Generating Abstract Explanations of Spurious Counterexamples in C Programs

Thomas Ball Sriram K. Rajamani
{tball,sriram}@microsoft.com

January 22, 2002

Technical Report
MSR-TR-2002-09

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Generating Abstract Explanations of Spurious Counterexamples in C Programs

Thomas Ball Sriram K. Rajamani
{tball,sriram}@microsoft.com

Microsoft Research

Abstract. Counterexample driven refinement is a promising technique to generate automatic abstractions for model checking software. A central problem in automating this approach is the refinement of models from spurious error traces. We present a solution to this problem for C programs. Our solution introduces compile time names for run time values, and handles all constructs in the C programming language. We present NEWTON, an implementation of our solution, and empirical results from running NEWTON on several C programs.

1 Introduction

There is considerable interest in using model checking to verify software written in common programming languages like C and Java [5, 7, 13]. Abstraction is the fundamental problem in making model checking viable for software. A popular approach for automatic abstraction is to construct a so-called *predicate abstraction*, where concrete states of a program are mapped to abstract states of a model according to their evaluation under a finite set of predicates.

Where do the predicates for predicate abstraction come from? A common way to generate predicates is through a process called iterative refinement: first construct a very abstract model (that, in the case of source code includes only a control skeleton of the source program) and use infeasible paths (spurious counterexamples) in the model to generate predicates iteratively. [14, 17, 6, 15]

This paper address the problem of generating predicates from infeasible paths in a C program. There are several challenges in solving this problem. First, C programs have a rich semantic structure, containing unbounded domains like integers and pointers. Second, C programs have a lexical scoping structure due to procedural abstraction. These features interact — for instance, when pointers are passed between procedures. Our experience is that predicates that *explain* infeasible paths are easier to generate if we assign compile time names to run-time values that can be generated in the C program.

We formalize what it means for a set of predicates to explain infeasibility of a path, and show how to generate *explanations* (the set of predicates) using symbolic simulation. The predicates generated using our technique have new names introduced to denote run time values in the C program. Annotations are used to bind these new names to the C program, and our predicate abstraction tool is able to handle such predicates and annotations [1, 2].

Suppose there are multiple explanations for an infeasible path p. Which explanation is better? Intuitively, an explanation for p is good if it has a small number of predicates and if it also serves as an explanation for other infeasible paths that overlap with p. We give a definition of a partial-order relation between explanations that formalizes the above intuition. We use a technique for giving fresh names to values as a way to effectively generate better explanations.

Given an infeasible path p, suppose our goal is to explain to a programmer (and not an automatic tool) why p is infeasible. It is natural in this context to consider projections (subsequences) of p that are relevant to explaining the infeasibility. In this case, the smaller the projection the better. Projections can be thought of as program slices (along a particular path) that preserve infeasibility of the path. In

fact, predicates explaining the infeasibility of a path can be formally related to projections that explain the infeasibility. We make that connection precise. Our results are useful in generating slices of paths where the relationship between a slice and the path is one of abstraction rather than equivalence, as is traditionally done in program slicing [18] (the link to program slicing is discussed in detail in related work).

The results of this paper are as follows:

- We define what it means for a set of predicates to *explain* the infeasibility of a path in a C program, using the concept of bit-vector abstraction, and show how to use strongest post-conditions to generate explanations of infeasible paths (Theorem 1). The predicates generated can mention both program variables, and new names generated called *symbolic constants*. The flexibility to mention these new names enables generation of such predicates using symbolic simulation.
- We define a criterion called *consistent path projections* for creating slices of infeasible paths that yield more abstract explanations of path infeasibility. We state and prove a theorem (Theorem 2) that links consistent path projections with abstract explanations.
- We present the NEWTON tool for generating explanations of infeasible paths through C programs, which is based on the above ideas. We give details on how NEWTON handles C programs with procedures and pointers.
- We present empirical results of running NEWTON on paths from Windows NT device drivers. These results show that NEWTON is very good at generating explanations of path infeasibility, achieving order-of-magnitude reductions over naive approaches to explaining path infeasibilities.

Outline. Section 2 introduces a simple example used to illustrate concepts throughout the paper. Section 3 sets up background material and defines what it means for a set of predicates to explain the infeasibility of a path. Section 4 presents a modified theory of strongest postconditions that uses "annotations" in order to introduce fresh names for run-time values. Section 5 defines the concept of consistent path projection that forms the basis for generating abstract explanations of path infeasibility. Section 6 describes the implementation of these ideas in the NEWTON tool. Section 7 presents our experimental results on running NEWTON in the context of the SLAM project. Section 8 discusses related work and Section 9 concludes the paper.

2 Example

Consider the "path program" p_1 in Figure 1(a). We call p_1 a "path program" because it represents a finite path through a program with **if-then-else** statements and loops. The **assume** statements in the path program represent the evaluation of a predicate in a conditional to true or false. An **assume** statement silently terminates execution if its expression evaluates to false. The path represented by the program p_1 is infeasible as it is impossible for any execution of this program to reach the end of the program (the program point after the last **assume** statement). Any successful execution of the first five statements establishes the condition $e_1 = (b > 0) \wedge (c = 2b) \wedge (a = b - 1)$. Since e_1 implies $a \neq c$ the last **assume** statement must fail. If we construct a predicate abstraction of this program using the set of predicates $E_1 = \{(b > 0), (c = 2b), (a = b), (a = b - 1)\}$, it turns out that the abstraction is precise enough to represent the infeasibility of this path. Thus E_1 is an explanation of this infeasibility. The formal definition of when a set of predicates is a sufficient explanation for infeasibility is given in Section 3.

