Unbounded Symbolic Execution for Program Verification

JOXAN JAFFAR[1], JORGE A. NAVAS[1], AND ANDREW E. SANTOSA[2]

[1] National University of Singapore
[2] University of Sydney, Australia
{joxan,navas}@comp.nus.edu.sg,santosa@it.usyd.edu.au

Abstract. Symbolic execution with interpolation is emerging as an alternative to CEGAR for software verification. The performance of both methods relies critically on interpolation in order to obtain the most general abstraction of the current symbolic or abstract state which can be shown to remain error-free. CEGAR naturally handles unbounded loops because it is based on abstract interpretation. In contrast, symbolic execution requires a special extension for such loops.
In this paper, we present such an extension. Its main characteristic is that it performs *eager subsumption*, that is, it always attempts to perform abstraction in order to avoid exploring other symbolic states. It balances this primary desire for more abstraction with the secondary desire to maintain the *strongest loop invariant*, for earlier detection of infeasible paths, which entails less abstraction. Occasionally certain abstractions are not permitted because of the reachability of error states; this is the underlying mechanism which then causes *selective unrolling*, that is, the unrolling of a loop along relevant paths only.

1 Introduction

Symbolic execution [21] is a method for program reasoning that uses *symbolic values* as inputs instead of actual data, and it represents the values of program variables as symbolic expressions as functions of the input symbolic values. A *symbolic execution tree* depicts all executed paths during the symbolic execution. A *path condition* is maintained for each path and it is a formula over the symbolic inputs by accumulating constraints which those inputs must satisfy in order for execution to follow that path. A path is *infeasible* if its path condition is unsatisfiable. Otherwise, the path is *feasible*.

Symbolic execution was first developed for program testing [21], but it has subsequently used for bug finding [8] and verification condition generation [3, 18], among others. Recently, symbolic execution has been used for software verification [20, 23, 15] as an alternative to existing model checking techniques based on *CounterExample-Guided Abstraction Refinement* (CEGAR) [9, 2]. Essentially, the general technique followed by symbolic execution-like tools starts with the concrete model of the program and then, the model is checked for the desired property via *symbolic execution* by proving that all paths to certain error nodes are infeasible (i.e., error nodes are unreachable).

The first challenge for symbolic execution is the exponential number of symbolic paths. The approaches of [20, 23, 15] tackle successfully this fundamental problem by eliminating from the concrete model those facts which are *irrelevant* or *too-specific* for

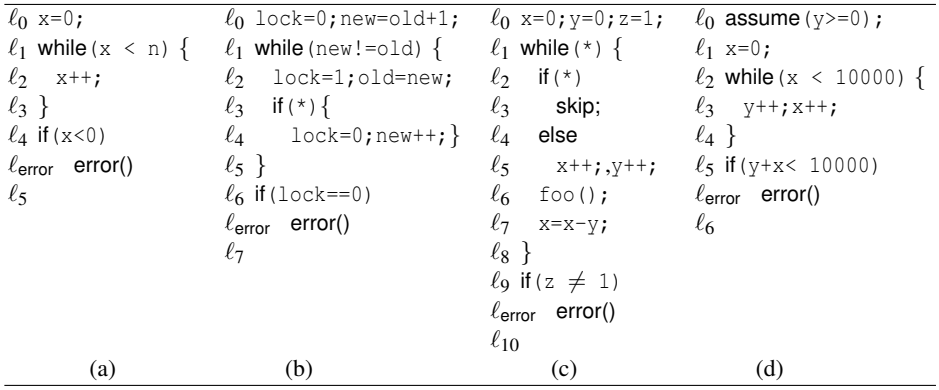

Fig. 1. Programs with Loops

proving the unreachability of the error nodes. This learning phase consists of computing *interpolants* in the same spirit of *conflict clause learning* in SAT solvers. The use of symbolic execution with interpolants is thus similar to CEGAR approaches [16, 22], but symbolic execution has some benefits [23]:

1. It does not explore *infeasible paths* avoiding the expensive refinement in CEGAR.
2. It avoids expensive *predicate image* computations of, for instance, the *Cartesian* [1, 7] and *Boolean* [5] abstract domains.
3. It can recover from *too-specific* abstractions in opposition to monotonic refinement schemes used in CEGAR.

The main remaining challenge for symbolic execution is due to unbounded loops which make the symbolic execution tree not just large, but *infinite*. This means that some of *abstraction* must be performed on the symbolic states in order to obtain finiteness. Previous work [20] assumed that loop invariants are inferred automatically by other means (e.g., abstract interpretation). The main disadvantage is the existence of false alarms. Another solution is proposed in [15] where abstraction refinement 'a la' CEGAR is performed as a separate process for loops but lacking of the benefits of symbolic execution mentioned above. Finally, [23] proposes a naive *iterative deepening* method which unwinds loops iteratively performing finite symbolic execution until a fixed depth, while inferring the interpolants needed to keep unreachable the error nodes. Amongst these interpolants, only those which are loop invariant are kept and it is checked whether they still prove unreachability of error nodes. If yes, the program is safe. Otherwise, the depth is increased and the process is repeated. Although simple this approach has the advantage of that it performs symbolic execution also within loops as [20] and without reporting false alarms.

Example 1 (Iterative Deepening). Consider the program in Fig. 1(a). To force termination, the iterative deepening method executes the program considering one iteration of the loop. Using interpolants, ℓ_4 is annotated with $x \geq 0$ by using weakest precondition. This interpolant preserves the infeasibility of the error path. Then, the remaining step is to check if the interpolant is invariant. Since $x \geq 0$ is an inductive interpolant, the symbolic execution is complete and we have proven the program safe.

This program illustrates the essence of the iterative deepening approach which obtains generalization by interpolation and relies on the heuristics that a bounded proof may highlight how to make the unbounded proof. However, this approach has one major drawback: its naive iterative deepening cannot terminate in programs like the one in Fig. 1(b) due to the impossibility of discovering disjunctive invariant interpolants. We elaborate on the reason below. Meanwhile, we mention that this example has been often used to highlight the strength of CEGAR [17] for handling unbounded loops. Further, its essential characteristic is present in real programs as we will show in Sec. 5.

In this paper, we propose a new method to enhance symbolic execution for handling unbounded loops but yet without losing the intrinsic benefits of symbolic execution. This method is based on three design principles: (1) *abstract loops* in order for the algorithm to attempt to terminate, (2) *preserve* as much as possible the inherent benefits of symbolic execution (mainly, earlier detection of infeasible paths) by propagating the *strongest loop invariants*, whenever possible, and (3) *refine progressively* imprecise abstractions in order to avoid reporting false alarms.

