Form Methods Syst Des (2016) 48:175–205
DOI 10.1007/s10703-016-0249-4

SMT-based model checking for recursive programs

Anvesh Komuravelli[1] · **Arie Gurfinkel**[1] ·
Sagar Chaki[1]

Published online: 25 August 2016

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.

Keywords Model checking · May-must · Satisfiability · Quantifier elimination · Recursion · Compositional

1 Introduction

We are interested in the problem of *safety* of recursive programs, i.e., deciding whether an assertion at a program location 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* [1] of SLAM [2], GOTO programs of CBMC [3], BOOGIEPL of BOOGIE [4], and, indirectly,

✉ Anvesh Komuravelli
anveshk12@gmail.com

1 Carnegie Mellon University, Pittsburgh, PA, USA

```
Main () {
  bool b := nd();
  Level<1> (b);
  Level<1> (b);
  assert (b);
}
```

```
Level<i> (bool b) {
  if (!b) {
      Level<i+1> (b);
      Level<i+1> (b);
  }
  b := !b;
}
```

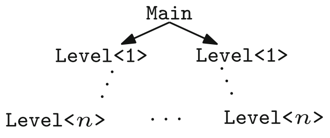


Fig. 1 A Boolean program with exponential unwinding size

internal representations of many tools such as UFO [5], HSF [6], 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 (with the stated assumption [7]), e.g., Boolean programs with (unbounded) recursion and the unbounded use of stack [1,8]. The general observation behind these algorithms is that one can *summarize* the input-output behavior of a procedure, where a *summary* of a procedure is an input-output relation describing what is currently known about its behavior. Thus, if a summary has enough details, it can be used to analyze a procedure call without considering the procedure body of the callee [9,10]. 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 analysis framework by Reps et al. (RHS) [8], recursive state machines [11], and symbolic BDD-based algorithms of Bebop [1] and Moped [12]. 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 [2,13]. In this paper, we are concerned with alternative approaches that work directly on the original program model without an explicit step of Boolean abstraction. Despite the undecidability, we are interested in proving safety for many programs in practice with a guarantee of finding a counterexample when the program is unsafe.

Several algorithms have been recently proposed for verifying recursive programs without Predicate Abstraction. Notable examples are Whale [14], HSF [6], GPDR [15], Ultimate Automizer [16,17] and Duality [18]. With the exception of GPDR, these algorithms are based on a combination of Bounded Model Checking (BMC) [19] and Craig Interpolation [20]. First, they use an SMT-solver to check for a counterexample of bounded size, where the bound is on the depth of the call stack (ie the number of nested procedure calls). Second, they use (tree) interpolation to obtain over-approximating summaries of procedures for the current bound. This is repeated with increasing values of the bound until a counterexample is found or the approximate summaries for the current bound are also invariant (i.e., independent of the bound). The reduction to BMC ensures that the algorithms are guaranteed to find a counterexample if one exists. However, the size of the SMT instance can grow exponentially with the bound on the call-stack depth in the worst-case, due to the tree-like unrolling of the call-graph.

To illustrate the exponential growth in the SMT instances created by existing approaches for BMC, consider the program in Fig. 1 with Boolean variables and finitely many $\texttt{Level} < \texttt{i} >$ procedures (adapted from [1]). Here, `nd` is a routine that returns an unknown Boolean value, i.e., assume that the behavior of `nd` is unknown and hence, for the purpose of verification, `nd` effectively returns either `true` or `false` non-deterministically. For a bound n on the number of such procedures, the figure also shows its tree-like unrolling which grows exponentially in n. With one Boolean parameter per procedure, note that the number of program states is linear in n, where a state corresponds to a valuation of the program counter and the variables in scope. Therefore, many of present day BMC-based model checking algorithms are at least worst-case exponential in the number of states for Boolean programs. However, note that the operational semantics of a Boolean program can be defined in terms of a pushdown automaton where the push and pop operations on the stack correspond to procedure calls and returns, respectively, and the accepting states denote the safe program states. This reduces safety in Boolean programs to state-reachability in pushdown automata and there exist polynomial-time (cubic) algorithms for the latter [1,8,21].

On the other hand, the algorithm GPDR [15] follows the approach of IC3 [22] by solving BMC incrementally without unrolling the call-graph. In GPDR, interpolation is used to obtain over-approximating summaries and partial models denoting undesirable reachable states are cached for future. For some configurations (e.g., explicit-state reasoning), GPDR is worst-case polynomial for Boolean Programs. However, it gets more challenging when the program operations and formulas are in a first-order language. In this case, GPDR might even fail to find a counterexample despite the presence of an SMT oracle, unlike the guarantee given by other BMC-based algorithms mentioned above (see Appendix for an example).

To address the problems mentioned above, we present a new SMT-based algorithm RecMC that analyzes the program *compositionally*. That is, RecMC iteratively checks safety properties of individual procedures by inferring and utilizing approximating summaries of procedures. Our main insight is to maintain not only over-approximating summaries but also *under-approximating* summaries of the procedures. Syntactically, our approximations are formulas over the parameters of a procedure and auxiliary variables denoting the initial values of the parameters. Clarke showed that such formulas are sufficient to obtain a relatively complete Hoare proof system by making use of a *Rule of Adaptation* [9].

We use the terms *may-summary* and *must-summary*, respectively, to refer to such an over- and under-approximation. While may-summaries are used to block spurious counterexamples, must-summaries are used to analyze a procedure call without inlining the body of the callee. Thus, if the under-approximations given by the must-summaries can be reused at call-sites, they help avoid redundant explorations of the state-space. However, given a bound on the call-stack depth, the must-summaries can be too strong and the may-summaries can be too weak to show falsification or satisfaction of safety. In this case, our compositional algorithm creates and checks new safety properties of the callee procedures, updates the approximations, and enters a new iteration.

For Boolean programs, as mentioned previously, the number of states is finite and hence, the summaries can only be updated finitely many times. As the summaries are reused at call-sites in a compositional manner, RecMC has a polynomial time complexity for Boolean Programs, by using an argument similar to that of RHS [8]. Moreover, in general, assuming an SMT oracle for the first-order language of the formulas and the program operations, we show that RecMC terminates for a given bound on the call-stack depth. To the best of our knowledge, this is the first SMT-based algorithm with such guarantees.

Almost every step of RecMC introduces existential quantifiers in the formulas created. RecMC tries to eliminate these quantified variables as, otherwise, they would accumulate exponentially in the value of the current bound.[1] This is because, if no quantified variable is eliminated, the compositional algorithm essentially breaks down into an algorithm that

[1] We assume that all quantified variables are eliminated to obtain the complexity for Boolean Programs mentioned above.

unrolls the call-graph into a tree where, as we mentioned earlier, the size of the SMT problems created may grow exponentially in the value of the bound. A naïve solution is to use algorithms for quantifier elimination (QE), which results in an equivalent quantifier-free formula, but which is also expensive in practice. Instead, we develop an alternative approach that *under-approximates* QE, i.e., obtains a quantifier-free formula stronger than the original formula. However, obtaining arbitrary under-approximations can lead to divergence of the algorithm. To this end, 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 satisfying *models* of $\varphi(\overline{x}, \overline{y})$ (see Sect. 5). We developed efficient MBPs for Linear Rational Arithmetic (LRA) and Presburger Arithmetic (also known as Linear Integer Arithmetic (LIA)) based on the QE methods by Loos-Weispfenning [23] for LRA and Cooper [24] for LIA. We use MBP to under-approximate existential quantification in RecMC. In the best case, a partial under-approximation suffices and a complete quantifier elimination can be avoided.

In summary, we present: (a) an efficient, *compositional* SMT-based algorithm for model checking recursive programs, that uses under- and over-approximating summaries of procedure behavior (Sect. 4), (b) MBP functions for under-approximating quantifier elimination for LRA and LIA (Sect. 5), (c) a new, complete algorithm for Boolean Programs, with complexity polynomial in the number of states, similar to the best known method [1] (see Sect. 4), and (d) an implementation and an empirical evaluation of the approach (Sect. 6).

2 Overview

In this section, we give an overview of RecMC and illustrate it on an example. Let $\mathcal{A}$ be a recursive program. We assume that there are no internal procedures and that procedures cannot be passed as parameters. Furthermore, 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 $|v_0| = |v|$, denoting the initial values of $\overline{v}$. A safety property for P is a formula $\varphi(\overline{v}_0, \overline{v})$. 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 [9] using a *Rule of Adaptation*. We say that φ is a *bounded safety* property for P and a natural number $n \geq 0$, denoted $P(\overline{v}) \models_n \varphi(\overline{v}_0, \overline{v})$, iff all executions of P using a call-stack bounded by n satisfy φ.

The key steps of RecMC are shown in Fig. 2. RecMC decides safety for the main procedure M of $\mathcal{A}$. RecMC maintains two *formula maps* σ_u and σ_o. The *must-summary* map σ_u maps each procedure $P(\overline{v}) \in \mathcal{A}$ to a set of formulas over $\overline{v}_0 \cup \overline{v}$ that *under-approximate* its

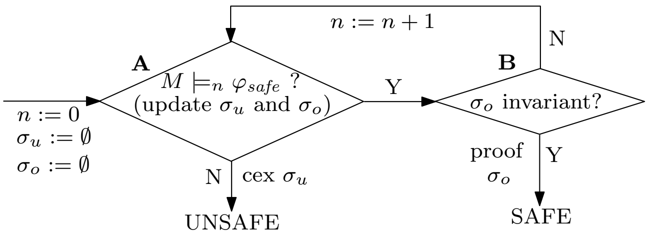


Fig. 2 Flow of the algorithm RecMC to check if $M \models \varphi_{safe}.\sigma_o$ and σ_u denote the may and must-summary maps

