Formalizing Counterexample-driven Refinement with Weakest Preconditions

Thomas Ball
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
tball@microsoft.com

December 10, 2004

Technical Report
MSR-TR-2004-134

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Formalizing Counterexample-driven Refinement with Weakest Preconditions

Thomas Ball
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
tball@microsoft.com

December 10, 2004

Abstract

To check a safety property of a program, it is sufficient to check the property on an abstraction that has more behaviors than the original program. If the safety property holds of the abstraction then it also holds of the original program.

However, if the property does not hold of the abstraction along some trace t (a counterexample), it may or may not hold of the original program on trace t. If it can be proved that the property does not hold in the original program on trace t then it makes sense to refine the abstraction to eliminate the "spurious counterexample" t (rather than a report a known false negative to the user).

The SLAM tool developed at Microsoft Research implements such an automated abstraction-refinement process. In this paper, we reformulate this process for a tiny **while** language using the concepts of weakest preconditions, bounded model checking and Craig interpolants. This representation of SLAM simplifies and distills the concepts of counterexample-driven refinement in a form that should be suitable for teaching the process in a few lectures of a graduate-level course.

1 Introduction

A classic problem in computer science is to automate, as fully as possible, the process of proving that a software program correctly implements a specification of its expected behavior.

Over thirty years ago, Tony Hoare and Edsger Dijkstra laid the foundation for logically reasoning about programs [Hoa69, Dij76]. A key idea of their approach (let us call it "programming with invariants") is to break the potentially complicated proof of a program's correctness into a finite set of small proofs. This proof technique requires the programmer to identify loop invariants that decompose the proof of a program containing loops into a set of proofs about loop-free program fragments.

Thirty years later, we do not find many programmers programming with invariants. We identify three main reasons for this:

- *Lack of Specifications*: it is hard to develop formal specifications for correct program behavior, especially for complex programs. Specifications of full correctness can be complicated and error-ridden, just like programs.

- *Minimal Tool Support*: there has been a lack of "push-button" verification tools that provide value to programmers. As a result, programmers must climb a steep learning curve and even become proficient in the underlying mechanisms of tools (such as automated deduction) to make progress. Programmers simply have not had the incentive to create formal specifications because tools have not provided enough value to them.

- *Large Annotation Burden*: finally, although invariants are at the heart of every correct program, eliciting them from people remains difficult. Identification of loop invariants and procedure pre- and postconditions is necessary to make programming with invariants scale and it is left up to the programmer to produce such annotations.

Despite these obstacles, automated support for program proving has steadily improved [SRW99, FLL+02] and has now reached a level of maturity such that software tools capable of proof are appearing on programmers' desktops. In particular, at Microsoft Research, a tool called SLAM [BR00, BR01] that checks temporal safety properties of C programs has been developed and successfully deployed on Windows device drivers. Three factors have made a tool like SLAM possible:

- *Focus on Critical Safety Properties*: The most important idea is to change the focus of correctness from proving the functional correctness of a program to proving that the program does not violate some critical safety property. In the domain of device drivers, such properties define what it means for a driver to be a good client of the Windows operating system (in which it is hosted). The properties state nothing about what the driver actually does; they merely state that the driver does nothing bad when it interacts with the operating system. Many of these properties are relatively simple, control-dominated properties without much dependence on program data. As a result, they are simpler to state and to check than functional correctness properties.

- *Advances in Algorithms and Horsepower*: SLAM builds on and extends a variety of analysis technologies. *Model checking* [CE81, QS81] and, in particular, symbolic model checking [BCM+92, McM93] greatly advanced our ability to automatically analyze large finite-state systems. *Predicate abstraction* [GS97] enables the automated construction of a finite-state system from an infinite-state system. SLAM uses *automatic theorem provers* [NO79, DNS03], which have steadily advanced in their capabilities over the past decades, to create predicate (or Boolean) abstractions of C programs and to determine whether or not a trace in a C program is executable. *Program analysis* [CC78, SP81, KS92, RHS95, Das00] has developed efficient techniques for analyzing programs with procedures and pointers. Last but not least, the tremendous computational capabilities of computers of today make more powerful analysis methods such as SLAM possible.

- *Invariant Inference*: Given a complex program and a simple property to check of that program, SLAM can automatically find loop invariants and procedure preconditions/postconditions that are strong enough to either prove that the program satisfies the property or that point to a real error in the program. This frees the programmer from having to annotate their code with invariants. Invariants are discovered in a goal-directed manner, based on the property to be checked, as described below.

To check a safety property of a program, it is sufficient to check the property on an abstraction that has more behaviors than (or overapproximates the behaviors of) the original program. This is a basic concept of abstract interpretation [CC77]. If the safety property holds of the abstraction then it also holds of the original program. However, if the property does not hold of the abstraction on some trace t, it may or may not hold of the original program. If it does not hold on the original program on trace t, we say the analysis yields a "false negative". Too many false negatives degrades the usefulness of an analysis. In abstract interpretation, if there are too many false negatives then the analyst's task is to find a better abstraction that reduces the number of false negatives. The abstraction should also be effectively computable and result in a terminating analysis.

