The Reachability-Bound Problem

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Florian Zuleger *
TU Darmstadt
zuleger@forstye.cs.tu-darmstadt.de

Abstract

We define the *reachability-bound problem* to be the problem of finding a symbolic worst-case bound on the number of times a given control location inside a procedure is visited in terms of the inputs to that procedure. This has applications in bounding resources consumed by a program such as time, memory, network-traffic, power, as well as estimating quantitative properties (as opposed to boolean properties) of data in programs, such as amount of information leakage or uncertainty propagation.

Our approach to solving the reachability-bound problem brings together two very different techniques for reasoning about loops in an effective manner. One of these techniques is an abstract-interpretation based iterative technique for computing precise disjunctive invariants (to summarize nested loops). The other technique is a non-iterative proof-rules based technique (for loop bound computation) that takes over the role of doing inductive reasoning, while deriving its power from use of SMT solvers to reason about abstract loop-free fragments.

Our solution to the reachability-bound problem allows us to compute precise symbolic bounds for several loops in .Net base-class libraries for which earlier techniques fail. We also illustrate the precision of our algorithm for disjunctive invariant computation (which has a more general applicability beyond the reachability-bound problem) on a set of benchmark examples.

1. Introduction

Program execution makes use of physical resources, and it is often important to compute worst-case bounds on usage of those resources as a function of the program inputs. For example, in memory-constrained environments such as embedded systems, it is important to bound the amount of *memory* required to run certain applications. In real-time systems, it is important to bound the worst-case *execution-time* of the program. Similarly, the applications running on low-power devices or low-bandwidth environments must use up little *power* or *bandwidth* respectively. One of the fundamental questions that need to be answered in these cases is: How many times is a given control-location inside the program that consumes these resources executed?

* The research of the second author was supported in part by Microsoft Research through its PhD Scholarship Programme.

Program execution also affects certain quantitative properties of data that it operates on. For example, the amount of *secret* leaked by a program depends on the number of times a certain operation that leaks the data, either by direct or indirect information flow, is executed [18]. Or the amount of *perturbation* in the output data values resulting from a small perturbation or uncertainty in the input values depends on the number of times additive error propagation operators are applied. This is the quantitative version of the boolean problem of continuity studied in [6]. Estimating such quantitative properties again requires addressing a similar question as above: How many times is a given control-location inside the program that performs certain operations executed?

We refer to the problem of bounding the number of times a given control-location π is visited as the *reachability-bound problem*. We present a two-step solution to this problem that brings together two very different techniques for reasoning about loops: an iterative technique for computing disjunctive invariants, and a non-iterative proof-rule based technique for computing bounds.

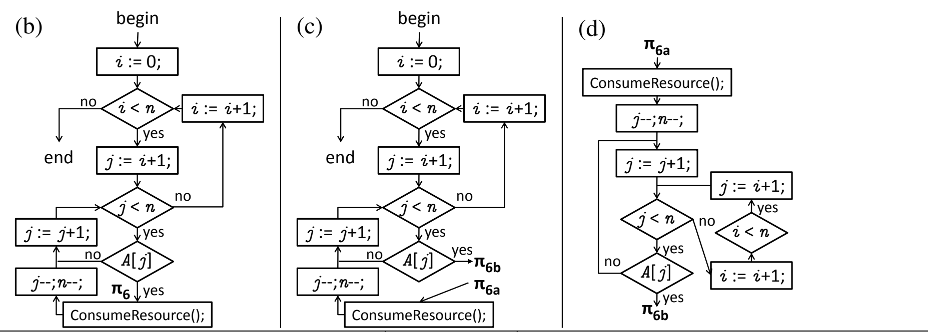
The first step consists of generating a disjunctive *transition-system* that describes relationships between values of program variables that are live at π and their values in the immediately next visit to π. This requires summarizing inner loops that lie on a path from π back to itself for which we present an abstract interpretation based iterative algorithm that generates disjunctive loop invariants. The precision of our algorithm relies on the *convexity-like assumption*, which appears satisfied by all instances that we came across in practice, and leads to an interesting completeness theorem (Theorem 1). We also experimentally evaluate the precision of this algorithm on benchmark examples taken from recent work on computing disjunctive invariants. Our algorithm can discover the required invariants in all examples, suggesting its potential for effective use in other applications requiring disjunctive invariants besides our application of bound analysis.

The second step consists of generating bounds for the disjunctive transition-system thus generated. For this, we propose non-iterative proof-rules based technique that requires discharging queries using an off-the-shelf SMT solver. These proof rules describe conditions that are sufficient for combining the ranking functions for individual transitions (to obtain bound for the transition-system consisting of those transitions) using fundamentally different mathematical operators, namely max, sum, and product. This is unlike existing work on termination analysis where the goal is to generate any ranking function for a transition-system with disregard to the precision of the ranking function. This methodology represents an interesting design choice for reasoning about loops, wherein SMT solvers are used to perform precise reasoning about transitions (loop-free code-fragments) thereby allowing a simple pattern-matching and proof-rules based technique to take over the role of performing inductive reasoning effectively. It will be interesting to consider applying such a methodology to other problems.

Contributions and Organization

(a)

```
   Ex1(int n, bool[] A)
1  i := 0;
2  while (i < n)
3      j := i + 1;
4      while (j < n)
5          if (A[j])
6              ConsumeResource();
7              j--; n--;
8          j := j + 1;
9      i := i + 1;
```



(e)

Transition-system T of inner loop in flowgraph (d) : $(j \geq n \;\wedge\; i < n-1 \;\wedge\; i' = i+1 \;\wedge\; j' = i+2)_{s_1}$

Transitive Closure(T): $(i' = i \;\wedge\; j' = j)_{s_1'} \;\vee\; (j \geq n \;\wedge\; i < n-1 \;\wedge\; i' \geq i+1 \;\wedge\; j' \geq i+2)_{s_2'}$

Transition-system T' of outer loop in flowgraph (d): $(j < n-1 \;\wedge\; j' = j+1 \;\wedge\; i' = i)_{s_1} \;\vee\; (i < n-1 \;\wedge\; i' \geq i+1 \;\wedge\; j' \geq i+2)_{s_2}$

Transitive Closure(T'): $(j' \geq j \;\wedge\; i' = i)_{s_1'} \;\vee\; (i < n-1 \;\wedge\; i' \geq i+1 \;\wedge\; j' \geq i+2)_{s_2'}$

Transition-system(π_6): $(n' = n-1 \;\wedge\; j < n-1 \;\wedge\; j' \geq j \;\wedge\; i' = i)_{s_3} \;\vee\; (n' = n-1 \;\wedge\; i < n-1 \;\wedge\; i' \geq i+1 \;\wedge\; j' \geq i+2)_{s_4}$

Figure 1. (a) A loop skeleton from .Net base-class library. (b) Flow-graph representation of the program in (a). (c) Flow-graph obtained from (b) by splitting location π_6 into π_{6a} and π_{6b}. (d) Part of the flow-graph(c) between π_{6a} and π_{6b} after re-drawing it. (e) Various steps involved in computation of the transition-system for location π_6 in flowgraph (b) from the flowgraph (d). Some transitions in (e) are named by a subscript (e.g., s_1') for future reference in the paper.

- We define the reachability-bound problem and the notion of a precise solution to that problem (Section 3). This can be an interesting point in the space of defining an entire quantitative logic (part of the *quantitative agenda*, as opposed to the Boolean agenda, set forth recently [17]).
- We describe a transition-system generation algorithm based on reducible flowgraph transformations for reducing the problem of reachability-bound computation to the problem of computing bound for a transition-system (Section 4).
- We describe an abstract-interpretation based iterative algorithm for computing transitive closure of a transition-system, or, equivalently, disjunctive invariants for a loop. (Section 5).
- We describe non-iterative proof rules (Section 7) that allow computation of precise symbolic bounds for a transition-system from the ranking functions of individual transitions, which can be obtained using techniques described in Section 6.
- We present experimental results evaluating the effectiveness of various aspects of our solution (Section 8).

2. Motivating Examples and Technical Overview

In this section, we discuss some examples that are representative of some challenges that arise during computation of symbolic bounds for the reachability-bound problem. We also provide a technical overview of our solution.

2.1 Bounding number of visits to a given control location

Consider the loop template shown in Figure 1, and consider the problem of computing symbolic bounds on the number of times the procedure ConsumeResource() is called at Line 6. One approach would be to approximate it by computing a bound on the number of iterations of the closest enclosing loop at Line 4 using techniques for loop bound computation (as in [13, 15]). However, this approach will yield quite conservative results since the number of iterations of the loop at Line 4 is bounded above by n^2, while the number of executions of Line 6 is bounded above by n.

Our approach first computes the following symbolic relationship between values of variables i, j, n at Line 6 with their values i', j', n' in the immediately next visit to Line 6. The relationship is expressed as a disjunction of two transitions s_3 and s_4.

$$(n' = n-1 \;\wedge\; j < n-1 \;\wedge\; j' \geq j \;\wedge\; i' = i)_{s_3}$$
$$\vee\; (n' = n-1 \;\wedge\; i < n-1 \;\wedge\; i' \geq i+1 \;\wedge\; j' \geq i+2)_{s_4}$$

This is done using the GenerateTransitionSystem algorithm described in Figure 4 in Section 4. The algorithm enumerates all paths in the control-flow graph (in Figure 1(d)) between locations π_{6a} and π_{6b} obtained after splitting the location π_6 in the original control-flow graph (in Figure 1(b)) into π_{6a} and π_{6b}. (Note that such a relationship is different from transition invariants [23] or variance assertions [3] that relate values of variables at a control-location with their values in *any* successive iteration, as opposed to the immediately next iteration). The challenge in such an enumeration is that the number of paths in presence of loops between control-locations π_{6a} and π_{6b} is not finite. For this purpose, the loops are summarized by disjunctive relationships between the inputs/outputs of the loop. These disjunctive relationships are generated by computing the transition-system of the loop (recursively, using the same algorithm applied to the control-location immediately after the loop-header) and then computing its transitive closure (using the algorithm TransitiveClosure described in Fig. 6 in Section 5). The transition-systems of the two loops in Figure 1(d) and their respective transitive closures are shown in Figure 1(e).