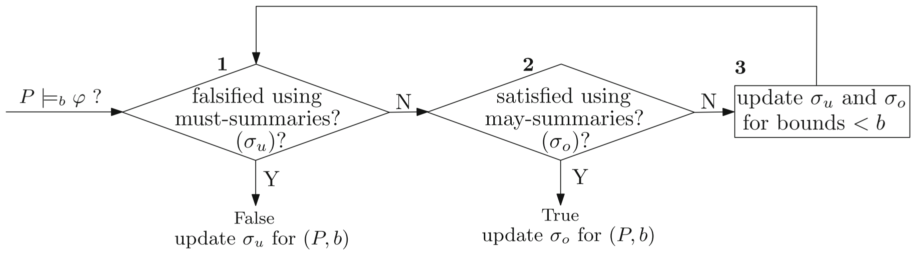

$P \models_b \varphi$?
1
falsified using must-summaries? (σ_u)?
N
2
satisfied using may-summaries? (σ_o)?
N
3
update σ_u and σ_o for bounds $< b$
Y
False update σ_u for (P, b)
Y
True update σ_o for (P, b)

Fig. 3 Flow of the algorithm BNDSAFETY to check $P \models_b \varphi$

```
                          T (t) {
M (m) {                       if (t>0) {
    T (m);                        t := t-2;           D (d) {
    D (m);                        T (t);                  d := d-1;
    D (m);                        t := t+1;           }
}                             }
                          }
```

Fig. 4 A recursive program with 3 procedures

behavior. Similarly, the *may-summary* map σ_o maps a procedure P to a set of formulas that *over-approximate* its behavior. Given P, the maps are partitioned according to the bound on the call-stack. Therefore, if $\delta(\overline{v}_0, \overline{v}) \in \sigma_u(P, n)$ for some $n \geq 0$, then δ under-approximates the behavior of all executions of P that use a call-stack of depth at most n. In other words, for every model m of δ, there is an execution of P that begins in $m(\overline{v}_0)$, the value of $\overline{v}_0$ under m, and ends in $m(\overline{v})$, the value of $\overline{v}$ under m, using a call-stack bounded by n. Similarly, if $\delta(\overline{v}_0, \overline{v}) \in \sigma_o(P, n)$, then δ over-approximates the behavior of all executions of P using a call-stack of depth at most n, i.e., $P(\overline{v}) \models_n \delta(\overline{v}_0, \overline{v})$.

RECMC alternates between two steps: (**A**) deciding bounded safety (that also updates σ_u and σ_o maps) and (**B**) checking whether the current proof of bounded safety also proves unbounded safety. It terminates when a counterexample or a proof is found.

Bounded safety, i.e., whether $P \models_b \varphi$, is decided using the algorithm BNDSAFETY shown in Fig. 3. Step **1** checks whether φ is falsified using the current must-summaries (σ_u) of the callees of P at bound $b - 1$. If so, it infers a new must-summary for P at bound b witnessing the falsification of φ. Step **2** checks whether φ is satisfied using the current may-summaries (σ_o) of the callees at bound $b - 1$. If so, it infers a new may-summary for P at bound b witnessing the satisfaction of φ. If the prior two steps fail, there is a potential counterexample π in P where the must-summaries of the callees are too strong to witness π but the may-summaries are too weak to block it. Step **3** checks the feasibility of such a path π by creating new bounded safety properties for the callees of P at bound $b - 1$, recursively checking the new properties, and updating the formula maps.

We conclude this section with an illustration of RECMC on the program in Fig. 4 (adapted from [9]). The program has 3 procedures: the main procedure `M`, and procedures `T` and `D`. The procedure `M` calls `T` and `D`. The procedure `T` modifies its argument `t` and calls itself recursively. The procedure `D` decrements its argument `d`. Suppose that we want to check if the (main procedure of the) program satisfies the safety property $\varphi \equiv m_0 \geq 2m + 4$. The formula maps σ_u and σ_o are initially empty.

In the first iteration of RECMC, the bound n on the call-stack is 0, i.e., the bounded safety problem is to check whether all executions that do not have any procedure calls are safe. Given that the only path in `M` has procedure calls, no such executions exist and safety

$M(m) \models_1 m_0 \geq 2m+4$?		
iteration 1	check new property: $D(d) \models_0 \bot$?	
	iteration 1	False; update $\sigma_u(D,0)$ with $(d = d_0 - 1)$
iteration 2	check new property: $T(t) \models_0 t_0 \geq 2t$?	
	iteration 1	True; update $\sigma_o(T,0)$ with $(t_0 \geq 2t)$
iteration 3	check new property: $D(d) \models_0 d \leq d_0 - 1$?	
	iteration 1	True; update $\sigma_o(D,0)$ with $(d \leq d_0 - 1)$
iteration 4	True; update $\sigma_o(M,1)$ with $(m_0 \geq 2m+4)$	

Fig. 5 A run of BndSafety for the program in Fig. 4 and the bounded safety property $M(m) \models_1 m_0 \geq 2m+4$

trivially holds for bound 0. Figure 5 shows the four iterations of BndSafety for the next bound $n = 1$, i.e., for checking whether $M(m) \models_1 \varphi$ holds or not. In the first iteration of BndSafety, the current may and must-summaries of the callees are insufficient to satisfy or falsify the property, and there is a potential counterexample along the only path in M. Next, we create a new property for a callee, by performing a backward analysis along the potential counterexample path beginning with the negation of the safety property, and making use of the current summaries of the callees. In practice, one need not be restricted to a backward analysis; see Sects. 4 and 6 for details. As shown in Fig. 5, assume that a new bounded safety property is created for D and $\sigma_u(\mathtt{D}, 0)$, the must-summary map of D at bound 0, is updated with a new must-summary that witnesses the falsification of the property. In the second iteration of BndSafety, the current summaries are still insufficient and assume that a new property is created for T and $\sigma_o(\mathtt{T}, 0)$ is updated with a new may-summary that witnesses the satisfaction of the property. To create the new property for T, we make use of the must-summary of D computed in the previous iteration for *both* the calls to D in M. This is where the compositionality of the algorithm helps avoid the potential re-computation of the must-summary of D. Similarly, in the third iteration of BndSafety, let $\sigma_o(\mathtt{D}, 0)$ be updated with a new may-summary. At this point, the may-summaries for T and D at bound 0 are sufficient to establish bounded safety at $n = 1$ in the fourth iteration of BndSafety, resulting in an update of $\sigma_o(\mathtt{M}, 1)$.

Now, the may-summary map σ_o is:

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

Ignoring the bounds, the may-summaries are invariant. For example, we can prove that the body of T satisfies $t_0 \geq 2t$, assuming that the calls do, i.e., $\{t = t_0\}\ \mathtt{T}(t)\ \{t_0 \geq 2t\} \vdash \{t = t_0\}\ Body(\mathtt{T})\ \{t_0 \geq 2t\}$, where *Body* denotes the body of a procedure. Thus, step **B** of RecMC succeeds and the algorithm terminates declaring the program SAFE.

In summary, RecMC checks safety of a recursive program in a compositional manner by inferring under- and over-approximations of the behavior of procedures. We use an SMT-solver for automating the steps of RecMC and BndSafety.

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 $dom(I)$, and assigns elements of $dom(I)$ to variables, and functions and predicates on $dom(I)$ to the symbols of $\mathcal{S}$. Let φ be a formula in the first-order language. 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 $dom(J) = dom(I)$ 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 $\{\overline{a} \in dom(I)^{|\overline{x}|} \mid I\{\overline{x} \mapsto \overline{a}\} \models \varphi\}$. For simplicity of presentation, we sometimes identify the truth value *true* with $dom(I)$ 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.

Definition 1 (Programs and Procedures) 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 $\langle \overline{\iota}_P, \overline{o}_P, \Sigma_P, \overline{\ell}_P, \beta_P \rangle$, where

1. $\overline{\iota}_P$, $\overline{o}_P$, and $\overline{\ell}_P$ are disjoint finite lists of variables denoting the input values of the parameters, the output values of the parameters, and the local variables, respectively,
2. Σ_P is a fresh predicate symbol of arity $|\overline{\iota}_P| + |\overline{o}_P|$,
3. β_P is a quantifier-free sentence over the signature $(\mathcal{S} \cup \{\Sigma_Q \mid Q \in \mathcal{A}\} \cup \overline{\iota}_P \cup \overline{o}_P \cup \overline{\ell}_P)$ denoting the body of the procedure, where a predicate symbol Σ_Q appears only positively, i.e., under even number of negations.

We use $\overline{v}_P$ to denote $\overline{\iota}_P \cup \overline{o}_P$.

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. 4 is represented as $\langle M, T, D \rangle$ with the main procedure $M = \langle m_0, m, \Sigma_M, \langle \ell_0, \ell_1 \rangle, \beta_M \rangle$, $T = \langle t_0, t, \Sigma_T, \langle \ell_0, \ell_1 \rangle, \beta_T \rangle$, and $D = \langle d_0, d, \Sigma_D, \emptyset, \beta_D \rangle$, where

$$\beta_M = \Sigma_T(m_0, \ell_0) \wedge \Sigma_D(\ell_0, \ell_1) \wedge \Sigma_D(\ell_1, m) \qquad \beta_D = (d = d_0 - 1)$$
$$\beta_T = (t_0 \leq 0 \wedge t_0 = t) \;\vee\; (t_0 > 0 \wedge \ell_0 = t_0 - 2 \wedge \Sigma_T(\ell_0, \ell_1) \wedge t = \ell_1 + 1) \qquad (1)$$

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 P_i has no parameters, i.e., $|\overline{v}_{P_i}| = 0$, or (ii) a subset of tuples from $dom(I)^{|\overline{v}_{P_i}|}$ 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 $Q \in \mathcal{A}$, $J(I, \overline{X}) \models \forall \overline{v}_Q \cup \overline{\ell}_Q \cdot (\beta_Q \implies \Sigma_Q(\overline{v}_Q))$. This has a well-known least fixed-point characterization [9].

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

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

Intuitively, $[\![P_i]\!]_I^0$ consists of all input-output values of the parameters of P_i reachable along paths that do not make any procedure calls, i.e., by interpreting every predicate symbol Σ_Q in the body β_{P_i} as $\emptyset$. Similarly, $[\![P_i]\!]_I^b$, for $b > 0$, consists of all input-output values of the parameters reachable along paths that use a stack of depth bounded by b.