Rather than appealing to a human for assistance, the SLAM process uses false negatives (also called spurious counterexamples) to automatically refine the abstraction. Kurshan introduced this idea [Kur94] which was extended and applied to finite-state systems by Clarke et al. [CGJ$^+$00] and to software by Ball and Rajamani [BR00, BR01].

In SLAM, program abstractions are represented by predicate abstractions. Predicate abstraction is a parametric abstraction technique that constructs a finite-state abstraction of an infinite state system S based on a set of predicates observations of the state space of S. If there are n predicates, then a state s of the original program (S) maps to a bit-vector of length n, with each bit having a value corresponding to the value of its corresponding predicate in state s. This finite-state abstraction is amenable to symbolic model checking techniques, using data structures such as binary decision diagrams [Bry86].

Suppose that the desired property does not hold of the predicate (Boolean) abstraction. In this case, the symbolic model checker produces an error trace t that demonstrates that the error state is reachable in the Boolean abstraction. SLAM uses automated theorem proving to decide whether or not trace t is a spurious or feasible counterexample of the original program. The basic idea is to build a formula $f(t)$ from the trace such that $f(t)$ is satisfiable if and only if trace t is a (feasible) execution trace of the original program. An automatic theorem prover decides if $f(t)$ is satisfiable or unsatisfiable. If it is satisfiable then the original program does not satisfy the safety property.

Otherwise, SLAM adds new predicates to the Boolean abstraction so as to eliminate the spurious counterexample. That is, the Boolean variables introduced into the Boolean abstraction to track the state of the new predicates make the trace t unexecutable in the refined Boolean abstraction.

SLAM then repeats the steps of symbolic model checking, path feasibility testing

and predicate refinement, as needed. SLAM is a semi-algorithm because it is not guaranteed to terminate, although it can terminate with success (proving that the program satifies the safety property) or failure (proving that the program does not satisfy the safety property).

The goal of this paper is to present the counterexample-driven refinement process for software in a declarative manner: we declare what the process does rather than how it does it. Here is what we will do:

- review Hoare logic and weakest preconditions for a tiny **while** language (Section 2);
- describe the predicate abstraction of a program with respect to a set of predicates and the symbolic model checking of the resulting Boolean abstraction in a single blow using weakest preconditions (Section 3);
- describe how the concept of a length-bounded weakest precondition serves to describe counterexamples in both the original program and Boolean abstraction (Section 4);
- show how refinement predicates can be found through the use of Craig interpolants [McM03, HJMM04] (Section 5);
- extend the **while** language and the process to procedures with call-by-value parameter passing (Section 6);
- extend the **while** language and the process to procedures with call-by-reference parameter passing (Section 7).

Along the way, we suggest certain exercises for the student to try (marked with "**Exercise**" and "**End Exercise**").

The process we present is not exactly what SLAM does. First, we have unified the steps of abstraction construction and symbolic reachability into a single step, which proceeds backwards from a (undesirable) goal state to determine if an initial state of the program can be found. In SLAM, the two steps are separate and the symbolic reachability analysis is a forward analysis. Second, SLAM computes a Cartesian approximation to predicate abstraction [BPR01], while the process we describe here defines the most precise predication abstraction. Third, SLAM abstracts a C program at each assignment statement. The process we describe here abstracts over much larger code segments, which yields a more precise abstraction. Fourth, inspired by work by McMillan [McM03], we analyze a set of counterexamples at once. SLAM works with one counterexample at a time. Finally, inspired by the BLAST project [HJMM04], we use the theory of Craig interpolants [Cra57] to describe the predicate refinement step.

2 A Small Language and Its Semantics

We consider a small **while** language containing structured control-flow constructs and integer variables, with the following syntax:

$$S \quad \rightarrow \quad \textbf{skip} \mid x := e \mid S\,;S \mid \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{while}\ b\ \textbf{do}\ S$$

$$\{Q\}\ \textbf{skip}\ \{Q\} \qquad \text{(SKIP)}$$

$$\{Q[x/e]\}\ x := e\ \{Q\} \qquad \text{(ASSIGNMENT)}$$

$$\frac{P \Rightarrow P' \qquad \{P'\}\ S\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}} \qquad \text{(CONSEQUENCE)}$$

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}} \qquad \text{(SEQUENCE)}$$

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \qquad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}} \qquad \text{(CONDITIONAL)}$$

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{\neg B \wedge I\}} \qquad \text{(ITERATION)}$$

Figure 1: Hoare axioms and rules of inference for the **while** language.