Is there a *better* explanation of why this path is infeasible? Consider the program p_2 in Figure 1(b), which is a projection of program p_1. This path program also is infeasible since any successful execution of the first 3 statements of this path establishes the condition $e_2 = (b > 0) \wedge (c = 2b) \wedge (a < b)$. Since e_2 implies $a \neq c$ we have that p_2 is infeasible. Notice that the condition e_1 implies the condition e_2

Program p_1	Program p_2
`assume(b>0);`	`assume(b>0);`
`c := 2*b;`	`c := 2*b;`
`a := b;`	
`a := a - 1;`	
`assume(a<b);`	`assume(a<b);`
`assume(a=c);`	`assume(a=c);`
(a)	(b)

Fig. 1. Two infeasible "path" programs (p_1 and p_2).

since $(a = b - 1)$ implies $a < b$. That is, e_2 is a consistent and weaker explanation of why program p_1 is infeasible, as any execution state satisfying e_1 also will satisfy e_2, and e_2 implies $a \neq c$. The program p_2 shows that the exact value of the variable a is irrelevant to explaining the infeasibility of the path p_1. To reiterate, this is captured by the fact that condition e_2 is more abstract (weaker) than the condition e_1. If we construct a predicate abstraction of the original program p_1 using the set of predicates $E_2 = \{(b > 0), (c = 2b), (a < b)\}$, it turns out that the abstraction is also precise enough to represent the infeasibility of this path. Since p_2 is a projection of p_1, we say that E_2 is a better explanation than E_1.

3 Explaining Infeasible Paths

Let X be a set of integer variables. Let *IntExpr* be the (infinite) set of side-effect-free arithmetic expressions over X and integer constants. Let *RelExpr* be the set of side-effect-free relational expressions over *IntExpr* using the relational operators $(=, \neq, <, >, \leq, \geq)$. A *path* p is a sequence $s_1, s_2, \ldots, s_n$ of **assume** statements (of the form **assume**(e), where $e \in RelExpression$) and assignment statements (of the form $x := e$, where $x \in X$ and $e \in ArithExpression$). This definition of a path can capture any finite path through a single procedure C program with integer variables, where assignment statements in the path in the C program are directly modeled in the path, and the evaluation of conditionals (in **while** loops and **if-then-else** statements) are modeled using the **assume** statement.

We give a semantics to a path using the standard formulation of strongest postconditions [12] for assignment and **assume** statements:

$$\begin{array}{lcl} SP(x := e) & = & \lambda f. \exists x'. f[x'/x] \wedge (x = e[x'/x]) \\ SP(\textbf{assume}(e)) & = & \lambda f. f \wedge e \end{array}$$

where $f[x'/x]$ is the formula f with x' substituted for every free occurrence of x. For a sequence of statements, $p = s_1, s_2, \ldots, s_n$, $SP(p) = SP(s_n) \circ SP(s_{n-1}) \circ \cdots \circ SP(s_1)$, where the functional composition $g \circ h$ of two functions g and h is defined from right to left (i.e., $g \circ h = \lambda x. g(h(x))$). We assume the existence of a quantifier elimination procedure for the computation of SP (we will return to this point in the next section).

A path p is *infeasible* if $SP(p)(\textbf{true}) = \textbf{false}$. For program p_1 of Figure 1(a),

$$\begin{array}{rl} SP(p_1)(\textbf{true}) & = (b > 0) \wedge (c = 2b) \wedge (a = b - 1)) \wedge (a < b) \wedge (a = c) \\ & = \textbf{false} \end{array}$$

Explaining Infeasible Paths. Suppose path p is infeasible. An *explanation* for the infeasibility of p is a set of expressions (predicates) from *RelExpr*. Informally, a set of predicates $\mathcal{P}$ explains the infeasibility of a path p if we can prove that p is infeasible when restricting the domain of our discourse to the

predicates in $\mathcal{P}$ (this is the essence of predicate abstraction). We formalize this intuition using the concept of *bit-vector abstraction*.

A set $\mathcal{P} = \{e_1, \cdots, e_n\}$ of predicates defines an abstract domain $\langle \mathcal{A}(\mathcal{P}), \preceq \rangle$, where $\mathcal{A}(\mathcal{P}) = \mathbf{false} \cup \{0,1\}^n$, and the partial order $\preceq$ is the least reflexive, transitive relation that satisfies the following axioms:

- $\mathbf{false} \preceq v$, for all $v \in \mathcal{A}(\mathcal{P})$.
- $u \preceq v$, if for every $0 \leq i \leq n$, we have either $v[i] = 0$ or $u[i] = v[i]$ for all $u, v \in \{0,1\}^n$.

Informally, for any bitvector v, and $0 \leq i \leq n$, we use $v[i] = 0$ to represent that we have no knowledge about the truth value of the predicate e_i, and we use we use $v[i] = 1$ to represent the knowledge that e_i is **true**. Given a set $\mathcal{P} = \{e_1, \cdots, e_n\}$ of predicates, we define a pair of abstraction and concretization functions $\alpha^{\mathcal{P}}_{\mathsf{bv}}$ and $\gamma^{\mathcal{P}}_{\mathsf{bv}}$ that connect the concrete domain *BoolExpr* (with implication as the partial ordering) and $\langle \mathcal{A}(\mathcal{P}), \preceq \rangle$. Let $0 \cdot e_i = 1$ and $1 \cdot e_i = e_i$, in

$\alpha^{\mathcal{P}}_{\mathsf{bv}} : \mathit{BoolExpr} \to \mathcal{A}(\mathcal{P})$		$\gamma^{\mathcal{P}}_{\mathsf{bv}} : \mathcal{A}(\mathcal{P}) \to \mathit{BoolExpr}$
$f \mapsto \mathbf{false}$	if f is unsatisfiable	$\langle v_1, \ldots, v_n \rangle \mapsto (v_1 \cdot e_1 \wedge \ldots \wedge v_n \cdot e_n)$
$\langle v_1, \ldots, v_n \rangle$	if f is satisfiable, where $v_i = 1$ if $f \implies e_i$ and $v_i = 0$ otherwise	

