Control-flow Refinement and Progress Invariants for Bound Analysis

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

Sagar Jain
IIT Kanpur
sagarj@iitk.ac.in

Eric Koskinen
University of Cambridge
ejk39@cam.ac.uk

Abstract

Symbolic complexity bounds help programmers understand the performance characteristics of their implementations. Existing work provides techniques for statically determining bounds of procedures with simple control-flow. However, procedures with nested loops or multiple paths through a single loop are challenging.

In this paper we describe two techniques, *control-flow refinement* and *progress invariants*, that together enable estimation of precise bounds for procedures with nested and multi-path loops. Control-flow refinement transforms a multi-path loop into a semantically equivalent code fragment with simpler loops by making the structure of path interleaving explicit. We show that this enables non-disjunctive invariant generation tools to find a bound on many procedures for which previous techniques were unable to prove termination. Progress invariants characterize relationships between consecutive states that can arise at a program location. We further present an algorithm that uses progress invariants to compute precise bounds for nested loops. The utility of these two techniques goes beyond our application to symbolic bound analysis. In particular, we discuss applications of control-flow refinement to proving safety properties that otherwise require disjunctive invariants.

We have applied our methodology to over 670,000 lines of code of a significant Microsoft product and were able to find symbolic bounds for 90% of the loops. We are not aware of any other published results that report experiences running a bound analysis on a real code-base.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement techniques; Reliability, availability, and serviceability; D.2.4 [*Software Engineering*]: Software/Program Verification; D.4.5 [*Operating Systems*]: Reliability—Verification; D.4.8 [*Operating Systems*]: Performance—Modeling and prediction; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Verification, Performance, Reliability

Keywords Bound analysis, Termination, Control-flow refinement, Progress invariants, Program verification, Formal verification

PLDI'09, June 15–20, 2009, Dublin, Ireland.

1. Introduction

Software engineers lack the tools they need to build robust, efficient software. They are therefore forced to rely on existing techniques such as testing and performance profiling which are limited, leaving many usage scenarios uncovered. In particular, as processor clock speeds begin to plateau, there is an increasing need to focus on software performance.

This paper addresses the problem of statically generating symbolic complexity bounds for procedures in a program, given a cost model for atomic program statements. Automated methods of generating symbolic complexity bounds offer a significant advance in aiding the performance aspects of the software life cycle. Though programmers are often cognizant of the intended complexity of an algorithm, concrete implementations can differ. Moreover, symbolic bounds can highlight the impact of changes and illuminate performance of unfamiliar APIs.

The most challenging aspect of computing complexity bounds is calculating a bound on the number of iterations of a given loop. Two kinds of techniques have been proposed for automatically bounding loop iterations: pattern matching [18] and counter instrumentation [17, 12, 15]. Unfortunately, these techniques have limitations: (i) They compute bounds for *simple* loops that have a single path (a straight-line sequence of statements) or a set of paths with similar effect, but not *multi-path* loops that have multiple paths with different effects and non-trivial interleaving patterns. (ii) They compute conservative bounds in presence of nested loops since they simply compose bounds for individual loops based on structural decomposition of the program. In this paper, we present techniques that address these limitations.

The first technical contribution of our work is a novel technique called *control-flow refinement*, which is a semantics- and bound-preserving transformation on procedures. Specifically, a loop that consists of multiple paths (arising from conditionals) is transformed into a code fragment with one or more loops in which the interleaving of paths is syntactically explicit. We describe an algorithm (Section 4) that explores all path interleavings in a recursive fashion using an underlying invariant generation tool. The procedure with transformed loop enables a more precise analysis (e.g. with the same invariant generation tool) than would have been possible with the original loop. The additional program points created by refinement allow the invariant generator to store more information about the procedure. Different invariants at related program points in the refined loop correspond to a disjunctive invariant at the original location in the original loop. We detail the application of the control-flow refinement technique for symbolic bound analysis (Section 4.3). The technique enables bound computation for loops for which no other technique can even establish termination (Section 2.1).

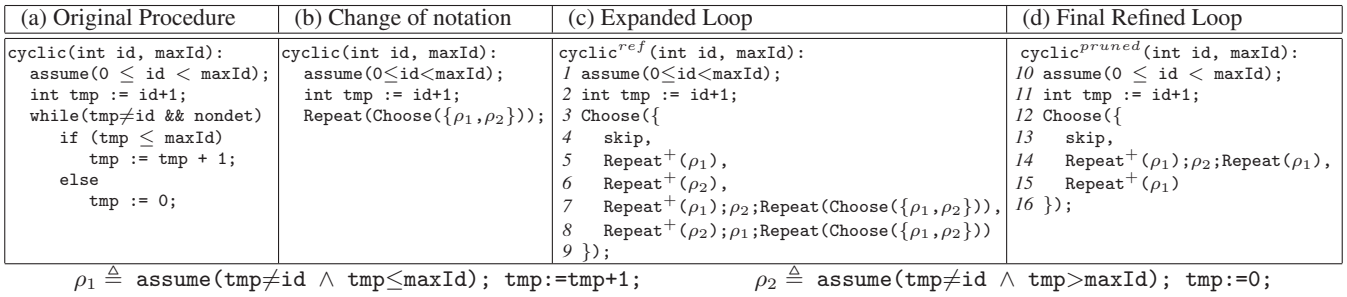

Figure 1: (a) illustrates an iteration pattern seen in product code. (b),(c), and (d) show control-flow refinement of the multi-path loop in (a).

The second technical contribution of our work is the notion of *progress invariants* that describe relationships between any two consecutive states that can arise at a given program location. We present an algorithm for computing such relationships using a standard invariant generator (Section 5). The algorithm runs the invariant generation tool over a procedure appropriately modified and instrumented with extra variables that copy the program state at appropriate locations. We observe that progress invariants are more precise than the related notion of *transition invariants* [24] or *variance analyses* [5] (recently described in literature for proving termination), which describe relationships between a state at a program location and *any other* previous state at that location. Transition invariants can be generated from progress invariants but not vice-versa. (See discussion in Section 9). We believe the notion of progress invariants to have applications beyond bound analysis.

A further contribution is that we show (in Section 6) how to use progress invariants to compute a precise bound for nested loops. This technique applies to procedures that may have been control-flow refined. The key idea is to use progress invariants to illuminate relationships between any two consecutive states of an inner loop per iteration of some dominating outer loop (as opposed to the immediately dominating outer loop). This information is then used to compute the amortized complexity of an inner loop. Such an amortized complexity yields a more precise bound when nested loops share same iterators, which occurs often in practice.

In summary, we make the following contributions:

1. We introduce *control-flow refinement*, a novel program transformation that allows standard invariant generators to reason about loops with structured interleaving between paths in the loop body. This transformation has applications beyond bound analysis, and briefly discuss one such application on proving non-trivial safety properties of procedures that otherwise require specialized analyses [13, 19, 14, 10, 17] (Section 8).
2. We define *progress invariants*, which describe relationships between a state at a program location and the previous state at the same location, and show how to compute them. Progress invariants have applications beyond bound analysis. For example they can be applied to the problem of fair termination (proving procedure termination under fairness constraints).
3. We define an algorithm for computing precise bounds of nested loops using progress invariants (Section 6).
4. To the best of our knowledge, we present the first experimental results for bound analysis on the source of a significant Microsoft product (Section 7). We have built a full interprocedural bound analysis for C++, using C# and F# on top of the Phoenix [1] compiler. Our results show that 90% of non-trivial procedures can be bounded with our technique (Section 7.2).

2. Overview

In this section we illustrate the challenges offered by multi-path loops and nested loops in computing precise bounds for procedures. We also briefly describe our two key techniques that address these challenges; these techniques are described in more detail in the subsequent sections. The examples are adapted from the source of a large Microsoft product. For clarity, we have distilled their core control flow and renamed some variables.

2.1 Multi-Path Loops

Consider the example in Figure 1(a), which is adapted from the product code. This procedure is a form of "cyclic" iteration: initially `tmp` is equal to `id+1`, `tmp` is incremented until it reaches `maxId+1` (along the `tmp` $\leq$ `maxId` branch), `tmp` is then reset to 0 (along the `else` branch), and finally `tmp` is incremented until it reaches `id`. We would like to automatically conclude that the total number of iterations for this loop is bounded above by `maxId+1`.

None of the bound analysis techniques that we know of can automatically compute a bound for such a loop. This is because path-sensitive disjunctive invariants are required to establish a bound. Recent work [15] proposes elaborate counter instrumentation strategies to reduce dependence on disjunctive invariants, yet would fail to compute a bound because the invariants required are path-sensitive. In fact, we do not know of any technique that can even prove termination of this procedure. Recent techniques [6, 5] based on disjunctively well-founded ranking functions [24] fail for this example because there does not exist a disjunctively well-founded linear ranking function.