We assume the usual operators over integer variables (addition, subtraction, multiplication and division). A *predicate* is either a relational comparison of integer-valued expressions or is a Boolean combination of predicates (constructed using the Boolean connectives $\wedge$, $\vee$ and $\neg$). A predicate p is said to be *atomic* if it contains no Boolean connectives (that is, it is a relational expression). Otherwise, it is said to be *compound*.

A program state is a mapping of variables to integer values. A predicate Q symbolically denotes the set of program states for which Q is true. For example, the predicate $(x < 5 \vee x > 10)$ denotes the infinite set of states for which x is less than 5 or greater than 10.

We use the familiar Floyd-Hoare triples and weakest-precondition transformers to specify the semantics of a program. The notation $\{P\}\ S\ \{Q\}$ denotes that if the predicate P is true before the execution of statement S then the predicate Q will be true if execution of statement S completes. (Note: this is a partial correctness guarantee, as it applies only to terminating computations of S).

Figure 1 presents the standard axiom of assignment and inference rules of Hoare logic for the **while** programming language. The Iteration inference rule shows the main difficulty of "programming with invariants": the invariant I appears out of nowhere. It is up to the programmer to come up with an appropriate loop invariant I.

Given a predicate Q representing a set of goal states and a statement S, the weakest (liberal) precondition transformer defines the weakest predicate P such that $\{P\}\ S\ \{Q\}$ holds. Figure 2 contains the weakest preconditions for the five statements in the example language. Of special importance is the transformer for the **while** loop, which requires the use of the greatest fixpoint operator $\nu X.\phi(X)$:

$$\nu X.\phi(X) \;=\; \bigwedge_{i=1,\cdots,\infty} \phi^i(true)$$

$$
\begin{array}{lcl}
wp(\textbf{skip}, Q) & = & Q \\
wp(x := e, Q) & = & Q[x/e] \\
wp(S_1; S_2, Q) & = & wp(S_1, wp(S_2, Q)) \\
wp(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2,\ Q) & = & (b \Rightarrow wp(S_1, Q)) \wedge (\neg b \Rightarrow wp(S_2, Q)) \\
wp(\textbf{while } b \textbf{ do } S,\ Q) & = & \nu X.((b \Rightarrow wp(S, X)) \wedge (\neg b \Rightarrow Q))
\end{array}
$$

Figure 2: Weakest preconditions

where $\phi^i(true)$ denotes the i-fold application of the function $\lambda X.\phi(X)$ to the predicate $true$. The greatest fixpoint is expressible in the language of predicates (where implication among predicates defines the lattice ordering) but is not, in general, computable (if it were, then there would be no need for programmers to provide loop invariants).

The weakest precondition transformer satisfies a number of properties:

- $wp(S, false) = false$;
- $(Q \Rightarrow R) \ \Rightarrow \ (wp(S, Q) \Rightarrow wp(S, R))$;
- $(wp(S, Q) \wedge wp(S, R)) = wp(S, Q \wedge R)$;

Exercise. Prove these three properties of the wp transformer. **End Exercise**

3 Predicate Abstraction and Symbolic Reachability

We are given a program S and a predicate Q that represents a set of "bad" final states. We want to determine if $wp(S, Q) = false$, in which case we say that program S is "safe" with respect to Q as there can be no initial state from which execution of S yields a state satisfying Q. The problem, of course, is that $wp(S, Q)$ is not computable in the presence of loops.

We will use predicate abstraction (with respect to a finite set of atomic predicates E) to define a $wp_E(S, Q)$ that is computable and is weaker than $wp(S, Q)$ (that is, $wp(S, Q) \Rightarrow wp_E(S, Q)$). Therefore, if $wp_E(S, Q) = false$ then $wp(S, Q) = false$. This gives us a sufficient (but not necessary) test for safety. We say that $wp_E(S, Q)$ is an *overapproximation* of $wp(S, Q)$.

We formalize the notion of overapproximation of a compound predicate e by a set of atomic predicates $E = \{e_1, \cdots, e_k\}$ with the *exterior cover* function $ec_E(e)$. The exterior cover function $ec_E(e)$ is the strongest predicate representable via a Boolean combination of the atomic predicates in E such that e implies $ec_E(e)$.

We now make the definition of $ec_E(e)$ precise. A *cube* over E is a conjunction $c_{i_1} \wedge \ldots \wedge c_{i_k}$, where each $c_{i_j} \in \{e_{i_j}, \neg e_{i_j}\}$ for some $e_{i_j} \in E$. The function $ec_E(e)$

is the disjunction of all clauses c over E such that $e \Rightarrow c$. There are exactly 2^k clauses over E. A naive implementation of the ec_E function enumerates all such clauses c and invokes a theorem prover to decide the validity of $e \Rightarrow c$.