The central idea is to unwind loops iteratively while computing *speculative loop invariants* which make converge quickly the symbolic execution of the loop. The algorithm attempts to minimize the loss of information (i.e., ability of detecting infeasible paths) by computing the *strongest* possible invariants and it checks whether error nodes are unreachable. If yes, the program is safe. Otherwise, a counterexample is produced and analyzed to test if it corresponds to a concrete counterexample in the original program.. If yes, the program is reported as unsafe. Otherwise, these speculative invariants are too coarse to ensure the safety conditions and the algorithm introduces a refinement phase similar to CEGAR in which it computes those interpolants needed to ensure the unreachability of the error nodes, resulting in *selective* unrolling only at points where the invariant can no longer be produced due to the strengthening introduced by the interpolants.

Example 2 (Selective Unrolling and Path Invariants). Consider our key example program in Fig. 1(b). We first explain why simple unrolling with iterative deepening does not work here. Essentially, there are two paths in the loop body, and the required safety property $lock \neq 0$ is not invariant along both paths. In fact, we require the disjunctive loop invariant $new \neq old \vee lock \neq 0$, and this entails far more than simple invariant discovery. Thus loop unrolling does not terminate with successive deepenings. In more detail, we execute first one iteration of the loop. Using interpolants, ℓ_1 is annotated with $new \neq old$. We test if the interpolant $new \neq old$ is inductive. Since it is not inductive we cannot keep it and we execute the program considering now two iterations of the loop. During the second iteration of the loop, both paths $\pi_1 \equiv \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_5 \rightarrow \ell_1$ and $\pi_2 \equiv \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_4 \rightarrow \ell_5 \rightarrow \ell_1$ must be unrolled. From π_1 the symbolic execution proves the unreachability of ℓ_{error} adding the interpolant $old = new \wedge lock \neq 0$. From π_2, simple unrolling with iterative deepening will add the interpolant $new \neq old$ after executing the second iteration of the loop. Since the interpolant is not inductive yet after this second iteration, we cannot keep it and the unrolling process runs forever.

In our algorithm, we proceed as follows. We also execute the symbolic path $\ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_5 \rightarrow \ell_{1'}$[1]. We then examine the constraints at the entry of the loop ℓ_1 (i.e., called *loop header*) to discover which abstraction at ℓ_1 makes possible that the symbolic state at $\ell_{1'}$ can entail (be *subsumed* by) the state at ℓ_1. We use the notion of *path-based loop invariant*. A path-based loop invariant is a formula whose truth value does not change after the symbolic execution of the path[2]. Clearly, the constraints $lock = 0$ and $new = old + 1$ at ℓ_1 are no longer path-invariants after the execution of the path. We then decide to generalize at ℓ_1 the constraints $lock = 0$ and $new = old + 1$ to *true*. As a consequence, the constraints at $\ell_{1'}$ can entail now the constraints at ℓ_1. The objective here is to achieve the convergence of the loop by forcing subsumption between a node and its ancestor.

Next, we backtrack and we execute the path $\ell_3 \rightarrow \ell_4 \rightarrow \ell_{5'}$. The symbolic state at $\ell_{5'}$ is subsumed by the interpolant computed at ℓ_5 since $lock = 0 \wedge old = new + 1$ trivially entails *true*. After we have executed the loop, we execute the path $\ell_1 \rightarrow \ell_6 \rightarrow \ell_{\mathsf{error}}$ which is now feasible due to the abstraction we performed at ℓ_1. We then trigger a counterexample-guided refinement phase 'a la' CEGAR. First, we check that the path $\ell_0 \rightarrow \ell_1 \rightarrow \ell_6 \rightarrow \ell_{\mathsf{error}}$ is indeed spurious due to the abstraction at ℓ_1. Next, we strengthen the abstraction at ℓ_1 in order to make the error node unreachable. It suffices here the interpolant $new = old + 1$. Finally, we ensure that the interpolant $new = old + 1$ cannot be generalized again at ℓ_1, and it we restart the process again.

After we restart we will reach the node $\ell_{1'}$ again and we will try to weaken the symbolic state at ℓ_1 s.t. the state at $\ell_{1'}$ can be subsumed, as we did before. However, the situation has now changed since we cannot abstract the interpolant $new = old + 1$ added by the refinement, and hence, we decide to unroll $\ell_{1'}$ with the symbolic state $lock = 1 \wedge old = new$. We prove that the error node is not reachable from $\ell_{1'}$ and during backtracking annotate both $\ell_{1'}$ and ℓ_5 with the interpolant $\overline{\Psi} \equiv old = new \wedge lock \neq 0$. This strengthening avoids now that the path $\ell_3 \rightarrow \ell_4 \rightarrow \ell_{5'}$ can be subsumed since $lock = 0 \wedge new = old + 1$ does not imply $\overline{\Psi}$. We continue the path and encounter $\ell_{1''}$. It is easy to see that $\ell_{1''}$ cannot be subsumed by its sibling $\ell_{1'}$ since the symbolic state at $\ell_{1''}$ ($lock = 0 \wedge new = old + 1$) does not entail $\overline{\Psi}$ neither. However, we can still force $\ell_{1''}$ to be subsumed by its ancestor ℓ_1 by abstracting the state at ℓ_1 using the notion of path-invariant again. Note that the subsumption already holds and hence, we can halt exploring the path without further abstraction.

Therefore, we have shown that selective unrolling only at points where we cannot force subsumption with an ancestor can help termination. The main advantage of selective unrolling is that it may achieve termination even if disjunctive invariants are needed. However, it also introduces some new challenges.

Example 3 (Strongest Invariant versus Speculative Subsumption). Consider the program in Fig. 1(c). Say we explore first the path $\ell_0 \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_6 \rightarrow \ell_7 \rightarrow \ell_8 \rightarrow \ell_{1'}$. Assuming that `foo()` only changes its local variables, the symbolic state at $\ell_{1'}$

[1] Note that $\ell_{1'}$ and ℓ_1 correspond to the same location where primed versions refer to different symbolic states in the symbolic execution tree.

[2] Do not be confused with the term "path invariants" used in [6].

is $x = 0 \wedge y = 0 \wedge z = 1$ which already entails the symbolic state at ℓ_1. As usual, during backtracking we annotate the symbolic states with their corresponding interpolants. The next path explored is $\ell_2 \rightarrow \ell_4 \rightarrow \ell_5 \rightarrow \ell_{6'}$. In principle, the symbolic state at $\ell_{6'}$ with constraints $x = 1 \wedge y = 1 \wedge z = 1$ entails *true*, the interpolant at ℓ_6. We therefore can stop the exploration of the path at $\ell_{6'}$ avoiding exploring `foo()`.