The mildly complex control flow in the loop foils all known approaches. A detailed analysis of the failure of these approaches on this example would illustrate that these approaches tend to consider all possible interleavings between the two paths through the loop. However, the two paths are interleaved in a more structured manner. Let us represent the control-flow using a regular expression, letting ρ_1 and ρ_2 denote the increment and reset branches, respectively. Then, the path interleavings in the example loop can be more precisely described by the refinement $(\rho_1^*\rho_2\rho_1^*)|(\rho_1^*)$ of the original control-flow $(\rho_1|\rho_2)^*$. While $(\rho_1|\rho_2)^*$ suggests that paths ρ_1 and ρ_2 can interleave in an arbitrary manner, the refinement $(\rho_1^*\rho_2\rho_1^*)|(\rho_1^*)$ explicitly indicates that path ρ_2 executes at most once. Next, we briefly describe how such a refinement can be carried out automatically, and how it enables bound computation.

Control-flow Refinement The first key idea of this paper is a technique called control-flow refinement. Rather than *abstracting* the control-flow, which blurs interleavings, we instead *refine* the control-flow by making interleavings more explicit. Subsequently, an invariant generation tool may determine that some paths are infeasible, often resulting in a procedure that is easier to analyze.

Figure 1(b) shows the original program re-written using our notation (formally described in Section 3) that uses `assume` statements to replace all conditionals with non-deterministic choice. `Repeat` repeatedly executes its argument a non-deterministic (0 or more) number of times, as long as the corresponding `assume` statements are satisfied. $\texttt{Repeat}^+$is identical to `Repeat` except that it executes its argument at least once. `Choose` selects non-deterministically among its arguments (i.e. among those that satisfy the corresponding `assume` statements).

Figure 1(c) illustrates the key ingredient of the control-flow refinement: a semantics and bound preserving expansion of a multi-path loop, wherein $\texttt{Repeat}(\texttt{Choose}(\{\rho_1, \rho_2\}))$ is replaced by a choice between one of the following:

- Loop does not execute: "`skip`"
- Only ρ_1 executes, at least once: "$\texttt{Repeat}^+(\rho_1)$"
- Only ρ_2 executes, at least once: "$\texttt{Repeat}^+(\rho_2)$"
- ρ_1 executes first, at least once, followed by the execution of ρ_2, and finally a non-deterministic interleaving of ρ_1 and ρ_2: "$\texttt{Repeat}^+(\rho_1)$;$\rho_2$;$\texttt{Repeat}(\texttt{Choose}(\{\rho_1,\rho_2\}))$"
- ρ_2 executes first, at least once, followed by the execution of ρ_1, and finally a non-deterministic interleaving of ρ_1 and ρ_2: "$\texttt{Repeat}^+(\rho_2)$;$\rho_1$;$\texttt{Repeat}(\texttt{Choose}(\{\rho_1,\rho_2\}))$".

A general form of this expansion for loops with more than two paths is described in Property 4.1.

Figure 1(d) shows the refined version of the program obtained from the expanded program in Figure 1(c) after simplification with the help of an invariant generation tool that can establish the following invariants: (i) The multi-path loop at Line 7 has the invariant $\texttt{tmp} \leq \texttt{id} < \texttt{maxId}$; hence only path ρ_1 is feasible inside the multi-path loop at Line 7. (ii) Line 3 has the invariant $\texttt{tmp} \leq \texttt{maxId}$; hence path ρ_2 is infeasible at the start of Lines 8, and 6. These invariants are easily computed by several standard (conjunctive, path-insensitive) linear relational analyses [22, 7].

The simplification used to obtain Figure 1(d) from Figure 1(c) may not always be possible after one expansion, but may require repeated expansion of multi-path loops. This raises the issue of termination of the expansion step, addressed in detail in Section 4.2.

We can easily bound the number of iterations of each loop in Figure 1(d) using our technique of progress invariants (described next). In particular, our technique can establish that the two $\texttt{Repeat}^+(\rho_1)$ loops at Line 14 run for at most $\texttt{maxId} - \texttt{id}$ and `id` iterations respectively, combined with the single execution of ρ_2 to yield a total of at most $\texttt{maxId} + 1$ iterations. Meanwhile, the $\texttt{Repeat}^+(\rho_1)$ loop on Line 15 runs for at most $\texttt{maxId} - \texttt{id}$ iterations. Thus, we can conclude a bound of $\texttt{maxId} + 1$ on the number of iterations of the loop in the original program in Figure 1(a).

2.2 Nested Loops

Consider the procedure in Figure 2, which is an example of nested loops with *related* iterator variables, seen commonly in product code. Such loops often arise when an inner loop is used to "skip ahead" through progress bounded by an outer loop.

It is not difficult to see that the values of the loop iterator variables i, j, and k increase in each iteration of the corresponding loop, and hence the complexity of the above loop is $O(n \times m \times N)$. However, this is an overly conservative bound. Observe that the total number of iterations of the innermost loop L_3 is bounded by N (as opposed to $n \times m \times N$) since the value of the iterator k at the entry to loop L_3 is greater than or equal to the value of k when loop L_3 was last executed. Hence, the total combined iterations of all the three loops is bounded above by $n + (m \times n) + N$.

We do not know of any existing bound analysis technique that can compute a precise bound for the above procedure. Recent work [15] proposes elaborate counter instrumentation strategies to compute a *counter-optimal* bound; it generates a bound of $(n + N) \times (1 + m)$, which is not the most precise bound, but still better than the conservative cubic bound[1]. Recent termination techniques [6, 5] based on disjunctively well-founded ranking functions [24] would come up with the termination argument that "either i increases or j increases or k increases at each cut-point, and all three of them are bounded." Such an argument would again imply a conservative cubic bound.

Consider the following triple-nested loop.

```
NestedLoop(int n, int m, int N):
1   assume(0 ≤ n ∧ 0 ≤ m ∧ 0 ≤ N);
2   i := 0;
3   L1: while (i < n && nondet)
4           j := 0;
5           L2: while (j < m && nondet)
6                   j := j + 1;
7                   k := i;
8                   L3: while (k < N && nondet)
9                           k := k + 1;
10                  i := k;
11          i := i + 1;
```

Figure 2: An example of nested loops with related iterator variables.

Our technique can compute the precise bound of $n + (m \times n) + N$ for the total number of all loop iterations. We illustrate here the challenging aspect of proving that the total number of iterations of the innermost loop are bounded above by N. Note that the procedure in Figure 2 is already control-flow refined (there are no multi-path loops) so that doesn't help here. Our bound computation technique uses the notion of progress invariants described below.

Progress Invariants The second key idea of this paper is the notion of *progress invariants* that characterize the sequence of states that arise at a given program location in between any two visits to another program location. Progress invariants are essential to our bound computation algorithm (described in Section 6), which finds more a precise bound than previous techniques based on simple structure decomposition. Our progress invariants (parameterized over an abstract domain D) are:

- $\textsc{Init}_D(\mathcal{P}, \pi_1, \pi_2)$ denotes the property of the initial state of procedure $\mathcal{P}$ that can arise during the first visit to location π_2 after any visit to location π_1.
- $\textsc{Next}_D(\mathcal{P}, \pi_1, \pi_2)$ denotes the relationship between a state (over program variables $\vec{x}$) at a given program location π_2 and the previous state (over fresh variables $\vec{x}_{\texttt{old}}$) at that location, without an intervening visit to location π_1.

We present algorithms in Section 5 to compute the progress invariants $\textsc{Init}_D$ and $\textsc{Next}_D$ given a standard invariant generation tool. For `NestedLoop` (Figure 2), standard relational linear analyses [22, 7] can generate the following progress invariants:

$$\begin{aligned} \textsc{Next}_D(\texttt{NestedLoop}, \pi_0, \pi_3) &: (k \geq k_{\texttt{old}} + 1) \wedge k < N \\ \textsc{Init}_D(\texttt{NestedLoop}, \pi_0, \pi_3) &: k \geq 0 \end{aligned}$$

where π_0 is the entry point of procedure `NestedLoop`, and π_3 is the program point just inside loop L_3.

Our bound analysis engine (presented in Section 6) can conclude from the above invariants that the number of times location 9 is visited (after the last visit to location 1) is bounded above by N.

[1] Counter instrumentation [15] computes a bound of $(n + N) \times (1 + m)$ because of its greedy heuristic to use up the smallest number of counters. It ends up using the same counter to count the total number of iterations of the loops L_1 and L_3, which gets bounded by $n + N$. This results in the number of iterations of loop L_2 to get bounded by $m \times (n + N)$.

3. Notation

We now turn to a formal model of our techniques. In this section, we introduce some notation which we will use in the subsequent sections when we present path refinement and our precise method of calculating procedure bounds.