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 main procedure M of a program $\mathcal{A}$ is also called a safety property of the program itself. Given a safety property $\psi(\overline{v}_M)$, a *safety proof* for ψ is an environment Π that is both safe and invariant:

$$\models_{Th} [\![\forall \overline{x} \cdot \Sigma_M(\overline{x}) \implies \psi(\overline{x})]\!]_\Pi \quad \text{(safety)} \tag{2}$$
$$\forall P \in \mathcal{A} \cdot \models_{Th} [\![\forall \overline{v}_P \cup \overline{\ell}_P \cdot (\beta_P \implies \Sigma_P(\overline{v}_P))]\!]_\Pi \quad \text{(invariance)} \tag{3}$$

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

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

$$U_m^b : \Sigma_P \mapsto \bigvee \left\{ \delta \in m(P, b') \mid b' \leq b \right\} \qquad O_m^b : \Sigma_P \mapsto \bigwedge \left\{ \delta \in m(P, b') \mid b' \geq b \right\}$$

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

4 Model checking recursive programs

In this section, we present our algorithm RECMC($\mathcal{A}$, φ_{safe}) for determining 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 show the soundness of RECMC and discuss its complexity guarantees. An efficient instantiation of RECMC to Linear Arithmetic is presented in Sect. 5.

```
RecMC($\mathcal{A}, \varphi_{safe}$)
    $n \leftarrow 0\,;\sigma_u \leftarrow \emptyset\,;\sigma_o \leftarrow \emptyset$
    while $true$ do
        $res, \sigma_u, \sigma_o \leftarrow$ BndSafety($\mathcal{A}, \varphi_{safe}, n, \sigma_u, \sigma_o$)
        if $res$ is $UNSAFE$ then
            return UNSAFE, $\sigma_u$
        else
            $ind, \sigma_o \leftarrow$ CheckInvariance($\mathcal{A}, \sigma_o, n$)
            if $ind$ then
                return SAFE, $\sigma_o$
            $n \leftarrow n + 1$

CheckInvariance($\mathcal{A}, \sigma_o, n$)
    $ind \leftarrow true$
    foreach $P \in \mathcal{A}$ do
        foreach $\delta \in \sigma_o(P, n)$ do
            if $\models [\![\beta_P]\!]^n_o \implies \delta$ then
                $\sigma_o \leftarrow \sigma_o \cup (\langle P, n+1\rangle \mapsto \delta)$
            else
                $ind \leftarrow false$
    return $(ind, \sigma_o)$
```

Fig. 6 Pseudo-code of RecMC

4.1 Top-level loop

RecMC maintains two *bounded formula maps* σ_u and σ_o for must and may-summaries, respectively. For brevity, for a first-order formula τ, we write $[\![\tau]\!]^b_u$ and $[\![\tau]\!]^b_o$ to denote $[\![\tau]\!]_{U^b_{\sigma_u}}$ and $[\![\tau]\!]_{O^b_{\sigma_o}}$, respectively, where the environments U^b_m and O^b_m, for a bounded formula map m, are as defined in Sect. 3. Intuitively, $[\![\tau]\!]^b_u$ and $[\![\tau]\!]^b_o$, respectively, under- and over-approximate τ using σ_u and σ_o.

The pseudo-code of the main loop of RecMC (corresponding to the flow diagram in Fig. 2) is shown in Fig. 6. 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 $M \models_n \varphi_{safe}$. BndSafety also updates the maps σ_u and σ_o. Whenever BndSafety returns *UNSAFE*, the must-summaries in σ_u are sufficient to construct a counterexample to safety and the loop terminates. Whenever BndSafety returns *SAFE*, the may-summaries in σ_o are sufficient to prove the absence of a counterexample for the current bound n on the call-stack. In this case, if σ_o is also invariant [see (3)], as determined by CheckInvariance, $O^n_{\sigma_o}$ 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 CheckInvariance, some may-summaries are propagated to the bound $n+1$. This is similar to the *push generalization* phase in the IC3 algorithm [22].

4.2 Bounded safety

We describe the routine BndSafety($\mathcal{A}, \varphi_{safe}, n, \sigma^{Init}_u, \sigma^{Init}_o$) as an abstract transition system [25] defined by the inference rules shown in Fig. 7. Here, n is the current bound on the

$$\textsc{Init}\ \frac{}{\{\langle M, \neg\varphi_{safe}, n\rangle\} \parallel \sigma_u^{Init} \parallel \sigma_o^{Init}}$$

$$\textsc{May}\ \frac{\mathcal{Q} \parallel \sigma_u \parallel \sigma_o \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi}{\mathcal{Q} \setminus \{\langle P, \eta, c\rangle \mid c \le b, \models [\![\Sigma_P]\!]_o^c \wedge \psi \implies \neg\eta\} \parallel \sigma_u \parallel \sigma_o \cup \{\langle P, b\rangle \mapsto \psi\}}$$

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

$$\textsc{Must}\ \frac{\mathcal{Q} \parallel \sigma_u \parallel \sigma_o \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \pi \in Paths(P) \qquad \not\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi}{\mathcal{Q} \setminus \{\langle P, \eta, c\rangle \mid c \ge b, \not\models \psi \implies \neg\eta\} \parallel \sigma_u \cup \{\langle P, b\rangle \mapsto \psi\} \parallel \sigma_o}$$

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

$$\textsc{Query}\ \frac{\begin{array}{c}\mathcal{Q} \parallel \sigma_u \parallel \sigma_o \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \models [\![\beta_P]\!]_u^{b-1} \implies \neg\varphi \qquad \pi \in Paths(P) \\ \pi = \pi_{pre} \wedge \Sigma_R(\overline{a}) \wedge \pi_{suf} \qquad \models [\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\Sigma_R(\overline{a})]\!]_u^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \implies \neg\varphi \\ \not\models [\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\Sigma_R(\overline{a})]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \implies \neg\varphi\end{array}}{\mathcal{Q} \cup \{\langle R, \psi, b-1\rangle\} \parallel \sigma_u \parallel \sigma_o}$$

$$\text{where } \begin{cases}\psi = \left(\exists \left(\overline{v}_P \cup \overline{\ell}_P\right) \setminus \overline{a} \cdot [\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \wedge \varphi\right)[\overline{a} \leftarrow \overline{v}_R] \\ \text{for all } \langle R, \eta, b-1\rangle \in \mathcal{Q}, \models \psi \implies \neg\eta\end{cases}$$

$$\textsc{Unsafe}\ \frac{\emptyset \parallel \sigma_u \parallel \sigma_o \qquad \not\models [\![\Sigma_M]\!]_u^n \implies \varphi_{safe}}{UNSAFE}$$

$$\textsc{Safe}\ \frac{\emptyset \parallel \sigma_u \parallel \sigma_o \qquad \models [\![\Sigma_M]\!]_o^n \implies \varphi_{safe}}{SAFE}$$

Fig. 7 Rules defining BNDSAFETY$(\mathcal{A}, \varphi_{safe}, n, \sigma_u^{Init}, \sigma_o^{Init})$

call-stack, and σ_u^{Init} and σ_o^{Init} are the maps of must and may-summaries input to the routine. A state of BNDSAFETY is a triple $\mathcal{Q} \| \sigma_u \| \sigma_o$, where σ_u and σ_o 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 $b \ge 0$. 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 must and may-summaries (rule INIT). It checks whether $M \models_n \varphi_{safe}$ by generating new queries as necessary (rule QUERY) and answering existing queries using existing summaries (rules MAY and MUST), the latter resulting in new summaries. When there are no queries left to answer, i.e., $\mathcal{Q}$ is empty, BNDSAFETY terminates with a result of either *UNSAFE* or *SAFE* (rules UNSAFE and SAFE). We explain the rules MAY, MUST and QUERY below.

MAY infers a new may-summary 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 bound b, obtained using the may-summaries of callees at bound $b - 1$, that is unsatisfiable with φ. That is, $\models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi$. The inference of the new summary is by interpolation [20] (denoted by ITP in the side-condition of the rule). Thus, the new may-summary ψ is a formula over $\overline{v}_P$ such that $\models \left([\![\beta_P]\!]_o^{b-1} \implies \psi(\overline{v}_P)\right) \wedge (\psi(\overline{v}_P) \implies \neg\varphi)$. 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 $O_{\sigma_o}^c(\Sigma_P)$ is immediately answered and removed.

MUST infers a new must-summary 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

Fig. 8 Approximations of the only path π of the procedure *M* in Fig. 4

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

must-summaries of callees at bound $b-1$, that is satisfiable with φ. That is, $\not\models [\![\beta_P]\!]_u^{b-1} \implies \neg\varphi$. In particular, there exists a path π in *Paths*(P) such that $\not\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi$. The new must-summary ψ is obtained by choosing such a path π non-deterministically and existentially quantifying all local variables from $[\![\pi]\!]_u^{b-1}$. 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 $U^c_{\sigma_u}(\Sigma_P)$ is immediately answered and removed.
Query creates a new query when an existing query $\langle P, \varphi, b\rangle$ cannot be answered using current summary maps σ_u and σ_o. 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, $\not\models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi$ and $\models [\![\beta_P]\!]_u^{b-1} \implies \neg\varphi$. In particular, there exists a path π in *Paths*(P) such that $\not\models [\![\pi]\!]_o^{b-1} \implies \neg\varphi$ and $\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi$. Intuitively, π is a potential counterexample path that needs to be checked for feasibility. Such a path π is chosen non-deterministically. π is guaranteed to have a conjunct $\Sigma_R(\overline{a})$, corresponding to a call to some procedure R, such that the under-approximation $[\![\Sigma_R(\overline{a})]\!]_u^{b-1}$ is too strong to witness an execution along π that satisfies φ but the over-approximation $[\![\Sigma_R(\overline{a})]\!]_o^{b-1}$ is too weak to block such an execution. That is, π can be partitioned into a prefix π_{pre}, a conjunct $\Sigma_R(\overline{a})$ corresponding to a call to R, and a suffix π_{suf} such that the following hold:

$$\models [\![\Sigma_R(\overline{a})]\!]_u^{b-1} \implies \left(\left([\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1}\right) \implies \neg\varphi\right)$$
$$\not\models [\![\Sigma_R(\overline{a})]\!]_o^{b-1} \implies \left(\left([\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1}\right) \implies \neg\varphi\right)$$

Note that the prefix π_{pre} and the suffix π_{suf} 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 $[\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \wedge \varphi$ except the arguments $\overline{a}$ of the call, and renaming appropriately. If the new query is answered negatively (using May), all executions along π where the values of $\overline{v}_P \cup \overline{\ell}_P$ satisfy $[\![\pi_{suf}]\!]_u^{b-1}$ 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. 4 represented by (1) and the query $\langle M, \varphi, 1\rangle$ where $\varphi \equiv m_0 < 2m + 4$. Let $\sigma_o = \emptyset$, $\sigma_u(D, 0) = \{d = d_0 - 1\}$ and $\sigma_u(T, 0) = \emptyset$. 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. Figure 8 shows $[\![\pi_i]\!]_u^0$ and $[\![\pi_i]\!]_o^0$ for each conjunct π_i of π. As the figure shows, $[\![\pi]\!]_o^0$ 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 $[\![\pi_2 \wedge \pi_3]\!]_u^0$. But, $[\![\pi_1]\!]_u^0$ 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 $[\![\pi_2 \wedge \pi_3]\!]_u^0$ spurious.