Next, bounds are computed for the transition-system thus generated. This involves computing the individual ranking functions of $n-1-j$ and $n-1-i$ for the two transitions s_3 and s_4 respectively. The ranking functions are computed using pattern matching technique described in Section 6. These ranking functions are then composed using one of the proof rules described in Section 7 (in this case, the proof rule in Theorem 3) to obtain a bound of $\mathtt{Max}(0, n-1-j, n-1-i)$ in terms of the inputs to the transition-system (For details, see Example 6). Using the invariants $i \geq 0 \wedge j \geq 1$ that hold during the first visit to π_6) (which can be obtained by generating invariants at control-location π_{6b} in Figure 1(d)), we obtain a bound of $n-1$ on the transition-system in terms of procedure inputs. This implies a bound of n on the number of visits to control-location π_6.

2.2 Bounding iterations of a loop

Computing bounds on number of loop iterations is a special case of the reachability-bound problem where the control location under consideration is the location immediately after the loop header.

`Ex2(uint n, uint m)`	`Ex3(uint n, bool[] A)`	`Ex4(uint n)`	`Ex5(uint n)`

`while (n > 0 ∧ m > 0)`
`    n--; m--;`
`    while (nondet())`
`        n--; m++;`
	`while (n > 0)`
	`    t := A[n];`
	`    while (n > 0 ∧ t = A[n])`
	`        n--;`
		`flag := true;`
		`while (flag)`
		`    flag := false;`
		`    while (n > 0 ∧ nondet())`
		`        n--; flag := true;`
			`i := 0;`
			`while (i < n)`
			`    flag := false;`
			`    while (nondet())`
			`        if (nondet()){flag:=true;n--;}`
			`    if (¬flag) i++;`

$n' \leq n \ \wedge \ m' \geq m$	$n' \leq n$	$(n' \leq n-1 \ \wedge \ \texttt{flag}')$ $\vee(\texttt{Same}(\{n, \texttt{flag}\}))$	$(n' \leq n\text{-}1 \ \wedge \texttt{flag}' \ \wedge \ i' = i)$ $\vee(\texttt{Same}(\{i, n, \texttt{flag}\}))$
$n{>}0 \wedge m{>}0 \wedge n'{\leq}n\text{-}1$	$(n{>}0 \wedge n'{\leq}n \wedge A[n]{\neq}A[n'])$ $\vee \ (n{>}0 \wedge n'{\leq}0)$	$(\texttt{flag} \wedge \texttt{flag}' \wedge n{>}0 \wedge n'{\leq}n\text{-}1)$ $\vee(\texttt{flag} \wedge \neg\texttt{flag}' \wedge n' = n)$	$(i{<}n \ \wedge \ \texttt{flag}' \ \wedge \ n'{\leq}n{-}1 \wedge i'{=}i)$ $\vee\, (i{<}n \ \wedge \ \neg\texttt{flag}' \wedge i'{\geq}i{+}1 \ \wedge \ n'{=}n)$
n	n	$n+1$	n

Figure 2. Loop templates from .Net class libraries where iterators of a loop are modified by inner loops. The second row shows the required transitive closure of inner loops to enable precise symbolic bound computation of respective outer loops. The third row shows the resultant transition-system generated for the outer loops after summarizing the respective inner loops by the transitive closure of their transition-system (using the algorithm in Figure 4). The final (fourth) row shows the bound computed from the transition-system by the algorithm in Figure 7. We use the predicate $\texttt{Same}(V)$ inside a transition to denote that the variables in V do not change their value, i.e., $\texttt{Same}(V) = \bigwedge_{x \in V} (x' = x)$.

`Ex6(int n, int x, int z)`	`Ex7(uint n, uint m)`

`while (x < n)`
`    if (z > x) x++;`
`    else z++;`
	`Assume(0 < n < m);`
	`j := n + 1;`
	`while (j < n ∨ j > n)`
	`    if (j > m) j := 0;`
	`    else j++;`

Figure 3. Loop templates Ex6 and Ex7 (from Microsoft product-code) taken respectively from recent work on proving termination [7] and loop bound computation [13]. Our proof rules for bound computation provide an alternative, but simpler, formalism for computing bounds. (For details see Example 7 and Example 9.)

Under that case, our technique outperforms recent techniques for loop bound computation and termination.

In particular, our technique is able to compute bounds for loops whose iterations is affected by inner loops for which existing bound techniques (such as [13, 15]) mostly fail (For details, see related work in Section 9). Such loops are quite common in .Net base-class library, and Figure 2 gives some examples. One of the key challenge addressed by our technique in such examples is the summarization of the inner loops by precise transitive-closure of the transition-system represented by these loops (in effect, disjunctive relationships between the inputs and outputs of the loop).

Also, even in case of loops with no nested loops, our technique is able to compute bounds for loops using a much simpler uniform algorithm compared to existing termination techniques or specialized bound computation techniques. Figure 3 shows two such examples that have been used as motivating examples by previous techniques. The computation of the transition-system for these examples is almost trivial, and the bound computation of the resultant transition-system is enabled by simple but precise proof rules (Theorem 4 and Theorem 5) for bound computation from ranking functions of individual transitions (For details, see Example 7 and 9).

3. Reachability-Bound Problem

There are two classical problems associated with reachability of a control-location π inside a procedure P with inputs $\vec{n}$.

- Safety Problem: Is control-location π never reached/visited?
- Liveness Problem: Is control-location π visited at most a finite number of times?

In this paper, we have motivated the following bound problem, which is different from the safety and liveness problems (for the simple reason that it is not a boolean problem).

- Bound Problem: Compute a worst-case symbolic bound $\mathcal{B}(\vec{n})$ on number of visits to control-location π for any execution of P.

The notion of a worst-case symbolic bound is defined below.

DEFINITION 1 (Worst-case symbolic bound). *An integer-valued function $\mathcal{B}(\vec{n})$ is a worst-case symbolic bound for a control-location π inside a procedure P with inputs $\vec{n}$ if for any input state $\vec{n_0}$, the number of times π is visited is at most $\mathcal{B}(\vec{n_0})$.*

There may be multiple worst-case symbolic bounds for a given problem. It is desirable to produce a bound that is precise in the sense that there exists a family $\phi(\vec{n})$ of worst-case inputs that exhibit the worst-case bound (up to some constant factor, as motivated by the definition of asymptotic complexity) formally defined as follows.

DEFINITION 2 (Precision of a worst-case symbolic bound). *A worst-case symbolic bound $\mathcal{B}(\vec{n})$ for a control-location π inside a procedure P with inputs $\vec{n}$ is said to be* precise *(up to multiplicative constant factors) if there exist positive integers c_1, c_2, and a formula $\phi(\vec{n})$ such that:*

E1. For any assignment $\vec{n_0}$ to variables $\vec{n}$ such that $\phi(\vec{n_0})$ holds, the number of times control-location π is visited (when procedure P is executed in the input state $\vec{n_0}$) is at least $\frac{\mathcal{B}(\vec{n_0})}{c_1} - c_2$.

E2. For any integer k, there exists a satisfying assignment $\vec{n_1}$ for $\phi(\vec{n})$ such that $\mathcal{B}(\vec{n_1}) > k$. In other words, the formula $\exists \vec{n} : (\mathcal{B}(\vec{n}) \geq k \wedge \phi(\vec{n}))$ has a satisfying assignment.

We refer to the triple (ϕ, c_1, c_2) as precision-witness *for bound $\mathcal{B}$.*

The following example explains and motivates the requirements E1 and E2 in the above definition.

EXAMPLE 1. *A precision-witness for bound of n on the number of times Line 6 is visited in Example Ex1 in Figure 1 can be $\phi = \forall k(0 \leq k < n \Rightarrow A[k])$, $c_1 = 1$ and $c_2 = 1$ since it can be shown that under the precondition ϕ, Line 6 is visited at least $n - 1$ times.*

A precision-witness for bound of n^2 on the number of times the inner loop (Line 5) is executed can be $\phi = \forall k(0 \leq k < n \Rightarrow \neg A[k])$, $c_1 = 4$ and $c_2 = 1$ since it can be shown that under the precondition ϕ, Line 5 is visited at least $n^2/4$ times. This is because, for example, i takes all values between 0 to $n/2 - 1$ at Line 2 (hence the number of visits to Line 2 is at least $n/2$), and for each of those visits, j takes all values between $n/2$ to $n - 1$ at Line 4 (i.e., the number of visits to Line 4 is at least $n/2$). Note that *if we did not relax the requirement E1 to allow for constants c_1 and c_2, then computation of a precise bound would have required us to compute the exact bound of $\frac{(n-1)(n-2)}{2}$. It would be impractical to find such exact closed-form solutions.*

A bound of, say, 100, on the number of times Line 6 is visited is not precise. It may appear that $\phi = (\forall k(0 \leq k < 100 \Rightarrow A[k]) \wedge n \leq 100)$, $c_1 = 1$ and $c_2 = 1$ is a precision-witness. However, note that it violates requirement E2 since for $k = 101$ (in fact, for any k greater than 100), there does not exist a satisfying assignment for the formula $\phi \wedge 100 \geq 101$.

In this paper, we describe an algorithm for computing a worst-case symbolic bound. Manual investigation of the bounds returned by our algorithm on our benchmark examples confirms that the bounds are precise (up to small multiplicative constants).

Automatically establishing the precision of a bound $\mathcal{B}$ returned by our algorithm is an orthogonal problem that we are currently working on. It requires identifying a precision-witness (ϕ, c_1, c_2) and establishing that $\frac{\mathcal{B}}{c_1} - c_2$ is a *lower* bound for all inputs that satisfy ϕ. The duality between the problems of computing a symbolic bound $\mathcal{B}$ and the problem of finding a witness ϕ to show that $\mathcal{B}$ is precise is similar to the duality between the problems of proving a given safety property, or finding a concrete counterexample/witness to the violation of a safety property. However, the challenge in our case is that the witness ϕ that establishes the precision of a given symbolic bound is symbolic as opposed to being concrete.

We next describe our overall algorithm for bound computation.

3.1 Algorithm

Our algorithm for the reachability-bound problem is as follows.

```
  ReachabilityBound(π)
1   T := GenerateTransitionSystem(π);
2   B := 1 + ComputeBound(T);
3   return TranslateBound(B, π);
```

Line 1 of the algorithm first computes a disjunctive transition-system T for the control location π that describes how the variables at π get updated in the immediately next visit to control-location π. This is done using the algorithm described in Figure 4 (Section 4), which in turn uses the algorithm for transitive closure computation described in Figure 6 (Section 5) to summarize any inner loops.

Line 2 of the algorithm computes a bound $\mathcal{B}$ for the transition-system T using the algorithm described in Figure 7 (Section 7), which in turn makes use of techniques described in Section 6 for computing ranking functions of individual transitions. The bound on number of visits to π is then given by $1 + \mathcal{B}$.