The abstraction of the operator $SP(s)$ over an atomic statement s (where s is an **assume** or assignment statement) is the operator $SP^{\mathcal{P}}_{\mathsf{bv}}(s)$ on sets of bitvectors defined by

$$SP^{\mathcal{P}}_{\mathsf{bv}}(s) = \alpha^{\mathcal{P}}_{\mathsf{bv}} \circ SP(s) \circ \gamma^{\mathcal{P}}_{\mathsf{bv}}$$

The abstraction of $SP(p)$, where p is a path, is defined to be

$$SP^{\mathcal{P}}_{\mathsf{bv}}(p) = SP^{\mathcal{P}}_{\mathsf{bv}}(s_n) \circ SP^{\mathcal{P}}_{\mathsf{bv}}(s_{n-1}) \circ \cdots \circ SP^{\mathcal{P}}_{\mathsf{bv}}(s_1)$$

If path p is infeasible, we say that a set of predicates $\mathcal{P} = \{e_1, \cdots, e_n\}$ is *sufficient* to explain the infeasibility of p if $SP^{\mathcal{P}}_{\mathsf{bv}}(p)(0^n) = \mathbf{false}$.

Returning to the example of path p_1 from Figure 1(a), the set of predicates $\{(b > 0), (c = 2b), (a = b - 1), \}$ is not an explanation of p_1's infeasibility because there is not enough information in this set for the abstraction to show that the sequence of statements `a:=b; a:=a-1;` makes the predicate $(a = b - 1)$ true. On the other hand, $\{(b > 0), (c = 2b), (a = b), (a = b - 1)\}$ is an explanation. Additionally, $\{(b > 0), (c = 2b), (a < b)\}$ also is an explanation.

4 Path Simulation via Strongest Postconditions

This section reformulates the definition of strongest postcondition (SP') so that it closely resembles the operation of a virtual machine that one would implement to symbolically execute a path. We want to be able to use such a symbolic simulator to automatically generate good explanations for infeasibility. Our reformulation of strongest postconditions introduces a fresh "symbolic constant" θ_x (a Skolem constant) when variable x is used without first being defined or used on a path. This allows us to operate in a logic without existential quantification. The definition of SP' uses an explicit store and substitutes x's value in the store for each occurrence of x in an expression.

Path p_1	Ω_1	Φ_1	Π_1
`b := `θ_b`;`	$\langle b, \theta_b \rangle$		
`assume(b>0);`	$\langle b, \theta_b \rangle$	$\theta_b > 0$	
`c := 2*b;`	$\langle b, \theta_b \rangle$, $\langle c, 2\theta_b \rangle$	$\theta_b > 0$	
`a := b;`	$\langle a, \theta_b \rangle$, $\langle b, \theta_b \rangle$, $\langle c, 2\theta_b \rangle$	$\theta_b > 0$	
`a := a-1;`	$\langle a, \theta_b - 1 \rangle$, $\langle b, \theta_b \rangle$, $\langle c, 2\theta_b \rangle$	$\theta_b > 0$	$\langle a, \theta_b \rangle$
`assume(a<b);`	$\langle a, \theta_b - 1 \rangle$, $\langle b, \theta_b \rangle$, $\langle c, 2\theta_b \rangle$	$\theta_b > 0$, $\theta_b - 1 < \theta_b$	$\langle a, \theta_b \rangle$
`assume(c=a);`	$\langle a, \theta_b - 1 \rangle$, $\langle b, \theta_b \rangle$, $\langle c, 2\theta_b \rangle$	$\theta_b > 0$, $\theta_b - 1 < \theta_b$, $2\theta_b = \theta_b - 1$	$\langle a, \theta_b \rangle$

Fig. 2. Path simulation of program p_1 from Figure 1(a) using SP'.

4.1 Annotations

Let $V(p)$ be the set of variables *referenced* (used or defined) by the statements of path p. Let $\Theta(p)$ be a set of symbolic constants in a one-to-one correspondence with the variables of $V(p)$: $\Theta(p) = \{\theta_x | x \in V(p)\}$. Let *Exp* denote the union of *RelExpr* and *IntExpr*, lifted to refer to $V(p)$, integer constants, and $\Theta(p)$.

Annotations are the mechanism by which we introduce a fresh name θ_x for the value of variable x at a point in a path. An *annotation* to a path $p = s_1, \ldots, s_n$ is a pair $\langle i, (x = \theta_x) \rangle$ where i is an integer, $1 \leq i \leq n$, $x \in V(p)$, and $\theta_x \in \Theta(p)$. Given a path p of n statements and an annotation $a = \langle i, (x = \theta_x) \rangle$, *applying* a to p inserts the assignment statement "$x := \theta_x$" immediately before statement s_i in p. Given a path p and a set A of annotations, let $A(p)$ denote the path obtained by applying every annotation $a \in A$ to p.

We pre-process path p so that no variable is used without first being defined. Given a statement s_j $(1 \leq j \leq n)$ in a path $p = s_1, s_2, \ldots, s_n$, let $exposed(p, j)$ represent the variables that are used in s_j without previously being assigned to or used in p. Given a path p, let E_p be the set of annotations representing the exposed variables in p

$$E_p = \{(j, (x = \theta_x)) | 1 \leq j \leq n \wedge x \in exposed(p, j)\}$$

Given a path p with annotations E_p, we say that the path $p' = E_p(p)$ is the *θ-closure* of p.

4.2 A New Version of Strongest Postconditions

We will define the strongest postcondition for path p in terms of its θ-closure p'. Additionally, we make explicit three separate components of the strongest postcondition representation via the notion of a *context*. A context is a triple $\langle \Omega, \Phi, \Pi \rangle$, where

- Ω is a partial function $V(p) \rightarrow Exp$ called the *store*
- Φ is a set of boolean expressions from *Exp* called the *conditions*, and
- $\Pi \subseteq V(p) \times Exp$ is a set called the *history*.

