SMT-Based Model Checking for Recursive Programs

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract. We present an SMT-based symbolic model checking algorithm for safety verification of recursive programs. The algorithm is modular and analyzes procedures individually. Unlike other SMT-based approaches, it maintains both *over-* and *under-approximations* of procedure summaries. Under-approximations are used to analyze procedure calls without inlining. Over-approximations are used to block infeasible counterexamples and detect convergence to a proof. We show that for programs and properties over a decidable theory, the algorithm is guaranteed to find a counterexample, if one exists. However, efficiency depends on an oracle for quantifier elimination (QE). For Boolean Programs, the algorithm is a polynomial decision procedure, matching the worst-case bounds of the best BDD-based algorithms. For Linear Arithmetic (integers and rationals), we give an efficient instantiation of the algorithm by applying QE *lazily*. We use existing interpolation techniques to over-approximate QE and introduce *Model Based Projection* to under-approximate QE. Empirical evaluation on SV-COMP benchmarks shows that our algorithm improves significantly on the state-of-the-art.

1 Introduction

We are interested in the problem of *safety* of recursive programs, *i.e.*, deciding whether a assertion always holds. The first step in Software Model Checking is to approximate the input program by a program model where the program operations are terms in a first-order theory $\mathcal{D}$. Many program models exist today, e.g., *Boolean Programs* [6] of SLAM [5], Goto programs of CBMC [14], BoogiePL of Boogie [7], and, indirectly, internal representations of many tools such as UFO [1], HSF [21], etc. Given a safety property and a program model over $\mathcal{D}$, it is possible to analyze bounded executions using an oracle for *Satisfiability Modulo Theories* (SMT) for $\mathcal{D}$. However, in the presence of unbounded recursion, safety is undecidable in general. Throughout this paper, we assume that procedures cannot be passed as parameters.

There exist several program models where safety is efficiently decidable[1], e.g., Boolean Programs with unbounded recursion and the unbounded use of stack [36,6]. The general observation behind these algorithms is that one can *summarize* the input-output behavior of a procedure. A summary of a procedure is an input-output relation describing what is currently known about its behavior. Thus,

[1] This is no longer true when we allow procedures as parameters [12].

A. Biere and R. Bloem (Eds.): CAV 2014, LNCS 8559, pp. 17–34, 2014.

a summary can be used to analyze a procedure call without inlining or analyzing the body of the callee [11,37]. For a Boolean Program, the number of states is finite and hence, a summary can only be updated finitely many times. This observation led to a number of efficient algorithms that are polynomial in the number of states, e.g., the RHS framework [36], recursive state machines [4], and symbolic BDD-based algorithms of Bebop [6] and Moped [19]. When safety is undecidable (e.g., when $\mathcal{D}$ is Linear Rational Arithmetic (LRA) or Linear Integer Arithmetic (LIA)), several existing software model checkers work by iteratively obtaining Boolean Program abstractions using Predicate Abstraction [13,5]. In this paper, we are interested in an alternative algorithm that works directly on the original program model without an explicit step of Boolean abstraction. Despite the undecidability, we are interested in an algorithm that is guaranteed to find a counterexample to safety, if one exists.

Several algorithms have been recently proposed for verifying recursive programs without predicate abstraction. Notable examples are Whale [2], HSF [21], GPDR [27], Ultimate Automizer [24,25] and Duality [33]. With the exception of GPDR, these algorithms are based on a combination of Bounded Model Checking (BMC) [8] and Craig Interpolation [16]. First, they use an SMT-solver to check for a bounded counterexample, where the bound is on the depth of the call stack (*i.e.*, the number of nested procedure calls). Second, they use (tree) interpolation to over-approximate procedure summaries. This is repeated with increasing values of the bound until a counterexample is found or the approximate summaries are inductive. The reduction to BMC ensures that the algorithms are guaranteed to find a counterexample. However, the size of the SMT instance grows exponentially with the bound on the call-stack (*i.e.*, linear in the size of the call tree). Therefore, for Boolean Programs, these algorithms are at least worst-case exponential in the number of states.

On the other hand, GPDR follows the approach of IC3 [9] by solving BMC incrementally without unrolling the call-graph. Interpolation is used to over-approximate summaries and caching is used to indirectly under-approximate them. For some configurations, GPDR is worst-case polynomial for Boolean Programs. However, even for LRA, GPDR might fail to find a counterexample [28].

In this paper, we introduce RecMC, the first SMT-based algorithm for model checking safety of recursive programs that is worst-case polynomial (in the number of states) for Boolean Programs while being a co-semidecision procedure for programs over decidable theories (see Section 4). Our main insight is to maintain not only over-approximations of procedure summaries (which we call *summary facts*), but also their under-approximations (which we call *reachability facts*). While summary facts are used to block spurious counterexamples, reachability facts are used to analyze a procedure call without inlining or analyzing the body of the callee. Our use of reachability facts is similar to that of *summary edges* of the RHS [36] algorithm. This explains our complexity result for Boolean Programs. However, our summary facts make an important difference. While the use of summary facts is an interesting heuristic for Boolean Programs that does not improve the worst-case complexity, it is crucial for richer theories.

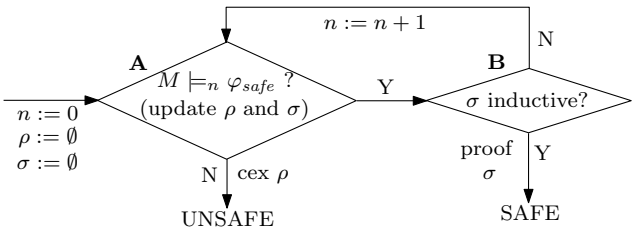

Fig. 1. Flow of the algorithm RecMC to check if $M \models \varphi_{safe}$

Almost every step of RecMC results in existential quantification of variables. RecMC tries to eliminate these variables, as otherwise, they would accumulate and the size of an inferred reachability fact, for example, grows exponentially in the bound on the call-stack. But, a naïve use of quantifier elimination (QE) is expensive. Instead, we develop an alternative approach that under-approximates QE. However, obtaining arbitrary under-approximations can lead to divergence of the algorithm. We introduce the concept of *Model Based Projection* (MBP), for *covering* $\exists \overline{x} \cdot \varphi(\overline{x}, \overline{y})$ by *finitely-many* quantifier-free under-approximations obtained using models of $\varphi(\overline{x}, \overline{y})$. We developed efficient MBPs (see Section 5) for Linear Arithmetic based on the QE methods by Loos-Weispfenning [31] for LRA and Cooper [15] for LIA. We use MBP to under-approximate reachability facts in RecMC. In the best case, only a partial under-approximation is needed and a complete quantifier elimination can be avoided.

We have implemented RecMC as part of our tool Spacer using the framework of Z3 [17] and evaluated it on 799 benchmarks from SV-COMP [38]. Spacer significantly outperforms the implementation of GPDR in Z3 (see Section 6).

In summary, our contributions are: (a) an efficient SMT-based algorithm for model checking recursive programs, that analyzes procedures individually using under- and over-approximations of procedure summaries, (b) MBP functions for under-approximating quantifier elimination for LRA and LIA, (c) a new, complete algorithm for Boolean Programs, with complexity polynomial in the number of states, similar to the best known method [6], and (d) an implementation and an empirical evaluation of the approach.