The exterior cover function has the following properties:

- $ec_E(e_1 \wedge e_2) \Rightarrow ec_E(e_1) \wedge ec_E(e_2)$;
- $ec_E(e_1 \vee e_2) = ec_E(e_1) \vee ec_E(e_2)$;
- $ec_E(\neg e) \Leftarrow \neg ec_E(e)$;
- $(e_1 \Rightarrow e_2) \Rightarrow (ec_E(e_1) \Rightarrow ec_E(e_2))$

Exercise. Prove these four properties of the ec_E function. **End Exercise**

An important property of the ec_E function is that there are at most 2^{2^k} semantically distinct compound predicates that can be constructed over the set of atomic predicates E. That is, while the domain of the ec_E function is infinite, the range of the ec_E function is finite. In general, computing the most precise ec_E is undecidable. Thus, in practice, we compute an overapproximation to ec_E.

Now, $ec_E(wp(S, Q))$ is the optimal (but incomputable) covering of wp with respect to E. It is a covering because by the definition of ec_E, $wp(S, Q) \Rightarrow ec_E(wp(S, Q))$. It is optimal also because of the definition of ec_E. In order to define a computable wp_E, we need to "push" ec_E inside wp, but not too far. For example, we prefer not to abstract between statements S_1 and S_2 in a sequence because:

$$wp(S_1, wp(S_2, Q)) \Rightarrow wp(S_1, ec_E(wp(S_2, Q)))$$

Following abstract interpretation, we wish to abstract only at loops. However,

$$ec_E(wp(\textbf{while } b \textbf{ do } S,\ Q)) \ = \ ec_E(\nu X.((b \Rightarrow wp(S, X)) \wedge (\neg b \Rightarrow Q)))$$

still is incomputable because of the greatest fixpoint inside the argument to ec_E. But because

$$ec_E(e_1 \wedge e_2) \Rightarrow ec_E(e_1) \wedge ec_E(e_2)$$

it follows that

$$ec_E(\nu X.((b \Rightarrow wp(S, X)) \wedge (\neg b \Rightarrow Q))) \Rightarrow \nu X.ec_E((b \Rightarrow wp(S, X)) \wedge (\neg b \Rightarrow Q))$$

That is, by pushing the ec_E function inside the greatest fixpoint, we get a function that is weaker than the ec_E function applied to the greatest fixpoint. (In fact, it is possible to gain even more precision by only applying ec_E to $wp(S, X)$; we leave it to the reader to show that this optimization is sound).

So, our desired wp_E transformer is the same as wp for every construct except for the problematic **while** loop, where we have:

$$wp_E(\textbf{while } b \textbf{ do } S,\ Q) \ = \ \nu X.ec_E((b \Rightarrow wp(S, X)) \wedge (\neg b \Rightarrow Q))$$

Thus, wp_E is computable (given an algorithm for approximating ec_E). If $wp_E(S, Q) = false$ then we declare that $wp(S, Q)$, which means that S is safe with respect to the

set of bad final states satisfying predicate Q. Referring back to the terminology of the Introduction, $wp_E(S, Q)$ is the result of the symbolic (backwards) reachability analysis of the Boolean abstraction. (We note that to be completely faithful to Boolean abstraction, we should check $ec_E(wp_E(S, Q)) = false$, which is in general weaker than $wp_E(S, Q) = false$. However, as shown above, we need only apply ec_E at **while** loops.)

Exercise. Determine how to efficiently implement wp_E. As a hint, in the SLAM process, we introduce a Boolean variable b_i for each predicate e_i in E and convert the predicate $ec_E(e)$ into a proposition formula (Boolean expression) over the b_i. We then can make use of binary decision diagrams [Bry86] to manipulate boolean expressions. **End Exercise**

4 Path Feasibility Analysis

Suppose that $wp_E(S, Q) \neq false$. From this result, we do not know if $wp(S, Q)$ is false or not. In order to refine our knowledge, we proceed as follows. We introduce the concept of a "bounded weakest precondition" that will serve to unify the concepts of counterexample traces of the Boolean abstraction and the original program.

Figure 3 shows the bounded weakest precondition computation, specified as a recursive function. The basic idea is to compute the weakest precondition of statement S with respect to predicate Q for all transformations up to and including k steps (and possibly some more) from the end of S. Each skip or assignment statement constitutes a single step. The bwp transformer returns a pair of a predicate and the numbers of steps remaining after processing S.

We see that the definition of bwp first checks if the step bound k has been exhausted. If so, bwp is defined to be "false". This stops the computation along paths of greater than k steps (because $wp(S, false) = false$ for all S). Otherwise, if statement S is an assignment or skip, the weakest precondition is computed as before and the step bound decremented by one. Statement sequencing is not noteworthy. The processing of the **if-then-else** statement proceeds by first recursing on the statements S_1 and S_2, resulting in corresponding outputs (Q_1, k_1) and (Q_2, k_2). The output predicate is computed as before and the returned bound is the maximum of k_1 and k_2. This guarantees that all (complete) computations of length less than or equal to k steps will be contained in bwp (and perhaps some of longer length as well).