Ω represents the current valuation to $V(p)$, Φ represents the constraints introduced by expressions in **assume** statements, and Π represents the past valuations to $V(p)$. Ω is extended to expressions over $V(p)$ in the usual way. Let $\Omega[x \rightarrow e]$ be defined as usual:

$$\Omega[x \rightarrow e](y) = \begin{array}{ll} \Omega(y) & \text{if } y \neq x \\ e & \text{if } y = x \end{array}$$

The *strongest postcondition* SP' for the two types of statements in a path maps a context to a new context:

$$\begin{array}{lcl} SP'(x := e) & = & \lambda\langle \Omega, \Phi, \Pi \rangle.\ \langle\ \Omega[x \rightarrow \Omega(e)],\ \Phi\ ,\ \Pi \cup \{\langle x, \Omega(x) \rangle | x \in dom(\Omega)\}\ \rangle \\ SP'(\textbf{assume}(e)) & = & \lambda\langle \Omega, \Phi, \Pi \rangle.\ \langle\ \Omega,\ \Phi \cup \Omega(e),\ \Pi\ \rangle \end{array}$$

Note that an assignment to variable x overwrites the old value of x in the store and appends the old value of x to the history. Unlike the store, which is a function, the history is a relation and is thus able to record all the old values of x.

For a path p with θ-closed version $p' = E_p(p) = s_1, s_2, \ldots, s_n$, we have:

$$SP'(p) \quad = \quad SP'(p') \quad = \quad SP'(s_n) \circ SP'(s_{n-1}) \circ \cdots \circ SP'(s_1)$$

Let $\emptyset^3$ abbreviate the empty context $\langle \emptyset, \emptyset, \emptyset \rangle$. Figure 2 shows $SP'(p)(\emptyset^3)$ for each prefix of (θ-closed) path p_1.

We now relate SP' to SP. For a pair $\langle x, e \rangle$ in $V(p) \times Exp$, let $\mathcal{C}(\langle x, e \rangle)$ denote the boolean expression $(x =_s e)$. We use the "tagged" logical equality $=_s$ to distinguish equalities that arise from the store and equalities that arise from **assume** statements. We generalize $\mathcal{C}$ to sets of pairs in the usual way.

Let the meaning function γ_{con} map a boolean expression e that is the conjunction of *RelExpr* from the sets e_s and e_o (where e_s is a set of relational expressions in e that are tagged equalities and e_o is a set of other relational expressions in e) to a context: $\gamma_{\mathsf{con}} = \lambda e. \langle e_s, e_o, \emptyset \rangle$. Let the abstraction of context to a boolean expression be denoted by:

$$\alpha_{\mathsf{con}} = \lambda \langle \Omega, \Phi, \Pi \rangle. \bigwedge_{e \in (\mathcal{C}(\Omega) \cup \Phi)} e$$

Note that $\gamma_{\mathsf{con}}(\alpha_{\mathsf{con}}(\langle \Omega, \Phi, \Pi \rangle)) = \langle \Omega, \Phi, \emptyset \rangle$. There are two interpretations of a context $\langle \Omega, \Phi, \Pi \rangle = SP'(\emptyset^3)(p)$ that are related to SP:

- First, the formula $F = \exists \Theta(p). \bigwedge_{c \in \Phi} c$ represents the conditions imposed by the **assume** statements in the path p. Path p is infeasible if-and-only-if F is not satisfiable (i.e., logically equivalent to **false**). For our example path p_1, $F = \exists \theta_b. (\theta_b > 0) \ \wedge \ (\theta_b - 1 < \theta_b) \ \wedge \ (2\theta_b = \theta_b - 1)$, which simplifies to $\exists \theta_b. (\theta_b > 0) \ \wedge \ (\theta_b = -1)$, which is unsatisfiable.
- Second, the boolean expression $\exists \Theta(p). \alpha_{\mathsf{con}}(SP'(p)(\emptyset^3))$ is equivalent to $SP(p)(\mathbf{true})$.

4.3 Explanations

We now show how to generate explanations from SP'. In order to do this, we first must reformulate the bit-vector abstraction to work with SP'. The abstraction of the operator $SP'(s)$ over an atomic statement s (where s is an **assume** or assignment statement) is the operator ${SP'}^{\mathcal{P}}_{\mathsf{bv}}(s)$ on sets of bitvectors defined by

$${SP'}^{\mathcal{P}}_{\mathsf{bv}}(s) = \alpha^{\mathcal{P}}_{\mathsf{bv}} \circ \alpha_{\mathsf{con}} \circ SP'(s) \circ \gamma_{\mathsf{con}} \circ \gamma^{\mathcal{P}}_{\mathsf{bv}}$$

The abstraction of $SP'(p)$, where p is a path, is defined to be

$${SP'}^{\mathcal{P}}_{\mathsf{bv}}(p) = {SP'}^{\mathcal{P}}_{\mathsf{bv}}(s_n) \circ {SP'}^{\mathcal{P}}_{\mathsf{bv}}(s_{n-1}) \circ \cdots \circ {SP'}^{\mathcal{P}}_{\mathsf{bv}}(s_1)$$

We now redefine explanations of infeasible paths in terms of ${SP'}^{\mathcal{P}}_{\mathsf{bv}}$. If path p is infeasible, we say that a set of predicates $\mathcal{P} = \{e_1, \cdots, e_n\}$ is *sufficient* to explain the infeasibility of p if ${SP'}^{\mathcal{P}}_{\mathsf{bv}}(p)(0^n) = \mathbf{false}$.

Theorem 1. Let p be a path and let $\langle \Omega, \Phi, \Pi \rangle = SP'(p)(\emptyset^3)$. Then, if p is infeasible, then the set of predicates $E = \mathcal{C}(\Omega) \cup \mathcal{C}(\Pi) \cup \Phi$, is sufficient to explain the infeasibility of p.