The bound $\mathcal{B}$ is expressed in terms of inputs to the transition-system, which may not necessarily be the procedure inputs. The function `TranslateBound` at Line 3 then translates the bound $\mathcal{B}$ at π in terms of the procedure inputs. This can be either done by using invariants (computed from an invariant generation tool) that relate the procedure inputs with the inputs to the transition-system T, or by using a backward symbolic engine to express the transition-system inputs in terms of the procedure inputs. We implemented the latter approach, which be found to be extremely effective in terms of both precision and efficiency. This technique is detailed in **??**.

4. Generation of Transition-System

We first define the notion of a transition-system and a transition.

DEFINITION 3 (Transition-System for a Control Location π). *Let $\vec{x}$ be the tuple of variables that are live at π. A transition-system for π is a relation $T(\vec{x}, \vec{x'})$ between variables $\vec{x}$ and their primed counterparts $\vec{x'}$ such that if $\vec{x}$ take values $\vec{v_1}$ and $\vec{v_2}$ during any two immediately successive/consecutive visits to π, then $T(\vec{v_1}, \vec{v_2})$ holds. Furthermore, a transition-system is always represented in DNF form as a disjunction of* transitions *s, where each transition s is a conjunctive relation over variables $\vec{x}$ and $\vec{x'}$.*

```
  GenerateTransitionSystem(π)
1 (π_a, π_b) := Split(π);
2 foreach top-level loop L:
3    π_L := location before header of L.
4    T := GenerateTransitionSystem(π_L)
5    T_c := TransitiveClosure(T);
6    Insert Summary(T_c) before header; Remove back-edges.
7 Initialize F[π_a] to the transition-system Id.
8 Propagate transitions F using Merge/Compose rules.
9 return F[π_b]
```

Figure 4. Generation of transition-system for a control location π.

We desire a disjunctive representation for our transition-system since our bound computation algorithm in Section 7 works by identifying precise ranking functions for a single transition/path, and then using proof rules to obtain the ranking function/bound for the entire transition-system.

The key idea for generating a transition-system for a control location π is to split the control location π into two locations (π_a, π_b) (using the `Split` transformation shown in Figure 5(a)) and enumerate all paths that start at π_a and end at π_b and take the disjunctions of the transitions represented by each path. The challenge that arises in such an enumeration is the presence of any nested loops. We address this challenge by replacing the nested loop by the transitive closure of the transition-system of the nested loop (using the `Summarize` transformation shown in Figure 5(b)). Since path enumeration leads to an exponential blowup, we generate transition-system on the flowgraph that has been sliced with respect to the statements on which π is control-dependent [20] (since these are the statements that determine the number of times π is executed). This usually leads to transition-systems with a very small number of transitions, as is exemplified by the statistics in Figure 8 in Section 8.1.

Figure 4 describes the algorithm for generation of transition-system for a control location π. The algorithm is described at flowgraph level. We make the assumption about the flowgraphs being reducible, but not necessarily structured. Our algorithm can be extended to irreducible flowgraphs too; but we avoid that for ease of presentation, and the fact that most flowgraphs in practice are in fact reducible [20]. However, it is important to consider the case of unstructured flowgraphs because even if the original flowgraph was structured, after the splitting transformation, the new flowgraph would no longer be structured. The splitting transformation, however, is reducibility-preserving. [1]

Line 1 transforms the flowgraph by splitting the input control-location π into two locations π_a and π_b using the `Split` transformation described in Figure 5(a). The loop in Line 2 iterates over each top-level loop L in the transformed flowgraph. (Recall that any graph can be decomposed into a DAG of maximal strongly-connected components.) Line 3 makes use of the fact that every loop in a reducible flow-graph has a unique header node. Line 4 recursively generates the transition-system for the loop L in the transformed flow-graph, while Line 5 generates its transitive closure (using the algorithm described in Figure 6 in Section 5). Line 6 replaces the loop L by its summary obtained by generating transitive closure of the transition-system represented by it (using the `Summarize` transformation shown in Figure 5(b)). The effect of the

[1] It is interesting to observe that the nesting structure of the loops inside which π was originally nested, is completely reversed after the splitting transformation, but the flowgraph stays reducible.

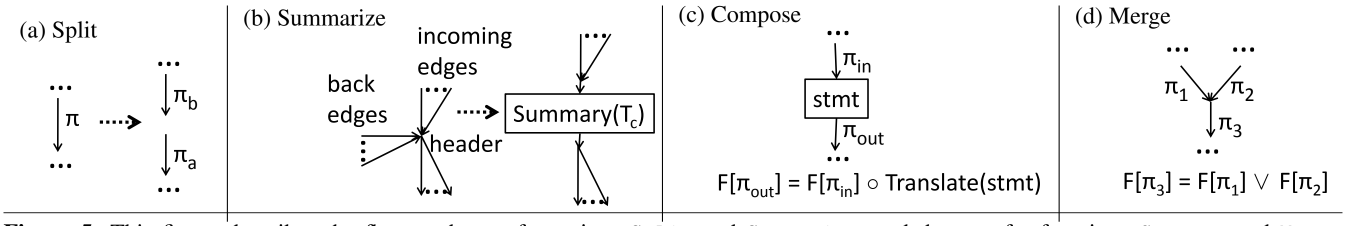


Figure 5. This figure describes the flowgraph transformations `Split` and `Summarize`, and the transfer functions `Compose` and `Merge` required in the algorithm `GenerateTransitionSystem` for computing the transition-system for any control location.

foreach-loop in Line 2 is to replace all loops on the paths between π_a and π_b by (disjunctive) loop-free abstract code-fragments. The transition-system can now simply be generated by enumerating all paths (which are now finite in number) between π_a and π_b.

Lines 7-9 generate the transition-system for an acyclic flowgraph by a simple forward dataflow analysis that associates a (disjunctive) transition-system $F[\pi]$ with each edge/control-location π in the transformed flowgraph. For this purpose, we associate the entry location π_a with the transition-system consisting of a single transition `Id`, which is the identity mapping between the variables and their primed versions. The transfer functions for performing this dataflow analysis are described in Figure 5. Without loss of any generality, lets assume that all conditional guards have been translated into `Assume` statements. The `Merge` transfer function simply returns the disjunctions of the transitions in the two input transition-systems. The `Compose` transfer function makes use of the compose operator $\circ$ that returns the composition of two transitions.

DEFINITION 4 (Composition of Transition-Systems). *The binary composition of two transition-systems $T(\vec{x},\vec{x'}) = \bigvee_i s_i$ and $T'(\vec{x},\vec{x'}) = \bigvee_j s'_j$, denoted by $T \circ T'$, is $\bigvee_{i,j} s_i \circ s'_j$, where $s_i \circ s'_j$ denotes the following transition.*

$$s_i(\vec{x},\vec{x'}) \circ s'_j(\vec{x},\vec{x'}) \stackrel{def}{=} \exists \vec{x''} \left(s_i[\vec{x''}/\vec{x'}] \wedge s'_j[\vec{x''}/\vec{x}] \right)$$

where $s_i[\vec{x''}/\vec{x'}]$ denotes the substitution of $\vec{x'}$ by $\vec{x''}$ in s_i.

The `Translate` function converts a statement into a transition-system as follows. Without loss of any generality, we assume that the only assignment statement is of the form $x := e$ since memory can be modeled using `Select` and `Update` expressions. The other kinds of statements can be either an `Assume` statement (obtained from the conditional guards) or `Summary` statement (obtained from summarization of nested loops).

$$\begin{array}{rcl} \texttt{Translate}(x := e) & = & (x' = e) \wedge (\bigwedge_{y \neq x} y' = y) \\ \texttt{Translate(Assume(guard))} & = & \texttt{Id} \wedge \texttt{guard} \\ \texttt{Translate(Summary}(T)) & = & T \end{array}$$

EXAMPLE 2. *The transition-system for control-location π_6 in Figure 1(b) is shown in Figure 1(e) along with the various steps required to obtain it from the flowgraph in Figure 1(d). These include computing the transition-system for the inner loop and then replacing the inner loop by its transitive closure. Next, the process is repeated for the outer loop.*

5. Computation of Transitive Closure

In this section, we describe an algorithm for computing a transitive closure (defined below) of a transition-system. This operation is required by the `GenerateTransitionSystem` algorithm described in Figure 6 in the previous section.

DEFINITION 5 (Transitive Closure). *We say that $T'(\vec{x},\vec{x'})$ is a* transitive closure *of a transition-system $T(\vec{x},\vec{x'})$ if*

$$Id \Rightarrow T' \qquad \textit{and} \qquad T' \circ T \Rightarrow T'$$

EXAMPLE 3. *Figure 2(e) provides an example of a transition-system T and its transitive closure. Note that $i' \geq i$ is another choice for the transitive closure for T. However, it is not as precise as one shown in Figure 2(e), and would lead to generation of a transition-system for location π_6 for which no bound exists.*

Generating transitive closure of a transition-system is like computing invariants for a loop representing the transition-system. Example 3 suggests the importance of these invariants to be precise, and hence disjunctive. There has been some work on discovering disjunctive invariants in general. We present below a technique that takes advantage of its particular application to bound analysis. (We also remark that our technique can be used in general for proving safety properties of programs. In Section 8.2, we present preliminary results that demonstrate the effectiveness of our technique on a set of benchmark examples taken from a variety of recent literature on generating disjunctive invariants.)

Our algorithm for computation of precise transitive closures is inspired by a *convexity-like* assumption that we found to hold true for all examples that we have come across in practice. (This includes the desired transitive closure of the transitions-systems of nested loops to compute precise bounds, as well as the benchmarks considered by previous work on computing disjunctive invariants.)

Recall that a theory is said to be *convex* iff for every quantifier-free formula ϕ in that theory, if ϕ implies a disjunction of equalities, then it implies one of those equalities, i.e.,

$$\left(\phi \Rightarrow \left(\bigvee_i (x_i = y_i) \right) \right) \implies \left(\bigvee_i (\phi \Rightarrow (x_i = y_i)) \right) \qquad (4)$$

Now, if $\bigvee_{j=1}^{m} s'_j$ is a transitive closure of $\bigvee_{i=1}^{n} s_i$, then from the definition of the transitive closure, it follows that for all $i \in \{1,..,n\}$ and $j \in \{1,..,m\}$, the following holds:

$$Id \Rightarrow \bigvee_{k=1}^{m} s'_k \qquad \text{and} \qquad s'_j \circ s_i \Rightarrow \bigvee_{k=1}^{m} s'_k$$

After distributing implication over disjunctions in the above equations (in a manner similar similar to in Equation 4), we obtain the *convexity-like* assumption, which is defined formally below.