2 Overview

In this section, we give an overview of RecMC and illustrate it on an example. Let $\mathcal{A}$ be a recursive program. For simplicity of presentation, assume no loops, no global variables and that arguments are passed by reference. Let $P(\overline{v}) \in \mathcal{A}$ be a procedure with parameters $\overline{v}$ and let $\overline{v}_0$ be fresh variables not appearing in P with $|\overline{v}| = |\overline{v}_0|$. A safety property for P is an assertion $\varphi(\overline{v}_0, \overline{v})$. We say that P satisfies φ, denoted $P(\overline{v}) \models \varphi(\overline{v}_0, \overline{v})$, iff the Hoare-triple $\{\overline{v} = \overline{v}_0\}\ P(\overline{v})\ \{\varphi(\overline{v}_0, \overline{v})\}$ is valid. Note that every Hoare-triple corresponds to a safety property in this sense, as shown by Clarke [11], using a *Rule of Adaptation*. Given a safety property φ and a natural number $n \geq 0$, the problem of *bounded safety* is to determine

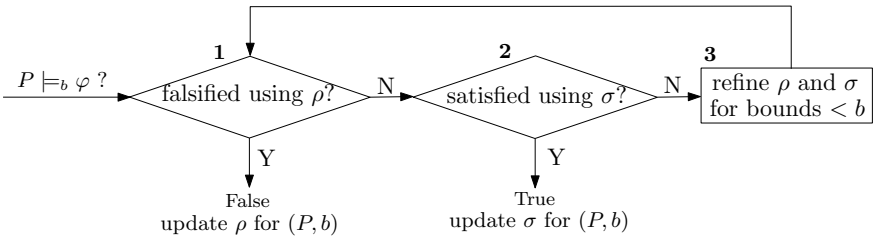

Fig. 2. Flow of the algorithm BndSafety to check $P \models_b \varphi$

```
M (m) {
    T (m);
    D (m);
    D (m); }

T (t) {
    if (t>0) {
        t := t-2;
        T (t);
        t := t+1; } }

D (d) {
    d := d-1;
}
```

Fig. 3. A recursive program with 3 procedures

whether all executions of P using a call-stack bounded by n satisfy φ. We use $P(\overline{v}) \models_n \varphi(\overline{v}_0, \overline{v})$ to denote bounded safety.

The key steps of RecMC are shown in Fig. 1. RecMC decides safety for the main procedure M of $\mathcal{A}$. RecMC maintains two *assertion maps* ρ and σ. The *reachability* map ρ maps each procedure $P(\overline{v}) \in \mathcal{A}$ to a set of assertions over $\overline{v}_0 \cup \overline{v}$ that under-approximate its behavior. Similarly, the *summary* map σ maps a procedure P to a set of assertions that over-approximate its behavior. Given P, the maps are partitioned according to the bound on the call-stack. That is, if $\delta(\overline{v}_0, \overline{v}) \in \rho(P, n)$ for $n \geq 0$, then for every model m of δ, there is an execution of P that begins in $m(\overline{v}_0)$ and ends in $m(\overline{v})$, using a call-stack bounded by n. Similarly, if $\delta(\overline{v}_0, \overline{v}) \in \sigma(P, n)$, then $P(\overline{v}) \models_n \delta(\overline{v}_0, \overline{v})$.

RecMC alternates between two steps: (**A**) deciding bounded safety (that also updates ρ and σ maps) and (**B**) checking whether the current proof of bounded safety is inductive (*i.e.*, independent of the bound). It terminates when a counterexample or a proof is found.

Bounded safety, $P \models_b \varphi$, is decided using BndSafety shown in Fig. 2. Step **1** checks whether φ is falsified by current reachability facts in ρ of the callees of P. If so, it infers a new reachability fact for P at bound b witnessing the falsification of φ. Step **2** checks whether φ is satisfied using current summary facts in σ of the callees. If so, it infers a new summary fact for P at bound b witnessing the satisfaction of φ. If the prior two steps fail, there is a potential counterexample π in P with a call to some procedure R such that the reachability facts of R are too strong to witness π, but the summary facts of R are too weak to block it. Step **3** updates ρ and σ by creating (and recursively deciding) a new bounded safety problem for R at bound $b - 1$.

We conclude this section with an illustration of RecMC on the program in Fig. 3 (adapted from [11]). The program has 3 procedures: the main procedure `M`, and procedures `T` and `D`. `M` calls `T` and `D`. `T` modifies its argument `t` and calls itself recursively. `D` decrements its argument `d`. Let the property be $\varphi = m_0 \geq 2m + 4$.

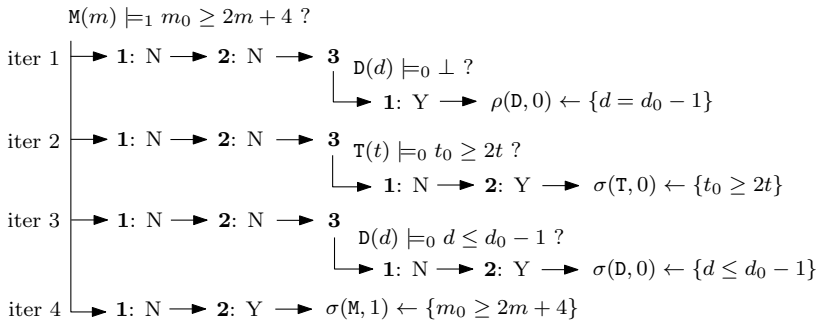

Fig. 4. A run of BndSafety on program in Fig. 3 and a bound 1 on the stack depth. Numbers in bold refer to the steps in Fig. 2.

The first iteration of RecMC is trivial. The bound $n = 0$ and since M has no call-free executions it vacuously satisfies any bounded safety property. Fig. 4 shows the four iterations of BndSafety for the second iteration of RecMC where $n = 1$. For this bound, the maps ρ and σ are initially empty. The first iteration of BndSafety finds a potential counterexample path in M and the approximation for D is updated with a new reachability fact: $d = d_0 - 1$. In the second iteration, the approximation for T is updated. Note that the two calls to D are "jumped over" using the reachability fact for D computed in the first iteration. The new summary fact for T is: $t_0 \geq 2t$. In the third iteration, the approximation for D is updated again, now with a summary fact $d \leq d_0 - 1$. Finally, the summary facts for T and D at bound 0 are sufficient to establish bounded safety at $n = 1$. At this point, the summary map σ is:

$$\sigma(\mathtt{M}, 1) = \{m_0 \geq 2m + 4\} \quad \sigma(\mathtt{T}, 0) = \{t_0 \geq 2t\} \quad \sigma(\mathtt{D}, 0) = \{d \leq d_0 - 1\}$$

Ignoring the bounds, σ is inductive. For example, we can prove that the body of T satisfies $t_0 \geq 2t$, assuming that the calls do. Thus, step **B** of RecMC succeeds and the algorithm terminates declaring the program SAFE. In the rest of the paper, we show how to automate RecMC using an SMT-oracle.

3 Preliminaries

Consider a first-order language with equality and let $\mathcal{S}$ be its signature, *i.e.*, the set of non-logical function and predicate symbols (including equality). An *$\mathcal{S}$-structure* I consists of a domain of interpretation, denoted $|I|$, and assigns elements of $|I|$ to variables, and functions and predicates on $|I|$ to the symbols of $\mathcal{S}$. Let φ be a formula. We assume the usual definition of satisfaction of φ by I, denoted $I \models \varphi$. I is called a *model* of φ iff $I \models \varphi$ and this can be extended to a set of formulas. A first-order *$\mathcal{S}$-theory* Th is a set of deductively closed $\mathcal{S}$-sentences. I satisfies φ modulo Th, denoted $I \models_{Th} \varphi$, iff $I \models Th \cup \{\varphi\}$. φ is *valid* modulo Th, denoted $\models_{Th} \varphi$, iff every model of Th is also a model of φ.