Processing of the **while** statement simply unrolls the loop by peeling off one iteration (modeled by the statement "**if** b **then** S_1 **else skip**") and recursing on the **while** statement. It is clear that the bwp will terminate after at most k steps.

We define bwp_E to be the same as bwp everywhere except at the **while** loop (as before), where we replace $bwp(S, (Q', k'))$ by $bwp(S, (ec_E(Q'), k'))$.

Exercise. If $wp_E(S, Q) \neq false$ then there is a smallest k such that $bwp_E\ (S, (Q, k))$ $\neq false$. Such a k can be found by various means during the computation of wp_E. Generalize the wp_E computation to produce such a k. **End Exercise**

let $bwp(S,(Q,k)) =$
  if $k \leq 0$ **then**
    $(false, 0)$
  else
    case S **of**
      "**skip**" $\rightarrow (Q, k-1)$

      "$x := e$" $\rightarrow (Q[x/e], k-1)$

      "$S_1; S_2$" $\rightarrow bwp(S_1, bwp(S_2,(Q,k)))$

      "**if** b **then** S_1 **else** S_2" $\rightarrow$
        let $(P_1, k_1) = bwp(S_1,(Q,k))$ **in**
        let $(P_2, k_2) = bwp(S_2,(Q,k))$ **in**
        $(((b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2)), max(k_1, k_2))$

      "**while** b **do** S_1" $\rightarrow$
        let $(Q', k') = bwp($"**if** b **then** S_1 **else skip**", $(Q,k))$ **in**
        $bwp($"**while** b **do** S_1", $(Q', k'))$

Figure 3: Bounded weakest precondition.

5 Predicate Discovery

If $bwp(S,(Q,k)) \neq false$ then we have found that there is an initial state in $wp(S,Q)$ from which the bad state Q can be reached in at least k steps. If, on the other hand, $bwp(S,(Q,k)) = false$ then there is no counterexample of less than or equal to k steps. In this case, we wish to refine the set of predicates E by finding a set of predicates E' such that $bwp_{E \cup E'}(S,(Q,k)) = false$.

To explain the generation of predicates, let us first focus on a simple scenario. We have a program $S = S_1; S_2$, post-condition Q and set of predicates E such that:

- $wp(S_1, ec_E(wp(S_2, Q))) \neq false$, and
- $wp(S_1, wp(S_2, Q)) = false$.

Now, we wish to find an E' such that

$$wp(S_1, ec_{E \cup E'}(wp(S_2, Q))) = false$$

One sufficient E' simply is the set of atomic predicates that occur in $wp(S_2, Q)$ (that is, $E' = atoms(wp(S_2, Q))$). With such an E', the predicate $wp(S_2, Q)$ is expressible as a Boolean function over the predicates in E' and so is expressible by the exterior cover $ec_{E \cup E'}$.

While the set of predicates ($atoms(wp(S_2, Q))$) is a correct solution to the predicate refinement problem, it may be too strong for our purposes. That is, there may be

many E'' such that for all e, $ec_{E'}(e) \Rightarrow ec_{E''}(e)$ and for which

$$wp(S_1, ec_{E \cup E''}(wp(S_2, Q))) = false$$

Craig interpolants [Cra57] are one way to find such E'' [HJMM04]. Given predicates A and B such that $A \wedge B = false$, an interpolant $\Theta(A, B)$ satisfies the three following points:

- $A \Rightarrow \Theta(A, B)$,
- $\Theta(A, B) \wedge B = false$,
- $V(\Theta(A, B)) \subseteq V(A) \cap V(B)$

That is, $\Theta(A, B)$ is weaker than A, the conjunction of $\Theta(A, B)$ and B is unsatisfiable ($\Theta(A, B)$ is not too weak), and all the variables in $\Theta(A, B)$ are common to both A and B.

To make use of interpolants in our setting, we decompose $wp(S_1, wp(S_2, Q))$ into A and B predicates as follows. Let X be the set of variables appearing in the program $S_1; S_2$ and let X' be primed versions of these variables (which do not appear in the program). Let $eq_X = \bigwedge_{x \in X}(x = x')$, which denotes the set of states in which each $x \in X$ has the same value as its primed version x'. We define A and B as follows:

- $A = wp(S_2, Q)[X/X']$;
- $B = wp(S_1, eq_X)$

That is, A is the weakest precondition of S_2 with respect to Q, with all unprimed variables replaced by their primed versions, while B is the weakest precondition of S_1 with respect to the predicate eq_X. It is clear that:

$$wp(S_1, wp(S_2, Q)) \iff (\exists X'.A \wedge B)$$

Let $E_\Theta = atoms(\Theta(A, B)[X'/X])$. From the definition of interpolants, it follows that $wp(S_1, ec_{E_\Theta}(wp(S_2, Q))) = false$. Therefore, $wp(S_1, ec_{E \cup E_\Theta}\ (wp(S_2, Q))) = false$.