3.1 Program Model

For simplicity of presentation, we assume that each procedure $\mathcal{P}$ is described as a statement s using the following structural language:

$$\begin{array}{rcl} s & ::= & s_1; s_2 \mid \texttt{Repeat}(s) \mid \texttt{Choose}(\{s_1, .., s_t\}) \\ & & \mid x := e \mid \texttt{assume(cond)} \mid \texttt{skip} \end{array}$$

where x is a variable from the set of all variables $\vec{x}$, e is some expression, and cond is some boolean expression. The expression e can contain procedure calls[2].

The above model has the following intuitive semantics. Since there are non-deterministic conditionals, its semantics can be characterized by showing its operational semantics on a set of states. The following function $[\![s]\!]\sigma$ illustrates how a statement s transforms a set σ of concrete states.

$$\begin{array}{rcl} [\![\texttt{skip}]\!]\sigma & = & \sigma \\ [\![s_1; s_2]\!]\sigma & = & [\![s_2]\!]([\![s_1]\!]\sigma) \\ [\![\texttt{Choose}(\{s_1, .., s_t\})]\!]\sigma & = & [\![s_1]\!]\sigma \cup .. \cup [\![s_t]\!]\sigma \\ [\![\texttt{Repeat}(s)]\!]\sigma & = & \sigma \cup [\![s; \texttt{Repeat}(s)]\!]\sigma \\ [\![x := e]\!]\sigma & = & \{\delta[x \mapsto \delta(e)] \mid \delta \in \sigma\} \\ [\![\texttt{assume(cond)}]\!]\sigma & = & \{\delta \mid \delta \in \sigma, \delta(\texttt{cond}) = \texttt{true}\} \end{array}$$

where $\delta(e)$ and $\delta(\texttt{cond})$ respectively denote the value of expression e or cond in state δ. We often use the notation $\texttt{Repeat}^+(s)$ to denote "$s; \texttt{Repeat}(s)$." Standard deterministic control flow in branches and loops can be modeled in our notation as follows.

$$\begin{array}{rcl} \text{if}(c)s_1 \text{ else } s_2 & \rightsquigarrow & \texttt{Choose}(\{(\texttt{assume}(c); s_1), (\texttt{assume}(\neg c); s_2)\}) \\ \text{while}(c)s_1 & \rightsquigarrow & \texttt{Repeat}(\texttt{assume}(c); s_1); \texttt{assume}(\neg c); \end{array}$$

3.2 Abstract Domain

Our framework is parameterized by a standard abstract domain D, with an abstract element denoted E. However operations in the abstract domain only occur behind the curtains of the invariant generator INVARIANT_D. The only abstract element which appears explicitly in our algorithms is the minimal element $\bot_D$. Our techniques are inter-operable with a variety of existing tools, so we will use some APIs throughout the paper. We assume an invariant generator $\text{INVARIANT}_D(\mathcal{P}, \pi, S_D(\vec{x})) \rightarrow S_D^R(\vec{x}, \vec{x}')$ which takes a procedure $\mathcal{P}$, a program point π, an abstract state S_D over program variables $\vec{x}$, and returns an invariant S_D^R which holds at π. This invariant generator can be for any abstract domain D.

4. Control-flow Refinement

In this section, we present a technique called *control-flow refinement*, which is a semantics-preserving and bound-preserving transformation of loops within a procedure. Specifically, a loop consisting of multiple paths (resulting from a conditional) is refined into one or more loops in which the interleaving of paths is syntactically explicit. Subsequently, an invariant generation tool may determine that some paths are infeasible, often resulting in an overall procedure that is easier to analyze.

Our algorithm is described in Section 4.2. It uses an operation called FLATTEN that we introduce next.

[2] Procedure calls may have side effects, but for simplicity we define the semantics of $[\![x := e]\!]\sigma$ assuming the absence of side effects.

```
REFINE(P:Procedure, s_loop:Repeat statement)
 1 let s_loop be Repeat(s) occurring at location π in P.
 2 E := INVARIANT_D(P, π, true);
 3 s := FLATTEN(s);
 4 Q := Push(E, Empty_Stack);
 5 (s_result, Z) := R(s, Q);
 6 return P with s_loop replaced by s_result;

R(s:Flattened stmt, Q:stack of abstract elements)
 1 let s be of the form Choose({ρ_1, .., ρ_t}).
 2 E := Top(Q);
 3 for i = 1 to t
 4   s_i := (Repeat^+(ρ_i); Choose({ρ_1, .., ρ_{i-1}, ρ_{i+1}, ρ_t}));
 5   π_ex := exit point of s_i;
 6   E' := INVARIANT_D(s_i, π_ex, E);
 7   if (E' = ⊥_D) s_i := ⊥;
 8   else if (∃E_t ∈ Q s.t. E' = E_t) Z_i := {E'};
 9   else (s', Z_i) := R(s, Push(Q, E')); s_i := s_i; s';
10 S_if := {skip}; S_wh := ∅;
11 for i = 1 to t
12   S_if := S_if ∪ {Repeat^+(ρ_i)};
13   if (s_i = ⊥) continue;
14   if (∃E_t ∈ Z_i s.t. E_t = E) S_wh := S_wh ∪ {s_i};
15   else S_if := S_if ∪ {s_i};
16   Z := Z ∪ Z_i − {E};
17 return (Choose(S_if ∪ Repeat(Choose(S_wh))), Z);
```

Figure 3: The algorithm REFINE for refining the control-flow of a loop $\texttt{Repeat}(s)$ in initial state E by invoking $\text{REFINE}(s, E)$.

4.1 Flattening of a statement

Definition 4.1 (Flatten)**.** *Given a statement s, we define* $\text{FLATTEN}(s)$ *to be a statement of the form* $\texttt{Choose}(\{\rho_1, .., \rho_t\})$ *such that for any set of states σ, we have:*

$$[\![s]\!]\sigma = [\![\texttt{Choose}(\{\rho_1, .., \rho_t\})]\!]\sigma$$

where each ρ_i *is a straight line sequence of atomic* $x := e$ *or* assume *statements or* Repeat *loops (and, importantly, no* Choose *statements). We refer to such* ρ_i *as a* path.

The flatten operation can be implemented as:

$$\text{FLATTEN}(s) = \texttt{Choose}(\mathcal{F}(s))$$

where the function $\mathcal{F}(s)$ maps a statement s into a set of straight-line sequences as follows:

$$\begin{array}{rcl} \mathcal{F}(s_1; s_2) & = & \{\rho_1; \rho_2 \mid \rho_1 \in \mathcal{F}(s_1), \rho_2 \in \mathcal{F}(s_2)\} \\ \mathcal{F}(\texttt{Choose}(\{s_1, .., s_t\})) & = & \mathcal{F}(s_1) \cup .. \cup \mathcal{F}(s_t) \\ \mathcal{F}(s) & = & \{s\} \quad \text{for all other } s \end{array}$$

Example 1. *Consider the following code fragment.*

$$s \stackrel{def}{=} \textit{if } c \textit{ then } s_1 \textit{ else } s_{11};\ \ s_2;\ \ \textit{if } c' \textit{ then } s_3;$$

Flattening of the above code fragment yields, in our notation:

$$\begin{array}{l} \texttt{Choose}(\{\ \texttt{assume}(c); s_1; s_2; \texttt{assume}(c'); s_3\ , \\ \qquad \texttt{assume}(\neg c); s_{11}; s_2; \texttt{assume}(c'); s_3\ , \\ \qquad \texttt{assume}(c); s_1; s_2; \texttt{assume}(\neg c')\ , \\ \qquad \texttt{assume}(\neg c); s_{11}; s_2; \texttt{assume}(\neg c')\ \}) \end{array}$$

4.2 Refinement of a loop

The REFINE algorithm in Figure 3 performs control-flow refinement of a multi-path loop s_{loop} in the initial state E, and returns a procedure that is semantically equivalent in the input state E (Theorem 4.1). The key idea is to use the following property that describes how a flattened, multi-path loop can be unfolded into $2t+1$ different cases depending on which loop path iterates first, and whether any other path iterates afterwards. This is the generalization of the two-path loop discussed in Section 2.1.