Let I be an $\mathcal{S}$-structure and $\overline{w}$ be a list of fresh function/predicate symbols not in $\mathcal{S}$. A $(\mathcal{S} \cup \overline{w})$-structure J is called an *expansion* of I to $\overline{w}$ iff $|J| = |I|$ and

J agrees with I on the assignments to all variables and the symbols of $\mathcal{S}$. We use the notation $I\{\overline{w} \mapsto \overline{u}\}$ to denote the expansion of I to $\overline{w}$ that assigns the function/predicate u_i to the symbol w_i. For an $\mathcal{S}$-sentence φ, we write $I(\varphi)$ to denote the truth value of φ under I. For a formula $\varphi(\overline{x})$ with free variables $\overline{x}$, we overload the notation $I(\varphi)$ to mean $\{\overline{a} \in |I|^{|\overline{x}|} \mid I\{\overline{x} \mapsto \overline{a}\} \models \varphi\}$. For simplicity of presentation, we sometimes identify the truth value *true* with $|I|$ and *false* with $\emptyset$.

We assume that programs do not have internal procedures and that procedures cannot be passed as parameters. Furthermore, without loss of generality, we assume that programs do not have loops or global variables. In the following, we define programs using a logical representation, as opposed to giving a concrete syntax. A *program* $\mathcal{A}$ is a finite list of procedures with a designated *main* procedure M where the program begins. A *procedure* P is a tuple $\langle \overline{\imath}_P, \overline{o}_P, \Sigma_P, \overline{\ell}_P, \beta_P \rangle$, where (a) $\overline{\imath}_P$ is the finite list of variables denoting the input values of the parameters, (b) $\overline{o}_P$ is the finite list of variables denoting the output values of the parameters, (c) Σ_P is a fresh predicate symbol of arity $|\overline{\imath}_P| + |\overline{o}_P|$, (d) $\overline{\ell}_P$ is the finite list of local variables, and (e) β_P is a quantifier-free sentence over the signature $(\mathcal{S} \cup \{\Sigma_Q \mid Q \in \mathcal{A}\} \cup \overline{\imath}_P \cup \overline{o}_P \cup \overline{\ell}_P)$ in which a predicate symbol Σ_Q appears only positively. We use $\overline{v}_P$ to denote $\overline{\imath}_P \cup \overline{o}_P$.

Intuitively, for a procedure P, Σ_P is used to denote its semantics and β_P encodes its body using the predicate symbol Σ_Q for a call to the procedure Q. We require that a predicate symbol Σ_Q appears only positively in β_P to ensure a fixed-point characterization of the semantics as shown later on. For example, for the signature $\mathcal{S} = \langle 0, Succ, -, +, \leq, >, = \rangle$, the program in Fig. 3 is represented as $\langle M, T, D \rangle$ with $M = \langle m_0, m, \Sigma_M, \langle \ell_0, \ell_1 \rangle, \beta_M \rangle$, $T = \langle t_0, t, \Sigma_T, \langle \ell_0, \ell_1 \rangle, \beta_T \rangle$ and $D = \langle d_0, d, \Sigma_D, \emptyset, \beta_D \rangle$, where

$$\begin{aligned} &\beta_M = \Sigma_T(m_0, \ell_0) \wedge \Sigma_D(\ell_0, \ell_1) \wedge \Sigma_D(\ell_1, m) \qquad \beta_D = (d = d_0 - 1) \\ &\beta_T = (t_0 \leq 0 \wedge t_0 = t) \ \vee \ (t_0 > 0 \wedge \ell_0 = t_0 - 2 \wedge \Sigma_T(\ell_0, \ell_1) \wedge t = \ell_1 + 1) \end{aligned} \tag{1}$$

Here, we abbreviate $Succ^i(0)$ by i and (m_0, t_0, d_0) and (m, t, d) denote the input and the output values of the parameters of the original program, respectively. For a procedure P, let $Paths(P)$ denote the set of all prime-implicants of β_P. Intuitively, each element of $Paths(P)$ encodes a path in the procedure.

Let $\mathcal{A} = \langle P_0, \ldots, P_n \rangle$ be a program and I be an $\mathcal{S}$-structure. Let $\overline{X}$ be a list of length n such that each X_i is either (i) a truth value if $|\overline{v}_{P_i}| = 0$, or (ii) a subset of $|I|^{|\overline{v}_{P_i}|}$ if $|\overline{v}_{P_i}| \geq 1$. Let $J(I, \overline{X})$ denote the expansion $I\{\Sigma_{P_0} \mapsto X_0\} \ldots \{\Sigma_{P_n} \mapsto X_n\}$. The *semantics* of a procedure P_i given I, denoted $[\![P_i]\!]_I$, characterizes all the terminating executions of P_i and is defined as follows. $\langle [\![P_0]\!]_I, \ldots, [\![P_n]\!]_I \rangle$ is the (pointwise) least $\overline{X}$ such that for all $Q \in \mathcal{A}$, $J(I, \overline{X}) \models \forall \overline{v}_Q \cup \overline{\ell}_Q \cdot (\beta_Q \Rightarrow \Sigma_Q(\overline{v}_Q))$. This has a well-known least fixed-point characterization [11].

For a bound $b \geq 0$ on the call-stack, the *bounded semantics* of a procedure P_i given I, denoted $[\![P_i]\!]^b_I$, characterizes all the executions using a stack of depth bounded by b and is defined by induction on b:

$$[\![P_i]\!]^0_I = J(I, \langle \emptyset, \ldots, \emptyset \rangle)(\exists \overline{\ell}_{P_i} \cdot \beta_{P_i}), \quad [\![P_i]\!]^b_I = J(I, \langle [\![P_0]\!]^{b-1}_I, \ldots, [\![P_n]\!]^{b-1}_I \rangle)(\exists \overline{\ell}_{P_i} \cdot \beta_{P_i})$$

An *environment* is a function that maps a predicate symbol Σ_P to a formula over $\overline{v}_P$. Given a formula τ and an environment E, we abuse the notation $[\![\cdot]\!]$ and write $[\![\tau]\!]_E$ for the formula obtained by instantiating every predicate symbol Σ_P by $E(\Sigma_P)$ in τ.

Let Th be an $\mathcal{S}$-theory. A *safety property* for a procedure $P \in \mathcal{A}$ is a formula over $\overline{v}_P$. P satisfies a safety property φ w.r.t Th, denoted $P \models_{Th} \varphi$, iff for all models I of Th, $[\![P]\!]_I \subseteq I(\varphi)$. A *safety property* ψ of the program $\mathcal{A}$ is a safety property of its main procedure. A *safety proof* for $\psi(\overline{v}_M)$ is an environment Π that is both safe and inductive:

$$\models_{Th} [\![\forall \overline{x} \cdot \Sigma_M(\overline{x}) \Rightarrow \psi(\overline{x})]\!]_\Pi, \quad \forall P \in \mathcal{A} \cdot \models_{Th} [\![\forall \overline{v}_P \cup \overline{\ell}_P \cdot (\beta_P \Rightarrow \Sigma_P(\overline{v}_P))]\!]_\Pi$$

Given a formula $\varphi(\overline{v}_P)$ and $b \geq 0$, a procedure P satisfies *bounded safety* w.r.t Th, denoted $P \models_{b,Th} \varphi$, iff for all models I of Th, $[\![P]\!]^b_I \subseteq I(\varphi)$. In this case, we also call φ a *summary fact* for $\langle P, b\rangle$. We call φ a *reachability fact* for $\langle P, b\rangle$ iff $I(\varphi) \subseteq [\![P]\!]^b_I$, for all models I of Th. Intuitively, *summary facts* and *reachability facts* for $\langle P, b\rangle$, respectively, over- and under-approximate $[\![P]\!]^b_I$ for every model I of Th.