$p'_{1\,A}$	p'_2	Ω_2	Φ_2
`b := `θ_b`;`	`b := `θ_b`;`	$\langle b, \theta_b \rangle$	
`[1] assume(b>0);`	`assume(b>0);`		$\theta_b > 0$
`[2] c := 2*b;`	`c := 2*b;`	$\langle c, 2\theta_b \rangle$	
`[3] a := b;`			
`[4] a := a - 1;`			
`a := `θ_a`;`	`a := `θ_a`;`	$\langle a, \theta_a \rangle$	
`[5] assume(a<b);`	`assume(a<b);`		$\theta_a < \theta_b$

Fig. 3. Symbolic execution of path p'_2 from Figure 1(b). Path p'_2 is a CPP of path p'_1 by the annotations $A = \{\ \langle 1, (b = \theta_b)\rangle,\ \ \langle 5, (a = \theta_a)\rangle\}$.

5 Abstract Explanations

We now define a partial-order that relates explanations to one another. Intuitively, the less information about the path the explanation uses, the more abstract it is. The advantage of abstract explanations of an infeasible path p is that the abstract explanation is likely to be also an explanation for other infeasible paths that overlap with p, and are infeasible for the same reason as p. We generate more abstract explanations by allowing the introduction of fresh symbolic value θ_x for variable x, *even if x has been defined previously.* Annotations are used to specify where such new symbolic values are introduced.

Theorem 1 says (unsurprisingly) that the full set of predicates generated by $SP'(p)$ is an explanation of p's infeasibility. There clearly are better (more "abstract") explanations of infeasibility. Such abstract explanations are generated by allowing annotations at arbitrary points in a path. Annotations are a way to ignore the particular symbolic value a variable x has at a point in a path, simply by renaming this value to be θ_x. If path p is infeasible, then we say that a pair $\langle \mathcal{P}, A \rangle$, where $\mathcal{P} = \{e_1, \cdots, e_n\}$ is a set of predicates and A is a set of annotations, is an *abstract explanation* of the infeasibility of P if $SP'^{\mathcal{P}}_{\mathsf{bv}}(p_A)(0^n) = \mathbf{false}$.

Abstract explanations can be related to projections of paths. We first define projections and then relate abstract explanations to projections. Let path $p = s_1, s_2, \ldots, s_n$, as before. Path q is a *projection* of path p if q is a subsequence of p. If q is a projection of p and j is the index of a statement in q, let j_p be the index of the corresponding statement in p. Given a projection q of path p, we wish to make q's annotation relevant in the context of p. We do so by defining $A_{p,q} = \{(j_p, (x = \theta_x)) | x \in \mathit{exposed}(q, j)\}$.

Path q is a *consistent path projection* (or CPP) of path p if (1) q is a projection of p, and (2) $\alpha_{\mathsf{con}}(SP'(\emptyset^3)(p_{A_{p,q}}))$ implies $\alpha_{\mathsf{con}}(SP'(\emptyset^3)(q))$. Consistent path projections form a partial order.

Now, given an infeasible path $p = p' :: \mathbf{assume}(e)$, where p' is feasible, we desire to find a least CPP q' of p' such that $q = q' :: \mathbf{assume}(e)$ is infeasible. Let us call such a path q an *infeasible consistent path projection* (ICPP) of p. The following theorem shows a strong correspondence between ICPPs and abstract explanations of infeasibility.

Theorem 2. Let p be an infeasible path, let q be an ICPP of p and let $\langle \Omega_q, \Phi_q, \Pi_q \rangle = SP'(q)(\emptyset^3)$. The set of predicates $E = \mathcal{C}(\Omega_q) \cup \mathcal{C}(\Pi_q) \cup \Phi_q$ along with the annotations $A_{p,q}$ is sufficient to explain the infeasibility of p.

Example. Path p_1 in Figure 1 is infeasible, as is path p_2. Path p_2 is a ICPP of p_1. As mentioned in the Introduction, the path p_2 shows that the exact value of variable a is not important in explaining the infeasibility.

We now show why p_2 is an ICPP of p_1. Let p'_1 and p'_2 be paths p_1 and p_2 without their last **assume** statements. The θ-closed version of p'_2 is shown in Figure 3. The annotations induced by the projection p'_2 of p'_1 are $A = \{\ \langle 1, (b = \theta_b)\rangle,\ \ \langle 5, (a = \theta_a)\rangle\}$. Applying these annotations to path p'_1 yields the path $p'_{1\,A}$

(also shown in Figure 3). It is straightforward to see that $\alpha_{\mathsf{con}}(SP'(\emptyset^3)(p'_{1\,A}))$ implies $\alpha_{\mathsf{con}}(SP'(\emptyset^3)(p'_2))$, so p'_2 is a CPP of p'_1.

This example shows that reinterpreting the symbolic value of a in the statement "`assume(a<b);`" of p'_2 as θ_a (rather than $\theta_b - 1$) allows the infeasibility of the path p_1 to be explained by a weaker (larger) set of states. Such a reinterpretation is made possible by the fact that the statement **assume**$(a < b)$ is redundant in path p_1 because the state just before this statement implies that the condition $a < b$ always is true. We will exploit this observation in the next section.

6 NEWTON: A tool for generating abstract explanations of infeasibility

The previous section described how abstract explanations for path infeasibility can be generated from infeasible consistent path projections (ICPPs). There are many possible ways to construct ICPPs. In this section, we describe the NEWTON tool, which implements the strongest-postcondition SP' (from Section 4) to check if a given path p is infeasible, and if it is infeasible find an abstract explanation for the infeasibility of p based on (implicitly) constructing ICPPs. We then briefly describe the issues in making NEWTON work for programs with procedures and pointers.

6.1 Basic Path Simulation

The internal state of NEWTON has three components that represent the path simulator's context (as described in Section 4): (1) Ω, which is a mapping from variables to symbolic expressions; (2) Φ, a set of predicates representing the evaluated expressions from the **assume** statements in the path; (3) Π, which is a set of past associations between variables and symbolic expressions.

