit{true}]$. Since $\{Q\}\ s;\overline{h_1}\ \{C\}$, there exists some φ such that $\{Q\}\ s\ \{\varphi\}$ and $\{\varphi\}\ \overline{h_1}\ \{C\}$. Thus, by the inductive hypothesis, we have

$$\texttt{assert}(\varphi); h_1[C] \equiv \texttt{assert}(\varphi); h_1[\mathit{true}]\ (**)$$

Now,

$$\begin{aligned}
&\texttt{assert}(Q)s; h_1[C] \\
\equiv\ &\texttt{assert}(Q); s; \texttt{assert}(\varphi); h_1[C]\ \ \text{(using the Hoare triple } \{Q\}\ s\ \{\varphi\}) \\
\equiv\ &\texttt{assert}(Q); s; \texttt{assert}(\varphi); h_1[\mathit{true}]\ \text{(using}(**)) \\
\equiv\ &\texttt{assert}(Q); s; h_1[\mathit{true}]\ \ \text{(using the Hoare triple } \{Q\}\ s\ \{\varphi\})
\end{aligned}$$

□

Now using Lemma 2, we can show Lemma 1:

Proof. We first prove

$$\text{If } \{Q\}\ \overline{B}\ \{C\}\text{, then } \texttt{while}(P \wedge Q)\{B[C]\} \equiv \texttt{while}(P \wedge Q)\{B[\mathit{true}]\}$$

$$\begin{aligned}
&\texttt{while}(P \wedge Q)\{B[C]\} \\
\equiv\ &\texttt{while}(P \wedge Q)\{\texttt{assert}(Q); B[C]\} \\
\equiv\ &\texttt{while}(P \wedge Q)\{\texttt{assert}(Q); B[\mathit{true}]\}\ \ \text{(using Lemma 2 and } \{Q\}\ \overline{B}\ \{C\}) \\
\equiv\ &\texttt{while}(P \wedge Q)\{B[\mathit{true}]\}
\end{aligned}$$

We now prove the second part of the lemma:

$$\text{If } \{Q\}\ \overline{B}\ \{\neg C\}\text{, then } \texttt{while}(P \wedge Q)\{B[C]\} \equiv \texttt{while}(P \wedge Q)\{B[\mathit{false}]\}$$

Let $B'[\neg C]$ be the statement obtained by replacing

$$\texttt{if}(C)\{s_1\}\ \texttt{else}\ \{s_2\}$$

by

$$\texttt{if}(\neg C)\{s_2\}\ \texttt{else}\ \{s_1\}$$

in $B[C]$. Observe that $B'[\neg C] \equiv B[C]$.

$$\begin{aligned}
&\texttt{while}(P \wedge Q)\{B[C]\} \\
\equiv\ &\texttt{while}(P \wedge Q)\{B'[\neg C]\} \\
\equiv\ &\texttt{while}(P \wedge Q)\{B'[\mathit{true}]\}\ \ \text{(using } \{Q\}\ \overline{B}\ \{\neg C\} \text{ and Lemma 2)} \\
\equiv\ &\texttt{while}(P \wedge Q)\{B[\mathit{false}]\}
\end{aligned}$$

□