However, a key observation is that the constraints $x = 0$ and $y = 0$ are not path-invariant if we would only consider the path $\ell_1 \rightarrow \ell_2 \rightarrow \ell_4 \rightarrow \ell_5 \rightarrow \ell_{6'}$. We face then here an important dilemma. On one hand, one of our design principles is to compute the strongest possible loop invariants. However, note that the constraint $x = 0$ is in fact invariant if subsumption would not take place at $\ell_{6'}$ due to the execution of $x = x - y$ at ℓ_7. On the other hand, we may suffer the path explosion problem if we would not subsume other paths.

We adopt the solution of subsuming other paths whenever possible while abstracting further the symbolic states of the loop headers even if we may lose the opportunity of computing the strongest path-based loop invariants.

Coming back to the example, subsumption takes place at $\ell_{6'}$ but we must also abstract the symbolic state at ℓ_1 discarding the constraints $x = 0$ and $y = 0$ although we still keep $z = 1$. In spite of this abstraction the transition $\ell_9 \rightarrow \ell_{\mathsf{error}}$ is infeasible. However, as with the program in Fig. 1(b), we now may have some interpolants that strengthen the path-based loop invariants in order to make unreachable the error nodes. For the sake of discussion, assume the condition at ℓ_9 is $x \neq 0$. Then, the path $\ell_1 \rightarrow \ell_9 \rightarrow \ell_{\mathsf{error}}$ would be feasible since $z = 1 \wedge x \neq 0$ is satisfiable. We then check that the path $\ell_0 \rightarrow \ell_1 \rightarrow \ell_9 \rightarrow \ell_{\mathsf{error}}$ is indeed spurious due the abstraction at ℓ_1 and discover that the interpolant $x = 0$ suffices to make unreachable the error node. As a consequence of this refinement, the subsumption at $\ell_{6'}$ now cannot take place since the constraint $x = 0$ is not allowed to be abstracted at ℓ_1. We therefore continue exploring the path $\ell_{6'} \rightarrow \ell_7 \rightarrow \ell_8 \rightarrow \ell_{1''}$. The symbolic state at $\ell_{1''}$ ($x = 0 \wedge y = 1 \wedge z = 1$) entails the one at ℓ_1 if we abstract the constraint $y = 1$ to true. As a result, the analysis of the loop can still terminate and the error can be proved unreachable.

Example 4 (Other Benefits of Strongest Invariants: Faster Convergence). It is well-studied that the discovery of loop invariants can speedup the convergence of loops [6]. The bounded program in Fig. 1(d) illustrates the potential benefits of propagating invariants by our symbolic execution-based approach wrt CEGAR.

CEGAR (e.g., [7, 24]) discovers the predicates $(x = 0), (x = 1), \ldots, (x = 10000 - 1)$ and also $(y \geq 0), (y \geq 1), \ldots, (y \geq 10000)$, and hence full unwinding of the loop is needed. Say symbolic execution explores the path $\ell_0 \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_4 \rightarrow \ell_{2'}$. It is straightforward to see that $y \geq 0$ is invariant. The next symbolic path is $\ell_2 \rightarrow \ell_5 \rightarrow \ell_{\mathsf{error}}$ with the generalized constraint $y \geq 0$ at ℓ_2. As a result, the symbolic path is infeasible since the formula $y \geq 0 \wedge x \geq 10000 \wedge y + x < 10000$ is unsatisfiable, and hence, we are done without unwinding the loop.

2 Related Work

Similar to [16, 22] our algorithm discovers invariant interpolants that prove the unreachability of error nodes. However, we differ from them because we abstract only at loops

discovering loop invariants as strong as possible and hence, we still explore a significant less amount of infeasible paths. Moreover, we avoid the expensive predicate image computation in predicate abstraction approaches [17, 2]. A recent paper [5] mitigates partially these problems by encoding large loop-free blocks into a Boolean formula relying on the capabilities of a SMT solver, although for loops the same issues still remain. Synergy/DASH/SMASH [14, 4, 13] use test-generation features to enhance the process of verification. The main advantage comes from the use of symbolic execution provided by DART [12] to make cheaper the refinement phase. The main disadvantage is that these methods cannot recover from too-specific refinements (see program diamond in [23]).

To the best of our knowledge, the works of [19, 20] are the first in using symbolic execution with interpolation in pursuit of verifying a target property. However, [19] does not consider loops and [20] relies on abstract interpretation in order to compute loop invariants, and as a result, false alarms can be reported. Alternatively, the verification problem can be seen as a translation to a Boolean formula that can then be subjected to a SAT or SMT solver. It is a fact that symbolic execution with interpolation can be considered analogous to conflict clause learning in DPLL style SAT solvers. [15] adopts this approach by mapping the verification problem of loop-free programs into a solving SMT instance. In presence of loops, [15] allows choosing between different methods. One is the use of abstract interpretation for discovering loop invariants that allow termination similar to [20]. Another alternative is the use of CEGAR but losing the ability of detecting eagerly infeasible paths within loops.

Our closest related work is McMillan at el. [23]. This work can be dissected in two parts. For loop-free fragments, this work is in fact covered by the earlier works [19, 20] and hence, equivalent to ours here. However, we differ in the way we handle unbounded loops. [23] follows the iterative deepening method explained in Sec. 1 and hence, may not converge for some realistic programs as we have shown. Finally, [23] computes summaries for functions and support recursive functions. Our implementation currently performs function inlining and does not cover recursive functions. We consider these extensions however to be an orthogonal issue which we can address elsewhere.

3 Background

Syntax. We restrict our presentation to a simple imperative programming language [3], where all basic operations are either assignments or assume operations, and the domain of all variables are integers. The set of all program variables is denoted by *Vars*. An *assignment* $\mathsf{x} := \mathsf{e}$ corresponds to assign the evaluation of the expression e to the variable x. In the *assume* operator, $\mathsf{assume(c)}$, if the boolean expression c evaluates to *true*, then the program continues, otherwise it halts. The set of operations is denoted by *Ops*.

We model a program by a *transition system*. A transition system is a quadruple $\langle \Sigma, I, \longrightarrow, O \rangle$ where Σ is the set of states and $I \subseteq \Sigma$ is the set of initial states. $\longrightarrow \subseteq \Sigma \times \Sigma \times Ops$ is the transition relation that relates a state to its (possible) successors executing operations. This transition relation models the operations that are executed when control flows from one program location to another. We shall use $\ell \xrightarrow{\mathsf{op}} \ell'$ to

[3] Our implementation supports most features of sequential C including function calls and pointers.

denote a transition relation from $\ell \in \Sigma$ to $\ell' \in \Sigma$ executing the operation $\mathsf{op} \in \mathit{Ops}$. Finally, $O \subseteq \Sigma$ is the set of final states.