Given a path p, NEWTON functions in three phases:

- Phase 1 computes $SP'(p)(\emptyset^3)$ until Φ becomes inconsistent. While adding a new condition $\Omega(e)$ to Φ, if $\Omega(e)$ is redundant (i.e, the current state implies $\Omega(e)$), then NEWTON introduces fresh symbolic constants for all variables in e. (This is illustrated later on.) As NEWTON computes SP' it keeps a *dependency map*, which maps each member m of Ω and Φ to the set of members from Ω and Π that were used to generate m. These dependencies are analogous to flow dependencies used in program slicing. They capture the flow of values from variable assignments to variable uses. For each **assume**(e) statement encountered in the path, NEWTON uses the Simplify theorem prover [16, 9] to check if $\exists\Theta(p).\bigwedge_{c\in\Phi} c \implies \neg\Omega(e)$ holds. If this formula is satisfiable then $\Phi \cup \Omega(e)$ is inconsistent and the path is declared to be infeasible. If this check does not succeed anywhere in the path, then the path may be feasible.
- Phase 2 starts if $\Phi \cup \Omega(e)$ becomes inconsistent while processing **assume**(e). The goal of this phase is to reduce the size of Φ while still maintaining the relationship that $\bigwedge_{c\in\Phi} c \implies \neg\Omega(e)$. Currently, this is done using a greedy heuristic that eliminates one condition at a time from Φ and checks to see if $\Phi \cup \Omega(e)$ is inconsistent. If so, NEWTON continues to eliminate conditions from Φ. Otherwise it returns the last inconsistent set $\Phi \cup \Omega(e)$. As the empirical results from Section 7 show, the greedy heuristic works well in practice.
- In Phase 3, NEWTON performs a backwards transitive closure from the set of minimized conditions (using the dependency map constructed in Phase 1) to find all the members of the environment (Ω) and history (Π) that flow into these conditions. For such member m in the closure, NEWTON generates a predicate.

We now illustrate the operation of NEWTON. Figure 4 shows three states of NEWTON, during different stages of processing the path p_1 from Figure 1. The statement `assume(b>0)` is processed by first assigning a symbolic constant θ_b for the variable `b`, and then introducing the condition $(\theta_b > 0)$. The number (1)

Ω: var	value	deps.
1. b:	θ_b	()
2. c:	$2\theta_b$	(1)
4. a:	$\theta_b - 1$	(3)

Φ: conds.	deps.
$(\theta_b > 0)$	(1)

Π: var hist.	deps
3. a: θ_b	(1)

(after the assignment `a:=a-1`)

Ω: var	value	deps.
1. b:	θ_b	()
2. c:	$2\theta_b$	(1)
5. a:	θ_a	()

Φ: conds.	deps.
$(\theta_b > 0)$	(1)

Π: var hist.	deps
3. a: θ_b	(1)

(after choosing an abstract value θ_a for a)

Ω: var	value	deps.
1. b:	θ_b	()
2. c:	$2\theta_b$	(1)
5. a:	θ_a	()

Φ: conds.	deps.
$(\theta_b > 0)$	(1)
$(\theta_a < \theta_b)$	(1,5)
$(2\theta_b = \theta_a)$	(2,5)

Π: var hist.	deps
3. a: θ_b	(1)

(after the entire path)

Fig. 4. NEWTON's data structures while simulating the path from Figure 1.

is given to the store entry that assigns θ_b to b. The dependency list for the condition $(\theta_b > 0)$ is recorded as (1) since the evaluation of the condition depended on that store entry. The first table in Figure 4 gives the state of NEWTON after simulating through the assignment `a:=a-1`. Note that the store entry mapping a to θ_b has been moved to the history since a has been assigned twice.

The second table illustrates the step, where NEWTON has decided to abstract the value $\theta_b - 1$ of a by a fresh symbolic constant θ_a. It does this because the condition $(a < b)$ is implied by the current state (that is, substituting the values of a and b in the expression yields $\theta_b - 1 < \theta_b$, which is true).

The third table shows the state of NEWTON after processing the last assume statement. At this stage NEWTON determines that the set of three conditions: $(\theta_b > 0)$, $(\theta_a < \theta_b)$ and $(2\theta_b = \theta_a)$ are inconsistent. Phase 2 of NEWTON determines that this set of conditions cannot be minimized any further, while still maintaining inconsistency. In phase 3, NEWTON collects all the dependencies of these conditions, namely 1 and 5. Finally, NEWTON generates 3 predicates $(\theta_b > 0)$, $(\theta_a < \theta_b)$, $(2\theta_b = \theta_a)$ and two annotations $\langle 5, (a = \theta_a)\rangle$ and $\langle 1, (b = \theta_b)\rangle$ as an abstract explanation of the infeasibility.

6.2 Procedures and Pointers

In the previous section, NEWTON was presented in the context of a simplified language of assignments and **assume** statements for simplicity. The NEWTON tool works for programs in C and handles interprocedural paths in the presence of structures and pointers. A preprocessing step converts any path to an equivalent one that contains only assignments, **assume** statements, procedure calls and procedure returns.

Interprocedural paths are handled by adding a stack to NEWTON. Predicates generated by newton with multiprocedure programs are classified as global predicates or as local to a particular procedure. For predicates to be well-formed with respect to scoping we impose the following conditions:

- A global predicate can only reference global variables or symbolic constants associated with global variables.
- A local predicate associated with a procedure P can only reference global variables, local variables of P, and symbolic constants associated with these variables.