DEFINITION 6 (Convexity-like Assumption). *Let $T' = \bigvee_{j=1}^{m} s'_j(\vec{x},\vec{x'})$ be a transitive closure for a transition-system $T = \bigvee_{i=1}^{n} s_i(\vec{x},\vec{x'})$, where each s_i and s'_j is a conjunctive relation. We say that the transitive closure $\bigvee_j s'_j$ satisfies the* convexity-like assumption *if there exists an integer $\delta \in \{1,..,m\}$ a map $\sigma : \{1,..,m\} \times \{1,..,n\} \mapsto \{1,..,m\}$, such that for all $i \in \{1,..,n\}$ and*

```
TransitiveClosure( ⋁_{i=1}^{n} s_i )
  for j ∈ {1,..,m} − {δ}: s'_j := false;
  s'_δ := Id;
  do {
       for i ∈ {1,..,n} and j ∈ {1,..,m}:
            s'_σ(j,i) := Join(s'_σ(j,i), s'_j ∘ s_i)
  } while any change in ⋁_{j=1}^{m} s'_j
  return ⋁_{j=1}^{m} s'_j;
```

Figure 6. Transitive closure computation of a transition-system.

$j \in \{1,..,m\}$, *the following holds:*

$$Id \Rightarrow s'_\delta \qquad \textit{and} \qquad (s'_j \circ s_i) \Rightarrow s'_{\sigma(j,i)}$$

The tuple (δ, σ) *is referred to as the* convexity-witness *of* $\bigvee_{j=1}^{m} s'_j$.

The convexity-like assumption essentially implies that no case-split reasoning is needed to prove inductiveness of transitive closure.

EXAMPLE 5. *All the transitive closures of the respective transition-systems described in Figure 1(e) and Figure 2 satisfy the convexity-like assumption. For example, the convexity-witness for the transitive closure of the transition-system* T *shown in Figure 1(e) is* $\delta = 1$ *and* $\sigma = \{(1,1) \mapsto 2, (2,1) \mapsto 2\}$. *The convexity-witness for the transitive closure of the transition-system* T' *shown in Figure 1(e) is* $\delta = 1$ *and* $\sigma = \{(1,1) \mapsto 1, (2,1) \mapsto 2, (1,2) \mapsto 2, (2,2) \mapsto 2\}$.

Given the convexity-witness (δ, σ) of any transitive-closure T' (that satisfies the convexity-like assumption) of a transition-system T, the algorithm in Figure 6 describes a way to compute a transitive closure that is at least as precise as T'. This property (stated formally in the following theorem) is quite significant in light of the fact that discovering disjunctive invariants has been quite a challenging task in literature and several merging heuristics based on semantics of the constituent dataflow facts have been suggested. The following theorem states the remarkable result that a semantic merging criterion cannot be better than a static syntactic criterion for merging data-flow facts.

THEOREM 1 (Precision of `TransitiveClosure` Algorithm). *Let* $\bigvee_{j=1}^{m} s''_j$ *be any transitive closure of a given transition-system* $\bigvee_{i=1}^{n} s_i$ *that satisfies the convexity-like assumption. Given the number of disjuncts* m *and the convexity-witness* (δ, σ)*, algorithm in Figure 6 outputs a transitive closure that is at least as precise as* $\bigvee_{j=1}^{m} s''_j$.

PROOF: We can prove that $s'_j \Rightarrow s''_j$ by induction on the number of loop iterations; the base case as well as the inductive case both follows easily from the definition of *convexity-like* assumption. □

The algorithm in Figure 6 performs abstract interpretation over the power-set extension of an underlying abstract domain (such as Polyhedra [8], Octagons [19], Conjunctions of a given set of predicates), where elements are restricted to at most m disjuncts. It uses the map σ to determine how to merge the $n \times m$ different disjuncts (into m disjuncts) that are obtained after propagation of m disjuncts across n transitions. The key distinguishing feature of the algorithm from earlier work on computing disjunctive invariants is that our algorithm uses a syntactic criterion based on σ to merge disjuncts as opposed to using semantic criterion based on notion of differences between disjuncts. This is justified by Theorem 1, which, in effect, says that no semantic merging criterion can be more powerful than a static syntactic criterion. There are two issues with the algorithm presented in Figure 6 that we discuss below.

Abstract Domains with Infinite Height The algorithm may not terminate when computing `Join` over domains with infinite height. The standard solution would be to replace apply `Widen` operator in place of `Join` operator, say after every 3 iterations of the loop. Since funny things are known to happen with widening, it is no longer possible to formally prove precision results as in Theorem 1. However, we show experimentally (in Section 8.2) that our algorithm is able to compute precise enough invariants with use of standard widening techniques when applied on benchmarks taken from recent work on computing disjunctive invariants.

Choice of m ***and the convexity-witness*** (δ, σ) Since we do not know the desired transitive closure and its convexity-witness (δ, σ) upfront, we have two options.

- We can enumerate all possible (δ, σ) for a specifically chosen m. There are m^{mn} such possible maps since without loss of any generality, we can assume that δ is 1. If m and n are small constants, say 2 (which is quite often an important special case), then there are 16 possibilities. Each choice for σ and δ results in some transitive closure computation by the algorithm. One can then select the strongest transitive closure among the various transitive closures thus obtained (or heuristically select between incomparable transitive closures). However, if m or n is large, then this approach quickly becomes prohibitive.
- We can use some heuristics to construct m, δ, σ. The following heuristic turns out to be the most effective for our application of bound computation. We choose m to be $n+1$, and δ to be m for all and select the map σ from the DAG of dependencies between transitions of T generated from bound computation of T (as described in Section 7). In particular, for any $i, j \in \{1,..,n\}$, we define $\sigma(n+1, i) := i$, $\sigma(i,i) := i$, and $\sigma(i,j) := i$ except when $\neg\texttt{NI}(s_j, s_i, r)$ (where $r \in \texttt{RankC}(s_i)$ was the ranking function that contributed to the bound computation of T) in which case we define $\sigma(i,j) := j$. It can be proved that such a choice of the map δ and σ would generate a transitive closure that would allow for computing the bound of $(T \circ \texttt{TransitiveClosure}(T))$ using our bound computation algorithm, provided it was able to generate a bound for the transition-system T. Such a transitive closure preserves important relationships (between program variables) for the application of compute bounds on the transition-system that is to be obtained after replacing the corresponding loop by the transitive closure. In particular, note that this heuristic for construction of convexity-witness, when used in conjunction with algorithm in Figure 6 discovers the required transitive-closures of the respective transition systems mentioned in Figure 1(e) and Figure 2.

6. Ranking Function for a Transition

In this section, show how to compute a *ranking function* for a transition. These ranking functions are made use of by the bound computation algorithm described in Section 7.

DEFINITION 7 (Ranking Function for a Transition). *We say that a real-valued function* $r(\vec{x})$ *is a* ranking function *for a transition* $s(\vec{x}, \vec{x'})$ *if it is bounded below by* 0 *and if it decreases by at least* 1 *in each execution of the transition, i.e.,*

- $s \Rightarrow (r > 0)$
- $s \Rightarrow (r[\vec{x'}/\vec{x}] \leq r - 1)$

We denote this by $\mathtt{Rank}(s, r)$.

We say that a ranking function $r_1(\vec{x})$ is *more precise* than a ranking function $r_2(\vec{x})$ if $r_1 \leq r_2$ (because in that case, r_1 provides a more precise bound for the transition than r_2).

We discuss below the design of a functionality RankC that takes as input a transition $s(\vec{x}, \vec{x'})$ and outputs a set of ranking functions $r(\vec{x})$ for that transition. We use a pattern-matching based technique that relies on making some queries that can be discharged using an SMT solver. We found this technique to be effective (fast and precise) for most of the transitions that we encountered during the process of bound computation on .Net base-class libraries. However, other techniques, such as constraint-based techniques [22] or counter instrumentation enabled iterative fixed-point computation based techniques [12, 15] can also be used for generating ranking functions. Clearly, there are examples where the constraint-based or iterative techniques that perform precise arithmetic reasoning would be more precise, but nothing beats the versatility of simple pattern matching that can handle non-arithmetic patterns with equal ease.

We list below some patterns that we found to be most effective.

6.1 Arithmetic Iteration Patterns

One standard way to iterate over loops is to use an arithmetic counter. Ranking functions for such an iteration pattern can be computed using the following pattern.

$$\boxed{\text{If } s \Rightarrow (e > 0 \ \wedge \ e[\vec{x'}/\vec{x}] < e), \text{ then } e \in \mathtt{RankC}(s)}$$

The candidates for expression e while applying the above pattern are restricted to those expressions that only involve variables from $\vec{x}$ and those that occur syntactically as an operator of conditionals when normalized to the form $(e > 0)$, after rewriting a conditional of the form $(e_1 > e_2)$ to $(e_1 - e_2 > 0)$. Following are some example transitions whose ranking functions can be computed using an application of this pattern.

- $\mathtt{RankC}(i'{=}i{+}1 \wedge i{<}n \wedge i{<}m \wedge n'{=}n \wedge m'{\leq}m){=}\{n{-}i{,}m{-}i\}$
- $\mathtt{RankC}(n > 0 \ \wedge \ n' \leq n \ \wedge \ A[n] \neq A[n']) = \{n\}$

The second example transition above (obtained from the transition-system generated for the loop in Example Ex3 in Figure 2) is a good illustration of how simple pattern matching is used to guess a ranking function, and an SMT solver (that can reason about combination of theory of linear arithmetic and theory of arrays) can be used to perform the relatively complicated reasoning of verifying the ranking function over loop-free code fragment.

Another common arithmetic pattern is the use of a multiplicative counter whose value doubles or halves in each iteration (as in case of binary search). A more precise ranking function for such a transition can be computed by using the pattern below.

$$\boxed{\text{If } s \Rightarrow (e \geq 1 \ \wedge \ e[\vec{x'}/\vec{x}] \leq e/2), \text{ then } \log e \in \mathtt{RankC}(s)}$$

The candidates for expression e while applying the above pattern are restricted to those expressions that only involve variables from $\vec{x}$ and those that occur syntactically as an operator of conditionals when normalized to the form $(e > 1)$, after rewriting a conditional of the form $(e_1 > e_2)$ that occurs in s to $(\frac{e_1}{e_2} > 1)$, provided e_2 is known to be positive. Following are some example transitions whose ranking functions can be computed using an application of this pattern.