A *bounded assertion map* maps a procedure P and a natural number $b \geq 0$ to a set of formulas over $\overline{v}_P$. Given a bounded assertion map m and $b \geq 0$, we define two special environments U^b_m and O^b_m as follows.

$$U^b_m : \Sigma_P \mapsto \bigvee\{\delta \in m(P, b') \mid b' \leq b\} \qquad O^b_m : \Sigma_P \mapsto \bigwedge\{\delta \in m(P, b') \mid b' \geq b\}$$

We use U^b_m and O^b_m to under- and over-approximate the bounded semantics. For convenience, let U^{-1}_m and O^{-1}_m be environments that map every symbol to $\bot$.

4 Model Checking Recursive Programs

In this section, we present our algorithm RecMC$(\mathcal{A}, \varphi_{safe})$ that determines whether a program $\mathcal{A}$ satisfies a safety property φ_{safe}. Let $\mathcal{S}$ be the signature of the first-order language under consideration and assume a fixed $\mathcal{S}$-theory Th. To avoid clutter, we drop the subscript Th from the notation $\models_{Th}$ and $\models_{b,Th}$. We also establish the soundness and complexity of RecMC. An efficient instantiation of RecMC to Linear Arithmetic is presented in Section 5.

Main Loop. RecMC maintains two *bounded assertion maps* ρ and σ for reachability and summary facts, respectively. For brevity, for a first-order formula τ, we write $[\![\tau]\!]^b_\rho$ and $[\![\tau]\!]^b_\sigma$ to denote $[\![\tau]\!]_{U^b_\rho}$ and $[\![\tau]\!]_{O^b_\sigma}$, respectively, where the environments U^b_ρ and O^b_σ are as defined in Section 3. Intuitively, $[\![\tau]\!]^b_\rho$ and $[\![\tau]\!]^b_\sigma$, respectively, under- and over-approximate τ using ρ and σ.

The pseudo-code of the main loop of RecMC (corresponding to the flow diagram in Fig. 1) is shown in Fig. 5. RecMC follows an *iterative deepening* strategy. In each iteration, BndSafety (described below) checks whether all executions of $\mathcal{A}$ satisfy φ_{safe} for a bound $n \geq 0$ on the call-stack, *i.e.*, if $M \models_n$

```
RecMC(A, φ_safe)
1  n ← 0 ; ρ ← ∅ ; σ ← ∅
2  while true do
3      res, ρ, σ ← BndSafety(A, φ_safe, n, ρ, σ)
4      if res is UNSAFE then
5          return UNSAFE, ρ
       else
6          ind, σ ← CheckInductive(A, σ, n)
7          if ind then
8              return SAFE, σ
9          n ← n + 1

CheckInductive(A, σ, n)
10 ind ← true
11 foreach P ∈ A do
12     foreach δ ∈ σ(P, n) do
13         if ⊨ [[β_P]]^n_σ ⇒ δ then
14             σ ← σ ∪ (⟨P, n + 1⟩ ↦ δ)
           else
15             ind ← false
16 return (ind, σ)
```

Fig. 5. Pseudo-code of RecMC

φ_{safe}. BndSafety also updates the maps ρ and σ. Whenever BndSafety returns *UNSAFE*, the reachability facts in ρ are sufficient to construct a counterexample and the loop terminates. Whenever BndSafety returns *SAFE*, the summary facts in σ are sufficient to prove the absence of a counterexample for the current bound n on the call-stack. In this case, if σ is also inductive, as determined by CheckInductive, O^n_σ is a safety proof and the loop terminates. Otherwise, the bound on the call-stack is incremented and a new iteration of the loop begins. Note that, as a side-effect of CheckInductive, some summary facts are propagated to the bound $n + 1$. This is similar to *push generalization* in IC3 [9].

Bounded Safety. We describe the routine $\text{BndSafety}(\mathcal{A}, \varphi_{safe}, n, \rho_{Init}, \sigma_{Init})$ as an abstract transition system [34] defined by the inference rules shown in Fig. 6. Here, n is the current bound on the call-stack and ρ_{Init} and σ_{Init} are the maps of reachability and summary facts input to the routine. A state of BndSafety is a triple $\mathcal{Q} \parallel \rho \parallel \sigma$, where ρ and σ are the current maps and $\mathcal{Q}$ is a set of triples $\langle P, \varphi, b\rangle$ for a procedure P, a formula φ over $\overline{v}_P$, and a number $b \geq 0$. A triple $\langle P, \varphi, b\rangle \in \mathcal{Q}$ is called a *bounded reachability query* and asks whether $P \not\models_b \neg\varphi$, *i.e.*, whether there is an execution in P using a call-stack bounded by b where the values of $\overline{v}_P$ satisfy φ.

BndSafety starts with a single query $\langle M, \neg\varphi_{safe}, n\rangle$ and initializes the maps of reachability and summary facts (rule Init). It checks whether $M \models_n \varphi_{safe}$ by inferring new summary and reachability facts to answer existing queries (rules Sum and Reach) and generating new queries (rule Query). When there are no queries left to answer, *i.e.*, $\mathcal{Q}$ is empty, it terminates with a result of either *UNSAFE* or *SAFE* (rules Unsafe and Safe).

Sum infers a new summary fact when a query $\langle P, \varphi, b\rangle$ can be answered negatively. In this case, there is an over-approximation of the bounded semantics of P at b, obtained using the summary facts of callees at bound $b - 1$, that is unsatisfiable with φ. That is, $\models [\![\beta_P]\!]^{b-1}_\sigma \Rightarrow \neg\varphi$. The inference of the new fact is by interpolation [16] (denoted by Itp in the side-condition of the rule). Thus, the new summary fact ψ is a formula over $\overline{v}_P$ such that $\models \left([\![\beta_P]\!]^{b-1}_\sigma \Rightarrow \psi(\overline{v}_P)\right) \ \wedge \ (\psi(\overline{v}_P) \Rightarrow \neg\varphi)$. Note that ψ over-approximates the bounded semantics of P at b. Every query $\langle P, \eta, c\rangle \in \mathcal{Q}$ such that η is unsatisfiable with the updated environment $O^c_\sigma(\Sigma_P)$ is immediately answered and removed.