4.3 Soundness of BndSafety and RecMC

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

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

Proof We only show the soundness of BndSafety; the soundness of RecMC easily follows. In particular, for BndSafety$(M, \varphi_{safe}, n, \emptyset, \emptyset)$ we show the following:

1. If the premises of Unsafe hold, then $M \not\models_n \varphi_{safe}$, and
2. If the premises of Safe hold, then $M \models_n \varphi_{safe}$.

It suffices to show that the environments $U^b_{\sigma_u}$ and $O^b_{\sigma_o}$, respectively, under- and over-approximate the bounded semantics of the procedures, for every $0 \leq b \leq n$. In particular, we show that the following is an invariant of BndSafety: for every model I of the background theory *Th*, for every procedure $Q \in \mathcal{A}$ and $b \in [0, n]$,

$$I(U^b_{\sigma_u}(\Sigma_Q)) \subseteq [\![Q]\!]^b_I \subseteq I(O^b_{\sigma_o}(\Sigma_Q)). \tag{4}$$

Initially, σ_u and σ_o are empty and the invariant holds trivially. BndSafety updates σ_o and σ_u in the rules May and Must, respectively. We show that these rules preserve (4). We only show the case of May. The case of Must is similar.

Let $\langle P, \varphi, b\rangle \in \mathcal{Q}$ be such that May is applicable, i.e., $\models [\![\beta_P]\!]^{b-1}_o \implies \neg\varphi$. Let $\psi = \text{Itp}([\![\beta_P]\!]^{b-1}_o, \neg\varphi)$. Note that φ, and hence ψ, does not depend on the local variables $\overline{\ell}_P$. Hence, we know that

$$\models \left(\exists \overline{\ell}_P \cdot [\![\beta_P]\!]^{b-1}_o\right) \implies \psi. \tag{5}$$

The case of $b = 0$ is easy and we will skip it. Let I be an arbitrary model of *Th*. Assume that (4) holds at $b - 1$ before applying the rule. In particular, assume that for all $Q \in \mathcal{A}$, $[\![Q]\!]^{b-1}_I \subseteq I(O^{b-1}_{\sigma_o}(\Sigma_Q))$.

We will first show that the new may-summary ψ over-approximates $[\![P]\!]^b_I$. Let $J(I, \overline{X})$ be an expansion of I as defined in Sect. 3.

$$\begin{aligned}
[\![P]\!]^b_I &= J\left(I, \langle [\![P_0]\!]^{b-1}_I, \ldots, [\![P_n]\!]^{b-1}_I\rangle\right)\left(\exists \overline{\ell}_{P_i} \cdot \beta_{P_i}\right) \\
&\subseteq J\left(I, \langle I(O^{b-1}_{\sigma_o}(\Sigma_{P_0})), \ldots, I(O^{b-1}_{\sigma_o}(\Sigma_{P_n}))\rangle\right)\left(\exists \overline{\ell}_{P_i} \cdot \beta_{P_i}\right) && \text{(hypothesis)} \\
&= I\left([\![\exists \overline{\ell}_P \cdot \beta_P]\!]_{O^{b-1}_{\sigma_o}}\right) && (O^{b-1}_{\sigma_o} \text{ is FO-definable}) \\
&= I\left(\exists \overline{\ell}_P \cdot [\![\beta_P]\!]_{O^{b-1}_{\sigma_o}}\right) && \text{(logic)} \\
&= I\left(\exists \overline{\ell}_P \cdot [\![\beta_P]\!]^{b-1}_o\right) && \text{(notation)} \\
&\subseteq I(\psi) && \text{(from (5))}
\end{aligned}$$

Next, we show that the invariant continues to hold. The map of may-summaries is updated to $\sigma'_o = \sigma_o \cup \{\langle P, b\rangle \mapsto \psi\}$. Now, σ'_o differs from σ_o only for the procedure P and for bounds in $[0, b]$. Let $b' \in [0, b]$ be arbitrary. Since (4) was true before applying May, we know that $[\![P]\!]^{b'}_I \subseteq I(O^{b'}_{\sigma_o}(\Sigma_P))$. As $[\![P]\!]^{b'}_I \subseteq [\![P]\!]^b_I \subseteq I(\psi)$, it follows that $[\![P]\!]^{b'}_I \subseteq I(O^{b'}_{\sigma_o}(\Sigma_P)) \cap I(\psi) \subseteq I(O^{b'}_{\sigma_o}(\Sigma_P) \wedge \psi) = I(O^{b'}_{\sigma'_o}(\Sigma_P))$. □

4.4 Termination and complexity of BNDSAFETY

We will now show that BNDSAFETY is complete relative to an oracle for satisfiability modulo *Th*. Intuitively, a must-summary inferred by BNDSAFETY corresponds to a path in a procedure and given a bound on the call-stack, the number of such formulas is finite. This bounds the number of may/must-summaries inferred by BNDSAFETY, guaranteeing termination. Throughout the following, assume an oracle for SAT modulo *Th*.

The following lemma shows that when a query is removed from $\mathcal{Q}$, it is actually answered. The proof is immediate from the definitions of $O^b_{\sigma_o}$ and $U^b_{\sigma_u}$ given in Sect. 3.

Lemma 1 (Answered queries) *Whenever* BNDSAFETY *removes a query from* $\mathcal{Q}$*, it is answered using the known must and may-summaries. In particular, for every query* $\langle P, \eta, b\rangle \in \mathcal{Q}$ *removed from* $\mathcal{Q}$ *by* BNDSAFETY*,*

1. *If the query is removed by* MAY*, then* $\models [\![\Sigma_P]\!]^b_o \implies \neg\eta$*, and*
2. *If the query is removed by* MUST*, then* $\not\models [\![\Sigma_P]\!]^b_u \implies \neg\eta$.

Next, we show that current summaries are insufficient to answer existing queries in $\mathcal{Q}$.

Lemma 2 (Pending queries) $\mathcal{Q}$ *only has the queries which cannot be immediately answered by* σ_u *or* σ_o*,* i.e.*, as long as* $\langle P, \eta, \ell\rangle$ *is in* $\mathcal{Q}$*, the following are invariants of* BNDSAFETY.

1. $\not\models [\![\Sigma_P]\!]^\ell_o \implies \neg\eta$*, and*
2. $\models [\![\Sigma_P]\!]^\ell_u \implies \neg\eta$.

Proof We first show that the invariants hold when a query is newly created by QUERY. Let P, η and ℓ be, respectively, R, $\psi[\overline{a} \leftarrow \overline{v}_R]$ and $b-1$, as in the conclusion of the rule. The last-but-one premise of QUERY is

$$\models [\![\pi_{pre}]\!]^{b-1}_o \wedge [\![\Sigma_R(\overline{a})]\!]^{b-1}_u \wedge [\![\pi_{suf}]\!]^{b-1}_u \implies \neg\varphi$$

which implies that

$$\models [\![\Sigma_R(\overline{a})]\!]^{b-1}_u \implies \neg\left([\![\pi_{pre}]\!]^{b-1}_o \wedge [\![\pi_{suf}]\!]^{b-1}_u \wedge \varphi\right).$$

The variables not in common, viz., $(\overline{v}_P \cup \overline{\ell}_P) \setminus \overline{a}$, can be universally quantified from the right hand side resulting in $\models [\![\Sigma_R]\!]^{b-1}_u \implies \neg\eta$. Similarly, $\not\models [\![\Sigma_R]\!]^{b-1}_o \implies \neg\eta$ follows from the last premise of the rule. Next, we show that MAY and MUST preserve the invariants.

Let MAY answer a query $\langle P, \varphi, \ell\rangle$ with a new may-summary ψ and let the updated map of may-summaries be $\sigma'_o = \sigma_o \cup \{\langle P, \ell\rangle \mapsto \psi\}$. Now, consider $\langle P, \eta, \ell'\rangle \in \mathcal{Q}$ after the application of the rule. If $\ell' > \ell$, $O^{\ell'}_{\sigma'_o} = O^{\ell'}_{\sigma_o}$ and the invariant continues to hold. So, assume $\ell' \leq \ell$. From the conclusion of MAY, we have $\not\models [\![\Sigma_P]\!]^{\ell'}_o \wedge \psi \implies \neg\eta$. Now, $O^{\ell'}_{\sigma'_o}(\Sigma_P) = O^{\ell'}_{\sigma_o}(\Sigma_P) \wedge \psi$. So, the invariant continues to hold.

Similarly, let MUST answer a query $\langle P, \varphi, \ell\rangle$ with a new must-summary ψ and let the updated map of must-summaries be $\sigma'_u = \sigma_u \cup \{\psi \mapsto \langle P, \ell\rangle\}$. Now, consider $\langle P, \eta, \ell'\rangle \in \mathcal{Q}$ after the application of the rule. If $\ell' < \ell$, $U^{\ell'}_{\sigma'_u} = U^{\ell'}_{\sigma_u}$ and the invariant continues to hold. So, assume $\ell' \geq \ell$. From the conclusion of MUST, we have $\models \psi \implies \neg\eta$. Assuming the invariant holds before the rule application, we also have $\models [\![\Sigma_P]\!]^{\ell'}_u \implies \neg\eta$. Therefore, we have $\models [\![\Sigma_P]\!]^{\ell'}_u \vee \psi \implies \neg\eta$. Now, $U^{\ell'}_{\sigma'_u}(\Sigma_P) = U^{\ell'}_{\sigma_u}(\Sigma_P) \vee \psi$. So, the invariant continues to hold. □

The next few lemmas show that the rules of the algorithm cannot be applied indefinitely, leading to a termination argument. Let N be the number of procedures in the program $\mathcal{A}$, p be the maximum number of paths in a procedure, and c be the maximum number of procedure calls along any path in $\mathcal{A}$.