We now generalize the generation of interpolants to the **while** language. The basic idea is simple: at any point where the ec_E function has been applied during bwp_E, we need to generate an (A, B) interpolant pair. Since the bwp_E function only applies ec_E after peeling one iteration off of a **while** loop, we need only interpolate at these places. Thus, we will generate at most $k \times n$ interpolant pairs, where n is the number of **while** loops in the program.

The interpolation function ($interp$ of Figure 4) follows the same basic structure as the bwp function with a few important differences. First, there is a mode parameter m that indicates whether the function currently is constructing the A predicate ($m = InA$) or is constructing the B predicate ($m = InB$). With each interpolant pair we also track the remaining number of steps k. Thus, the $interp$ function takes two arguments: a statement S and a four-tuple (m, A, B, k). Second, the function returns a list of four tuples of the same form (m, A, B, k). It returns a list because each iteration peeled from a **while** loop yields a new (A, B) pair.

```
let interp S (m, A, B, k) =
  if k ≤ 0 then
    list (m, false, false, 0)
  else
    case S of
      "skip" → list (m, A, B, k − 1)

      "x := e" →
        if m = InA then
          list (m, A[x/e], B, k − 1)
        else
          list (m, A, B[x/e], k − 1)

      "S1; S2" → flatten(map (interp S1) (interp S2 (m, A, B, k)))

      "if b then S1 else S2" →
        let (InA, A1, B1, k1) :: tl1 = interp S1 (m, A, B, k) in
        let (InA, A2, B2, k2) :: tl2 = interp S2 (m, A, B, k) in
        let res0 = list (InA, ((b ⇒ A1) ∧ (¬b ⇒ A2)), false, max(k1, k2)) in
        let res1 = map (λ(m′, A′, B′, k′). (m′, A′, b ∧ B′, k′)) tl1 in
        let res2 = map (λ(m′, A′, B′, k′). (m′, A′, ¬b ∧ B′, k′)) tl2 in
        res0@res1@res2

      "while b do S1" →
        let res = interp "if b then S1 else skip"(m, A, B, k) in
        let (InA, A′, B′, k′) :: tl = res in
        let res′ = (InA, A′, B′, k′) :: (InB, A′[X/X′], eqX, k′) :: tl in
        flatten (map (interp "while b do S1") res′)
```

Figure 4: Generation of (A, B) pairs for interpolation.

In the case that k is less than or equal to zero, the $interp$ function replaces A and B by $false$ and returns $list(m, false, false, 0)$ (the function $list$ takes a tuple and returns a list containing that single tuple). Processing of a **skip** statement simply reduces the k bound. Processing of an assignment statement switches on the mode variable m to determine whether or not to apply the substitution ($[x/e]$) to A or B.

Processing of statement sequencing first recursively makes the call $(interp\ S_2\ (m, A, B, k))$, which results in a list of four-tuples. The curried function $(interp\ S_1)$ then is mapped over this list (via map), resulting in a list of lists of four-tuples, which is then flattened (via $flatten$) to result in a list of four-tuples.

We now skip to the processing of the **while** loop. The $interp$ function maintains the invariant that (assuming it initially is called with $m = InA$) that the output list begins with a tuple with mode InA and is followed by tuples with mode InB. A loop iteration

is peeled (as before) and a call to $interp$ yields the list res, which is then split into a head (InA, A', B', k') and tail tl. A new list res' is defined that is the same as res except for the addition of the new tuple $(InB, A'[X/X'], eq_X, k')$ which represents the new interpolant pair that we must consider for this loop iteration. Finally, this list is processed (as in statement sequencing, using map and $flatten$).

Finally, we come to the processing of the **if-then-else** statement. As before, the $interp$ function recurses on the statements S_1 and S_2, yielding two lists. By construction, the first tuple of each list has mode InA. These tuples are combined together to make a new list res_0 containing one tuple of mode InA, in the expected way (note that since the mode is InA, it is safe to substitute $false$ in the B position). The list tl_1 contains the InB tuples from the **then** branch. The $interp$ function maps each tuple (m', A', B', k') in tl_1 to $(m', A', b \wedge B', k')$ in res_1 to reflect the semantics of the conditional statement. The list res_2 is created in a symmetric fashion. Finally, the concatenation of lists res_0, res_1 and res_2 is returned as the result.

Recall that we are given that for some k, $bwp_E(S, (Q, k)) \neq false$ and that $bwp(S, (Q, k)) = false$. To derive a new set of predicates we invoke $(interp\ S\ (InA, Q, false, k))$. The first tuple in the result list is discarded (as it has mode InA and does not represent an interpolant pair). The remaining tuples in the list represent (A, B) interpolant pairs. Further recall, that each $\Theta(A, B)$ that is an interpolant of (A, B) yields a set of refinement predicates $E_\Theta = atoms(\Theta(A, B)[X'/X])$.