$$\textsc{Init}\ \frac{}{\{\langle M, \neg\varphi_{safe}, n\rangle\} \parallel \rho_{Init} \parallel \sigma_{Init}}$$

$$\textsc{Sum}\ \frac{\mathcal{Q} \parallel \rho \parallel \sigma \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \models [\![\beta_P]\!]_\sigma^{b-1} \Rightarrow \neg\varphi}{\mathcal{Q} \setminus \{\langle P, \eta, c\rangle \mid c \leq b, \models [\![\Sigma_P]\!]_\sigma^c \wedge \psi \Rightarrow \neg\eta\} \parallel \rho \parallel \sigma \cup \{\langle P, b\rangle \mapsto \psi\}}$$

where $\psi = \textsc{Itp}([\![\beta_P]\!]_\sigma^{b-1}, \neg\varphi)$

$$\textsc{Reach}\ \frac{\mathcal{Q} \parallel \rho \parallel \sigma \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \pi \in \mathit{Paths}(P) \qquad \not\models [\![\pi]\!]_\rho^{b-1} \Rightarrow \neg\varphi}{\mathcal{Q} \setminus \{\langle P, \eta, c\rangle \mid c \geq b, \not\models \psi \Rightarrow \neg\eta\} \parallel \rho \cup \{\langle P, b\rangle \mapsto \psi\} \parallel \sigma}$$

where $\psi = \exists \overline{\ell}_P \cdot [\![\pi]\!]_\rho^{b-1}$

$$\textsc{Query}\ \frac{\begin{array}{c}\mathcal{Q} \parallel \rho \parallel \sigma \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \models [\![\beta_P]\!]_\rho^{b-1} \Rightarrow \neg\varphi \qquad \pi \in \mathit{Paths}(P) \\ \pi = \pi_u \wedge \Sigma_R(\overline{a}) \wedge \pi_v \qquad \models [\![\pi_u]\!]_\sigma^{b-1} \wedge [\![\Sigma_R(\overline{a})]\!]_\rho^{b-1} \wedge [\![\pi_v]\!]_\rho^{b-1} \Rightarrow \neg\varphi \\ \not\models [\![\pi_u]\!]_\sigma^{b-1} \wedge [\![\Sigma_R(\overline{a})]\!]_\sigma^{b-1} \wedge [\![\pi_v]\!]_\rho^{b-1} \Rightarrow \neg\varphi\end{array}}{\mathcal{Q} \cup \{\langle R, \psi, b-1\rangle\} \parallel \rho \parallel \sigma}$$

$$\text{where } \begin{cases} \psi = \left(\exists\, (\overline{v}_P \cup \overline{\ell}_P) \setminus \overline{a} \cdot [\![\pi_u]\!]_\sigma^{b-1} \wedge [\![\pi_v]\!]_\rho^{b-1} \wedge \varphi\right)[\overline{a} \leftarrow \overline{v}_R] \\ \text{for all } \langle R, \eta, b-1\rangle \in \mathcal{Q}, \models \psi \Rightarrow \neg\eta \end{cases}$$

$$\textsc{Unsafe}\ \frac{\emptyset \parallel \rho \parallel \sigma \qquad \not\models [\![\Sigma_M]\!]_\rho^n \Rightarrow \varphi_{safe}}{\mathit{UNSAFE}} \qquad \textsc{Safe}\ \frac{\emptyset \parallel \rho \parallel \sigma \qquad \models [\![\Sigma_M]\!]_\sigma^n \Rightarrow \varphi_{safe}}{\mathit{SAFE}}$$

Fig. 6. Rules defining $\textsc{BndSafety}(\mathcal{A}, \varphi_{safe}, n, \rho_{Init}, \sigma_{Init})$

Reach infers a new reachability fact when a query $\langle P, \varphi, b\rangle$ can be answered positively. In this case, there is an under-approximation of the bounded semantics of P at b, obtained using the reachability facts of callees at bound $b-1$, that is satisfiable with φ. That is, $\not\models [\![\beta_P]\!]_\rho^{b-1} \Rightarrow \neg\varphi$. In particular, there exists a path π in $\mathit{Paths}(P)$ such that $\not\models [\![\pi]\!]_\rho^{b-1} \Rightarrow \neg\varphi$. The new reachability fact ψ is obtained by choosing such a π non-deterministically and existentially quantifying all local variables from $[\![\pi]\!]_\rho^{b-1}$. Note that ψ under-approximates the bounded semantics of P at b. Every query $\langle P, \eta, c\rangle \in \mathcal{Q}$ such that η is satisfiable with the updated environment $U_\rho^c(\Sigma_P)$ is immediately answered and removed.

Query creates a new query when a query $\langle P, \varphi, b\rangle$ cannot be answered using current ρ and σ. In this case, the current over-approximation of the bounded semantics of P at b is satisfiable with φ while its current under-approximation is unsatisfiable with φ. That is, $\not\models [\![\beta_P]\!]_\sigma^{b-1} \Rightarrow \neg\varphi$ and $\models [\![\beta_P]\!]_\rho^{b-1} \Rightarrow \neg\varphi$. In particular, there exists a path π in $\mathit{Paths}(P)$ such that $\not\models [\![\pi]\!]_\sigma^{b-1} \Rightarrow \neg\varphi$ and $\models [\![\pi]\!]_\rho^{b-1} \Rightarrow \neg\varphi$. Intuitively, π is a potential counterexample path that needs to be checked for feasibility. Such a π is chosen non-deterministically. π is guaranteed to have a call $\Sigma_R(\overline{a})$ to a procedure R such that the under-approximation $[\![\Sigma_R(\overline{a})]\!]_\rho^{b-1}$ is too strong to witness π but the over-approximation $[\![\Sigma_R(\overline{a})]\!]_\sigma^{b-1}$

	π_i	$[\![\pi_i]\!]^0_\rho$	$[\![\pi_i]\!]^0_\sigma$
$i=1$	$\Sigma_T(m_0,\ell_0)$	$\bot$	$\top$
$i=2$	$\Sigma_D(\ell_0,\ell_1)$	$\ell_1 = \ell_0 - 1$	$\top$
$i=3$	$\Sigma_D(\ell_1,m)$	$m = \ell_1 - 1$	$\top$

Fig. 7. Approximations of the only path π of the procedure M in Fig. 3

is too weak to block it. That is, π can be partitioned into a prefix π_u, a call $\Sigma_R(\overline{a})$ to R, and a suffix π_v such that the following hold:

$$\models [\![\Sigma_R(\overline{a})]\!]^{b-1}_\rho \Rightarrow \left(([\![\pi_u]\!]^{b-1}_\sigma \wedge [\![\pi_v]\!]^{b-1}_\rho) \Rightarrow \neg\varphi\right)$$
$$\not\models [\![\Sigma_R(\overline{a})]\!]^{b-1}_\sigma \Rightarrow \left(([\![\pi_u]\!]^{b-1}_\sigma \wedge [\![\pi_v]\!]^{b-1}_\rho) \Rightarrow \neg\varphi\right)$$

Note that the prefix π_u and the suffix π_v are over- and under-approximated, respectively. A new query $\langle R, \psi, b-1\rangle$ is created where ψ is obtained by existentially quantifying all variables from $[\![\pi_u]\!]^{b-1}_\sigma \wedge [\![\pi_v]\!]^{b-1}_\rho \wedge \varphi$ except the arguments $\overline{a}$ of the call, and renaming appropriately. If the new query is answered negatively (using Sum), all executions along π where the values of $\overline{v}_P \cup \overline{\ell}_P$ satisfy $[\![\pi_v]\!]^{b-1}_\rho$ are spurious counterexamples. An additional side-condition requires that ψ "does not overlap" with η for any other query $\langle R, \eta, b-1\rangle$ in $\mathcal{Q}$. This is necessary for termination of BndSafety (Theorem 2). In practice, the side-condition is trivially satisfied by always applying the rule to $\langle P, \varphi, b\rangle$ with the smallest b.