Lemma 3 (Finitely-many must summaries) *Given a predicate symbol* Σ_P *and a bound* b*, the environment* $U^b_{\sigma_u}$ *is updated only* $O(N^b \cdot p^{b+1})$*-many times.*

Proof The environment $U^b_{\sigma_u}$ can be updated for Σ_P and b whenever a must-summary is inferred for P at a bound $b' \leq b$. Now, a must-summary is obtained per path (after eliminating the local variables) of a procedure, using the currently known must-summaries about the callees. Moreover, Lemmas 1 and 2 imply that no must-summary is inferred twice. This is because whenever a query is answered using MUST, the query could not have been answered using already existing must-summaries and a new must-summary is inferred.

This gives the following recurrence $Must(b)$ for the number of updates to $U^b_{\sigma_u}$ for a given Σ_P:

$$Must(b) = \begin{cases} p, & b = 0 \\ (p \cdot N + 1) \cdot Must(b-1), & b > 0. \end{cases}$$

In words, for $b = 0$, the number of updates is given by the number of must-summaries that can be inferred, which is bounded by the number of paths p in the procedure P. For $b > 0$, the environment $U^b_{\sigma_u}$ is updated when a must-summary is learnt for the procedure at a bound smaller than or equal to b. For the former, the number of updates is simply $Must(b-1)$. For the latter, a new must-summary is inferred at bound b along a path whenever $U^{b-1}_{\sigma_u}$ changes for a callee. For N procedures and p paths, this is given by $(p \cdot N \cdot Must(b-1))$.

This gives us $Must(b) = O(N^b \cdot p^{b+1})$. □

Lemma 4 (Finitely-many queries) *For* $\langle P, \varphi, b\rangle \in \mathcal{Q}$, QUERY *is applicable only* $O(c \cdot N^b \cdot p^{b+1})$*-many times.*

Proof First, assume that the environments $U^{b-1}_{\sigma_u}$ and $O^{b-1}_{\sigma_o}$ are fixed. The number of possible queries that can be created for a given path of P is bounded by the number of ways the path can be divided into a prefix, a procedure call, and a suffix. This is bounded by c, the maximum number of calls along the path. For p paths, this is bounded by $c \cdot p$.

Consider a path π and its division, and let a query be created for a callee R along π. Now, while the query is still in $\mathcal{Q}$, updates to the environments $O^{b-1}_{\sigma_o}$ and $U^{b-1}_{\sigma_u}$ do not result in a new query for R for the same division along π. This is because, the new query would overlap with the existing one and this is disallowed by the second side-condition of QUERY.

Suppose that the new query is answered by MAY. With the updated map of may-summary, the last premise of QUERY can be shown to fail for the current division of π. If $O^{b-1}_{\sigma_o}$ is updated, the last premise continues to fail. So, a new query can be created for the same prefix and suffix along π only if $U^{b-1}_{\sigma_u}$ is updated for some callee along π. The other possibility is that the query is answered by MUST which updates $U^{b-1}_{\sigma_u}$ as well.

Thus, for a given path, and a given division of it into prefix and suffix, the number of queries that can be created is bounded by the number of updates to $U^{b-1}_{\sigma_u}$ which is $(N \cdot Must(b-1))$. Here, $Must$ is as in Lemma 3. So, the number of times QUERY is applicable for a given query $\langle P, \varphi, b\rangle$ is $O(p \cdot c \cdot N \cdot Must(b-1))$. As $Must(b) = N^b \cdot p^{b+1}$, we obtain the bound $O(c \cdot N^b \cdot p^{b+1})$. □

Lemma 5 (Progress) *As long as $\mathcal{Q}$ is non-empty, either* May, Must *or* Query *is always applicable.*

Proof First, we show that for every query in $\mathcal{Q}$, either of the three rules is applicable, without the second side-condition in Query. Let $\langle P, \varphi, b\rangle \in \mathcal{Q}$. If $\models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi$, then May is applicable. Otherwise, there exists a path $\pi \in \mathit{Paths}(P)$ such that $[\![\pi]\!]_o^{b-1}$ is satisfiable with φ, i.e., $\not\models [\![\pi]\!]_o^{b-1} \implies \neg\varphi$. Now, if $[\![\pi]\!]_u^{b-1}$ is also satisfiable with φ, i.e., $\not\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi$, Must is applicable. Otherwise, $\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi$. Note that this can only happen if $b > 0$, as otherwise, there will not be any procedure calls along π and $[\![\pi]\!]_o^{b-1}$ and $[\![\pi]\!]_u^{b-1}$ would be equivalent.

Let $\pi = \pi_0 \wedge \pi_1 \wedge \ldots \pi_l$ for some finite l. Then, $[\![\pi]\!]_o^{b-1}$ is obtained by taking the conjunction of the formulas

$$\langle [\![\pi_0]\!]_o^{b-1}, [\![\pi_1]\!]_o^{b-1}, \ldots\rangle.$$

Similarly, $[\![\pi]\!]_u^{b-1}$ is obtained by taking the conjunction of the formulas

$$\langle [\![\pi_0]\!]_u^{b-1}, [\![\pi_1]\!]_u^{b-1}, \ldots\rangle.$$

From Theorem 1, we can think of obtaining the latter sequence of formulas by conjoining $[\![\pi_i]\!]_u^{b-1}$ to $[\![\pi_i]\!]_o^{b-1}$ for every i. When this is done backwards for decreasing values of i, an intermediate sequence looks like

$$\langle [\![\pi_0]\!]_o^{b-1}, \ldots, [\![\pi_{j-1}]\!]_o^{b-1}, [\![\pi_j]\!]_u^{b-1} \ldots\rangle.$$

As $[\![\pi]\!]_u^{b-1}$ is unsatisfiable with φ, there exists a maximal j such that the conjunction of constraints in such an intermediate sequence are unsatisfiable with φ. Moreover, π_j must be a literal of the form $\Sigma_R(\overline{a})$ as otherwise, $[\![\pi_j]\!]_o^{b-1} = [\![\pi_j]\!]_u^{b-1}$ violating the maximality condition on j. Thus, all premises of Query hold and the rule is applicable.

Now, the second side-condition in Query can be trivially satisfied by always choosing a query in $\mathcal{Q}$ with the smallest bound for the next rule to apply. This is because, if $\langle R, \eta, b-1\rangle$ is the newly created query, there is no other query in $\mathcal{Q}$ for R and $b-1$. □

Lemmas 4 and 5 imply that every query in $\mathcal{Q}$ is eventually answered by May or Must, as shown below.

Lemma 6 (Eventual answer) *Every $\langle P, \varphi, b\rangle \in \mathcal{Q}$ is eventually answered by* May *or* Must, *in $O(b \cdot c^b \cdot (Np)^{O(b^2)})$ applications of the rules.*

Proof Firstly, to answer any given query in $\mathcal{Q}$, Lemma 4 guarantees that the algorithm can only create finitely many queries. Lemma 5 guarantees that some rule is always applicable, as long as $\mathcal{Q}$ is non-empty. Thus, when Query cannot be applied for any query in $\mathcal{Q}$, either May or Must must be applicable for some query. Thus, eventually, all queries are answered.

The total number of rule applications to answer $\langle P, \varphi, b\rangle$ is then linear in the cumulative number of applications of Query, which has the following recurrence:

$$T(b) = \begin{cases} Q(0), & b = 0 \\ Q(b)(1 + T(b-1)), & b > 0. \end{cases}$$

where $Q(b)$ denotes the number of applications of Query for a fixed query in $\mathcal{Q}$ at bound b. From Lemma 4, $Q(b) = O(c \cdot N^b \cdot p^{b+1})$. This gives us $T(b) = O(b \cdot c^b \cdot (Np)^{O(b^2)})$. □

The main termination theorem is an immediate consequence of the above lemma:

Theorem 2 *Given a satisfiability oracle for Th,* BndSafety$(\mathcal{A}, \varphi, n, \emptyset, \emptyset)$ *decides bounded safety in finitely many iterations and terminates.*

As an immediate corollary, RecMC is guaranteed to find a counterexample if one exists.

Corollary 1 RecMC$(\mathcal{A}, \varphi)$ *is guaranteed to return UNSAFE with a counterexample if* $\mathcal{A} \not\models \varphi$.

In contrast, the closest related algorithm GPDR [15], mentioned briefly in Sect. 1, does not have such guarantees. 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 summaries, is finite. Let N denote the number of procedures of a program $\mathcal{A}$ and $k = \max\{|\overline{v}_P| \mid P(\overline{v}_P) \in \mathcal{A}\}$.

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.* 7.

Proof First, assume a bound n on the call-stack. The number of queries that can be created for a procedure at any given bound is $O(2^k)$, the number of possible valuations of the parameters (note that Query disallows overlapping queries to be present simultaneously in $\mathcal{Q}$). For N procedures and n possible values of the bound, the complexity of BndSafety$(\mathcal{A}, \varphi, n, \emptyset, \emptyset)$, for a Boolean program, is $O(N \cdot 2^k \cdot n)$.

Now, the total number of may-summaries that can be inferred for a procedure is also bounded by $O(2^k)$. As $O^b_{\sigma_o}$ is monotonic in b, the number of iterations of RecMC is bounded by $O(N \cdot 2^k)$, the cumulative number of states of all procedures. Thus, we obtain the complexity of RecMC as $O(N^2 \cdot 2^{2k})$. □

Note that the number of states of a Boolean program is $O(N \cdot 2^k)$, so the above complexity is polynomial in the number of states. In contrast, other SMT-based algorithms, such as Whale [14], are worst-case exponential in the number of states of a Boolean program. Also, note that the complexity is quadratic in the number of procedures as opposed to the known upper-bound which has a linear dependency [1]. This is a manifestation of the iterative deepening strategy of RecMC and in particular, the may-summaries computed by the algorithm, which is necessary for handling programs over first-order theories. In contrast, the known optimal algorithms for Boolean programs do not compute may-summaries.