Symbolic Execution. A *symbolic state* σ is a triple $\langle \ell, s, \Pi \rangle$. The symbol $\ell \in \Sigma$ corresponds to the current program counter (with special program counter $\ell_{\mathsf{end}} \in O$ to denote a final location). The symbolic store s is a function from program variables to terms over input symbolic variables. Each program variable is initialized to a fresh input symbolic variable. The *evaluation* $[\![e]\!]_s$ of an arithmetic expression e in a store s is defined as usual: $[\![v]\!]_s = s(v)$, $[\![n]\!]_s = n$, $[\![e + e']\!]_s = [\![e]\!]_s + [\![e']\!]_s$, $[\![e - e']\!]_s = [\![e]\!]_s - [\![e']\!]_s$, etc. The evaluation of Boolean expression $[\![b]\!]_s$ can be defined analogously. Finally, Π is called *path condition* and it is a first-order formula over the symbolic inputs and it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular corresponding path. The set of first-order formulas and symbolic states are denoted by *FO* and *SymStates*, respectively. Given a transition system $\langle \Sigma, I, \longrightarrow, O \rangle$ and a state $\sigma \equiv \langle \ell, s, \Pi \rangle \in \mathit{SymStates}$, the symbolic execution of $\ell \xrightarrow{\mathsf{op}} \ell'$ returns another symbolic state σ' defined as:

$$\sigma' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge [\![c]\!]_s \rangle & \text{if } \mathsf{op} \equiv \mathsf{assume(c)} \text{ and } \Pi \wedge [\![c]\!]_s \text{ is satisfiable} \\ \langle \ell', s[x \mapsto [\![e]\!]_s], \Pi \rangle & \text{if } \mathsf{op} \equiv \mathsf{x := e} \end{cases} \quad (1)$$

Note that Eq. (1) queries a *theorem prover* for satisfiability checking on the path condition. We assume the theorem prover is sound but not complete. That is, the theorem prover must say a formula is unsatisfiable only if it is indeed so.

Abusing notation, given a symbolic state $\sigma \equiv \langle \ell, s, \Pi \rangle$ we define $[\![\sigma]\!] : \mathit{SymStates} \rightarrow \mathit{FO}$ as the formula the projection of the formula $(\bigwedge_{v \in \mathit{Vars}} [\![v]\!]_s) \wedge [\![\Pi]\!]_s$ on the set of program variables *Vars*. The projection is performed by the elimination of existentially quantified variables.

A *symbolic path* $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n$ is a sequence of symbolic states such that $\forall i \bullet 1 \leq i \leq n$ the state σ_i is a *successor* of σ_{i-1}. The set of symbolic paths is denoted by *SymPaths*. A symbolic state $\sigma' \equiv \langle \ell', \cdot, \cdot \rangle$ is a successor of another $\sigma \equiv \langle \ell, \cdot, \cdot \rangle$ if there exists a transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$. A path $\pi \equiv \sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_n$ is *feasible* if $\sigma_n \equiv \langle \ell, s, \Pi \rangle$ such that $[\![\Pi]\!]_s$ is satisfiable. If $\ell \in O$ and σ_n is feasible then σ_n is called *terminal* state. Otherwise, if $[\![\Pi]\!]_s$ is unsatisfiable the path is called *infeasible* and σ_n is called *infeasible* state. A state $\sigma \equiv \langle \ell, \cdot, \cdot \rangle$ is called *subsumed* if there exists another state $\sigma' \equiv \langle \ell, \cdot, \cdot \rangle$ such that $[\![\sigma]\!] \models [\![\sigma']\!]$. If there exists a feasible path $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n$ then we say σ_k $(0 \leq k \leq n)$ is *reachable* from σ_0 in *k steps*. We say σ'' is reachable from σ if it is reachable from σ in some number of steps.

A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a transition system by triggering Eq. (1). The nodes represent symbolic states and the arcs represent transitions between states. We say a symbolic execution tree is *complete* if it is finite and all its leaves are either terminal, infeasible or subsumed.

Bounded Program Verification via Symbolic Execution. We follow the approach of [20].We will assume a program is annotated with assertions of the form `if (!c) then` error(), where `c` is the safety property. Then the verification process consists of constructing a complete symbolic execution tree and proving that error is unreachable from all symbolic paths in the tree. Otherwise, the program is unsafe. One of the

challenges to build a complete tree is the exponential number of symbolic paths. An interpolation-based solution to this problem was first proposed in [20] which we also follow in this paper. Given an infeasible state $\sigma \equiv \langle \ell, s, \Pi \rangle$ s.t. $[\![\Pi]\!]_s$ is unsatisfiable we can generate a formula $\overline{\Psi}$ (called *interpolant*) which still preserves the infeasibility of the state but using a weaker (more general) formula than the original $[\![\sigma]\!]$. The main purpose of using $\overline{\Psi}$ rather than the original formula associated to the symbolic state σ is to increase the likelihood of subsumption.

Definition 1 (Interpolant). *Given two first-order logic formulas Π_1 and Π_2 such that $\Pi_1 \wedge \Pi_2$ is unsatisfiable a Craig* interpolant *[10] is another first-order logic formula $\overline{\Psi}$ such that (a) $\Pi_1 \models \overline{\Psi}$, (b) $\overline{\Psi} \wedge \Pi_2$ is unsatisfiable, and (c) all variables in $\overline{\Psi}$ are common variables in Π_1 and Π_2.*

The symbolic execution of a program can be augmented by annotating each symbolic state with its corresponding interpolant such that the interpolant represents the sufficient conditions to preserve the unreachability of the error nodes. Then, the notion of subsumption can be redefined as follows.

Definition 2 (Subsumption with Interpolants). *Given two symbolic states σ and σ' such that σ is annotated with the interpolant $\overline{\Psi}$, we say that σ' is* subsumed *by σ if $[\![\sigma']\!]$ implies $\overline{\Psi}$ (i.e., s.t. $[\![\sigma']\!] \models \overline{\Psi}$).*

4 Algorithm

A full description of our algorithm is given in Fig. 2 and Fig. 3. For clarity and making the reader familiar with our algorithm, we start by explaining only the parts corresponding to the bounded symbolic execution engine used in [20, 23]. Having done this, we will explain how this basic algorithm can be augmented for supporting unbounded programs which is the main technical contribution of this paper.