Property 4.1. *Let s and s_i (for $1 \leq i \leq t$) be as follows.*

$$s \stackrel{def}{=} \mathit{Choose}(\{\rho_1, .., \rho_t\})$$
$$s_i \stackrel{def}{=} \mathit{Repeat}^+(\rho_i); \mathit{Choose}(\{\rho_1, .., \rho_{i-1}, \rho_{i+1}, .., \rho_t\}); \mathit{Repeat}(s)$$
$$s'_i \stackrel{def}{=} \mathit{Repeat}^+(\rho_i);$$

Then, for any set of states σ, we have:

$$[\![\mathit{Repeat}(s)]\!]\sigma = [\![\mathit{Choose}(\{\mathit{skip}, s_1, .., s_t, s'_1, .., s'_t\})]\!]\sigma$$

Of these $2t + 1$ cases, there are t cases (corresponding to $s_1, .., s_t$) that have multi-path loops, which are then further refined recursively. To ensure termination, we use an underlying invariant generator INVARIANT_D to compute the state before each newly created multi-path loop. We then either stop the recursive exploration (if INVARIANT_D can establish unreachability), put a back-edge (if INVARIANT_D finds a state already seen), or use widening heuristics (in case INVARIANT_D generates invariants over an infinite domain). Note that although our algorithm is exponential in the number of paths through the body of the loop, we use a strict slicing technique to keep the constants small. Our slicing technique (described in Section 7.1) reduces the number of paths by collapsing branches that do not impact the iterations of the loop into a single path.

The REFINE algorithm invokes a recursive algorithm $\mathcal{R}$ on the flattened body s of the input loop, along with a stack containing the element E, which is the only input configuration seen before any loop. $\mathcal{R}$ consumes a flattened loop body s and a stack Q of abstract elements. Q represent the input abstract states immediately before the while loop $\texttt{Repeat}(s)$ seen during the earlier (but yet unfinished) recursive calls to $\mathcal{R}$. $\mathcal{R}$ returns a pair (s'', Z) where s'' is a statement and Z is a set of input abstract states that were revisited by the recursive algorithm during the refinement and used to terminate exploration at the promise of arranging a nested loop at appropriate places.

The first loop in $\mathcal{R}$ (Lines 3-9) recursively refines the t cases $(s_1, .., s_t)$ from Property 4.1 that have multi-path loops, one by one. $\mathcal{R}$ refines s_i by choosing between one of the following 3 possibilities depending on the element E' computed before the multi-path loop in s_i:

- Stop exploration (Line 7) if $E' = \bot_D$, denoting unreachability.
- Create a nested loop (Line 8) if E' belongs to stack Q (i.e. it is an input state that has been seen before). Further exploration is stopped and E' is returned to denote the place where the nested loop needs to be created.
- Pursue more exploration (Line 9) otherwise, recursively.

If the abstract domain D is a finite domain, then the first loop in $\mathcal{R}$ terminates because the algorithm is never recursively invoked with the same input state E twice. Otherwise additional measures are required to ensure termination. A trivial way to ensure termination this would be to override the equality check in Line 8 with `return true` if the size of stack Q_i becomes equal to some preselected constant. A better way to accomplish this is with a widening algorithm associated with the domain D, wherein the contents of the stack Q_i are treated as that of the corresponding widening sequence for purpose of checking equality.

The second loop in $\mathcal{R}$ (Lines 11-16) puts together the result of refining the t recursive cases along with the other $t + 1$ cases. $S_{\texttt{wh}}$ collects all the cases to be put together inside a loop at the current level of exploration (thereby meeting the promise of arranging a nested loop), while $S_{\texttt{if}}$ collects all other cases.

The following theorem states that control-flow refinement is semantics- and bound-preserving.

Original	**Refined**
Figure 1(a) with `maxId` renamed by `n`.	Figure 1(d) *Bound*: `n`

Example 2.
`assume(n>0 ∧ m>0);`
`v1:= n; v2:= 0;`
`while (v1>0 && nondet)`
`  if (v2<m)`
`  v2++; v1--;`
`  else`
`  v2:=0;`
	`assume(n > 0 ∧ m > 0);`
	`v1:= n; v2:= 0;`
	`Choose({ skip,`
	`  Repeat(Repeat`$^+(\rho_1)$`;`ρ_2`),`
	`  Repeat`$^+(\rho_1)$
	`});`
	`assume(v1≤ 0);`
	where $\rho_2 \triangleq$ `assume(v1>0); v2:=0;`
	$\rho_1 \triangleq$ `assume(v1>0∧v2<m);v2++;v1--;`
	Bound: $\frac{n}{m} + n$

Example 3.
`assume (0<m<n);`
`i := 0; j := 0;`
`while (i<n && nondet)`
`  if (j<m) j++;`
`  else j := 0; i++;`
	`assume(0<m<n);`
	`i := n;`
	`Choose({ skip,`
	`  Repeat(Repeat`$^+(\rho_1)$`; `ρ_2`),`
	`  Repeat`$^+(\rho_1)$
	`})`
	where $\rho_1 \triangleq$ `assume(i<n∧j<m);j++;`
	$\rho_2 \triangleq$ `assume(i<n∧j≥m);j:=0;i++;`
	Bound: $n \times m$

Example 4.
`assume (0<m<n);`
`i := n;`
`while (i>0 && nondet)`
`  if (i<m) i--;`
`  else i := i-m;`
	`assume(0<m<n);`
	`i := n;`
	`Choose({ skip,`
	`  Repeat`$^+(\rho_2)$`; Repeat(`ρ_1`),`
	`  Repeat`$^+(\rho_2)$
	`})`
	where $\rho_1 \triangleq$ `assume(i>0∧i<m);i--;`
	$\rho_2 \triangleq$ `assume(i>0∧i≥m);i:=i-m;`
	Bound: $\frac{n}{m} + m$

Example 5.
`assume(0 < m < n);`
`i := m;`
`while (0 < i < n)`
`  if (dir=fwd) i++;`
`  else i--;`
	`assume(0 < i < n);`
	`Choose({ skip,`
	`  Repeat`$^+(\rho_1)$`,`
	`  Repeat`$^+(\rho_2)$`,`
	`})`
	where $\rho_1 \triangleq$ `assume(dir=fwd);i++;`
	$\rho_2 \triangleq$ `assume(dir≠fwd);i--;`
	Bound: $\max(m, n - m)$

Figure 4: Some non-trivial iterator patterns from product code that all have a similar multi-loop structure with 2 paths, but very different path-interleavings, and as a result, different bounds.

Theorem 4.1. (Control-flow Refinement) *For any loop s_{loop} inside a procedure P, and any set of initial states σ*

$$[\![\text{REFINE}(P, s_{loop})]\!]\sigma \;=\; [\![P]\!]\sigma$$

Also, $\text{REFINE}(P, s_{loop})$ *and P have the same complexity bound.*

4.3 Case Studies

The table in Figure 4 shows several non-trivial iterator patterns found in product code that share very similar syntactic structure: a single multi-path loop with 2 paths (iterating over variables that range over 0 to n or m). However, the process of control-flow refinement results in significantly different looping structures, because of the different ways in which the 2 paths interleave (which is made explicit by our control-flow refinement technique). In particular, we obtain nested loops for 2nd and 3rd example, sequential loops for 1st and 4th example, and a choice of loops for the 5th example. This leads to significantly different bounds.

5. Progress Invariants

As discussed in Section 2, existing techniques for computing complexity bounds are often imprecise. In this section, we introduce a special form of invariants, we call *progress invariants*: the $\text{INIT}_D(\mathcal{P}, \pi_1, \pi_2)$ and $\text{NEXT}_D(\mathcal{P}, \pi_1, \pi_2)$ relations, which are associated with two program locations π_1 and π_2 inside a procedure $\mathcal{P}$. While progress invariants may have other applications, we use them in this paper to be able to reason about the progress of one particular loop with respect to another loop. As a result, our bound computation algorithm (discussed in the next section) can be precise.

We will refer back to Figure 2 throughout this section. `NestedLoop` is triple-nested and the innermost loop (effectively) increments the same counter as the outermost loop. As discussed in Section 2, previous techniques would compute an overly conservative bound of $m \times n \times N$ rather than $n + (m \times n) + N$.

We start by describing a simple transformation on a procedure called SPLIT that is useful for computing INIT_D and NEXT_D. $\text{SPLIT}(\mathcal{P}, \pi)$ takes a procedure $\mathcal{P}$ and a program location π (inside $\mathcal{P}$) as inputs and returns $(\mathcal{P}', \pi', \pi'')$, where $\mathcal{P}'$ is the new procedure obtained from $\mathcal{P}$ by splitting program location π into two locations π' and π'' such that the predecessors of π are connected to π' and the successors of π are connected to π'', and there is no connection between π' and π''. The SPLIT transformation is a fundamental building block that is used to compute the two progress invariant relations we describe in the remainder of the section.