- $\mathtt{RankC}(i' \leq i/2 \ \wedge \ i > 1) = \{\log i\}$
- $\mathtt{RankC}(i' = 2{\times}i \wedge i > 0 \wedge n > i \wedge n' = n) = \{\log{(n/i)}\}$

The above two patterns are good enough to compute ranking functions for most loops that iterate using arithmetic counters. However, for purpose of completeness, we describe below two examples (taken from some recent work on proving termination) that cannot be matched using the above two patterns, and hence illustrate the limitations of pattern-matching. However, we can find ranking functions or bounds for these examples using the counter instrumentation and invariant generation techniques described in [12].

- Consider the terminating transition-system $(x' = x + y \wedge y' = y + 1 \wedge x < n \wedge n' = n)$ from [5], which uses the principle of polyranking lexicographic functions for proving its termination. Note that the reason why the transition-system terminates is because even though y is not known to be always positive, it will eventually become positive by virtue of the assignment $y' = y + 1$.
- Consider the terminating transition-system $(x' = y \wedge y' = x{-}1 {\wedge} x > 0)$. This transition-system can be proven terminating by monotonicity constraints as introduced in [2]). Note, that the reason why the transition-system terminates is because in every two iterations the value of x decreases by 1.

6.2 Boolean Iteration Patterns

Often loops contain a path/transition that is meant to execute just once. The purpose of such a transition is to switch between different phases of a loop, or to perform the cleanup action immediately before loop termination. Such an iteration pattern can be captured by the following rule/lemma, where the operator $\mathtt{Bool2Int}(e)$ maps boolean values `true` and `false` to 1 and 0 respectively.

$$\boxed{\text{If } s \Rightarrow (e \ \wedge \ \neg(e[\vec{x'}/\vec{x}])), \text{ then } \mathtt{Bool2Int}(e) \in \mathtt{RankC}(s)}$$

The candidates for boolean expression e while applying the above pattern are restricted to those expressions that only involve variables from $\vec{x}$ and those that occur syntactically in the transition s. Following are some example transitions whose ranking functions can be computed using an application of this pattern.

- $\mathtt{RankC}(\mathtt{flag}' = \mathtt{false} \ \wedge \ \mathtt{flag}) = \{\mathtt{Bool2Int}(\mathtt{flag})\}$
- $\mathtt{RankC}(x' = 100 \ \wedge \ x < 100) = \{\mathtt{Bool2Int}(x < 100)\}$

6.3 Bit-vector Iteration Patterns

One standard way to iterate over a bit-vector is to change the position of lsb, i.e., least significant one bit (or msb, i.e., most significant one bit). Such an iteration pattern can be captured by the following rule/lemma, where the function $\mathtt{LSB}(x)$ denotes the position of the least significant 1-bit, counting from 1, and starting from most significant bit-position. $\mathtt{LSB}(x)$ is defined to be 0 if there is no 1-bit in x. Note that $\mathtt{LSB}(x)$ is bounded above by the total number of bits in bit-vector x.

$$\boxed{\text{If } s \Rightarrow (\mathtt{LSB}(x') < \mathtt{LSB}(x) \ \wedge \ x \neq 0), \text{ then } \mathtt{LSB}(x) \in \mathtt{RankC}(s)}$$

The candidates for variable x while applying the above pattern are all bit-vector variables that occur in the transition s. The query in the above pattern can be discharged using an SMT solver that provides support for bit-vector reasoning, and, in particular, the LSB operator. (If the SMT solver does not provide first-class support for the LSB operator, then one can encode the LSB operator using bit-level manipulation as described in [25].) Following are two common example transitions whose bound can be computed using the above rule.

- $\mathtt{RankC}(x' = x << 1 \ \wedge \ x \neq 0) = \{\mathtt{LSB}(x)\}$
- $\mathtt{RankC}(x' = x \& (x - 1) \ \wedge \ x \neq 0) = \{\mathtt{LSB}(x)\}$

6.4 Data-structure Iteration Patterns

Iteration over data-structures or collections is quite common, and one standard way to iterate over a data-structure is to follow field dereferences until some designated object is reached. Such an iteration pattern can be captured by the following rule/lemma, where the function $\texttt{Dist}(x, z, f)$ denotes the number of field dereferences along field f required to reach z from x.

$$\boxed{\text{If } s \Rightarrow (x \neq z \ \wedge \ (\texttt{Dist}(x', z, f) < \texttt{Dist}(x, z, f))), \quad \text{then } \texttt{Dist}(x, z, f) \in \texttt{RankC}(s).}$$

The candidates for variables x, z and field f, while applying the above pattern are all variables $\vec{x}$ and field names that occur in s. The query in the above pattern can be discharged using an SMT solver that implements a decision procedure for the theory of reachability and can reason about its cardinalities (e.g., [14]). Note that $\texttt{Dist}(x, z, f)$ denotes the cardinality of the set of all nodes that are reachable from x before reaching z along field f. Following are some example transitions whose ranking functions can be computed using an application of this pattern.

- $\texttt{RankC}(x \neq \texttt{Null} \wedge x' = x.\texttt{next}) \ = \ \{\texttt{Dist}(x, \texttt{Null}, \texttt{next})\}$

- $\texttt{RankC}(\texttt{Mem}'\texttt{=Update}(\texttt{Mem}, x.\texttt{next}, x.\texttt{next}.\texttt{next}) \wedge$
  $x \neq \texttt{Null} \wedge x.\texttt{next} \neq \texttt{Null}) \ = \ \{\texttt{Dist}(x, \texttt{Null}, \texttt{next})\}$

7. Bound Computation of Transition-System

In this section, we show how to compute a bound for a transition-system T.

If a transition-system consists of a single transition s, then a bound for the transition-system can be obtained simply from any ranking function r of the transition s using the following theorem.

THEOREM 2. *Let* $\mathit{Rank}(s, r)$. *Then,*

$$\mathit{Bound}(s) = \mathit{Max}(0, r)$$

where the Max *operator returns the maximum of its arguments.*

PROOF: If the transition s is ever taken, then r denotes an upper bound on number of iterations of s (since, by our definition of a ranking function, transition s implies that r is bounded below by 0 and decreases by at least 1 in each iteration). The other case is when s is never executed (i.e., the number of iterations of s is 0). Combining these two cases, we obtain the result. □

The significance of sanitizing the bound by applying the $\texttt{Max}$ operator in Theorem 2 is illustrated in Example 7.

Obtaining bound for a transition-system consisting of multiple transitions is not as straight-forward. We cannot simply, for example, add the ranking functions of all individual transitions to obtain the bound for the transition-system, since interleaving of those transitions with each other can invalidate the decreasing measure of the ranking function. An alternative can be to define the notion of lexicographic ranking functions [5] or disjunctively well-founded ranking functions [23] for transition-systems consisting of multiple transitions. Such an approach may sometimes work for proving termination, but would usually not be precise for yielding bounds.

For purpose of precise bound computation, we distinguish between the different ways in which two transitions of a transition-system can interact with one another. These cases (described in Section 7.1, Section 7.2, and Section 7.3) result in composition of the ranking functions using three fundamentally different mathematical operators, namely max, sum, and product.

7.1 Max Composition of Ranking Functions

The bound for a transition-system consisting of two transitions $s_1 \vee s_2$ can be obtained by applying the $\texttt{Max}$ operator to ranking functions for the individual transitions under cases when the transitions are either disjoint, or they decrease each other's ranking functions. In fact, the criterion is a bit more general, and is formalized in Theorem 3, which makes use of the following definition.

DEFINITION 8 (Cooperative-interference). *We say there is* cooperative interference *between transitions* s_1 *and* s_2 *through their ranking functions* r_1 *and* r_2 *if any of the following conditions hold:*

- *(Non-enabling condition)* $s_1 \circ s_2 = \mathtt{false}$.
- *(Rank-decrease condition)* $s_1 \Rightarrow r_2[\vec{x'}/\vec{x}] \leq \mathit{Max}(r_1, r_2) - 1$.

We denote such a cooperative-interference by $\mathit{CI}(s_1, r_1, s_2, r_2)$.

THEOREM 3 (Proof Rule for Max-Composition). *Let* $r_1 \in \mathit{RankC}(s_1)$ *and* $r_2 \in \mathit{RankC}(s_2)$. *If* $\mathit{CI}(s_1, r_1, s_2, r_2) \wedge \mathit{CI}(s_2, r_2, s_1, r_1)$, *then*

$$\mathit{Bound}(s_1 \vee s_2) \quad = \quad \mathit{Max}(0, r_1, r_2)$$

PROOF: We consider four cases below. (1) If both transitions s_1 and s_2 satisfy the non-enabling condition, then either only transition s_1 can execute or only transition s_2 can execute. Hence, the result. (2) If both transitions satisfy the rank-decrease condition, then it can be shown that $\texttt{Max}(r_1, r_2)$ is a ranking function for both the transitions s_1 and s_2. Hence, the result. (3) Now suppose transition s_1 satisfies the non-enabling condition, while transition s_2 satisfies the rank-decrease condition. The only possibility is that a sequence of transitions s_2 is followed by a sequence of transitions s_1. The result now follows from the fact that $\texttt{Max}(r_1, r_2)$ is a ranking function for s_2, while r_1 is a ranking function for s_1. (4) The last case is similar to (3). □

EXAMPLE 6. *Consider the transition-system* $s_1 \vee s_2$ *(for the control-location at Line 6 in Example* $\texttt{Ex1}$ *in Figure 1) with the following two transitions:*

$$s_1 \quad \stackrel{def}{=} \quad (n' = n - 1 \ \wedge \ j < n - 1 \ \wedge \ j' \geq j \ \wedge \ i' = i)$$

$$s_2 \quad \stackrel{def}{=} \quad (n' = n - 1 \wedge i < n - 1 \ \wedge \ i' \geq i + 1 \ \wedge \ j' \geq i + 2)$$

We can compute $\mathit{RankC}(s_1) = \{n - j - 1\}$ *and* $\mathit{RankC}(s_2) = \{n - i - 1\}$. *We can prove* $\mathit{CI}(s_1, n - j - 1, s_2, n - i - 1)$ *and* $\mathit{CI}(s_2, n - i - 1, s_1, n - j - 1)$. *An application of max-composition theorem yields a bound of* $\mathit{Max}(0, n - i - 1, n - j - 1)$ *for the transition-system* $s_1 \vee s_2$.