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

The algorithm RecMC described in the previous section works for an arbitrary first-order signature $\mathcal{S}$ and a $\mathcal{S}$-theory *Th* as long as there is an oracle for satisfiability modulo *Th*. In this section, we consider the case of *Th* being either Linear Rational Arithmetic (LRA) or Linear Integer Arithmetic (LIA). Note that the program can also have Boolean parameters or local variables. RecMC can be used as-is, but recall that BndSafety introduces quantifiers in the formulas maintained by the algorithm. This is because the may and must-summaries are formulas over the parameters of a procedure and auxiliary variables denoting their initial values, and when creating a new summary, all other variables will be quantified away. Same is the case with creating new bounded safety properties. Unless eliminated, these quantifiers accumulate exponentially in the value of the bound corresponding to the bounded safety problem. This is because, if no quantified variable is eliminated, the compositional algorithm essentially breaks down into an algorithm that unrolls the call-graph into a tree where, as we mentioned earlier, the size of the SMT problems created may grow exponentially in the bound on the call-stack. On the other hand, it is expensive to use quantifier elimination (QE) to obtain an equivalent quantifier-free formula. Instead, we propose an alternative approach that *approximates* QE with quantifier-free formulas *lazily* and efficiently.

In particular, we (a) introduce a model-based under-approximation of QE for existentially quantified formulas, called *Model Based Projection* (MBP), (b) give efficient (linear in the size of formulas involved) MBP procedures for Propositional Logic, Linear Rational Arithmetic (LRA), and Linear Integer Arithmetic (LIA; also well known as Presburger Arithmetic), and (c) present a modified version of BndSafety that uses MBP to under-approximate the existential quantification of variables out of scope, and show that it remains sound and terminating. Our MBP procedures for LRA and LIA are based on the QE algorithms by Loos and Weispfenning [23] and Cooper [24], respectively.

Definition 2 (Model Based Projection) Let $\eta(\overline{y}) = \exists \overline{x} \cdot \eta_m(\overline{x}, \overline{y})$ be an existentially quantified formula where η_m is quantifier free. A function $Proj_\eta$ from models of η_m to quantifier-free formulas over $\overline{y}$ is a *Model Based Projection* (for η) iff

1. $Proj_\eta$ has a finite image,
2. $\eta \equiv \bigvee_{M \models \eta_m} Proj_\eta(M)$, and
3. for every model M of η_m, $M \models Proj_\eta(M)$.

In other words, $Proj_\eta$ covers the space of all models of $\eta_m(\overline{x}, \overline{y})$ by a finite set of quantifier-free formulas over $\overline{y}$. Note that there is a trivial MBP that maps every model of η_m to a quantifier-free formula equivalent to η. However, when QE is expensive, it is not the most efficient MBP and our objective is to obtain an MBP that maps models to quantifier-free *under-approximations* of η. In the following, we describe MBP procedures whose computation is linear in time and space given a model.

5.1 MBP for propositional logic

Let $\eta(\overline{y}) = \exists \overline{x} \cdot \eta_m(\overline{x}, \overline{y})$ be an existentially quantified formula where the quantified variables in $\overline{x}$ are all Boolean (propositional). Without loss of generality, assume that $\overline{x}$ is singleton. Our MBP procedure is based on the following equivalence:

$$\exists x \cdot \eta_m(x, \overline{y}) \equiv \eta_m[x \mapsto \bot] \vee \eta_m[x \mapsto \top] \tag{6}$$

where $\eta_m[x \mapsto t]$ denotes the result of substituting the term t for x in η_m.

We now define an MBP $BoolProj_\eta$ for Propositional Logic as a map from models of η_m to one of the disjuncts above depending on the assignment to x in the given model:

$$BoolProj_\eta(M) = \begin{cases} \eta_m[x \mapsto \bot], & M \models x = \bot \\ \eta_m[x \mapsto \top], & M \models x = \top \end{cases}$$

This procedure is also used in the GPDR model checking algorithm [15] implemented in the tool Z3 [26] and a similar approach is used in SAT-based iterative quantifier elimination in hardware verification [27]. The following is now immediate.

Theorem 4 *$BoolProj_\eta$ is a Model Based Projection.*

5.2 MBP for linear rational arithmetic (LRA)

We begin with a brief overview of Loos-Weispfenning (LW) method [23] for quantifier elimination in LRA. We borrow our presentation from Nipkow [28] to which we refer the reader for more details. Let $\eta(\overline{y}) = \exists \overline{x} \cdot \eta_m(\overline{x}, \overline{y})$ as above. Assume that *Th* is LRA. Without loss of generality, assume that $\overline{x}$ is singleton containing a rational variable x, η_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 $lits(\eta)$ denote the literals of η. The LW-method states that $\exists x \cdot \eta_m(x, \overline{y}) \equiv$

$$\bigvee_{(x=e)\in lits(\eta)} \eta_m[x \mapsto_v e] \vee \bigvee_{(\ell<x)\in lits(\eta)} \eta_m[x \mapsto_v (\ell + \epsilon)] \vee \eta_m[x \mapsto_v -\infty] \quad (7)$$

where $\eta_m[x \mapsto_v t]$ denotes the result of a *virtual substitution* of the term t for x in η_m. Note that the symbols ϵ and $-\infty$ do not appear in the results of the substitutions (which is why the substitution is called *virtual*). Intuitively, $\eta_m[x \mapsto_v e]$ 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.

To perform the virtual substitution, the LW-method defines a substitution map Sub_t of literals containing x corresponding to $\eta_m[x \mapsto_v t]$, defined as follows:

$$Sub_e(x = e) = \top, \quad Sub_e(\ell < x) = (\ell < e), \quad Sub_e(x < u) = (e < u) \quad (8)$$

$$Sub_{\ell+\epsilon}(x = e) = \bot, \quad Sub_{\ell+\epsilon}(\ell' < x) = (\ell' \le \ell), \quad Sub_{\ell+\epsilon}(x < u) = (\ell < u) \quad (9)$$

$$Sub_{-\infty}(x = e) = \bot, \quad Sub_{-\infty}(\ell < x) = \bot, \quad Sub_{-\infty}(x < u) = \top \quad (10)$$

For example, let η_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, $\eta_m[x \mapsto_v e]$

$$= (Sub_e(x = e) \wedge \phi_1) \vee (Sub_e(\ell < x) \wedge Sub_e(x < u)) \vee (Sub_e(x < u) \wedge \phi_2)$$
$$\equiv \big(\phi_1 \vee (\ell < e \wedge e < u) \vee (e < u \wedge \phi_2)\big),$$

$\eta_m[x \mapsto_v (\ell + \epsilon)]$

$$= (Sub_{\ell+\epsilon}(x = e) \wedge \phi_1) \vee (Sub_{\ell+\epsilon}(\ell < x) \wedge Sub_{\ell+\epsilon}(x < u)) \vee (Sub_{\ell+\epsilon}(x < u) \wedge \phi_2)$$
$$\equiv \big(\ell < u \vee (\ell < u \wedge \phi_2)\big),$$

and $\eta_m[x \mapsto_v -\infty]$

$$= (Sub_{-\infty}(x = e) \wedge \phi_1) \vee (Sub_{-\infty}(\ell < x) \wedge Sub_{-\infty}(x < u)) \vee (Sub_{-\infty}(x < u) \wedge \phi_2)$$
$$\equiv \phi_2.$$

Together, we obtain $\exists x \cdot \eta_m \equiv \phi_1 \vee (\ell < u) \vee \phi_2$.

We now define an MBP $LRAProj_\eta$ for LRA as a map from models of η_m to disjuncts in (7). Given $M \models \eta_m$, $LRAProj_\eta$ 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 $\ell' < x$).

$$LRAProj_\eta(M) = \begin{cases} \eta_m[x \mapsto_v e], & \text{if } (x = e) \in lits(\eta) \wedge M \models x = e \\ \eta_m[x \mapsto_v (\ell + \epsilon)], & \text{else if } (\ell < x) \in lits(\eta) \wedge M \models \ell < x \wedge \\ & \forall (\ell' < x) \in lits(\eta) \cdot \\ & M \models \big((\ell' < x) \implies (\ell' \le \ell)\big) \\ \eta_m[x \mapsto_v -\infty], & \text{otherwise} \end{cases}$$

Theorem 5 *$LRAProj_\eta$ is a Model Based Projection.*

Proof By definition, $LRAProj_\eta$ has a finite image, as there are only finitely many disjuncts in (7). Thus, it suffices to show that for every $M \models \eta_m$, $M \models LRAProj_\eta(M)$.

Let $M \models \eta_m$ and $LRAProj_\eta(M) = \eta_m[x \mapsto_v t]$ where t is either e or $\ell + \epsilon$ or $-\infty$. As η_m is in NNF, it suffices to show that for every literal μ of η_m containing x, the following holds:

$$M \models (\mu \implies Sub_t(\mu)) \tag{11}$$

We consider the different possibilities of t below. For a term v, let $M[v]$ denote the value of v in M.

In this case, we know that $M \models x = e$. Now, for a literal $\ell < x$,

$$\begin{aligned} M[\ell < x] &\implies M[\ell] < M[x] \\ &= M[\ell] < M[e] \\ &= M[\ell < e] \\ &= M[Sub_t(\ell < x)] \qquad \{Sub_t(\ell < x) = (\ell < e)\}. \end{aligned}$$

Similarly, literals of the form $x < u$ and $x = e'$ can be considered.

Case $t = e$.Case $t = \ell+\epsilon$. In this case, we know that $M[\ell < x]$ is true, i.e., $M[\ell] < M[x]$ and whenever $M[\ell' < x]$ is true, $M[\ell' \leq \ell]$ is also true. Now, for a literal $\ell' < x$,

$$\begin{aligned} M[\ell' < x] &\implies M[\ell' \leq \ell] \\ &= M[Sub_t(\ell' < x)] \quad \{Sub_t(\ell' < x) = (\ell' \leq \ell)\}. \end{aligned}$$

For a literal $x < u$,

$$\begin{aligned} M[x < u] &\implies M[x] < M[u] \\ &\implies M[\ell] < M[u] \qquad \{M[\ell] < M[x]\} \\ &\implies M[\ell < u] \\ &= M[Sub_t(x < u)] \qquad \{Sub_t(x < u) = (\ell < u)\} \end{aligned}$$

For a literal $x = e$, (11) vacuously holds as $M[x = e]$ is false.