The input of the algorithm is an initial symbolic state $\sigma_k \in \mathit{SymStates}$, the transition system $\mathcal{P}$, an initial empty path π, and an empty *subsumption table* $\mathcal{M}$. We use the key k to refer unambiguously to the symbolic state σ in the symbolic execution tree. In order to perform subsumption tests our algorithm maintains the table $\mathcal{M}$ stores entries of the form $\langle \ell, k \rangle : \overline{\Psi}$, where $\overline{\Psi}$ is the interpolant at program location ℓ associated to a symbolic state k in the symbolic execution tree. The interpolants are generated by a procedure $\mathsf{Interp} : \mathit{FO} \times \mathit{FO} \rightarrow \mathit{FO}$ that takes two formulas and computes a Craig interpolant following Def. 1, Sec. 3. The output of the algorithm is the subsumption table if the program is safe. Otherwise, the algorithm aborts..

Bounded Verification via Symbolic Execution with Interpolation. The algorithm for bounded verification using symbolic execution with interpolants consists of building a complete symbolic execution tree while testing error nodes are not reachable.
The algorithm starts by testing if the path is infeasible at line 1. If yes, an interpolant is generated to avoid exploring again paths which have the same infeasibility reason. Next, if the error node is reachable (line 3) then the error must be real since for bounded programs no abstraction is done and hence, the program is reported as unsafe at line 7. The next case is when the end of a path (i.e, terminal node) has been encountered.

UNBOUNDEDSYMEXEC($\sigma_k \equiv \langle \ell, s, \Pi \rangle, \mathcal{P}, \pi, \mathcal{M}$)

1: if $[\![\Pi]\!]_s$ is unsat then /* infeasible path */
2: return $\mathcal{M} \cup \{\langle \ell, k \rangle : \mathit{false}\}$
3: else if ($\ell = \ell_{\mathsf{error}}$) then
4: if $\exists\, \sigma_h \equiv \langle \ell_h, \cdot, \cdot \rangle$ in π s.t. ℓ_h is a *loop header* and $[\![\sigma_h]\!] \wedge [\![\Pi]\!]_s$ is unsat then /* spurious error */
5: return REFINEANDRESTART($\sigma_h, \sigma_k, \mathcal{P}, \pi, \mathcal{M}$)
6: else
7: printf("The program is unsafe") and abort() /* real error */
8: else if ($\ell = \ell_{\mathsf{end}}$) then /* end of path */
9: return $\mathcal{M} \cup \{\langle \ell, k \rangle : \mathit{true}\}$
10: else if $\overline{\Psi}$:= (SUBSUMED($\sigma_k, \mathcal{M}$) $\neq \bot$) then /* sibling-sibling subsumed */
11: return $\mathcal{M} \cup \{\langle \ell, k \rangle : \overline{\Psi}\}$
12: else if $\exists\, \langle \ell, \cdot, \cdot \rangle$ in π then /* cyclic path */
13: foreach σ_h in π s.t. $\sigma_h \equiv \langle \ell, \cdot, \cdot \rangle$ do
14: if (NONPATHINV($\sigma_h, \sigma_k, \mathcal{M}$) $\neq \bot$) then /* child-ancestor subsumed */
15: return $\mathcal{M} \cup \{\langle \ell, k \rangle : \mathit{true}\}$
16: endfor
17: goto 19
18: else
19: $\overline{\Psi} := \mathit{true}$
20: foreach transition relation $\ell \xrightarrow{\mathsf{op}} \ell' \in \mathcal{P}$ do /* forward symbolic execution */
21: $\sigma'_{k'} \triangleq \begin{cases} \langle \ell', s, \Pi \wedge [\![c]\!]_s \rangle & \text{if op} \equiv \mathsf{assume(c)} \text{ and fresh } k' \\ \langle \ell', s[x \mapsto [\![e]\!]_s], \Pi \rangle & \text{if op} \equiv \mathsf{x := e} \text{ and fresh } k' \end{cases}$
22: $\mathcal{M}$:= UNBOUNDEDSYMEXEC($\sigma'_{k'}, \mathcal{P}, \pi \cdot \sigma_k, \mathcal{M}$)
23: $\overline{\Psi} := \overline{\Psi} \wedge (\bigwedge_{\langle \cdot, k' \rangle : \overline{\Psi}' \in \mathcal{M}} \widehat{wp}(\mathsf{op}, \overline{\Psi}'))$ /* backward symbolic execution */
24: endfor
25: return $\mathcal{M} \setminus \{\langle \ell, k \rangle : \overline{\Psi}''\} \cup \{\langle \ell, k \rangle : \overline{\Psi} \wedge \overline{\Psi}''\}$

Fig. 2. Algorithm for Unbounded Symbolic Execution with Interpolation

The algorithm simply adds an entry in the *subsumption table* whose interpolant is *true* (line 9) since the symbolic path is feasible and hence, there is no false paths to preserve. Otherwise, a subsumption test at line 10 is done in order for the symbolic execution to attempt at halting the exploration of the path. For bounded programs, this test is quite straightforward because it suffices whether the current symbolic state entails any interpolant computed previously at the same program location following Def. 2, Sec. 3. This is done at line 46, Fig. 3. If the test holds, it returns the interpolant associated with the subsuming node (line 50). Otherwise, it returns $\bot$ at line 51 to point out that the subsumption test failed.

In the remaining case, the symbolic execution moves forward one level in the symbolic execution tree. The foreach loop (lines 20-24) executes one symbolic step for each successor node [4] and it calls recursively to the main procedure UNBOUNDEDSYMEXEC

[4] Note that the rule described in line 21 is slightly different from the one described in Sec. 3 because no satisfiability check is performed. Instead, this check is postponed and done by line 1.

NONPATHINV($\sigma_h \equiv \langle \ell, s, \cdot \rangle, \sigma_k, \mathcal{M}$)

26: let $\overline{\Psi}$ be s.t. $\langle \ell, h \rangle : \overline{\Psi} \in \mathcal{M}$. Otherwise, let $\overline{\Psi}$ be *true*
27: let $I \equiv c_1 \wedge \ldots \wedge c_n$ be $[\![\sigma_h]\!]$
28: $i :=$ 1, $NonInv := \emptyset$
29: repeat
30: if $[\![\sigma_k]\!] \models I \models \overline{\Psi}$ then
31: foreach $\sigma \equiv \langle \cdot, s, \cdot \rangle$ s.t. σ is k-reachable ($k > 0$) from σ_h
32: replace s with HAVOC(s, MODIFIES($NonInv$))
33: endfor
34: return
35: else
36: $I = I - c_i$ /* delete from I the constraint c_i */
37: $NonInv := NonInv \cup \{c_i\}$
38: $i := i + 1$
39: until ($i > n$)
40: return $\bot$