7.2 Additive Composition of Ranking Functions

The bound for a transition-system consisting of two transitions $s_1 \vee s_2$ can be obtained by adding together the ranking functions for the two transitions under cases when the transitions do not interfere with each other's ranking functions. To state this formally (Theorem 4), we first define the notion of *non-interference* of a transition with respect to the ranking function of another transition.

DEFINITION 9 (Non-interference). *We say that a transition* s_1 *does not interfere with the ranking function* r_2 *of another transition* s_2 *if any of the following conditions hold:*

- *(Non-enabling condition)* $s_1 \circ s_2 = \mathtt{false}$
- *(Rank-preserving condition)* $s_1 \Rightarrow (r_2[\vec{x'}/\vec{x}] \leq r_2)$

We denote such a non-interference by $\mathit{NI}(s_1, s_2, r_2)$.

The following theorem holds. We use the notation $\texttt{Iter}(s)$ denote the total number of iterations of transition s inside its transition-system.

THEOREM 4 (Proof Rule for Additive-Composition). *Let* $r_1 \in \mathit{RankC}(s_1)$, $r_2 {\in} \mathit{RankC}(s_2)$. *If* $\mathit{NI}(s_1, s_2, r_2) \wedge \mathit{NI}(s_2, s_1, r_1)$, *then*

$$\begin{aligned} \mathit{Bound}(s_1 \vee s_2) \quad &= \quad \mathit{Iter}(s_1) + \mathit{Iter}(s_2), \ \textit{where} \\ \mathit{Iter}(s_1) \quad &= \quad \mathit{Max}(0, r_1) \\ \mathit{Iter}(s_2) \quad &= \quad \mathit{Max}(0, r_2) \end{aligned}$$

PROOF: The non-interference conditions $\mathtt{NI}(s_2, s_1, r_1)$ ensure that the value of the ranking function r_1 for transition s_1 is not increased by any interleaving of transition s_2. Hence, the total number of iterations of the transition s_1 is given by $\mathtt{Max}(0, r_1)$ (based on an argument similar to that in proof of Theorem 2). Similarly, the total number of iterations of the transition s_2 is given by $\mathtt{Max}(0, r_2)$. Hence, the result.

□

EXAMPLE 7. *Consider the transition-system $s_1 \vee s_2$ (obtained from the loop in Example Ex6 in Fig. 3 with following 2 transitions:*

$$s_1 \stackrel{def}{=} z > x \ \wedge \ x < n \ \wedge \ x' = x + 1 \ \wedge \ \mathit{Same}(\{z, n\})$$
$$s_2 \stackrel{def}{=} z \leq x \ \wedge \ x < n \ \wedge \ z' = z + 1 \ \wedge \ \mathit{Same}(\{x, n\})$$

We can compute $\mathit{RankC}(s_1) = \{n - x\}$ and $\mathit{RankC}(s_2) = \{n - z\}$. We can prove $\mathit{NI}(s_1, s_2, n - z)$ and $\mathit{NI}(s_2, s_1, n - x)$. An application of additive-composition theorem yields a bound of $\mathit{Max}(0, n - x) + \mathit{Max}(0, n - z)$ for the transition-system $s_1 \vee s_2$.

We now explain the importance of using the Max operators in the statement of Theorem 2 and Theorem 4. If we defined $\mathtt{Iter}(s)$ to simply r instead of $\mathit{Max}(0, r)$, then we would incorrectly conclude the bound on the transition-system $s_1 \vee s_2$ to be $2n - x - z$. This is incorrect because, for example, suppose that the transition-system was executed in the initial state $n = 100, x = 0, z = 200$, then the expression $n - x - z$ evaluates to 0, while the transition-system $s_1 \vee s_2$ executes for 100 iterations.

This example is also a good illustration of how our technique differs significantly from (and, in fact, provides a simpler alternative to) recently proposed techniques for proving termination [7] and loop bound analysis [13]. The control-flow refinement technique used in [13] unravels the exact interleaving pattern between the two transitions to conclude that s_1 and s_2 interleave in lock-steps, only after which it is able to derive the bound. In contrast, our proof rule stated in Theorem 4 only requires to establish the non-interference property between the two transitions. The principle of disjunctively well-founded ranking functions used in [7] requires computing the transitive closure of the transition-system only to conclude a quadratic bound. In contrast, our proof rule stated in Theorem 4 does not require computing any transitive-closure, and is even able to obtain a precise linear bound. (Transitive-closure is required in our technique to only summarize any inner nested loops, which are not present in the loop in Example Ex6).

Observe that the Additive-Composition and Max-Composition Theorems provide quite orthogonal proof-rules. The bound for the transition-system in Example 6 can be computed using Max-Composition Theorem, but not using Additive-Composition Theorem. Similarly, the bound for the transition-system in Example 7 can be computed using Additive-Composition Theorem, but not using Max-Composition Theorem.

7.3 Multiplicative Composition of Ranking Functions

If we cannot establish mutual cooperative-interference or mutual non-interference properties of two transitions, then it is still possible to compute bounds provided one of the transition satisfies the non-interference property. The bound in such a case is obtained by multiplying together the ranking functions for the two transitions, as made precise in the following theorem. This is a common case for bounding iterations of an inner loop when its iterators are re-initialized inside the outer loop leading to a multiplicative bound.

THEOREM 5 (Proof Rule for Multiplicative-Composition). *Let $r_1 \in \mathit{RankC}(s_1)$, and $r_2 \in \mathit{RankC}(s_2)$. If $\mathit{NI}(s_2, s_1, r_1)$, then*

$$\begin{aligned} \mathit{Bound}(s_1 \vee s_2) &= \mathtt{Iter}(s_1) + \mathtt{Iter}(s_2), \text{ where} \\ \mathtt{Iter}(s_1) &= \mathit{Max}(0, r_1) \\ \mathtt{Iter}(s_2) &= \mathit{Max}(0, r_2) + \mathit{Max}(0, u_2) \times \mathtt{factor} \\ & \quad \text{where } \mathtt{factor} = \mathit{Max}(0, r_1) \end{aligned}$$

where $u_2(\vec{x})$ denotes an upper bound on expression $r_2(\vec{x'})$ in terms of $\vec{x}$ as implied by $\mathit{TC}(s_1)$. For the special case when $(r_1 > 0) \wedge s_2$ is unsatisfiable, we can choose $\mathtt{factor}$ to be 1.

PROOF: From the non-interference condition $\mathtt{NI}(s_2, s_1, r_1)$, we can conclude that $\mathtt{Iter}(s_1) \leq \mathtt{Max}(0, r_1)$ (same argument as in proof of Theorem 4). However, the same thing cannot be s_2. Instead we observe that the maximum number of iterations of s_2 *in between any two interleavings of* s_1 is bounded above by $\mathtt{Max}(0, u_2)$ (since the starting value of the ranking function r_2 is reset to u_2 by any execution of s_1). However, the number of iterations of s_2 before any interleaving of s_1 is still bounded by $\mathtt{Max}(0, r_2)$. Hence, the total number of iterations of s_2 is bounded by $\mathtt{Max}(0, r_2) + \mathtt{Max}(0, u_2) \times \mathtt{Max}(0, r_1)$. The special case follows from the observation that even though s_1 interferes with the ranking function r_2 of s_2, it can interfere at most once since s_2 is enabled only after completion of all iterations (as opposed to somewhere in the middle) of s_1. In other words, the worst-case possibility is a sequence of transitions s_2, followed by a sequence of transitions s_1, followed by a sequence of transitions s_2.

□

EXAMPLE 8. *Consider the transition-system with the following two transitions s_1 and s_2.*

$$s_1 \stackrel{def}{=} i'{=}i{-}1 \ \wedge \ i{>}0 \ \wedge \ j'{=}j{-}1 \ \wedge \ j{>}0 \ \wedge \ \mathit{Same}(\{k', m'\})$$
$$s_2 \stackrel{def}{=} j' = m \ \wedge \ k'{=}k{-}1 \ \wedge \ k > 0 \ \wedge \ \mathit{Same}(\{i', m'\})$$

We can compute $\mathit{RankC}(s_1) = \{i, j\}$ and $\mathit{RankC}(s_2) = \{k\}$. We can prove $\mathit{NI}(s_1, s_2, k)$ and $\mathit{NI}(s_2, s_1, i)$. An application of additive-composition theorem yields a bound of $\mathit{Max}(0, i) + \mathit{Max}(0, k)$ for the transition-system $s_1 \vee s_2$. An application of multiplicative-composition theorem yields an incomparable bound of $\mathit{Max}(0, j) + \mathit{Max}(0, m) \times \mathit{Max}(0, k)$.

7.4 Combining the Composition Rules

In this section, we discuss how to compute bounds for a transition-system with multiple (including more than 2 transitions) by putting together the proof rules mentioned in Theorem 3, 4, and 5.

First observe that an optimal way of applying the proof rules in Additive-Composition Theorem and Multiplicative-Composition Theorems (Theorem 4 and Theorem 5) is to compute the total number of iterations for each transition individually, and then sum them up together. The algorithm described in Figure 7 implements such a strategy based on a simple extension of Theorem 4 and Theorem 5 to the case when a transition-system contains more than 2 transitions. The algorithm iteratively computes an array $\mathtt{Iter}$ such that $\mathtt{Iter}[s_i]$ denotes a bound on the total number of iterations taken by the transition s_i during any execution of the transition-system $s_1 \vee \ldots \vee s_n$. The array J at Line 4 contains the indices of all transitions that interfere with the ranking function r of transition s_i. If a bound on the total iterations of all those transitions is known (test on Line 5), then the iterations of s_i is obtained using a generalization of Theorem 4 and Theorem 5 (Line 9). A bound on the entire transition-system is obtained by simply summing up the bound on the total number of iterations of the individual transitions (Line 11). For simplicity, we have presented the algorithm to output

ComputeBound($\bigvee_{i=1}^{n} s_i$)

1 for $i \in \{1,..,n\}$: $\texttt{Iter}[s_i] := \perp$;
2 do {
3   for $i \in \{1,..,n\}$ and $r \in \texttt{RankC}(s_i)$:
4         $J := \{j \mid \neg\texttt{NI}(s_j, s_i, r)\}$;
5   if $(\texttt{Iter}[s_i] = \perp) \wedge (\forall j \in J : \texttt{Iter}[s_j] \neq \perp)$
6             $\texttt{factor} := 0$;
7   foreach $j \in J$: $\texttt{factor:=factor+Iter}[s_j]$;
8   Let $u(\vec{x})$ be an upper bound on $r[\vec{x'}/\vec{x}]$ as implied by $\texttt{TC}(\bigvee_{j \neq i} s_j)$.
9             $\texttt{Iter}[s_i] := \texttt{Max}(0, r) + \texttt{Max}(0, u) \times \texttt{factor}'$;
10 } while any change in Iter array;
11 if $(\forall j \in \{1,..,n\} : \texttt{Iter}[s_j] \neq \perp)$, return $\sum_j \texttt{Iter}[s_j]$;
12 else return ''Potentially Unbounded''

Figure 7. Bound Computation for a Transition-System $\bigvee_{i=1}^{n} s_i$ from ranking functions $\texttt{RankC}(s_i)$ of individual transitions.

only one bound, but the algorithm can be easily extended to output multiple bounds by relaxing the condition $\texttt{Iter}[s_i] = \perp$ in Line 5 and by associating a set of bounds (as opposed to a single bound) with $\texttt{Iter}[s_i]$.