Case $t = e$.Case $t = -\infty$. In this case, we know that $M[x = e]$ and $M[\ell < x]$ are false for every literal of the form $x = e$ and $\ell < x$. So, for such literals (11) vacuously holds. For a literal $x < u$, $Sub_t(x < u) = \top$ and hence, (11) holds again. □

5.3 MBP for linear integer arithmetic (LIA)

We will now present our MBP $LIAProj_\eta$ for LIA. It is based on Cooper's method for Quantifier Elimination procedure for LIA [24]. Let $\eta(\overline{y}) = \exists x \cdot \eta_m(x, \overline{y})$, where η_m is quantifier free and in negation normal form. Assume that Th is LIA and x is an integer variable. Without loss of generality, let the only literals containing x be the form $\ell < x$, $x < u$, $x = e$ or $(d \mid \pm x + w)$, where $a \mid b$ denotes that a divides b, the terms ℓ, u, e and w are x-free, and $d \in \mathbb{Z} \setminus \{0\}$. Let $E = \{e \mid (x = e) \in lits(\eta_m)\}$ be the set of equality terms of x and $L = \{\ell \mid (\ell < x) \in lits(\eta_m)\}$ be the set of lower-bounds of x. Then, by Cooper's method, $\exists x \cdot \eta_m(x, \overline{y}) \equiv$

$$\bigvee_{(x=e)\in lits(\eta)} \eta_m[x \mapsto_v e] \vee \bigvee_{(\ell<x)\in lits(\eta)} \left(\bigvee_{i=0}^{D-1} \eta_m[x \mapsto_v (\ell + 1 + i)] \right) \vee \bigvee_{i=0}^{D-1} \eta_m^{-\infty}[x \mapsto_v i] \tag{12}$$

where D is the least common multiple of all the divisors in the divisibility literals of η_m, $[x \mapsto_v t]$ denotes a *virtual substitution* of the term t for x and $\eta_m^{-\infty}$ is obtained from η_m by substituting all non-divisibility literals as follows:

$$(\ell < x) \mapsto \bot \qquad (x < u) \mapsto \top \qquad (x = e) \mapsto \bot \tag{13}$$

Intuitively, the disjunction partitions the space of the possible values of x. A disjunct for $(x = e)$ covers the case when x is equal to an equality term. Otherwise, the lower-bounds identify various intervals in which x can be present. The disjuncts for $(\ell < x)$ cover the case when x satisfies a lower-bound, and the last disjunct is for the case when x is smaller than all lower-bounds. The disjunction over the possible values of i covers the different ways in which the divisibility literals can be satisfied.

Model based projection $\mathit{LIAProj}_\eta$ is defined as follows, conflicts are resolved by some arbitrary, but fixed, syntactic ordering on terms:

$$\mathit{LIAProj}_\eta(M) = \begin{cases} \eta_m[x \mapsto_v e], & \text{if } x = e \in \mathit{lits}(\eta) \wedge M \models (x = e) \\ \eta_m[x \mapsto_v (\ell + 1 + i_\ell)], & \text{else if } (\ell < x) \in \mathit{lits}(\eta) \wedge M \models (\ell < x) \wedge \\ & \forall (\ell' < x) \in \mathit{lits}(\eta) \cdot \\ & \quad \big(M \models ((\ell' < x) \implies (\ell' \leq \ell))\big) \\ \eta_m^{-\infty}[x \mapsto_v i_{-\infty}], & \text{otherwise} \end{cases}$$

where $i_\ell = M[x - (\ell + 1)] \bmod D$, $i_{-\infty} = M[x] \bmod D$, and $M[x]$ is the value of x in M.

The following theorem shows that $\mathit{LIAProj}_\eta$ is indeed a model based projection. The proof is similar to that of Theorem 5.

Theorem 6 *$\mathit{LIAProj}_\eta$ is a Model Based Projection.*

5.4 Bounded safety with MBP

Given an MBP Proj_η for an existentially quantified formula η, we have seen above that each quantifier-free formula in the image of Proj_η under-approximates η. As above, we use η_m for the quantifier-free matrix of η. We can now modify the side-condition $\psi = \eta$ of MUST and QUERY in the algorithm BNDSAFETY to use quantifier-free under-approximations as follows: (i) for MUST, the new side-condition is $\psi = \mathit{Proj}_\eta(M)$ where $M \models \eta_m \wedge \varphi$, and (ii) for QUERY, the new side-condition is $\psi = \mathit{Proj}_\eta(M)$ where $M \models \eta_m \wedge [\![\Sigma_R(a)]\!]_o^{b-1}$. Note that to avoid redundant applications of the rules, we require M to satisfy a formula stronger than η_m. Intuitively, (i) ensures that the newly inferred reachability 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_η.

Theorem 7 *Assuming an oracle and an MBP for Th,* BNDSAFETY *is sound and terminating after modifying the rules as described above.*

Proof sketch Here, we show that BNDSAFETY with MBP is sound and terminating.

First of all, in presence of MBP, MAY is unaffected and a reachability fact inferred by MUST is only strengthened. Thus, soundness of BNDSAFETY (Theorem 1) is preserved.

Then, it is easy to show that the modified side-conditions to MUST and QUERY preserve Lemmas 1 and 2 and we skip the proof.

Then, we will show that the *finite-image* property of an MBP preserves the finiteness of the number of reachability facts inferred and the number of queries generated by the algorithm. Let d be the size of the image of an MBP. In the proof of Lemma 3, the recurrence relation will now have an extra factor of d. The rest of the proof of finiteness of the number of reachability facts remains the same. Similarly, in the proof of Lemma 4, the number of times QUERY can be applied along a path for a fixed division and fixed environments O_σ^{b-1} and U_ρ^{b-1} will increase by a factor of d. Again, the rest of the proof of finiteness of the number of queries generated remains the same. That is, Lemmas 3 and 4, and hence, Lemma 6, are preserved with scaled up complexity bounds.

Note that Theorem 5 is unaffected by under-approximations.

Together, we have that Theorem 2 is preserved, with a scaled up complexity bound. □

Thus, BNDSAFETY with a linear-time MBP (such as $LRAProj_\eta$) 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 our tool SPACER. The back-end is based on Z3 [26] 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 the tool UFO [29]. It encodes safety of a C program by converting it to the Horn-SMT format of Z3, which corresponds to the logical program representation described in Sect. 3. Loops are handled by creating fresh predicate symbols denoting the loop invariants and encoding the corresponding verification conditions. The implementation and benchmarks are available online.[2] We evaluated SPACER on three sets of benchmarks:

(a) 2908 Boolean programs obtained from the SLAM toolkit,[3]
(b) 1535 procedural programs from Microsoft's SDV project,[4] and
(c) 797 C programs from the Software Verification Competition (SVCOMP) 2014 [30].

The numbers of programs mentioned for the second and third sets of benchmarks above exclude programs with non-linear operations and memory-related properties as SPACER cannot handle them yet. The 797 programs in the third set of benchmarks also exclude programs that can be easily verified by our frontend (which converts a C program to the Horn-SMT format) using common compiler optimizations. Note that the programs in the last set have non-recursive procedures (which may still have loops) and our current frontend inlines all procedure calls. We call the resulting set of encodings SVCOMP-1. Note that SVCOMP-1 essentially corresponds to while-programs. We introduced procedural modularity in SVCOMP-1 by two distinct means: (a) factoring out maximal loop-free fragments into new loop-free, recursion-free procedures (the main procedure may still have loops) to obtain SVCOMP-2, and (b) factoring out loops into tail-recursive procedures (in an inside-out fashion for nested loops) to obtain SVCOMP-3. Our simple outlining procedure could not handle some large programs and SVCOMP-3 has 45 fewer programs.

Figure 9 shows some characteristics of the Horn-SMT encodings for the benchmarks, when viewed according to the logical program representation described in Sect. 3. The number of

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

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

[4] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/ALIA/sdv.

	#calls along a path		#calls of a procedure	
	max (over all paths)	*avg* (over all paths)	*max* (over all procedures)	*avg* (over all procedures)
Slam	2	1	95.1	1.4
Sdv	11.9	1.18	21.6	1.7
Svcomp-1	1	0.7	4.1	1.4
Svcomp-2	2	0.8	3.9	1.1
Svcomp-3	1.5	0.8	8.4	3.2

Fig. 9 Some characteristics of the Horn-SMT encodings of the benchmarks, averaged over all programs in the corresponding set

	Slam		Sdv	
	SAFE	UNSAFE	SAFE	UNSAFE
Spacer	1,721	985	1,303	232
Z3	1,727	992	1,302	232
Spacer or Z3	1,727	992	1,303	232

Fig. 10 Number of programs verified for Slam and Sdv benchmarks

	Svcomp-1		Svcomp-2		Svcomp-3	
	SAFE	UNSAFE	SAFE	UNSAFE	SAFE	UNSAFE
Spacer	249	509	213	497	234	482
Z3	245	509	208	493	234	477
Spacer or Z3	252	509	225	500	240	482

Fig. 11 Number of programs verified for SVCOMP benchmarks

calls along a path roughly identifies the procedural modularity of the encodings. The number of calls of a procedure (in the entire program) identifies the potential number of times a summary (may or must) of the procedure can be reused for a given bound on the call-stack. Note that summaries can also be reused across different bounds on the call-stack. Despite the fact that the average number of procedure calls is low from the figure, we can show the practical advantage of RecMC using these benchmarks, as we show below.

We compared Spacer against the implementation of GPDR in Z3 [15]. GPDR is inspired by the IC3 hardware model checking algorithm [22] and avoids unrolling the call-graph. Thus, it creates and checks reachability queries for individual procedures similar to RecMC. However, it only computes may summaries and because of the lack of must summaries, its query creation mechanism is quite different from the rule Query. Moreover, it does not use MBP.