Exercise. Prove that the set of refinement predicates E' generated by interpolating the (A, B) pairs returned by $(interp\ S\ (InA, Q, false, k))$ has the property that $bwp_{E'}(S, (Q, k)) = false$. **End Exercise**

6 Procedures

We now add to the language procedures with call-by-value formal parameters (of type integer). As in the C language, all procedures are defined at the same lexical level (no nesting of procedures is allowed). To simplify the exposition, procedures do not have return statements. Any side-effect of a procedure must be accomplished through assignment to a global variable (which can be used to simulate returning an integer result).

Exercise. Extend the technique to deal with return statements. **End Exercise**

We assume a set of global integer variables G and we will find it useful to have primed (G') and temporary (G^T) versions of the global variables. Given a procedure p, let $F_p = \{f_1 \cdots f_m\}$ be the formal (integer) parameters of p and let $L_p = \{l_1 \cdots l_n\}$ be the local (integer) variables of p.

Now that we have procedures, it makes sense to talk about the "scope" of predicates. A predicate has "global" scope if it only refers to global variables and constants. A predicate has "local" scope (with respect to a procedure p) if it mentions a formal parameter or local variable of procedure p. Predicates that mention variables from the (local) scopes of different procedures are not allowed. Let E_p be the predicates with global scope together with the predicates that are locally scoped to procedure p.

We wish to compute a "summary" of the effect of a procedure p. A summary is simply a predicate containing both unprimed and primed variables of the global variables. Thus, as summary is a (transition) relation between pre-states and post-states. Let S_p be the body of procedure p (in which all formal parameters and local variables of p are in scope). The summary of p is

$$\Delta_p = \exists L_p.wp(S_p, \bigwedge_{g \in G}(g = g'))$$

That is, the summary is the weakest precondition of the procedure body with respect to a predicate in which the values of variables g and g' are the same, for all global variables g in G and their primed counterparts. Finally, the local variables of p are eliminated, resulting in a predicate mentioning only global variables and their primed counterparts, as well as the formal parameters of procedure p. Procedure summaries make it possible to deal with recursive procedures, as is well-known in program analysis [SP81, RHS95].

The weakest precondition of a call to procedure p from procedure q is

$$wp(p(a_1, \cdots, a_m), Q) \;=\; \exists G^T.(\Delta_p[G'/G^T] \wedge Q[G/G^T])[f_i/a_i]$$

The formula $(\Delta_p[G'/G^T] \wedge Q[G/G^T])$ equates the pre-state of Q (G renamed to G^T) with the post-state of Δ_p (G' renamed to G^T) in order to perform a join of Δ_p with Q. Then, the formal parameters f_i of procedure p are replaced by their corresponding actually arguments a_i. Finally, the temporary globals G^T are eliminated, resulting in a predicate that mentions only global variables (and their primed counterparts), as well as the local variables of procedure q.

If the program has no recursion then we can simply apply the weakest precondition calculation as given above. If it has recursion then, as with **while** loops, we must apply the exterior cover to achieve termination of symbolic backwards reachability. We redefine Δ_p to use the exterior cover as follows:

$$\Delta_p = \exists L_p.ec_{E_p}(wp(S_p, \bigwedge_{g \in G}(g = g')))$$

Of course, not every procedure body need be abstracted. By a depth-first search of the program's callgraph we can identify a set of procedures that are the targets of backedges. Only these procedures's bodies need be abstracted.

Exercise. We have generalized the weakest precondition calculation to procedures with call-by-value parameter passing. Generalize the bwp and bwp_E functions and the generation of refinement predicates via the $interp$ function. **End Exercise**

7 Call-by-reference Parameter Passing

To complicate matters a bit more, we introduce a call-by-reference parameter passing mechanism to the language, which permits aliasing. So far, the type system of our

language (such as it is) permits only the base type "integer". We now extend the type system to permit references to integers:

$$\tau \;\rightarrow\; int \;\mid\; ref\ int$$

References are created via parameter passing of a local variable l (of type int) to formal parameter f of a procedure with type declaration "**out** int". By definition, the type of f is $ref\ int$ and f is initialized to be a reference to variable l. It is possible to copy a reference from caller to callee using the type declaration "**ref** int" for formal parameter f. In this case, the actual argument corresponding to f must be a formal parameter of the caller with type $ref\ int$.

Only formal parameters may have type $ref\ int$. If f is a formal parameter of type $ref\ int$ then the expression $*f$ denotes the contents of the location that f references. Two formal parameters with reference type may be compared for equality.