5.1 The $\text{NEXT}_D(\mathcal{P}, \pi_1, \pi_2)$ Relation

We define $\text{NEXT}_D(\mathcal{P}, \pi_1, \pi_2)$ to be a relation over variables $\vec{x}$ (those that are live at location π_2) and their counterparts $\vec{x}_{\texttt{old}}$ that describes the relationship between any two consecutive states that arise at π_2 without an intervening visit to location π_1. More formally, let $\sigma_1, \sigma_2, \ldots,$ denote any sequence of program states that arise at location π_2 after any visit to location π_1, but before any other visit (to π_1). Let $\sigma_{i,i+1}$ denote the state over $\vec{x} \cup \vec{x'}$ such that for any variable $x \in \vec{x'}$, $\sigma_{i,i+1}(x_{\texttt{old}}) = \sigma_i(x)$ and $\sigma_{i,i+1}(x) = \sigma_{i+1}(x)$. Then, for all i, $\sigma_{i,i+1}$ satisfies the relation $\text{NEXT}_D(\mathcal{P}, \pi_1, \pi_2)$. We can compute NEXT_D as follows using an invariant generator INVARIANT_D.

```
NEXT_D(P, π1, π2):
1 E1 := INVARIANT_D(P, π2, true);
2 (P1, π1', π1'') := SPLIT(P, π1);
3 (P2, π2', π2'') := SPLIT(P1, π2);
4 Let P3 be P2 with entry point changed to π1''
  and instrumented with x⃗old := x⃗ at π2'';
5 E2 := INVARIANT_D(P3, π2', E1);
6 return E2;
```

This algorithm begins by using an invariant generation procedure to generate an abstract element as a loop invariant for π_2 (Line 1). We then perform two transformations on the flow graph: the region of interest (all paths from π_2 to π_2 which do not pass through π_1) is isolated by eliminating the path from π_1 to π_2 (Lines 2 and 4), and π_2 is instrumented with $\vec{x}_{\texttt{old}} := \vec{x}$ (Lines 3 and 4). Finally, we compute a new invariant at π_2' (Line 5) seeded with the original loop invariant.

We now return to the example in Figure 2. As we will describe in the next section, it is useful to obtain a NEXT_D invariant for each nested loop L with respect to its dominating loops L'. Let π_1 be the program point just inside loop L_1; similar for π_2 and π_3. Let π_0 be the entry point of procedure `NestedLoop`. For this example, an invariant generator may find (among other things):

$\text{NEXT}_D(\texttt{NL}, \pi_0, \pi_1) \quad : \quad i \geq i_{\texttt{old}} + 1 \wedge i < n$
$\text{NEXT}_D(\texttt{NL}, \pi_1, \pi_2) \quad : \quad j = j_{\texttt{old}} + 1 \wedge j < m$
$\text{NEXT}_D(\texttt{NL}, \pi_0, \pi_3) \quad : \quad k \geq k_{\texttt{old}} + 1 \wedge k < N$

We later explain (Section 6) how to use these invariants to obtain a bound. However, for now note that these expressions describe the *progress* of variables with respect to outer loop iterations. For example, we see that at π_3, k is always greater than or equal to $k_{\texttt{old}} + 1$, and the loop invariant is that $k < N$. From this, along with initial conditions on k, we will later (Section 6) conclude that the *total* number of loop iterations of L_3 is bounded by N.

5.2 The $\text{INIT}_D(\mathcal{P}, \pi_1, \pi_2)$ Relation

We define $\text{INIT}_D(\mathcal{P}, \pi_1, \pi_2)$ to be a relation over variables $\vec{x}$ (those that are live at location π_2) that describes the state that can arise during the first visit to π_2 after the last visit to location π_1. We can compute INIT_D as follows using an invariant generator INVARIANT_D.

```
INIT_D(P, π1, π2):
1 E1 := INVARIANT_D(P, π1, true);
2 (P1, π1', π1'') := SPLIT(P, π1);
3 (P2, π2', π2'') := SPLIT(P1, π2);
4 Let P3 be P2 with entry point changed to π1''.
5 E2 := INVARIANT_D(P3, π2', E1);
6 return E2;
```

This algorithm is similar to the algorithm used to compute NEXT_D, but has important differences. First, the initial abstract element E_1 holds at π_1 (Line 1). Second, the transformation preserves the path from π_1 to π_2 (Line 4) and `false` holds on all edges out of π_2''. Finally, we are not interested in computing invariants over relationships over the value of variables between two successive states (hence there is no instrumentation step). The algorithm therefore computes invariants which hold the first time π_2 is reached coming from π_1, rather than loop invariants over π_2.

We again return to Figure 2, where a standard invariant generation tool may find (among other things):

$\text{INIT}_D(\texttt{NL}, \pi_0, \pi_1) \quad : \quad i = 0$
$\text{INIT}_D(\texttt{NL}, \pi_1, \pi_2) \quad : \quad j = 0$
$\text{INIT}_D(\texttt{NL}, \pi_0, \pi_3) \quad : \quad k \geq 0$

The purpose of INIT_D is to study properties of the first element represented in the sequence NEXT_D (invoked with the same arguments). We later explain (Section 6) how to use these invariants to obtain a bound.

6. Bound Computation

In this section, we describe how progress invariants (introduced in Section 5) can be used to compute *precise* bounds. This technique can be applied to any procedure, but we apply it to procedures for which we first perform control-flow refinement (introduced in Section 4) to reason about path interleavings.

We introduce some useful notation. For any loop L in procedure $\mathcal{P}$, we define $T(L)$ to be the upper bound on the total number of iterations of L in procedure $\mathcal{P}$. For any loops L, L' such that L is nested inside L', we define $I(L, L')$ to be the upper bound on the total number of iterations of L for each iteration of L'.

6.1 Bounding Loop Iterations

Fundamental to computing complexity bounds is the task of calculating the number of iterations of a loop. We denote this procedure BOUNDFINDER[3]. It consumes an abstraction of the initial state of the loop (given in some abstract domain D) as well as an abstraction of the relation between any two successive states in a loop. These abstractions are given by the progress invariants INIT_D and NEXT_D described in Section 5. The output is both

[3] This name follows the spirit of RankFinder [23], which accomplishes a similar task of finding a ranking function for a transition invariant.

$$
\begin{array}{rcll}
\mathcal{B}(s) & = & (1, \emptyset) & (1) \\
 & & \text{where } s \in \{\texttt{skip}, x\texttt{:=}e, \texttt{assume}(c)\} & \\
\mathcal{B}(s_1; s_2) & = & (c_1 + c_2, Z_1 \cup Z_2) & (2) \\
 & & \text{where } (c_1, Z_1) = \mathcal{B}(s_1) \text{ and } (c_2, Z_2) = \mathcal{B}(s_2) & \\
\mathcal{B}(\texttt{Choose}(\{s_1, .., s_t\})) & = & (\texttt{Max}\{c_1, .., c_t\}, Z_1 \cup .. \cup Z_t) & (3) \\
 & = & \text{where } (c_i, Z_i) = \mathcal{B}(s_i) & \\
\mathcal{B}(L : \texttt{Repeat}(s')) & = & (0, Z \cup (c, L)) & (4) \\
 & & \text{where } c = c' + \sum_{(c'', L'') \in Z', Parent(L'') = L} (c'' \times I(L'', L)) & \\
 & & \text{and } Z = \{(c'', L'') \textit{ where } (c'', L'') \in Z', Parent(L'') \neq L\} & \\
 & & \text{and } (c', Z') = \mathcal{B}(s') &
\end{array}
$$

$$
\textsc{Bound}(s) = c + \sum_{(c', L') \in Z} c' \times T(L')
$$
$$
\text{where } (c, Z) = \mathcal{B}(s)
$$

Figure 5: Calculating the precise bound $\textsc{Bound}(s)$ on a statement s.

$$
\begin{array}{l}
I(L, L') = \textsc{BoundFinder}_D(\textsc{Init}_D(\mathcal{P}, \pi', \pi), \textsc{Next}_D(\mathcal{P}, \pi', \pi), V) \\
T(L) = \textsc{BoundFinder}_D(\textsc{Init}_D(\mathcal{P}, \pi_{\texttt{en}}, \pi), \textsc{Next}_D(\mathcal{P}, \pi_{\texttt{en}}, \pi), V)
\end{array}
$$

where π is the first location inside loop L, π' is the first location inside loop L', $\pi_{\texttt{en}}$ is the entry point of procedure $\mathcal{P}$, and V is the set of all input variables.

Continuing with the example in Figure 2, from the progress invariants given in Section 5, BOUNDFINDER would bound the total number of loop iterations as: $T(L_3) = N$ and $T(L_1) = n$. Moreover, BOUNDFINDER would conclude that the number of iterations of loop L_2 per iteration of L_1 is: $I(L_2, L_1) = m$. These quantities allow us to compute an overall bound of $n + (m \times n) + N$ using the equations given in the next section.

BOUNDFINDER can be implemented in a variety of ways. One potential way to implement BOUNDFINDER is with counter instrumentation by using ideas from previous work [12, 15]. Alternatively it can be implemented via unification against a database of known loop iteration lemmas. We implemented the latter technique, as we expected it would be more efficient and comprehensive for the experiments discussed in Section 7.