EXAMPLE 9. *Consider the transition-system $s_1 \vee s_2 \vee s_3$ (obtained from the loop in Ex7 in Figure 3) with the following 3 transitions:*

$$s_1 = j < n \wedge j < m \wedge j'{=}j{+}1 \wedge 0 < n < m \wedge \textit{Same}(\{n, m\})$$
$$s_2 = j > n \wedge j < m \wedge j'{=}j{+}1 \wedge 0 < n < m \wedge \textit{Same}(\{n, m\})$$
$$s_3 = j \geq m \ \wedge\ j' = 0 \ \wedge\ 0 < n < m \ \wedge\ \textit{Same}(\{n, m\})$$

We can compute $\textit{RankC}(s_1) = \{n-j, m-j\}$, $\textit{RankC}(s_2) = \{m-j\}$, $\textit{RankC}(s_3) = \{\texttt{Bool2Int}(j \geq m)\}$. Since $\textit{NI}(s_1, s_2, m-j)$ and $\textit{NI}(s_3, s_2, m-j)$, the algorithm in Figure 7 first computes $\texttt{Iter}[s_2] = \textit{Max}(0, m-j)$. Using $\textit{NI}(s_1, s_3, \texttt{Bool2Int}(j \geq m))$, the algorithm now computes $\texttt{Iter}[s_3] = \texttt{Bool2Int}(j \geq m) \times (1+1) \leq 2$. From $\textit{NI}(s_2, s_1, n-j)$, the algorithm now computes $\texttt{Iter}[s_1] = \textit{Max}(0, n-j) + \textit{Max}(0, n) \times 2$. The algorithm now returns a total bound of $\textit{Max}(0, m-j) + 2 + \textit{Max}(0, n-j) + \textit{Max}(0, n) \times 2$. This bound can be translated to in terms of the inputs in Example Ex7 by substituting $n+1$ for j (as obtained from the initial state before the loop) to yield $m+1+n$, which is a factor of 2 away from the real bound of $m+1$ (since $n < m$).

This example also illustrates how our technique differs significantly from (and, in fact, provides a simpler alternative to) recently proposed techniques for termination and loop bound analysis. The control-flow refinement technique used in [13] uses a sophisticated machinery to unravel the exact interleaving pattern between the three transitions (in particular, s_2 follows s_3 which in turn follows s_1) and is able to obtain the exact bound of $m+1$. In contrast, our proof rules yield a bound of $m{+}1{+}n$, but using a much simpler formalism. We do not know of any other technique (including [7, 15]) that can even prove termination of this example.

We now discuss an extension to the above-described algorithm that also takes advantage of the proof rule in Max-Composition Theorem (Theorem 3). For this purpose, before running the algorithm, we simply extend $\texttt{RankC}(s)$ for any transition s with $\texttt{Max}(r, r')$, where $r \in \texttt{RankC}(s)$ and $r' \in \texttt{RankC}(s')$ for some other transition s', provided $\texttt{Rank}(s, \texttt{Max}(r, r'))$ holds.

# Transitions	1	2	3	4	5	6	7	8	9	≥10
# Loops	1561	224	107	44	25	11	9	5	8	191

Figure 8. Number of loops for respective number of transitions.

8. Experiments

We have implemented our proposed solution to the reachability-bound problem in C# using Phoenix Compiler Infrastructure [21] and the SMT solver Z3 [1]. Our implementation runs on .Net binaries. We present below two different sets of experimental results that measure the effectiveness of various aspects of our solution.

8.1 Loop Bound Computation

We considered the problem of computing symbolic bounds on the number of loop iterations, which is an instance of the reachability-bound problem where the control-location under consideration is the loop header. We chose mscorlib.dll (a .Net base-class library), which had 2185 loops, as our benchmark. Our tool analyzes these 2185 loops in less than 5 minutes and is able to compute bounds for 1677 loops. The problem of loop bound computation is especially challenging under the following two cases for which earlier techniques for bound computation do not perform as well.

Case 1: Iterations of outer loops depend on inner loops (examples of the kind described in Figure 2). There were 113 such loops out of the total 2185 loops. The key idea of our paper to address such challenges is to replace the inner loops by their transitive-closure that preserves required relationships between the inputs and outputs of the loop. The *effectiveness of our transitive closure computation algorithm* is illustrated by the fact that our success ratio for such cases (80 out of 113, i.e., 70%) is similar to our overall success ratio (1677 out of 2185, i.e., 76%).

Case 2: Loop bound computation for nested loops. The challenge here is to compute precise amortized bounds on the total number of iterations of those loops, as opposed to the number of iterations per iteration of the immediately outer loop (the latter is an easier problem than the former). This is the same issue as exemplified by the example in Figure 1. There were 250 such loops out of the total 2185 loops. Unfortunately, we cannot evaluate the precision of our bounds automatically. As described in Section 3, the problem of computing a precision-witness for a given symbolic bound is an orthogonal problem that we are currently working on. Instead, we manually investigated the generated bounds for most of these loops and found all these bounds to be precise (according to Definition 2). This points out the *effectiveness of our bound-computation algorithm* based on the three proof rules presented in Section 7.

Another interesting statistic is the distribution of the number of transitions generated for each loop, as shown in Figure 8. The small number of transitions validates the design choice behind our *transition-system generation algorithm* that enumerates all paths between two program points (in order not to lose any precision) after slicing has been performed.

Out of the 508 loops for which we failed to compute a bound, the failure for 503 loops is attributed to not being able to compute ranking functions for some transition in the transition-system corresponding to the loop. There were two main causes: (i) our implementation is intra-procedural, meaning that our transition-system generation algorithm fails when the value of loop iterators gets modified because of procedure calls. This problem can be addressed by simply inlining the procedure, provided there are no recursive calls. (ii) We only implemented the arithmetic and boolean iteration patterns, while several transitions were iterating using field dereferences or bit-vector manipulation. A sound handling of field dereferences would require use of an alias analysis. A more optimistic way to read this statistic is to observe the *effectiveness of the pattern-matching technique for finding ranking*

Original Example	Various Details

Gopan and Reps 06. P. 3, F. 1
`x:=0, y:=0;`
`while (*)`
`  if (x ≤ 50) y++;`
`  else y--;`
`  if (y<0) break;`
`  x++;`
`assert(x=102)`
	$(x \leq 50 \wedge y+1 \geq 0 \wedge y' = y+1 \wedge x' = x+1)_{s_1} \vee (x > 50 \wedge y-1 \geq 0 \wedge y' = y-1 \wedge x' = x+1)_{s_2}$
	$Init \equiv x = 0 \wedge y = 0$
	$(0 \leq x' \leq 51 \wedge x' = y')_{s'_1} \vee (52 \leq x' \leq 102 \wedge x' + y' = 102)_{s'_2}$
	$\delta = 1, \sigma = \{(1,1) \mapsto 1, (1,2) \mapsto 2, (2,2) \mapsto 2, (2,1) \mapsto 1\}$

Beyer *et al.* **07.** P. 306, F. 4.
`x:=0; y:=50;`
`while (x<100)`
`  if (x<50) x++;`
`  else x++; y++;`
`assert(y=100);`
	$(x \leq 50 \wedge x' = x+1 \wedge y' = y)_{s_1} \vee (51 \leq x \leq 100 \wedge x' = x+1 \wedge y' = y+1)_{s_2}$
	$Init \equiv x = 0 \wedge y = 50$
	$(0 \leq x' \leq 50 \wedge y' = 50)_{s'_1} \vee (51 \leq x' \leq 100 \wedge x' = y')_{s'_2}$
	$\delta = 1, \sigma = \{(1,1) \mapsto 1, (1,2) \mapsto 2, (2,2) \mapsto 2, (2,1) \mapsto 1\}$

Gulavani *et al.* **06.** P. 5, F. 3. **Henzinger** *et al.* **02.** P. 2, F. 1.
`lock:=0;assume(`$x \neq y$`)`
`while (`$x \neq y$`)`
`  lock := 1; x := y;`
`  if (*)`
`    lock := 0; y++;`
`assert(lock = 1);`
	$(x \neq y \wedge lock' = 1 \wedge x' = y \wedge y' = y)_{s_1} \vee (x \neq y \wedge lock' = 0 \wedge x' = y \wedge y' = y+1)_{s_2}$
	$Init \equiv x \neq y \wedge lock = 0$
	$(x' = y' \wedge lock' = 1)_{s'_1} \vee (x' + 1 = y' \wedge lock' = 0)_{s'_2}$
	$\delta = 1, \sigma = \{(1,1) \mapsto 1, (2,1) \mapsto 1, (1,2) \mapsto 2, (2,2) \mapsto 2\}$

Popeea and Chin 06. P. 2
`x := 0; upd := 0;`
`while (`$x < N$`)`
`  if (*)`
`    l := x; upd := 1;`
`  x++;`
`assert(`$upd = 1 \Rightarrow 0 \leq l < N$`);`
	$(x < N \wedge x' = x+1 \wedge l' = l \wedge upd' = upd)_{s_1} \vee (x < N \wedge x' = x+1 \wedge l' = x \wedge upd' = 1)_{s_2}$
	$Init \equiv x = 0 \wedge upd = 0$
	$(x' \geq 0 \wedge l' = l \wedge upd' = 0 \wedge N' = N)_{s'_1} \vee (x' \geq 1 \wedge upd' = 1 \wedge N' = N \wedge 0 \leq l' < N)_{s'_2}$
	$\delta = 1, \sigma = \{(1,1) \mapsto 1, (1,2) \mapsto 2, (2,2) \mapsto 2, (2,1) \mapsto 2\}$

Figure 9. Prominent disjunctive invariant challenges from recent literature. Entries in 2nd column show following details in that order: transition-system representation of the loop, initial condition `Init`, transitive closure of the transition-system required to prove the assertion, and the convexity-witness (δ, σ).

functions: a handful of patterns are sufficient to compute ranking functions for the various transitions for 76% of the examples.