In order to reason about references to variables we need to add the concept of an "address" of a variable. Let L be the set of all local variables in the program. Let every variable l in L be assigned a unique integer index $ind(l)$. To distinguish integers that represent addresses from program integers, we introduce the constructor $addr(i)$. The following axioms define the meaning of $addr$ and dereference $*$:

$$\forall\, l \in L : *addr(ind(l)) = l \qquad \text{(ADDR/DEREF)}$$

$$\forall\, i, j : i = j \iff addr(i) = addr(j) \qquad \text{(ADDREQ)}$$

The first axiom states that the dereference of $addr(i)$, where i is the index of local variable l is equal to the value of variable l. The second axiom states that two addresses are equal if and only if their integer indices are equal.

We now need to define the weakest precondition for two new cases that arise with dereferences: procedure calls to procedures with **out** parameters; and assignment through a dereference of a formal parameter ($*f := e$). The weakest precondition for a direct assignment to a variable (local or global) ($x := e$) does not change.

We assume (without loss of generality) that the formal parameters of procedure p are $(f_1, \cdots, f_m,\ g_1, \cdots, g_n)$ where the f_i are not **out** parameters and the g_j are **out** parameters. The weakest precondition of a call to procedure p from procedure q is

$$wp(p(a_1, \cdots, a_m, l_1, \cdots, l_n), Q)$$

$$=$$

$$\exists G^T.(\Delta_p[G'/G^T] \wedge Q[G/G^T])[f_i/a_i][g_j/addr(ind(l_j))]$$

The procedure call passes actual expressions $(a_1, \cdots, a_m)$ corresponding to the formal parameters $(f_1, \cdots, f_m)$ and locals variables $(l_1, \cdots, l_n)$ of procedure q corresponding to the formal **out** parameters $(g_1, \cdots, g_n)$. The only change to the wp transformer (compared to the call-by-value transformer) is to replace each **out** parameter g_j by the address of the corresponding local l_j in the call.

Let $*f := e$ be an assignment statement in procedure p, where f is a formal parameter of type $ref\ int$ and e is an integer expression. Let $Y = \{*g_1, \cdots * g_k\}$ be the dereference expressions mentioned in expression e.

Given a predicate Q, variables f and g of type $ref\ int$ and expression e of type int, we consider the effect of assigning e to $*f$ on predicate Q under potential aliasing between f and g. There are two cases to consider: either f and g reference the same location (are aliases), and hence the assignment of e to $*f$ will cause the value of $*g$ to become e; or they are not aliases, and the assignment to $*f$ leaves $*g$ unchanged. The following formula captures this choice:

$$Q[f, e, g] = (f = g \ \wedge \ Q[*g/e]) \ \vee \ (f \neq g \ \wedge \ Q)$$

We now define $wp(*f := e, Q)$ as follows:

$$wp(*f := e, \ Q) = Q[f, e, g_1][f, e, g_2] \cdots [f, e, g_k]$$

This generalization of the assignment statement to consider potential aliases is due to Morris [Mor82].

Exercise. We have generalized the weakest precondition calculation to procedures with call-by-reference parameter passing. Generalize the bwp and bwp_E functions and the generation of refinement predicates via the $interp$ function. **End Exercise**

8 Conclusions

We have presented a counterexample-driven refinement process for software using the concepts of weakest precondition, predicate abstraction, bounded model checking and Craig interpolants. This presentation drew from our experience with the SLAM project as well as recent results from other researchers working on abstraction/refinement techniques. We thank them all for their work and inspiration.

We have left a number of interesting problems open for the reader to solve in the exercises. Just in case these are not enough, here are some other problems to consider:

- consider how to make the process incremental, ala the BLAST project [HJMS02];
- our use of interpolants does not localize the scope of the refinement predicates to the syntax of the program as in [HJMM04] - consider how to do this to improve the efficiency of the process;
- consider how to generalize the type system we have given to permit references to references (as in the C language), and to allow the creation of references by mechanisms other than parameter passing (such as the C address-of operator);
- consider how to generalize the process for concurrent programs.

Acknowledgements

Thanks to Tony Hoare for suggesting that we formalize the counterexample-driven refinement process for a simple structured language using the weakest precondition transformer and for his comments on this paper. Thanks to Sriram K. Rajamani for the enjoyable years we spent together working on the SLAM project and for his comments on this paper. Thanks also to Mooly Sagiv and Orna Kupferman and her students for their comments on this work during my first and too brief visit to Israel.

References

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BPR01] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 268–283. Springer-Verlag, 2001.

[BR00] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, January 2000.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057, pages 103–122. Springer-Verlag, 2001.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[CC78] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

[Cra57] W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *J. Symbolic Logic*, 22:250–268, 1957.

[Das00] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DNS03] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, January 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC 92: Compiler Construction*, pages 125–140, 1992.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993.

[McM03] K.L. McMillan. Interpolation and sat-based model checking. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 1–13. Springer-Verlag, 2003.

[Mor82] J. M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, Lecture Notes of an International Summer School, pages 25–34. D. Reidel Publishing Company, 1982.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.