6.2 Intraprocedural Bound Computation

In order to compute a precise bound $\textsc{Bound}(s)$ on a statement s, we define $\mathcal{B}(s)$ recursively as shown in Figure 5. For any loop L, we use $Parent(L)$ to denote the outermost dominating loop L' such that $I(L, L') \neq \infty$, if any such loop L' exists *and* if $T(L) = \infty$. Otherwise $Parent(L) = \texttt{undefined}$.

$\mathcal{B}$ recurs over the annotated syntax of the statement s. It is aided by $I(L, L')$ and $T(L)$ computed as described in the previous section. $\mathcal{B}$ returns a pair (c, Z), where c denotes the bound of s excluding the contribution of any loop L_i such that $(c_i, L_i) \in Z$. Furthermore, for any loop L_i, there is at most one entry of the form (c_i, L_i) in Z, and c_i denotes the bound of the body of loop L_i.

The base cases are `skip`, assignment, and `assume` statements (Eqn. 1) where the bound is one and there are no loops excluded. Sequential composition (Eqn. 2) is merely the sum of the bounds and combines loop exclusions; non-deterministic choice is similar (Eqn. 3). When the $\mathcal{B}$ reaches a loop L (Eqn. 4), bound calculation is more subtle. The bound in this case is not given directly because the *context* of the loop is unknown. Instead, the bound is deferred by accumulating a pair (c, L) where c is the bound of the body of the loop, which will be multiplied in a future recursive call by outer loops where the context is known. However, we must process the bound of other inner loops L'' that have been deferred to be processed in the *current* context of L. Ultimately, we reach the base case, where $\textsc{Bound}(s)$ can now be obtained directly (right-hand side of Figure 5).

Theorem 6.1. (Bound Computation via Progress Invariants) *The complexity of a procedure, assuming a unit cost model for all atomic statements and procedure calls, is bounded by* $\textsc{Bound}(P)$.

6.3 Examples

Example 6. *Consider the following procedure $\mathcal{P}$ with two disjoint parallel inner loops $L1$ and $L2$ nested inside an outer loop L.*

```
i:=j:=k:=0; while(i++<n) { if (nondet) while(j++<m);
                           else        while(k++<m); }
```

Given that $T(L1) = T(L2) = m$ and $T(L) = n$, we obtain $\textsc{Bound}(\mathcal{P}) = n + 2m$. (Note $n + m$ is not a correct answer, while $n \times m$ is correct but conservative.) This example demonstrates a subtle aspect of $\mathcal{B}$. The elements of a pair of cost and deferred loop (c, Z) (arising from recursive invocations on sub-structures of s) must be tallied differently. Where Z is tallied identically under sequential composition (Eqn. 2) and non-deterministic choice (Eqn. 3), c is instead aggregated as summation and max*, respectively.*

Previous Examples. We return to the example in Figure 2, where we concluded (in Section 6.1) that $T(L_3) = N$, $T(L_1) = n$, and $I(L_2, L_1) = m$. Using the above definitions of BOUND and $\mathcal{B}$ it is easy to show that $\textsc{Bound}(\texttt{NestedLoop}) = n + (m \times n) + N$.

Let us also consider the original example in Figure 1, listed in Figure 1(a), and then refined in Figure 1(d). Let L_{14a} and L_{14b} be the first and second loops on Line 14, and let L_{15} be the loop on Line 15. There are no nested loops, but using $\textsc{Init}_D$ and $\textsc{Next}_D$, BOUNDFINDER would find that $T(L_{14a}) = T(L_{15a}) = \texttt{maxId} - \texttt{id}$ and that $T(L_{14b}) = \texttt{id}$. It is now easy to check that $\textsc{Bound}(\texttt{cyclic}) = \texttt{maxId} + 1$.

6.4 Interprocedural Extension

The bound computation described in the above section assigns a unit cost to all atomic statements including procedure calls. However, in order to obtain an interprocedural computation complexity, we can compute the cost for a procedure call $x := P(y)$ using the following standard process [15, 3]. We replace the formal inputs of procedure P by actuals y in the bound expression $\textsc{Bound}(P)$, and then translate this to a bound only in terms of the inputs of the enclosing procedure by using the invariants at the procedure call site that relate y with the procedure inputs. This process works only for non-recursive procedures that need to be analyzed in a top-down order of the call-graph.

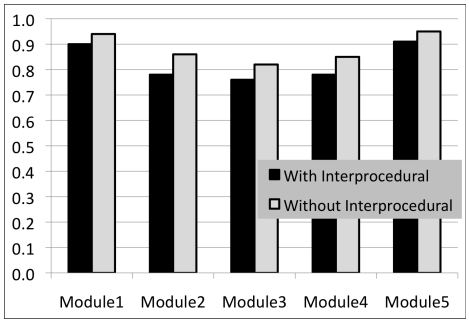

Figure 6: Success rates for non-trivial procedure bounds.

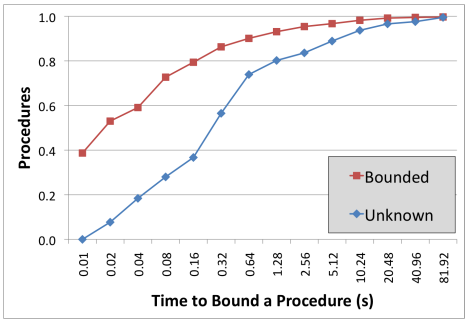

Figure 7: Performance of our tool.

7. Evaluation

7.1 Implementation

We implemented a static interprocedural analysis for computing symbolic complexity bounds of C and C++ procedures, based on the Phoenix [1] compiler framework. Our tool includes support for standard C++ control-flow structures (e.g. `if`, `switch`, `for`, `while`, `do-while`).

An important heuristic is our slicing technique. We slice each loop by preserving only statements that control the loop iteration behavior – this is done by computing a backward slice starting with the conditionals that exit the loop. Slicing is an important optimization that helps generate small loop skeletons that usually do not incur a blowup when flattening is applied to the loop body. We implemented procedure slicing, and flattening in C#. Bound computation (including control-flow refinement and progress invariants) is then accomplished in F#. This library consists of modules for manipulating relational flow graphs and an abstract interpreter, which uses the Z3 [2] theorem prover.

Implementation of BOUNDFINDER**.** We implemented the search for bound expressions in a style similar to unification. We have implemented several lemma "patterns" for each of the iteration classes described below, and search for a pattern which matches the output of progress invariants NEXT_D and INIT_D[4].

- *Arithmetic Iteration*. Many loops use simple arithmetic addition for iteration, consisting of an initial value for the iterator, a maximum (or minimum) loop condition, and an increment (or decrement) step in the body of the loop.
- *Bit-wise Iteration*. Some loop bodies either consist of a left/right shift or an inclusive OR operation with a decreasing operand.
- *Data Structure Iteration*. We implemented patterns for iterations over linked list fields (e.g. `x = x->next`), encapsulated iterators (e.g. `x = GetNext(l)`), and destructive iteration (e.g. `x = RemoveHead(l)`).

Loop iterators beyond these categories are discussed in our limitations (Section 7.3) and an area for future work.

7.2 Experiments

We evaluated our technique by running several experiments over the source code of a large Microsoft product. All experiments below were run on a Hewlett-Packard XW4600, with a Dual Core 3 GHz processor. The hardware included 4 GB of RAM and 250 GB of hard disk space. Our software stack consisted of Windows Vista, the Phoenix April 2008 SDK, F# version 1.9.4.19, and Z3 [2] version 1.3.5. Our analysis was run over a variety of modules from a large Microsoft product; the line count of each module is given in Figure 8(a).

[4] Note that BOUNDFINDER can also be implemented using counter instrumentation [15], though we found unification to be more efficient.

(a) Lines of Code

Module	L.O.C.
Module1	110,469
Module2	132,803
Module3	80,348
Module4	221,120
Module5	126,028
Total:	670,768

(b) Individual Loop Bounds

Module	Loops	Bounded	Success
Module1	1574	1513	0.96
Module2	1749	1570	0.90
Module3	1165	1035	0.89
Module4	535	491	0.92
Module5	2511	2410	0.96
Total:	7534	7019	0.93

Figure 8: (a) Lines of Code and (b) effectiveness of computing loop bounds for a variety of modules from the product source code. For legal reason, the module name is suppressed.

Module	Proc.	Non-Triv.	Isolated Proc. Count	Isolated Proc. Rate	Inter-procedural Count	Inter-procedural Rate
Module1	7192	1746	1639	0.94	1578	0.90
Module2	10816	1956	1674	0.86	1527	0.78
Module3	6280	1181	973	0.82	897	0.76
Module4	4871	744	629	0.85	578	0.78
Module5	9363	2862	2714	0.95	2601	0.91
Total:	38522	8489	7629	0.90	7181	0.84

Figure 9: Effectiveness of computing procedure bounds for a variety of modules from the product source code. The chart considers two cases: (1) procedures individually ("Isolated Proc."), without regard to procedure call sites and (2) effectiveness after including an interprocedural analysis.