For example, consider the program in Fig. 3 represented by (1) and the query $\langle M, \varphi, 1\rangle$ where $\varphi \equiv m_0 < 2m + 4$. Let $\sigma = \emptyset$, $\rho(D, 0) = \{d = d_0 - 1\}$ and $\rho(T, 0) = \emptyset$. Let $\pi = (\Sigma_T(m_0, \ell_0) \wedge \Sigma_D(\ell_0, \ell_1) \wedge \Sigma_D(\ell_1, m))$ denote the only path in the procedure M. Fig. 7 shows $[\![\pi_i]\!]^0_\rho$ and $[\![\pi_i]\!]^0_\sigma$ for each conjunct π_i of π. As the figure shows, $[\![\pi]\!]^0_\sigma$ is satisfiable with φ, witnessed by the execution $e \equiv \langle m_0 = 3, \ell_0 = 3, \ell_1 = 2, m = 1\rangle$. Note that this execution also satisfies $[\![\pi_2 \wedge \pi_3]\!]^0_\rho$. But, $[\![\pi_1]\!]^0_\rho$ is too strong to witness it, where π_1 is the call $\Sigma_T(m_0, \ell_0)$. To create a new query for T, we first existentially quantify all variables other than the arguments m_0 and ℓ_0 from $\pi_2 \wedge \pi_3 \wedge \varphi$, obtaining $m_0 < 2\ell_0$. Renaming the arguments by the parameters of T results in the new query $\langle T, t_0 < 2t, 0\rangle$. Further iterations of BndSafety would answer this query negatively making the execution e spurious. Note that this would also make all other executions where the values to $\langle m_0, \ell_0, \ell_1, m\rangle$ satisfy $[\![\pi_2 \wedge \pi_3]\!]^0_\rho$ spurious.

Soundness and Complexity. Soundness of RecMC follows from that of BndSafety, which can be shown by a case analysis on the inference rules[2].

Theorem 1. BndSafety *and* RecMC *are sound.*

BndSafety is complete relative to an oracle for satisfiability modulo *Th*. Even though the number of reachable states of a procedure is unbounded in general, the number of reachability facts inferred by BndSafety is finite. This is because a reachability fact corresponds to a path (see Reach) and given a bound on the call-stack, the number of such facts is bounded. This further bounds the number of queries that can be created.

[2] Proofs of all of the theorems can be found in the extended version of the paper [28].

Theorem 2. *Given an oracle for Th,* BndSafety$(\mathcal{A}, \varphi, n, \emptyset, \emptyset)$ *terminates.*

As a corollary of Theorem 2, RecMC is a co-semidecision procedure for safety, *i.e.*, RecMC is guaranteed to find a counterexample if one exists. In contrast, the closest related algorithm GPDR [27] is not a co-semidecision procedure [28]. Finally, for Boolean Programs RecMC is a complete decision procedure. Unlike the general case, the number of reachable states of a Boolean Program, and hence the number of reachability facts, is finite and independent of the bound on the call-stack. Let $N = |\mathcal{A}|$ and $k = \max\{|\overline{v}_P| \mid P \in \mathcal{A}\}$.

Theorem 3. *Let $\mathcal{A}$ be a Boolean Program. Then* RecMC$(\mathcal{A}, \varphi)$ *terminates in $O(N^2 \cdot 2^{2k})$-many applications of the rules in Fig. 6.*

Note that due to the iterative deepening strategy of RecMC, the complexity is quadratic in the number of procedures (and not linear as in [6]). In contrast, other SMT-based algorithms, such as Whale [2], are worst-case exponential in the number of states of a Boolean Program.

In summary, RecMC checks safety of a recursive program by inferring the necessary under- and over-approximations of procedure semantics and using them to analyze procedures individually.

5 Model Based Projection

RecMC, as presented in Section 4, can be used as-is, when *Th* is Linear Arithmetic. But, note that the rules Reach and Query introduce existential quantifiers in reachability facts and queries. Unless eliminated, these quantifiers accumulate and the size of the formulas grows exponentially in the bound on the call-stack. Eliminating them using quantifier elimination (QE) is expensive. Instead, we suggest an alternative approach that under-approximates existential quantification with quantifier-free formulas *lazily* and efficiently. We first introduce a model-based under-approximation of QE, called *Model Based Projection* (MBP). Second, we give an efficient (linear in the size of formulas involved) MBP procedure for Linear Rational Arithmetic (LRA). Due to space limitations, MBP for Linear Integer Arithmetic (LIA) is described in the extended version [28]. Finally, we show a modified version of BndSafety that uses MBP instead of existential quantification and show that it is sound and terminating.

Model Based Projection *(MBP).* Let $\lambda(\overline{y})$ be the existentially quantified formula $\exists \overline{x} \cdot \lambda_m(\overline{x}, \overline{y})$ where λ_m is quantifier free. A function $Proj_\lambda$ from models (modulo *Th*) of λ_m to quantifier-free formulas over $\overline{y}$ is a *Model Based Projection* (for λ) iff it has a finite image, $\lambda \equiv \bigvee_{M \models \lambda_m} Proj_\lambda(M)$, and for every model M of λ_m, $M \models Proj_\lambda(M)$.

In other words, $Proj_\lambda$ covers the space of all models of $\lambda_m(\overline{x}, \overline{y})$ by a finite set of quantifier-free formulas over $\overline{y}$. MBP exists for any theory that admits quantifier elimination, because one can first obtain an equivalent quantifier-free formula and map every model to it.

MBP for Linear Rational Arithmetic. We begin with a brief overview of Loos-Weispfenning (LW) method [31] for quantifier elimination in LRA. We borrow our presentation from Nipkow [35] to which we refer the reader for more details. Let $\lambda(\overline{y}) = \exists \overline{x} \cdot \lambda_m(\overline{x}, \overline{y})$ as above. Without loss of generality, assume that $\overline{x}$ is singleton, λ_m is in Negation Normal Form, and x only appears in the literals of the form $\ell < x$, $x < u$, and $x = e$, where ℓ, u, and e are x-free. Let $lits(\lambda)$ denote the literals of λ. The LW-method states that

$$\exists x \cdot \lambda_m(x) \equiv \left(\bigvee_{(x=e)\in lits(\lambda)} \lambda_m[e] \vee \bigvee_{(\ell<x)\in lits(\lambda)} \lambda_m[\ell+\epsilon] \vee \lambda_m[-\infty] \right) \quad (2)$$

where $\lambda_m[\cdot]$ denotes a *virtual substitution* for the literals containing x. Intuitively, $\lambda_m[e]$ covers the case when a literal $(x = e)$ is true. Otherwise, the set of ℓ's in the literals $(\ell < x)$ identify intervals in which x can lie which are covered by the remaining substitutions. We omit the details of the substitution and instead illustrate it on an example. Let λ_m be $(x = e \wedge \phi_1) \vee (\ell < x \wedge x < u) \vee (x < u \wedge \phi_2)$, where $\ell, e, u, \phi_1, \phi_2$ are x-free. Then,

$$\begin{aligned}
\exists x \cdot \lambda_m(x) &\equiv \lambda_m[e] \vee \lambda_m[\ell+\epsilon] \vee \lambda_m[-\infty] \\
&\equiv \big(\phi_1 \vee (\ell < e \wedge e < u) \vee (e < u \wedge \phi_2)\big) \vee \big(\ell < u \vee (\ell < u \wedge \phi_2)\big) \vee \phi_2 \\
&\equiv \phi_1 \vee (\ell < u) \vee \phi_2
\end{aligned}$$