There were only 5 cases (out of 1682 cases) for which we were able to compute a ranking functions for each transition, but were not able to compute a bound for the transition-system. This points out the *effectiveness of our proof rules for bound computation* from composition of ranking functions of individual transitions.

8.2 Disjunctive Invariant Computation

We also evaluated the effectiveness of our transitive closure algorithm on a variety of benchmark examples chosen by recent state-of-the-art papers on computing disjunctive invariants. Figure 9 describes these four examples that have been used as flagship examples to motivate new techniques for proving non-trivial safety assertions. Proving validity of the assertions in all these examples requires disjunctive loop invariants. It turns out that the required disjunctive invariant for each of these examples satisfies the convexity-like assumption, and hence can be discovered by our transitive closure algorithm in Figure 6. We adapt our algorithm slightly to take advantage of the initial condition (as is done by all the other approaches) by initializing s'_1 to $\texttt{Init} \wedge \texttt{Id}$ at Line 2, instead of only `Id` since `Init` is known at the beginning of each loop. This allows our algorithm to establish the desired assertion using a disjunctive invariant with fewer disjuncts. (For a more detailed discussion on this adaptation, see the end of this section).

Given that the number of disjuncts in the desired transitive closure is 2 for all examples, and the number of transitions in the transition-system represented by the loop is either 2 or 3, the total number of possibilities for the map σ is 16 or 64 respectively. Hence, by trying out all possible maps, the algorithm in Figure 6 can discover the desired disjunctive invariants.

Instead we experimented with a heuristic for dynamic construction of map σ that we found to be effective for all examples. We choose $m = 1$ and initialize s'_1 to $\texttt{Init} \wedge \texttt{Id}$. We maintain a partial map σ that is completely undefined to start with, and use the following heuristic to construct σ on the fly. For each choice of (i, j) on Line 4 in the algorithm, if $\sigma(j, i)$ is undefined, we compute $s = s'_j \circ s_i$ in the abstract domain. If s is not equal to false, then we use a semantic-merging criterion to find any k such that s is close to an existing disjunct s'_k and define $\sigma(j, i)$ to be k. If no such k exists, we increase m by 1 and define $\sigma(j, i)$ to the new value of m. The semantic-merging criterion that we used for our experiments was one that checks agreements on variable equalities (as opposed to the more general inequality relationships expressible in the octagon domain [19] used by our prototype implementation). This heuristic is an excellent example of combining the strengths of semantic-merging criterions in light of the importance of having a static syntactic merging criterion as suggested by Theorem 1 (if we do not want to iterate over all maps σ). We implemented this heuristic and our prototype implementation is able to validate the assertion in each of the examples in less than 0.2sec.

We now return to the discussion on what would happen if we do not adapt our algorithm to make use of the initial condition `Init` while computing a loop summary. We can still prove the desired assertion, but the required transitive closure would consist of more disjuncts, and would involve elements from a numerical domain richer than the Octagon abstract domain. For example, for the first example, we would require the following disjunctive invariant:

$$
\begin{aligned}
&(\texttt{Id})_{s'_1} \;\vee\; (x \leq 50 \wedge x' \leq 51 \wedge x' - x = y' - y)_{s'_2} \\
\vee\;& (x \geq 51 \wedge x' \geq 52 \wedge x' - x = y' - y)_{s'_3} \\
\vee\;& (x \leq 50 \wedge x' \geq 52 \wedge 102 - x' - x = y - y')_{s'_4}
\end{aligned}
$$

Observe that the above invariant again satisfies the convexity-like assumption, where the convexity-witness σ is as follows: $\sigma = \{(1,1) \mapsto 2, (2,1) \mapsto 2, (3,1) \mapsto 3, (4,1) \mapsto 4, (1,2) \mapsto 3, (2,2) \mapsto 4, (3,2) \mapsto 3, (4,2) \mapsto 4\}$. Hence, our approach can be used to discover this invariant. In contrast, none of the techniques presented for the respective examples can analyze the loops in such a *modular* setting where the initial condition is not initially known. Further discussion on use of our technique for modular analysis is beyond the scope of this paper.

9. Comparison with Related Work

Disjunctive Invariant Generation Variety of techniques exist to lift classical abstract domains (like intervals, octagons [19], and polyhedra [8], which infer conjunctive invariants) to the powerset extension or some approximation of it for discovering disjunctive invariants [10, 11, 16, 24]. These techniques address the hardness inherent in this problem by proposing various semantic-merging heuristics. In contrast, we present a result that calls for working with a static syntactic merge criterion under the convexity-like assumption (which appears to be satisfied by benchmark examples).

Some syntactic techniques based on program restriction [4] or control-flow refinement [13] have also been suggested for discovering disjunctive invariants. These can be viewed as instantiations of our more general framework based on the convexity-witness σ.

Symbolic Bound Generation There has been some recent work on generating symbolic bounds on the number of loop iterations [12, 13, 15], but none of these techniques directly address the more general problem of reachability-bound that we introduce in our paper. Our solution reduces the reachability-bound problem to the problem of computing bounds of an outer loop, but one whose iterations are influenced by inner loops. None of the techniques presented in [12, 13, 15] directly address the challenge of computing bounds for such loops, and hence would fail to compute bounds for most of the examples presented in the paper. In particular, [15] would fail to compute bounds for the examples `Ex1`, `Ex3`, `Ex4`, `Ex5`, `Ex7` because the invariants required for establishing bounds on the counters are disjunctive. (It can only compute bounds for `Ex2` and `Ex6`.) The multiplicative counter instrumentation strategies that are meant to alleviate the problem of computing disjunctive invariants do not help in this case because there is only one back-edge for the outer loop and only one counter can be instrumented. Similarly, [13] would fail to compute bounds for `Ex1`, `Ex3`, `Ex4`, `Ex5` for the same reason of requiring disjunctive invariants for performing desired reasoning of inner loops. (It can only compute bounds for `Ex2`, `Ex6` and `Ex7`.) The control-flow refinement strategy is meant to alleviate the problem of computing disjunctive invariants, but it does not help in any of these cases since the control-flow is already refined, and it cannot be refined any further. In contrast, our technique can compute bounds for all the motivating examples presented in [12, 13, 15]. The approach described in [12] requires user annotations to identify interesting non-linear and disjunctive expressions to compute bounds for transition-systems with multiple transitions. Our technique addresses these challenges by means of novel proof rules. However, the technique described in [12] can be used in a synergistic manner with our technique, in particular, as an extension to the pattern-matching based technique to compute bounds/ranking-functions for single transitions.

We report the first implementation of symbolic bound generation for .Net binaries, while [12, 13, 15] all work for C++ programs. Our implementation scales to large programs, while [12, 15] have been applied to only small benchmarks.

[9] computes symbolic bounds by curve-fitting timing data obtained from profiling. Their technique has the advantage of measuring real time in seconds for a representative workload, but does not provide worst-case bounds. There is a large body of work on estimating worst case execution time (WCET) in the embedded and real-time systems community [26]. 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 or simple numerical analysis. These WCET techniques cannot compute bounds for the most of the examples considered in this paper.

Termination Analysis There has been a large body of work on proving termination of programs and the standard approach used has been that of finding ranking functions. We also use ranking functions to compute bounds, but our focus is on finding *precise* ranking functions, using composition by `Max` or $+$ operators if possible, that can yield precise symbolic bounds. Bounds can also be obtained from the standard lexicographic ranking functions or disjunctively well-founded ranking relations [7], but only using multiplicative-composition, which is imprecise compared to the bounds that can be obtained from max- or additive-composition.

In fact, our proof rules can also be regarded as an alternative new technique for proving termination. For example, the recently proposed approach based on variance assertions or disjunctively well-founded ranking relations cannot be used to prove termination of the loop in Example `Ex7`, while our technique can.

There is superficial similarity between termination techniques based on computing variance assertions [3], transition invariants [23] and disjunctively well-founded ranking relations [7] in that they also summarize relationships between two different visits to a control-location, and often require disjunctive invariants. However, there are two key technical differences: (a) Our technique requires computing relationships between two immediate visits to a control-location, while the approach based on transition invariants or variance assertions requires computing relationships between *any* two visits to a control-location. (b) Our technique requires use of disjunctive invariants only to summarize nested loops. In particular, for examples `Ex6` and `Ex7` with no nested loops, our technique would not require computing disjunctive invariants unlike the technique based on disjunctively well-founded ranking relations.

10. Future Work and Conclusion

This paper defined and motivated the reachability-bound problem. The paper also presented a solution to the reachability-bound problem in the context of non-recursive and sequential programs. The next technical challenge is to address the reachability-bound problem in context of recursive procedures and concurrent execution.

On the applications side, we are working on integrating the proposed solution to the reachability-bound problem with other specific techniques to provide an integrated solution for resource bound analysis in some contexts such as memory bound analysis, and active-task graph size analysis in asynchronous programs.

References

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

[2] A. M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In *CAV*, pages 109–123, 2009.

[3] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL*, 2007.

[4] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[5] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.

[6] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[7] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.

[8] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL*, 1978.

[9] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *ESEC/SIGSOFT FSE*, 2007.

[10] D. Gopan and T. W. Reps. Lookahead widening. In *CAV*, 2006.

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

[12] B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384, 2008.

[13] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009.

[14] S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, 2009.

[15] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[16] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, pages 200–214, 1998.

[17] T. Henzinger. From boolean to quantitative system specifications, keynote. In *1st Workshop on Quantitative Analysis of Software. http://research.microsoft.com/users/sumitg/qa09/keynote.pdf*.

[18] P. Malacaria. Assessing security threats of looping constructs. In *POPL*, pages 225–235, 2007.

[19] A. Miné. The octagon abstract domain. In *WCRE*, 2001.

[20] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[21] Microsoft Phoenix Compiler, research.microsoft.com/phoenix/.

[22] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'04*.

[23] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS '04*.

[24] C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345, 2006.

[25] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[26] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. In *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.