Loop Bounds. For our first experiment, we quantify the utility of our technique for bounding multi-path loops, by measuring how frequently BOUNDFINDER is able to find a bound for individual loops. We ran our analysis on several modules and counted the number of loops L for which our technique could compute a bound. Successful bound computation for a loop L means being able to compute $T(L)$ if L is an outermost loop, or $I(L, L')$ where L' is the loop that immediately dominates L. The results are summarized in Figure 8(b). Each module consists of many source files, themselves each consisting of several procedures with loops. Out of the total number of loops in the second column, our technique found a bound for the amount in the third column, yielding the success rate in the final column. Across all modules, our technique found a bound for 93% of the loops.

Isolated Procedures. Procedures are more difficult to bound, because they may contain multiple (possibly nested) loops, all of which must be bounded. Our next experiment tests BOUND, which calculates a cumulative bound across arbitrary procedure

structures. To study this problem, we measured our tool's ability to compute a bound for individual procedures, without regard to call sites to other procedures. Many procedures are trivial (contain no loops), so our analysis focuses on the non-trivial procedures. The results of our experiment for several of the largest modules is given in Figure 9 (and pictorially in Figure 6) labeled "Isolated Proc."

Inter-procedural Analysis. We then evaluated the effectiveness of our inter-procedural technique, which properly accounts the cost of call sites (see Section 6.4). While this is a more accurate measure of a procedure's cost, it decreased our success rate to 84%. This is because of a "cascade effect": if we fail to compute a bound for procedure A, then any other procedure B that involves a call site to A will also be unbounded. The results of this experiment are also given in Figures 9 and 6, labeled "Inter-procedural."

Limitation. One experimental limitation is that, due to the size of the source, we were unable to comprehensively check the precision of the complexity bounds. However, we manually inspected many of the bounds and confirmed that they were, indeed, precise.

Performance. For each non-trivial procedure we also measured the time it took to find a bound for the flattened version of the procedure. This includes control-path refinement via REFINE, progress invariant generation via INIT_D and NEXT_D (which use INVARIANT_D), finding loop bounds via BOUNDFINDER, and finally calculating the total bound via BOUND. Across all modules, the performance is given in Figure 7. This graph illustrates the time it takes to find a bound for a single procedure (in seconds). A perfectly efficient tool would calculate bounds for 100% of procedures instantly. When our technique is successful, over 90% of procedures are bounded within 640ms. For failed attempts, only 70% of procedures are bounded with 640ms. This suggests a possible improvement in the performance of our tool by aborting the search for a bound after, say, 640ms.

Figure 7 also illustrates the efficacy of our slicing heuristic. Most non-trivial functions have at most 8 paths after slicing; thus our algorithm typically completes in a fraction of a second. In less than 10 cases (among the 670,000 lines of code we evaluated) the number of paths was large enough for the analysis to time-out.

7.3 Limitations

There are some loops (roughly 7%) for which our tool is unable to find a bound. As with any large code base, the modules vary in coding styles and paradigms, yet we were surprised by how widely applicable our technique was. We categorized the unsuccessful loops (somewhat automatically) into the following challenges, postponed to future work:

- *Concurrency.* Many procedures contained concurrent algorithms, such as spin-locks or work queues, in which case the the number of loop iterations depends on other threads.
- *I/O.* Some modules contained procedures which interacted with the filesystem. In these rare cases, the bound again depends on the size (or availability) of non-deterministic input.
- *Recursion*. We currently do not address the issue of computing bounds for recursive procedures (though we believe that ideas presented in this paper can potentially be used to compute bounds for recursive procedures).
- *Procedure Calls*. Usually, procedure calls inside a loop do not affect the value of loop iterators. But when they do, we can either inline the appropriately sliced version of the procedure, or use an interprocedural invariant generation tool. We currently do not implement any such strategy.
- *Exponential Paths*. Slicing drastically reduces the number of paths in a flattened loop body; however, in rare cases, flattening generates an intractable, exponential number of paths.

8. Other Application: Safety Properties

The control-flow refinement technique presented in Section 4 is more fundamental than the sole application of bound analysis. In particular, it can be used to prove safety properties that otherwise require disjunctive invariants or a path-sensitive analysis. For this purpose, we simply use a given simple (path-insensitive) invariant generation tool I to first refine the control-flow of the procedure, and then analyze the refined procedure using I.

We need a small extension of our control-flow refinement algorithm described in Figure 3 for it to be powerful enough to establish non-trivial safety properties at the end of the loop. We explicitly add any post-dominating `assume` statement at the end of the multi-path loop to all top-level choices in the expansion of the loop. This is done to enable a path-insensitive invariant generation tool to filter out paths that leave the loop prematurely. This extension is not required for bound analysis because the focus there was to reason about what happens inside the loop, and not outside the loop.

Figure 10 presents a list of some examples, each of which has been used as a flagship example to motivate a new technique for proving non-trivial safety assertions. Proving validity of the assertions in all these examples requires disjunctive loop invariants.

Figure 10 also shows the resulting (semantically equivalent) procedure after control-flow refinement is applied using either the octagonal [22] or polyhedra [26] analysis as the invariant generation tool. The safety assertions in all these procedures can now be validated by running either the octagonal or the polyhedra analysis on the control-refined procedure. (Note that running these analyses on the original procedure would fail to validate any of these assertions with the exception that octagonal domain can validate the assertion in the last procedure.) For the first example, the (inductive) loop invariants $d = t \leq 3$ and $d = s \leq 2$ for the loops $\texttt{Repeat}(\rho_1)$ and $\texttt{Repeat}(\rho_2)$ respectively imply the assertion. The dotted portion denotes irrelevant code that does not contain any assertions.

For the second example, the loop invariant $x = y \wedge x \leq 50$ for the first loop $\texttt{Repeat}^+(\rho_1)$ helps establish $x = y = 51$ at the end of the loop; and then the loop invariant $x + y = 102 \wedge x \geq 52 \wedge y \geq 0$ for the second loop $\texttt{Repeat}(\rho_2)$ helps establish the desired assertion after the loop. For the third example, the loop invariant $x \leq 50 \wedge y = 50$ for the first loop helps establish $x = y = 50$ at the end of the first loop $\texttt{Repeat}^+(\rho_1)$, and the loop invariant $x = y \wedge x \leq 100$ for the second loop $\texttt{Repeat}^+(\rho_2)$ helps establish $y = 100$ at the end of the second loop. For the fourth example, the assertion is trivially established.

For the last example, the invariant $x \leq y$ gets established after the inner loop on ρ_2, which then propagates itself as the loop invariant for the outer loop too.

9. Related Work

Control Flow Refinement Control-flow refinement is related to other approaches that have been proposed for doing a more precise program analysis given an underlying invariant generation tool. This includes *widening strategies* (such as "look-ahead widening" [10] and "upto widening" [17]) or *disjunctive extensions* of domains [13, 25, 11, 14]. The primary goal of these techniques is to compute precise invariants at different program points in the original program, while we instead focus on creating a precise expansion of the program into one with simpler loops. Our technique is useful for bound computation, a process that is more effective for simple loops, as opposed to complex loops annotated with precise invariants. Other work [25] does a CFG elaboration, but only as a means to perform efficient computation over some refinement of the powerset extension. In particular k bounded elab-

Original Example	After Path Refinement

Halbwachs *et al.* **1997, P. 14, F. 7.**
`t:=0; d:=0; s:=0;`
`while (*)`
`  if (sec)`
`  s:=0;`
`  if (t++ = 4) break;`
`  if (met)`
`  if (s++ = 3) break;`
`  assert(d++ ≠ 10);`
	`t:=0, d:=0, s:=0;`
	`Choose({`
	`  asm(sec&&met);Repeat(`ρ_1`),`
	`  asm(¬sec&&met);Repeat(`ρ_2`),`
	`  asm(sec&&¬met); ....`
	`  asm(¬sec&&¬met); ....`
	`});...`
	where
	$\rho_1 \triangleq$ `s:=0;asm(t++ ≠ 4);`
	`asm(s++≠3);assert(d++≠10);`
	$\rho_2 \triangleq$ `asm(s++≠3);assert(d++≠10);`

Gopan and Reps 2006, P. 3, F. 1.
`x:=0, y:=0;`
`while (*)`
`  if (x ≤ 50) y++;`
`  else y--;`
`  if (y<0) break;`
`  x++;`
`assert(x=102)`
	`x:=0; y:=0;`
	`Repeat`$^+(\rho_1)$`;` ρ_2`; Repeat(`ρ_2`);`
	`if (x ≤ 50) y++;`
	`else y--;`
	`assume (`$y < 0$`);`
	`assert(x=102);`
	where
	$\rho_1 \triangleq$ `asm(x≤50);y++;asm(y≥0);x++;`
	$\rho_2 \triangleq$ `asm(x>50);y--;asm(y≥0);`
	`x++;`