In order to ensure that all generated predicates are well-formed, NEWTON introduces fresh symbolic constants when necessary. We give an example to illustrate this. Consider the path that leads to `ERROR` in the program shown in Figure 5(a). While simulating the first call to procedure `foo`, NEWTON computes the actual parameter to be θ_a, the value of `a` on entry to `main`. If $\Omega(x) = \theta_a$ then there is a potential for a predicate $x = \theta_a$ to be generated. This predicate is not well-formed by the above definition. To prevent

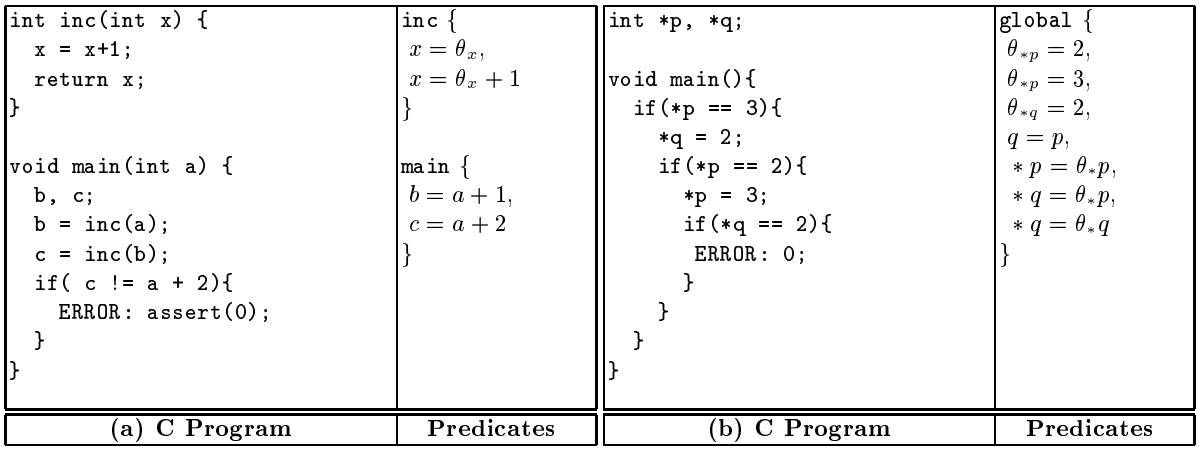

Fig. 5. (a) A multiprocedure C program and predicates generated by NEWTON for the path that leads to `ERROR`; (b) A C program with pointers and predicates generated by NEWTON for the path that leads to `ERROR`

this, NEWTON generates a fresh symbolic constant θ_x for x and associates θ_x with θ_a in a separate mapping maintained at each procedure call.

As a result, NEWTON is able to generate the predicates $\theta_b = \theta_a + 1$, $\theta_c = \theta_a + 2$ as local predicates for procedure `main` and predicates $x = \theta_x$, $x = \theta_x + 1$ as local predicates for procedure `foo`. Using post-processing on the predicates, NEWTON finds that θ_b can be replaced with b and θ_a with a. For a description of how our predicate abstraction tool C2BP uses these predicates see [2].

Pointers are handled by generalizing the store (Ω) to be a map from locations to values. Expressions can also evaluate to locations. Care must be taken to generate proper explanations of infeasibility in the presence of pointer aliasing. Whenever NEWTON needs to generate annotations for two pointers p and q, it consults the results of a pointer-analysis. If p and q can never be aliased then it generates distinct annotations for p and q. If p and q may alias, then NEWTON considers both the aliased and not aliased cases. The resulting conditions generated to Simplify can contain disjunctions and Simplify is able to handle them using case analysis.

Figure 5(b) shows a C program with pointers p and q, where p and q can possibly be aliased. Regardless of whether p and q are aliased, the path to `ERROR` is infeasible. Figure 5(b) also shows the predicates generated by NEWTON by considering both alias possibilities.

7 Implementation and Experiments

In this section, we describe the context of the SLAM process and toolkit in which we run the NEWTON tool and present experimental results of applying NEWTON to paths from device drivers and the SLAM regression test suite. The SLAM toolkit checks safety properties of software without the need for user-supplied annotations or user-supplied abstractions. Given a safety property ϕ to check on a C program P, the SLAM process [5] iteratively refines a *boolean program* abstraction of P using three tools:

- C2BP, a predicate abstraction tool that abstracts P into a boolean program $\mathcal{BP}(P, E_i)$ with respect to a set of predicates E_i over P [1]. The boolean programs constructed by C2BP are guaranteed to be *abstractions* of the input C program P in that every feasible path of P is a feasible path of the boolean program. However, because of the imprecision of the abstraction, the boolean program may contain feasible paths that are not feasible in the C program.

driver	Loc	Insts	Avg. $\mid \Omega \mid$	Avg. $\mid \Phi \mid$	Avg. Preds	Iter
apmbatt	2273	238	173	16	1	1
pnpmem	3930	256	175	18	1	3
iscsiprt-server	4855	252	143	25	3	23
floppy	7631	179	133	19	3	10
serial	26981	408	268	46	1	22

Table 1. Results on running NEWTON on paths from a Windows NT device drivers. See text of paper for explanation.

- BEBOP, a tool for model checking boolean programs [4]. If BEBOP reports that $\mathcal{BP}(P, E_i)$ satisfies a property ϕ then so does the C program P. Otherwise, BEBOP reports a counterexample path through P, which may or may not be feasible.
- NEWTON, the tool that is the topic of this paper. The SLAM toolkit uses the NEWTON tool to generate a set of predicates F_i (and annotations) that explains why a counterexample is infeasible (if it is infeasible). These predicates (and annotations) are sufficient to guarantee that the next iteration of the SLAM process (run on the set of predicates $E_{i+1} = E_i \cup F_i$) will not encounter the same infeasible path. The weaker (more abstract) an explanation is, the more infeasible paths it will rule out in a single iteration of the process.

The C2BP and NEWTON tools were implemented in the OCAML programming language, on top of the AST Toolkit, one-level-flow points-to algorithm [8], and the Simplify theorem prover [9]. The BEBOP tool was implemented in C++, on top of the CMU and Colorado BDD packages.

We report the results of running NEWTON on five device drivers from Windows NT. The property that was checked in these runs had to with conformance to a state machine for I/O request packet (IRP) completion in the device driver. Details of the property are not relevant for understanding the numbers. The interpretation of the numbers in each column is given below.

- **Loc**: Number of lines of code. The drivers range from 2273 to 26981 lines of C code (without counting number of lines in header files).
- **Insts**: Number of assignments and **assume** instructions in the path (i.e., the length of the path).
- **Avg.** $\mid \Omega \mid$: Average number of entries in Ω after path simulation.
- **Avg.** $\mid \Phi \mid$: Average number of entries in Φ after path simulation.
- **Avg. Preds**: Average number of predicates in the explanation generated by NEWTON.
- **Iter**: Number of iterations of NEWTON run by SLAM. Averages in the previous columns are computed over these runs.