REFINEANDRESTART($\sigma_h \equiv \langle \ell, \cdot, \cdot \rangle, \sigma_k, \mathcal{P}, \pi, \mathcal{M}$)

41: let π be $\sigma_0 \cdots \sigma_{h-1} \cdot \sigma_h \cdots$
42: $\overline{\Psi} :=$ INTERP($[\![\sigma_h]\!], [\![\sigma_k]\!]$)
43: $\mathcal{M} := \mathcal{M} \setminus \{\langle \ell', k' \rangle : \overline{\Psi}' \mid \langle \ell', k' \rangle : \overline{\Psi}'$, $\sigma_{k'}$ is k-reachable ($k > 0$) from $\sigma_h\}$
44: $\mathcal{M} := \mathcal{M} \setminus \{\langle \ell, h \rangle : \overline{\Psi}'\} \cup \{\langle \ell, h \rangle : \overline{\Psi} \wedge \overline{\Psi}'\}$
45: return UNBOUNDEDSYMEXEC($\sigma_h, \mathcal{P}, \sigma_0 \cdots \sigma_{h-1}, \mathcal{M}$)

SUBSUMED($\sigma_k \equiv \langle \ell, \cdot, \cdot \rangle, \mathcal{M}$)

46: if $\exists \langle \ell', k' \rangle : \overline{\Psi} \in \mathcal{M}$ s.t. ($\ell = \ell'$) and ($[\![\sigma_k]\!] \models \overline{\Psi}$) then
47: if k and k' have a common *loop header ancestor* σ_h in the tree then
48: if (NONPATHINV($\sigma_h, \sigma_k, \mathcal{M}$) $\neq \bot$) then return $\overline{\Psi}$
49: else return $\bot$
50: else return $\overline{\Psi}$
51: return $\bot$

Fig. 3. NONPATHINV, REFINEANDRESTART and SUBSUMED Procedures

with each successor state (line 22). Once the recursive call returns the key remaining step is to compute an interpolant that generalizes the symbolic execution tree at the current node while preserving the unreachability of the error nodes. The procedure $\widehat{wp} : Ops \times FO \rightarrow FO$ computes ideally the *weakest precondition (wp)* [11] which is the weakest formula on the initial state ensuring the execution of an operation in a final state, assuming it terminates. In practice, we approximate *wp* by making a linear number of calls to a theorem prover following the techniques described in [20]. The final interpolant $\overline{\Psi}$ added in the *subsumption table* is a first-order logic formula consisting of the *conjunction* of the result of $\widehat{wp}$ on each child's interpolant (line 23).

Unbounded Verification via Symbolic Execution with Interpolation. For handling unbounded loops we need to augment the basic algorithm described so far in several ways.

Loop abstractions. The main abstraction is done whenever a *cyclic path* is detected by forcing subsumption between the current node and any of its ancestors. The mechanism to force subsumption takes the constraints from the symbolic state associated with the *loop header* (i.e., entry point of the loop) and it abstracts any non-invariant constraint for that particular path. By doing this, we ensure that the symbolic state of the current node entails the symbolic state of its ancestor. Here, we use the concept of *path-based loop invariant*. Using Floyd-Hoare notation, given a path π and a constraint c, we say c is path-based invariant along π if $\{c\}\ \pi\ \{c\}$. That is, whenever c holds of the symbolic state before the execution of *path*, then c will hold afterward.

Let us come back to the algorithm in Fig. 2. In line 12 we have detected a cyclic path. The foreach loop (lines 13-16) forces subsumption between the current symbolic state and any of its ancestors. The procedure NONPATHINV in Fig. 3 attempts the current state σ_k to entail some generalization of its ancestor state σ_h. This generalization is basically to discover a loop invariant at the symbolic context of the loop header. Clearly, this procedure has a huge impact in the symbolic execution since our ability of detecting infeasible paths depends on the precision of this generalization. This task is, in general, undecidable and even if the strongest invariants can be computed it is, in general, exponential.

Our method followed in NONPATHINV is quite simple but it works well in practice and it requires a linear number of calls to a theorem prover. The invariant is a subset of the constraints at the symbolic state σ_h, called I. Initially, I contains all constraints σ_h (line 27). At each iteration of the repeat loop (lines 29-39), we test if the symbolic state at σ_k entails I (line 30). If yes, we are done. Otherwise, we delete one constraint from I and repeat the process.

If we discover a generalization of the symbolic state of the loop header in order for the test at line 30 to hold then NONPATHINV needs to propagate it to all reachable states from the header by abstracting their symbolic stores at line 31. For this purpose, we use $\textsc{Havoc}(s, Vars) \triangleq \forall v \in Vars \bullet s[v \mapsto z]$, where z is a fresh variable, and $\textsc{Modifies}(c_1 \ldots c_n)$ which takes a sequence of constraints and it returns the set of variables that may be modified during its execution. Note that this generalization is a bit more elaborated since it needs to be propagated from the symbolic state of the loop header to all symbolic states reachable from it both in the current path and also in future paths. We omit this process from our description since although trivial it is quite tedious. We assume that whenever a generalization for a loop header is done all reachable symbolic states update their symbolic stores accordingly.

It is also important to notice that the invariance property is not closed under intersection since the intersection of two invariants may not be an invariant, in general. However, we construct loop invariants by testing path-by-path and discarding non-invariant constraints from the symbolic state of the loop header. This is equivalent to compute path-based loop invariants for each path within a loop and then intersect them at the loop header. This is correct because NONPATHINV keeps only invariants which are closed under intersection. This limitation, in principle, preclude us to compute the strongest invariants but based on our experience it is not a problem and it is vital for an efficient implementation.

Refine and Restart. Clearly, the use of abstractions can mitigate the termination problems of symbolic execution but it may introduce false alarms. We therefore add a new case at line 4 in our algorithm to test whether abstract counterexamples correspond to counterexamples in the concrete model of the program. Clearly, this case resembles to the refinement phase in CEGAR. Whenever a counterexample is found, we test if its constraints are indeed satisfiable. If yes, the error must be real. Otherwise, we inspect all loop headers in the counterexample and find out which one introduced an abstraction that cannot keep unreachable the error node. Once we have found the loop header, the procedure REFINEANDRESTART, described in Fig. 3, infers an interpolant that excludes that particular counterexample (line 42) and restarts the symbolic execution from that loop header at line 45. It is worth mentioning that although our algorithm can then perform expensive refinements as CEGAR, the refinements are confined only to loop headers as in opposition to CEGAR where refinements may involve any program point. This is an important feature for performing more efficient refinements.