Gulwani and Jojic 2007. P. 7, F. 3.
`x:=0; y:=50;`
`while (x<100)`
`  if (x<50) x++;`
`  else x++; y++;`
`assert(y=100);`
	`x:=0; y:=50;`
	`Repeat`$^+(\rho_1)$`;` ρ_2`; Repeat`$^+(\rho_2)$`;`
	`asm(x ≥ 100);`
	`assert(y=100);`
	where
	$\rho_1 \triangleq$ `asm(x<100&&x<50); x++;`
	$\rho_2 \triangleq$ `asm(x<100&&x≥50);x++;y++;`

Gulavani *et al.* **2006. P. 5, F. 3.**
Henzinger *et al.* **2002. P. 2, F. 1.**
`lock := 0;`
`assume (x ≠ y);`
`while (x ≠ y)`
`  lock := 1; x := y;`
`  if (*)`
`  lock := 0; y++;`
`assert(lock = 1);`
	`lock := 0; assume (x ≠ y);`
	`Choose({`
	`  Repeat`$^+(\rho_2);\rho_1$`,`
	`  Repeat`$^+(\rho_1)$
	`});`
	`asm(x = y);`
	`assert(lock = 1);`
	where
	$\rho_1 \triangleq$ `asm(x ≠ y); lock := 1; x := y;`
	$\rho_2 \triangleq$ `asm(x ≠ y); lock := 1; x := y;`
	`lock := 0; y++;`

Figure 10: Prominent disjunctive invariant challenges from recent literature. Our technique finds suitable disjunctive invariants for each example. For brevity, `assume(`c`)` is denoted `asm(`c`)`.

oration, wherein a program location is duplicated at most k times. In contrast, the expansion decision in our algorithm is independent of a different exploration branch. More significantly, our expansion granularity focuses on interleavings between different paths, as opposed to what happens in different iterations of the same path.

Part of our algorithm is based on pruning infeasible paths. Balakrishnan *et al.* [4] present a technique for finding infeasible paths with backward and forward analyses. Infeasible paths are removed from the transition system ("static language refinement"). The fundamental distinction of our work is that we prune infeasible *unwound* paths, rather than simple, statically occurring paths.

With respect to proving safety properties (as discussed in Section 8), our technique is more precise but less efficient than widening approaches (since CFG expansion allows our technique to effectively compute disjunctive invariants). Our technique is orthogonal to techniques based on disjunctive extensions (each of which is unique w.r.t. the number of disjuncts, and their merging).

Progress Invariants Our notion of progress invariants is related to *transition invariants* [24, 6, 21] or *variance analyses* [5] (recently described in literature for proving termination), which describe relationships between a state at a program location and *any other* (as opposed to *immediately*) previous state at that location. Hence, theoretically, progress invariants are more precise in the sense that transition invariants can be generated from progress invariants but not vice-versa.

While both transition invariants and progress invariants are used to measure how a program state evolves by studying relations over pairs of states (as opposed to just a single state), it is interesting to observe the differences in the methodologies involved. Transition invariants have been used for proving termination by computing disjunctive transition invariants and then showing that each disjunct is well-founded. In contrast, progress invariants are used for computing a bound after refining the program, which obviates the need for disjunctive invariants. We believe that the separation of concerns of dealing with non-regular (disjunctive) program behavior and relations on pairs of states enables our approach to go beyond proving termination (i.e. computing a precise bound). For example, it is trivial to prove the nested loop in Section 2.2 terminating, but computing a precise bound requires a more sophisticated machinery, such as our notion of progress invariants. On the other hand, our technique can compute bounds for programs for which existing termination techniques fail to even prove termination (e.g. the cyclic iteration example in Figure 1).

Symbolic Bound Computation Recent work by Gulwani *et al.* [15] describes elaborate counter instrumentation strategies for computing a bound on loop iterations using a linear arithmetic invariant generation tool. However, the strategy is not effective for multi-path loops, which seem to occur frequently in practice. For example, the strategy cannot compute a bound for any of the multi-path loops (except the 3rd example) in Figure 4. For nested loops, the strategy is not as precise as the ideas presented here (see further discussion in Section 2.2). Moreover, counter instrumentation works well only for arithmetic programs. To overcome this restriction, Gulwani *et al.* also introduced the notion of user-defined quantitative functions for data-structures (such as length of list, height of tree) and their updates to express and enable computation of bounds for loops that iterate over data-structures. But the strategy is still limited to computing polynomial bounds as opposed to, say, logarithmic (e.g. binary search) or square-root bounds. In contrast, since we do not use counter instrumentation, we need not arithmetize a program using quantitative functions; neither are we restricted to computing polynomial bounds. However, we do require that the underlying invariant generation tool support an additional interface of generating bounds from progress invariants.

Gulavani and Gulwani [12] have described the design of a rich numerical domain to generate non-linear disjunctive invariants, and they have applied it to generating bounds for timing analysis. However, it requires the user to describe *important expressions* (over which linear relationships are to be tracked) as well as the set of variables that should occur within a max operator for each loop. Furthermore, the technique only applies to arithmetic programs.

There is a large body of work on estimating worst case execution time (WCET) in the embedded and real-time systems community [27]. The WCET research is largely orthogonal, focused on distinguishing between the complexity of different code-paths and low-level modeling of architectural features such as caches, branch prediction, instruction pipelines. For establishing loop bounds, WCET techniques either require user annotation, or use simple techniques based on pattern matching [18] or some numerical analysis (e.g., relational linear analysis to compute linear bounds on the delay or timer variables of the system [17], interval analysis based

approach [16], and symbolic computation of integer points in a polyhedra [20]). These WCET techniques cannot compute precise bounds for the examples considered in this paper.

Goldsmith *et al.* [9] compute symbolic bounds by curve-fitting timing data obtained from profiling. Their technique has the advantage of measuring real amortized complexity; however the results are not sound for all inputs. Crary Weirich [8] presents a type system for certifying (as opposed to inferring) resource consumption, including time.

10. Conclusion and Future Work

We have introduced novel techniques for automatically determining symbolic complexity bounds of procedures. We first showed how *control-flow refinement* enables standard invariant generators to reason about mildly complex control-flow, which would otherwise require impractical disjunctive invariants. For example, we have proven symbolic complexity bounds for procedures which no previous technique can even prove termination. We then introduced *progress invariants* and showed how to use them to compute precise procedure bounds. Finally, our experience with a large Microsoft product illustrates the effectiveness of our techniques: our tool was able to find bounds for 90% of procedures that involve loops.

We believe that a productive direction for future work would be to study broader applications for control-flow refinement and progress invariants. As discussed in Section 8 there seem to be applications in proving safety as well as liveness properties.

Acknowledgments. We thank Matthew Parkinson and the anonymous reviewers for their valuable feedback which improved this paper. We also thank the product teams at Microsoft for their assistance with this project and acknowledge funding from a Gates scholarship (Koskinen).

References

[1] Phoenix Compiler. `research.microsoft.com/Phoenix/`.

[2] Z3 Theorem Prover. `research.microsoft.com/projects/Z3/`.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *SAS*, 2008.

[4] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. *Lecture Notes in Computer Science*, 5079, 2008.

[5] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Variance analyses from invariance analyses. In *POPL*, 2007.

[6] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[7] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[8] Karl Crary and Stephanie Weirich. Resource bound certification. In *POPL*, 2000.

[9] Simon Goldsmith, Alex Aiken, and Daniel Shawcross Wilkerson. Measuring empirical computational complexity. In *ESEC/SIGSOFT FSE*, 2007.

[10] Denis Gopan and Thomas W. Reps. Lookahead widening. In *CAV*, 2006.

[11] Denis Gopan and Thomas W. Reps. Guided static analysis. In *SAS*, 2007.

[12] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, 2008.

[13] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: A new algorithm for property checking. In *FSE*, 2006.

[14] Sumit Gulwani and Nebojsa Jojic. Program verification as probabilistic inference. In *POPL*, 2007.

[15] Sumit Gulwani, Krishna Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[16] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, 2006.

[17] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *FMSD*, 1997.

[18] Christopher A. Healy, Mikael Sjodin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3), 2000.

[19] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, 2002.

[20] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *WCET*, 2003.

[21] Ali Mili. Reflexive transitive loop invariants: A basis for computing loop functions. In *WING*, 2007.

[22] Antoine Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.

[23] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. *LNCS*, 2003.

[24] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.

[25] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *SAS*, 2006.

[26] Chao Wang, Zijiang Yang, Aarti Gupta, and Franjo Ivancic. Using counterexamples for improving the precision of reachability computation with polyhedra. In *CAV*, 2007.

[27] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. In *TECS*, 2007.