We now define an MBP $LRAProj_\lambda$ for LRA as a map from models of λ_m to disjuncts in (2). Given $M \models \lambda_m$, $LRAProj_\lambda$ picks a disjunct that covers M based on values of the literals of the form $x = e$ and $\ell < x$ in M. Ties are broken by a syntactic ordering on terms (e.g., when $M \models \ell' = \ell$ for two literals $\ell < x$ and $\ell' < x$).

$$LRAProj_\lambda(M) = \begin{cases} \lambda_m[e], & \text{if } (x = e) \in lits(\lambda) \wedge M \models x = e \\ \lambda_m[\ell+\epsilon], & \text{else if } (\ell < x) \in lits(\lambda) \wedge M \models \ell < x \wedge \\ & \forall (\ell' < x) \in lits(\lambda) \cdot M \models ((\ell' < x) \Rightarrow (\ell' \leq \ell)) \\ \lambda_m[-\infty], & \text{otherwise} \end{cases}$$

Theorem 4. *$LRAProj_\lambda$ is a Model Based Projection.*

Note that $LRAProj_\lambda$ is linear in the size of λ. An MBP for LIA can be defined similarly [28] based on Cooper's method [15].

Bounded Safety with MBP. Intuitively, each quantifier-free formula in the image of $Proj_\lambda$ under-approximates λ. As above, we use λ_m for the quantifier-free matrix of λ. We modify the side-condition $\psi = \lambda$ of Reach and Query to use quantifier-free under-approximations as follows: (i) for Reach, the new side-condition is $\psi = Proj_\lambda(M)$ where $M \models \lambda_m \wedge \varphi$, (ii) for Query, the new side-condition is $\psi = Proj_\lambda(M)$ where $M \models \lambda_m \wedge [\![\Sigma_R(a)]\!]_\sigma^{b-1}$. Note that to avoid redundant applications of the rules, we require M to satisfy a formula stronger than λ_m. Intuitively, (i) ensures that the newly inferred reachability

	Slam		Svcomp-1		Svcomp-2		Svcomp-3	
	SAFE	UNSAFE	SAFE	UNSAFE	SAFE	UNSAFE	SAFE	UNSAFE
Spacer	1,721	985	249	509	213	497	234	482
Z3	1,722	997	245	509	208	493	234	477
VBS	1,727	998	252	509	225	500	240	482

Fig. 8. Number of programs verified by Spacer, Z3 and the Virtual Best Solver

fact answers the current query and (ii) ensures that the new query cannot be immediately answered by known facts. In both cases, the required model M can be obtained as a side-effect of discharging the premises of the rules. Soundness of BndSafety is unaffected and termination of BndSafety follows from the image-finiteness of $Proj_\lambda$.

Theorem 5. *Assuming an oracle and an MBP for Th,* BndSafety *is sound and terminating with the modified rules.*

Thus, BndSafety with a linear-time MBP (such as $LRAProj_\lambda$) keeps the size of the formulas small by efficiently inferring only the necessary under-approximations of the quantified formulas.

6 Implementation and Experiments

We have implemented RecMC for analyzing C programs as part of the tool Spacer. The back-end is based on Z3 [18] which is used for SMT-solving and interpolation. It supports propositional logic, linear arithmetic, and bit-vectors (via bit-blasting). The front-end is based on UFO [3]. It converts C programs to the Horn-SMT format of Z3, which corresponds to the logical program representation of Section 3. The implementation and benchmarks are available online[3].

We evaluated Spacer on two sets of benchmarks. The first set contains 2,908 Boolean Programs obtained from the SLAM toolkit[4]. The second contains 799 C programs from the Software Verification Competition (SVCOMP) 2014 [38]. We call this set Svcomp-1. We also evaluated on two variants of Svcomp-1, which we call Svcomp-2 and Svcomp-3, obtained by factoring out parts of the program into procedures and introducing more modularity. We compared Spacer against the implementation of GPDR in Z3. We used a time limit of 30 minutes and a memory limit of 16GB, on an Ubuntu machine with a 2.2 GHz AMD Opteron(TM) Processor 6174 and 516GB RAM. The results are summarized in Fig. 8. Since there are programs verified by only one of the tools, Fig. 8 also reports the number of programs verified by at least one, *i.e.*, the Virtual Best Solver (VBS).

Boolean Program Benchmarks. On most of the SLAM benchmarks, the runtimes of Spacer and Z3 are similar (within 2 minutes). We then evaluated on a Boolean Program from [6] in which the size of the call-tree grows exponentially

[3] `http://www.cs.cmu.edu/~akomurav/projects/spacer/home.html`

[4] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/BOOL/slam.zip`

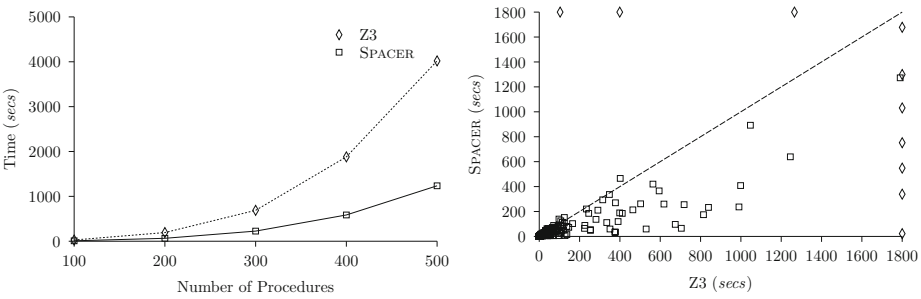

Fig. 9. SPACER vs. Z3 for (a) BEBOP example and (b) SVCOMP-1 benchmarks

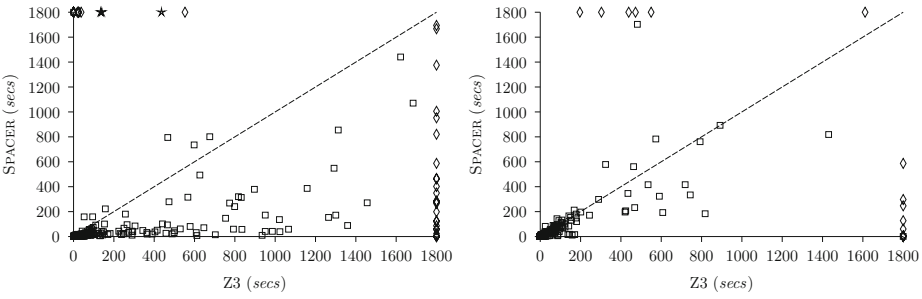

Fig. 10. SPACER vs. Z3 for the benchmarks (a) SVCOMP-2 and (b) SVCOMP-3

in the number of procedures. As Fig. 9(a) shows, SPACER handles the increasing complexity in the example significantly better than Z3.

SVCOMP 2014 Benchmarks. Fig. 9(b), 10(a) and 10(b) show the scatter plots for SVCOMP-1, SVCOMP-2 and SVCOMP-3 benchmarks. A diamond indicates a time-out and a star indicates a mem-out. The plots show that SPACER is significantly better on most of the programs. This shows the practical advantage of the approximations and MBP of RECMC.