Interestingly now, the interpolants added by REFINEANDRESTART can affect the abstractions done by procedure NONPATHINV explained so far. Let us come back to NONPATHINV. In principle, we can always find a path invariant *true* by deleting all constraints. However, note that the test at line 30 is restrained by ensuring that the candidate invariant must entail the interpolant associated with the loop header obtained possibly from a previous refinement. If this entailment does not hold, the procedure NONPATHINV fails. This is, in fact, our mechanism to unroll selectively those points where the invariant can no longer be produced due to the strengthening introduced by the interpolants.

Subsumption. The program of Fig. 1(c) in Sec. 1 illustrated that in presence of loops the subsumption test (SUBSUMED, Fig. 3) cannot be simply an entailment test. Whenever we attempt at subsuming a symbolic state within a loop, we need additionally to be aware of which constraints may not be path-invariant anymore and generalize the symbolic state of the nearest loop header accordingly. That is the reason of SUBSUMED calling NONPATHINV at line 48. If NONPATHINV fails (i.e., it could not generalize the state of the loop header) then subsumption cannot take place.

Moreover, the correctness of SUBSUMED assumes that whenever a loop header h is annotated with its interpolant $\overline{\Psi}$, the subsumption table $\mathcal{M}$ is updated in such way that all entries associated with program points within the loop with entry h must *conjoin* their interpolants with $\overline{\Psi}$.

Example 5 (Running example). We finally show how our algorithm executes the program in Fig 1(b), Sec. 1 and proves the program is safe. The initial algorithm state is $\sigma_0 \equiv \langle \ell_0, [lock \mapsto S_{lock}, old \mapsto S_{old}, new \mapsto S_{new}], true \rangle$, $\pi \equiv nil$, and $\mathcal{M} \equiv \emptyset$.

First iteration. We first execute the successor of σ_0 obtaining $\sigma_1 \equiv \langle \ell_1, [lock \mapsto 0, old \mapsto S_{old}, new \mapsto S_{old}+1], true \rangle$. Then, we continue augmenting the path by running the foreach loop (lines 20-24) and calling recursively to UNBOUNDEDSYMEXEC (line 22) until we find a cyclic path (line 12) $\ell_0 \to \ell_1 \to \ell_2 \to \ell_3 \to \ell_5 \to \ell_1$ with $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdot \sigma_5$, $\mathcal{M} \equiv \emptyset$, and $\sigma_{1'} \equiv \langle \ell_1, [lock \mapsto 1, old \mapsto S_{old}+1, new \mapsto S_{old}+1], S_{old}+1 \neq S_{old} \rangle$. We call NONPATHINV($\sigma_1$,$\sigma_{1'}$,$\mathcal{M}$). The formulas $[\![\sigma_1]\!] \equiv lock = 0 \wedge new = old + 1$ and $[\![\sigma_{1'}]\!] \equiv lock = 1 \wedge old = new$ are obtained by projecting the symbolic states of σ_1 and $\sigma_{1'}$ onto the variables *lock*, *old* and *new*. The test at line 30 holds after deleting

from I the constraints $lock = 0$ and $new = old + 1$, since these two constraints are not path-invariant. We backtrack up to σ_3 which after the loop abstraction is $\langle \ell_3, [lock \mapsto 1, old \mapsto S_{new}, new \mapsto S_{new}], true \rangle$. We then execute $\ell_3 \rightarrow \ell_4 \rightarrow \ell_5$, obtaining the state $\sigma_{5'} \equiv \langle \ell_5, [lock \mapsto 0, old \mapsto S_{old}, new \mapsto S_{old} + 1], true \rangle$ but $\mathcal{M}$ contains now two new entries $\{\langle \ell_1, 1' \rangle : true, \langle \ell_5, 5 \rangle : true\}$. As a result, SUBSUMED($\sigma_{5'}$,$\mathcal{M}$) (line 10) succeeds since the interpolant associated with ℓ_5 is *true*. In addition, since the symbolic states σ_5 and $\sigma_{5'}$ are within a loop whose header is denoted by σ_1 we call NONPATHINV(σ_1,$\sigma_{5'}$,$\mathcal{M}$) which also succeeds without making further generalization.

We continue backtracking up to σ_1 with $\mathcal{M} \equiv \{\langle \ell_2, 2 \rangle : true, \langle \ell_3, 3 \rangle : true, \langle \ell_4, 4 \rangle : true, \langle \ell_5, 5 \rangle : true, \langle \ell_5, 5' \rangle : true, \langle \ell_1, 1' \rangle : true\}$. Prior to executing $\ell_1 \rightarrow \ell_6$ recall that the symbolic state σ_1 was generalized, and hence, the reachable state σ_6 is, in fact, $\sigma_6 \equiv \langle \ell_6, [lock \mapsto S_{lock}, new \mapsto S_{new}, old \mapsto S_{old}], S_{new} = S_{old} \rangle$. Then, we continue executing symbolically until we finally reach ℓ_{error} with $\sigma_{\text{error}} \equiv \langle \ell_{\text{error}}, [lock \mapsto S_{old}, new \mapsto S_{new}, old \mapsto S_{old}], S_{new} = S_{old} \wedge S_{lock} = 0 \rangle$. We test at line 4 whether $[\![\sigma_1]\!] \equiv lock = 0 \wedge new = old + 1 \wedge [\![\sigma_{\text{error}}]\!] \equiv old = new \wedge lock = 0$ is satisfiable. We then call REFINEANDRESTART (line 5) with σ_1, σ_{error}, $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \sigma_6$, and $\mathcal{M}$. We compute the interpolant (line 42) $new = old + 1$ calling INTERP($[\![\sigma_1]\!]$, $[\![\sigma_{\text{error}}]\!]$), delete all elements from $\mathcal{M}$ which were added by any state reachable from σ_1 (line 43), add a new element with the new interpolant (i.e., $\mathcal{M} \equiv \{\langle \ell_1, 1 \rangle : new = old + 1\}$) (line 44), and finally, we restart by calling UNBOUNDEDSYMEXEC (line 45).

Second iteration. After restart, we detect again the cyclic path $\ell_0 \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_5 \rightarrow \ell_1$ with σ_1 and $\sigma_{1'}$ as before, and call NONPATHINV. The key difference is that $\mathcal{M} \equiv \{\langle \ell_1, 1 \rangle : new = old + 1\}$. Therefore, the test at line 30 always fails and the procedure returns $\perp$ (line 40) without performing any generalization.