We find that in each case NEWTON generates a very small explanation containing an average of at most 3 predicates per iteration. The total number of predicates from which these predicates were chosen is the sum $\mid \Omega \mid + \mid \Phi \mid$ which ranges from 152 (floppy) to 314 (serial). Every iteration of NEWTON took under a minute to run and consumed less than 10MB of memory in a 996Mhz Pentium PC with 256MB RAM.

We have also run NEWTON on 73 test programs that are part of a regression test suite for SLAM. The test suite contains small programs that test several specific features such as structure fields, pointer dereferences, taking an address of a local and passing it to a procedure where the local is updated indirectly, dynamic memory allocation, and complex interactions between such features. NEWTON finds a small set of predicates to explain the infeasibility in all these cases. A complete listing of our regression suite and the results or running all the SLAM tools (including NEWTON) on this suite can be downloaded from `http://research.microsoft.com/slam/`.

8 Related Work

Counter-example driven refinement. Counterexample driven refinement with automatic predicate generation has been studied before in [14, 17, 6, 15]. In comparison with these efforts, the main novelty of our work is the use of new names (symbolic constants) to denote run time values and generate predicates in terms of these symbolic constants. We needed to do this in order to handle the rich features of a programming language with pointers and procedure calls. The paths given to NEWTON (by the BEBOP tool) are control paths— a sequence of control locations through a program. A control path is equivalent to a *set* of abstract counterexamples when compared to [6] and [15]. Since NEWTON maintains data dependencies, it requires just one iteration to exclude all such abstract counterexamples unlike [6] and [15]. In addition, the notions of abstract explanations, the partial order between explanations and connections with program slicing are novel.

Discovering invariants. The predicates E that NEWTON discovers are input to the C2BP predicate abstraction tool that creates a boolean program abstraction $\mathcal{BP}(P, E)$ of a C program P. The BEBOP tool then computes the reachable states of $\mathcal{BP}(P, E)$, which are inductive invariants. In this way, the three tools combine to discover invariants about the C program P. These invariants often are powerful enough to prove interesting properties of programs.

Houdini is a tool for discovering invariants about Java programs to assist the ESC-Java tool, which detects defects in Java programs [11]. Houdini generates invariants based on the input source program and various syntactically-based heuristics (i.e., a reference variable should be tested for being null). Incorrectly-guessed invariants are refuted by the ESC-Java tool. Invariants that are not refuted are guaranteed to be true invariants. Daikon is a tool for discovering program invariants by analyzing the run-time behavior of a program. [10] Daikon uses a library of heuristics to guess likely invariants based on the values that variables take on during an execution of a program. Both the above approaches can be thought of as "eager" approaches to invariant discovery. Invariants are guessed based on the hope that they will be helpful to some client analysis. In contrast, our work is demand-driven, discovering the predicates (and invariants) based on the goal of explaining the infeasibility of a program path.

Program slicing. Consistent path projections (CPPs) are a form of program slice, but differ in a crucial aspect. Program slicing typically computes program projections that are semantically *equivalent* to the programs they come from. [18] Consider programs p_1 and p_2 from Figure 1. Program p_2 is not semantically equivalent to program p_1. Consider the possible values of variable a at the statement `assume(a<b)` in the two programs, given the initial state $[a = 2, b = 1, c = 0]$. In program p_1, the state before the **assume** statement is $[a = 0, b = 1, c = 2]$. In program p_2, the state before the **assume** statement is $[a = 2, b = 1, c = 2]$. These states differ on the value of variable a.

The relationship between program p_1 and p_2 (and CPPs in general) is one of *abstraction* rather than equivalence. That is, the set of reachable states at each program point in p_1 is a subset of the set of reachable states at each corresponding program point (as defined by the projection relationship) in p_2. For example, the set of reachable states in p_1 just before the statement `assume(a<b);` is $b > 0 \wedge c = 2b \wedge a = b - 1$, while the set of reachable states in p_2 just before the statement `assume(a<b);` is $b > 0 \wedge c = 2b$, a strictly weaker condition than the former condition. This abstraction relationship forms a partial order over programs related by syntactic projection.

Termination of iterative refinement. Given a control path p that is infeasible, NEWTON produces a set of predicates that explain the infeasibility of p. When NEWTON is used in the context of property checking using iterative refinement, this property alone does not say anything about the termination of iterative refinement. This paper takes a very local view of iterative view of iterative refinement— how to rule out a single control path. The more global issue— termination of iterative refinement, which involves multiple invocations of NEWTON— is addressed in [3].

9 Conclusions

The question "Where do predicates come from?" is a critical one in counter-example driven refinement. Ours is the first work to address this question for C programs. Our key idea is to name the values of variables at certain points in a path so as to generate explanations of a path's infeasibility. This enables us to use symbolic simulation for generating predicates and handle all the features of the C language. We have introduced consistent path projections as a way to form a partial-order of explanations, and related these with sets of predicates given as explanations. We have implemented our ideas in a tool, NEWTON, and demonstrated its efficacy on Windows NT device drivers.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.
2. T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, 2001.
3. T. Ball, A. Podelski, and S. K. Rajamani. On the relative completeness of abstraction refinement. In *TACAS 02: Tools and Algorithms for Construction and Analysis of Systems*, LNCS (to appear). Springer-Verlag, 2002.
4. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
5. 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.
6. E. M. 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.
7. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *ICSE 00: Software Engineering*, 2000.
8. M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.
9. D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover – http://research.compaq.com/src/esc/simplify.html.
10. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering*, 27(2):1–25, February 2001.
11. C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters (to appear)*, 2001.
12. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
13. G. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.
14. R. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.
15. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 98–112. Springer-Verlag, 2001.
16. G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.
17. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *TACAS 99: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1579, pages 178–192. Springer-Verlag, 1999.
18. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.