7 Related Work

There is a large body of work on interprocedural program analysis. It was pointed out early on that verification of recursive programs is reducible to the computation of a fixed-point over relations (called *summaries*) representing the input-output behavior of each procedure [11,37]. Such procedure summaries are called *partial correctness relations* in [11], and are part of the *functional approach* of [37]. Reps, Horwitz, and Sagiv [36] showed that for a large class of finite interprocedural dataflow problems the summaries can be computed in time polynomial in the number of *facts* and procedures. Ball and Rajamani [6] adapted the

RHS algorithm to the verification of Boolean Programs. Following the SLAM project, other software model checkers – such as BLAST [26] and MAGIC [10] – also implemented the CEGAR loop with predicate abstraction. None used under-approximations of procedure semantics as we do.

Recently, several SMT-based algorithms have been proposed for safety verification of recursive programs, including WHALE [2], HSF [21], Duality [33], Ultimate Automizer [24], and Corral [30]. While these algorithms have been developed independently, they share a similar structure. They use SMT-solvers to look for counterexamples and interpolation to over-approximate summaries. Corral is an exception, which relies on user input and heuristics to supply the summaries. The algorithms differ in the SMT encoding and the heuristics used. However, in the worst-case, they completely unroll the call graph into a tree.

The work closest to ours is Generalized Property Driven Reachability (GPDR) of Hoder and Bjørner [27]. GPDR extends the hardware model checking algorithm IC3 of Bradley [9] to SMT-supported theories and recursive programs. Unlike RECMC, GPDR does not maintain reachability facts. In the context of Fig. 6, this means that ρ is always empty and there is no REACH rule. Instead, the QUERY rule is modified to use a model M that satisfies the premises (instead of our use of the path π when creating a query). Furthermore, the answers to the queries are cached. In the context of Boolean Programs, this ensures that every query is asked at most once (and either cached or blocked by a summary fact). Since there are only finitely many models, the algorithm always terminates. However, in the case of Linear Arithmetic, a formula can have infinitely many models and GPDR might end up applying the QUERY rule indefinitely. In contrast, RECMC creates only finitely many queries for a given bound on the call-stack and is guaranteed to find a counterexample if one exists.

Combination of over- and under-approximations for analysis of procedural programs has also been explored in [23,20]. However, our notion of an under-approximation is very different. Both [23,20] under-approximate summaries by *must transitions*. A must transition is a pair of formulas $\langle \varphi, \psi \rangle$ that under-approximates the summary of a procedure P iff for every state that satisfies φ, P has an execution that ends in a state satisfying ψ. In contrast, our reachability facts are similar to *summary edges* of RHS [36]. A reachability fact is a single formula φ such that every satisfying assignment to φ captures a terminating execution of P.

8 Conclusion

We presented RECMC, a new SMT-based algorithm for model checking safety properties of recursive programs. For programs and properties over decidable theories, RECMC is guaranteed to find a counterexample if one exists. To our knowledge, this is the first SMT-based algorithm with such a guarantee while being polynomial for Boolean Programs. The key idea is to use a combination of under- and over-approximations of the semantics of procedures, avoiding re-exploration of parts of the state-space. We described an efficient instantiation

of RecMC for Linear Arithmetic (over rationals and integers) by introducing *Model-Based Projection* to under-approximate the expensive quantifier elimination. We have implemented it in our tool Spacer and shown empirical evidence that it significantly improves on the state-of-the-art.

In the future, we would like to explore extensions to other theories. Of particular interest are the theory EUF of uninterpreted functions with equality and the theory of arrays. The challenge is to deal with the lack of quantifier elimination. Another direction of interest is to combine RecMC with *Proof-based Abstraction* [32,22,29] to explore a combination of the approximations of procedure semantics with transition-relation abstraction.

Acknowledgment. We thank Edmund M. Clarke and Nikolaj Bjørner for many helpful discussions. Our definition of MBP is based on the idea of projected implicants co-developed with Nikolaj. We thank Cesare Tinelli and the anonymous reviewers for insightful comments. This research was sponsored by the National Science Foundation grants no. DMS1068829, CNS0926181 and CNS0931985, the GSRC under contract no. 1041377, the Semiconductor Research Corporation under contract no. 2005TJ1366, the Office of Naval Research under award no. N000141010188 and the CMU-Portugal Program. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense. This material has been approved for public release and unlimited distribution. DM-0000973.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From Under-Approximations to Over-Approximations and Back. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 157–172. Springer, Heidelberg (2012)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 39–55. Springer, Heidelberg (2012)
3. Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., Chechik, M.: UFO: Verification with Interpolants and Abstract Interpretation. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 637–640. Springer, Heidelberg (2013)
4. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of Recursive State Machines. TOPLAS 27(4), 786–818 (2005)
5. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic Predicate Abstraction of C Programs. SIGPLAN Not. 36(5), 203–213 (2001)
6. Ball, T., Rajamani, S.K.: Bebop: A Symbolic Model Checker for Boolean Programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)

7. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Advances in Computers 58, 117–148 (2003)
9. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
10. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular Verification of Software Components in C. IEEE Trans. Software Eng. 30(6), 388–402 (2004)
11. Clarke, E.M.: Program Invariants as Fixed Points. Computing 21(4), 273–294 (1979)
12. Clarke, E.M.: Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Axiom Systems. JACM 26(1), 129–147 (1979)
13. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: CAV (2000)
14. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
15. Cooper, D.C.: Theorem Proving in Arithmetic without Multiplication. In: Machine Intelligence, pp. 91–100 (1972)
16. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. Symbolic Logic 22(3), 269–285 (1957)
17. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
18. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
19. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient Algorithms for Model Checking Pushdown Systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
20. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In: POPL, pp. 43–56 (2010)
21. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing Software Verifiers from Proof Rules. In: PLDI, pp. 405–416 (2012)
22. Gupta, A., Ganai, M.K., Yang, Z., Ashar, P.: Iterative Abstraction using SAT-based BMC with Proof Analysis. In: ICCAD, pp. 416–423 (2003)
23. Gurfinkel, A., Wei, O., Chechik, M.: Model checking recursive programs with exact predicate abstraction. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 95–110. Springer, Heidelberg (2008)
24. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate Automizer with SMTInterpol. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 641–643. Springer, Heidelberg (2013)
25. Heizmann, M., Hoenicke, J., Podelski, A.: Nested Interpolants. SIGPLAN Not. 45(1), 471–482 (2010)
26. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. of POPL, pp. 58–70 (2002)

27. Hoder, K., Bjørner, N.: Generalized Property Directed Reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
28. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based Model Checking for Recursive Programs. CoRR, abs/1405.4028 (2014)
29. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 846–862. Springer, Heidelberg (2013)
30. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
31. Loos, R., Weispfenning, V.: Applying Linear Quantifier Elimination. Computing 36(5), 450–462 (1993)
32. McMillan, K.L., Amla, N.: Automatic Abstraction without Counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
33. McMillan, K.L., Rybalchenko, A.: Solving Constrained Horn Clauses using Interpolation. Technical Report MSR-TR-2013-6, Microsoft Research (2013)
34. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)
35. Nipkow, T.: Linear Quantifier Elimination. J. Autom. Reason. 45(2), 189–212 (2010)
36. Reps, T.W., Horwitz, S., Sagiv, S.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: POPL, pp. 49–61 (1995)
37. Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow Analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–233. Prentice-Hall (1981)
38. Software Verification Competition. TACAS (2014), `http://sv-comp.sosy-lab.org`