We then execute $\ell_1 \rightarrow \ell_2$ (second loop unroll) obtaining the infeasible symbolic state $\sigma_{2'} \equiv \langle \ell_2, [lock \mapsto 1, old \mapsto S_{old} + 1, new \mapsto S_{old} + 1], S_{old} + 1 \neq S_{old} \wedge S_{old} + 1 \neq S_{old} + 1 \rangle$. We then backtrack and execute the path $\ell_1 \rightarrow \ell_6 \rightarrow \ell_{\text{error}}$. The state at ℓ_{error} is infeasible now. We backtrack again, adding in $\mathcal{M} \equiv \{\langle \ell_1, 1' \rangle : old = new \wedge lock \neq 0, \langle \ell_5, 5 \rangle : old = new \wedge lock \neq 0, \ldots\}$, until we execute the path $\ell_3 \rightarrow \ell_4 \rightarrow \ell_5$ again (first loop unroll). We call SUBSUMED as we did in the first iteration but now the symbolic state $\sigma_{5'}$ cannot be subsumed because the formula $[\![\sigma_{5'}]\!] \equiv lock = 0 \wedge new = old + 1$ does not entail the interpolant $\langle \ell_5, 5 \rangle \in \mathcal{M}$. We execute another transition reaching ℓ_1 and detect again a cyclic path with the state $\sigma_{1''} \equiv \langle \ell_1, lock \mapsto 0, old \mapsto S_{old}, new \mapsto S_{old} + 1], S_{old} + 1 \neq S_{old} \rangle$. We call NONPATHINV($\sigma_1$,$\sigma_{1''}$,$\mathcal{M}$). The formulas associated with σ_1 and $\sigma_{1''}$ are $[\![\sigma_1]\!] \equiv lock = 0 \wedge new = old + 1$ and $[\![\sigma_{1''}]\!] \equiv lock = 0 \wedge new = old + 1$. Therefore, it is easy to see that NONPATHINV succeeds without any further abstraction and hence, we can obtain a complete symbolic execution tree without error nodes.

5 Results

We report the results of the evaluation of our prototype implementation called TRACER on several real-world C programs, commonly used in the verification community and compared with the iterative deepening algorithm used in McMillan at el. [23]. The first two programs are Linux device drivers: qpmouse and tlan. The next four programs are Microsoft Windows device drivers: kbfiltr, diskperf, floppy, and cdaudio. The program

tcas is an implementation of a traffic collision avoidance system. The program is instrumented with ten safety conditions, of which five are violated. We omit the unsafe cases since there is no differences between TRACER and iterative deepening. Finally, ssh_clnt.1 and ssh_srvr.2 are a client and server implementation of the ssh protocol.

TRACER models the heap as an array. A theorem prover is used to decide linear arithmetic formulas over integer variables and array elements in order to check the satisfiability and entailment of formulas. Functions are inlined and external functions are modeled as having no side effects and returning an unknown value.

		ITERDEEP		TRACER		
Program	LOC	S	T(s)	S	T(s)	R
qpmouse	400	1033	1.5	1033	1.99	1
tlan	8069	4892	12.3	4892	13.5	0
kbfiltr	5931	1396	1.56	1396	2.59	0
diskperf	6984	5465	16.8	5465	18.46	0
floppy	8570	4965	8.33	4995	13.26	2
cdaudio	8921	13512	27.98	13814	34.48	3
tcas-1a	394	5386	6.59	5386	7.08	0
tcas-1b	394	5405	6.42	5405	6.89	0
tcas-2a	394	5386	6.36	5386	6.84	0
tcas-3b	394	5375	6.33	5375	6.87	0
tcas-5a	394	5386	6.38	5386	6.88	0
ssh_clnt.1	2521	∞	∞	47825	593	77
ssh_srvr.2	2516	∞	∞	44213	462	63

Table 1. *Iterative Deepening* vs TRACER.

The results on Intel 2.33Ghz 3.2GB are summarized in Table. 1. We present two set of numbers. For ITERDEEP (Iterative Deepening) the number of nodes of the symbolic execution tree (S) and the total time in seconds (T), and for TRACER these two numbers and also the column R that shows the number of *restarts* performed by TRACER. A restart occurs when an abstraction for a loop discovered by TRACER is too coarse to prove the program is safe.

Based on the numbers shown in Fig. 1 we can conclude that the overhead of our approach pays off. The main overhead comes basically from the frequent use of the procedure NONPATHINV since this procedure is used whenever the algorithm attempts at subsuming a node. In spite of this, the overhead is quite reasonable. More importantly, the key difference is that our approach can terminate with the programs ssh_clnt.1 and ssh_srvr.2. However, an iterative deepening cannot terminate the proof after 2 hours or 2.5Gb of memory consumption. The reason is similar to the one present in the program of Fig. 1(b) in Sec. 1: disjunctive invariant interpolants are needed for the proof.

Finally, it is worth mentioning that in the current version of TRACER we have not implemented any heuristics in the refinement phase. It is well known that heuristics can have a huge impact in the convergence of the algorithm reducing the number of refinements.

6 Conclusions

We extended symbolic execution with interpolation to address unbounded loops in the context of program verification. The algorithm balances *eager subsumption* in order to prune symbolic paths with the desire of discovering the *strongest loop invariants*, in order to detect earlier infeasible paths. Occasionally certain abstractions are not permitted because of the reachability of error states; this is the underlying mechanism which

then causes *selective unrolling*, that is, the unrolling of a loop along relevant paths only. Moreover, we implemented our algorithm in a prototype called TRACER and presented some experimental evaluation.

References

1. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM. In *IFM'2004*.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, pages 203–213.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
4. N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA '08*, pages 3–14.
5. D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, 2009.
6. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *PLDI'07*, pages 300–309.
7. D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. *Int. J. STTT*, 9:505–525, 2007.
8. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically Generating Inputs of Death. In *CCS '06*, pages 322–335.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterrExample-Guided Abstraction Refinement. In *CAV'00*.
10. W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.
11. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.
12. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.
13. P. Godefroid, A. V. Nori, S. K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. *POPL'10*, pages 43–56.
14. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT '06/FSE-14*, pages 117–127, 2006.
15. W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL'10*, pages 71–82, 2010.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, pages 232–244. ACM Press, 2004.
17. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL'02*, pages 58–70.
18. B. Jacobs and F. Piessens. The Verifast Program Verifier, 2008.
19. J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *23rd AAAI*, pages 297–303. AAAI Press, 2008.
20. J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *15th CP*, volume 5732 of *LNCS*. Springer, 2009.
21. James C. King. Symbolic Execution and Program Testing. *Com. ACM' 76*, pages 385–394.
22. K. L. McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136.
23. K. L. McMillan. Lazy annotation for program testing and verification. In *22nd CAV*, 2010.
24. A. Podelski and A. Rybalchenko. ARMC. In *PADL'07*.