In our experiments, the resource limits were set to 30 min of time and 16GB of memory, on an Ubuntu machine with a 2.2 GHz AMD Opteron(TM) Processor 6174 and 516GB RAM. Figure 10 and 11 show a high level summary of the results in terms of the number of programs verified by Spacer and Z3. Since there are some programs verified by only one of the tools, the figures also report the number of programs verified by at least one tool in the third row. We provide a more detailed discussion of the experimental results in the following. In the scatter plots shown below, a diamond indicates a time-out and a star indicates a mem-out.

6.1 Boolean program benchmarks

Figure 12 shows the scatter plot of runtimes for Spacer and Z3 for the SLAM benchmarks. The runtimes of both the tools are within ±5 min for over 98% of the benchmarks. Of the remaining, Spacer is better on 1 benchmark, Z3 is better on 42 benchmarks which includes 13 benchmarks where Spacer runs out of time. Recall that Z3 utilizes may summaries which

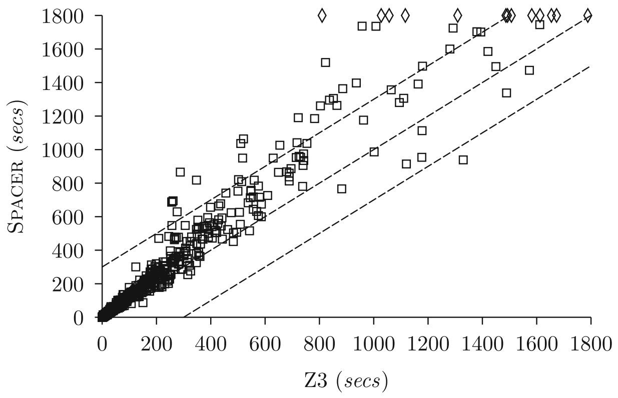


Fig. 12 SPACER versus Z3 for the SLAM benchmarks (with ±5 min boundaries)

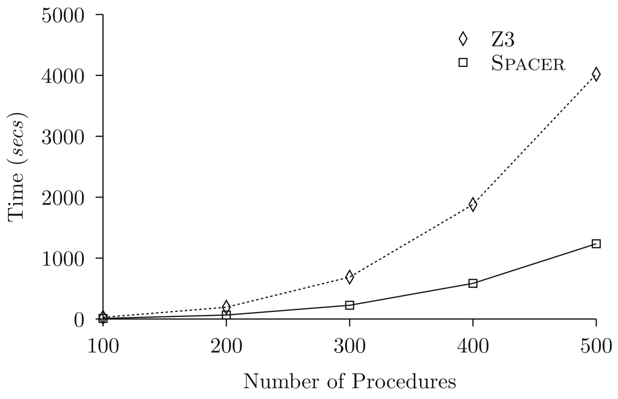


Fig. 13 SPACER versus Z3 for the Boolean program in Fig. 1 which is parametric in the number of procedures

heuristically avoid the possible exponential blow-up associated with unwinding the call-graph and as the plot shows, such a heuristic approach can be better than SPACER in some cases. However, when we compared the tools on the parametric Boolean program from Fig. 1, in which the size of the unrolled call-tree necessarily grows exponentially in the number of procedures, SPACER handles the increasing complexity significantly better than Z3, as shown in Fig. 13.

6.2 SDV Benchmarks

Figure 14 shows the scatter plot of runtimes for SPACER and Z3 for the SDV benchmarks. SPACER clearly outperforms Z3 including a benchmark where Z3 runs out of time.

6.3 SVCOMP 2014 Benchmarks

We begin with the scatter plot in Fig. 15 for SVCOMP-1 encodings. As mentioned above, SVCOMP-1 encodings correspond to while-programs and therefore, do not require must summaries. As the GPDR algorithm also computes may summaries, the plot in Fig. 15 essentially

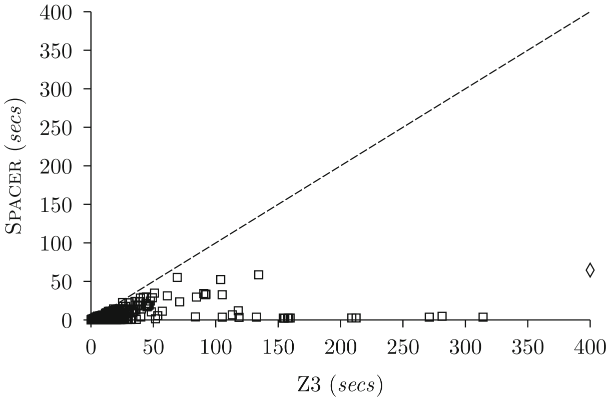


Fig. 14 SPACER versus Z3 for the SDV benchmarks

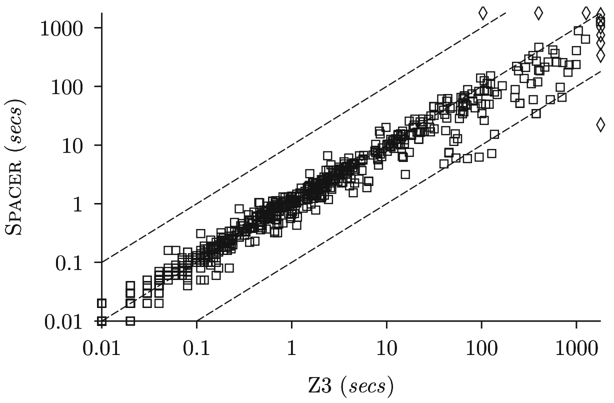


Fig. 15 The advantage of MBP over SVCOMP-1 encodings. As each of these encodings corresponds to a single procedure, must summaries are not required and MBP is the only key difference between SPACER and Z3. Axes are logarithmically scaled

shows the advantage of using MBP in creating a new query as opposed to Z3's variable substitution based on a given model.[5]

To understand the effect of must summaries, we also created a version of SPACER that only infers and utilizes may summaries. We obtained this by modifying Z3 to use MBP in creating new queries. As shown in Fig. 16, the advantage of using must summaries is quite significant on SVCOMP-2 encodings.

So, a combination of MBP and must summaries is expected to result in significant improvements over using may summaries alone. This is shown experimentally in Fig. 17 and 18 for the SVCOMP-2 and SVCOMP-3 encodings which show that SPACER is significantly better than Z3 on most of the programs.

Recall that the rule QUERY checks the feasibility of a potential counterexample path π by recursively creating a new reachability query for a procedure R called along π. Due to our logical representation of a program, one can consider an arbitrary permutation of the

[5] Z3 first tries to eliminate existential quantifiers by using equalities with ground terms present in the input formula and resorts to model substitution otherwise.

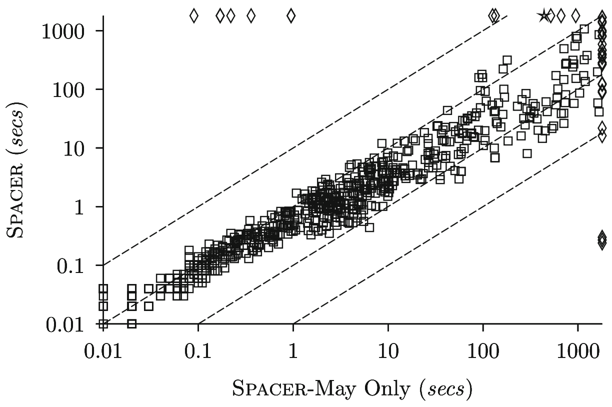


Fig. 16 The advantage of must summaries over SVCOMP-2 encodings. Axes are logarithmically scaled

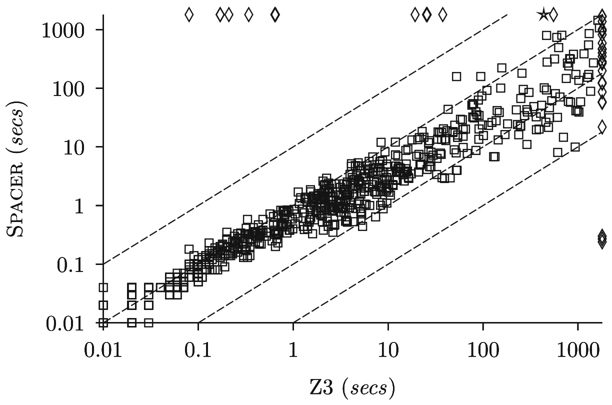


Fig. 17 SPACER versus Z3 for SVCOMP-2 encodings. Axes are logarithmically scaled

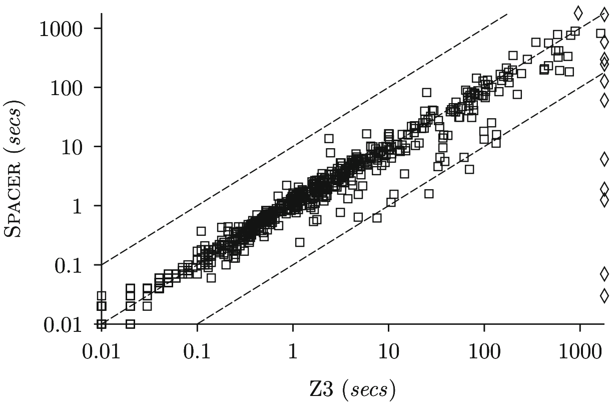
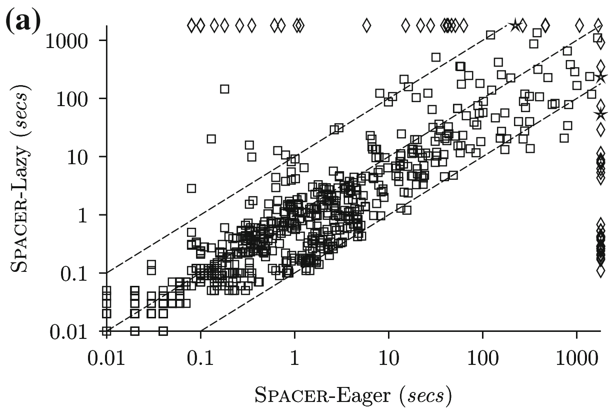


Fig. 18 SPACER versus Z3 for SVCOMP-3 encodings. Axes are logarithmically scaled



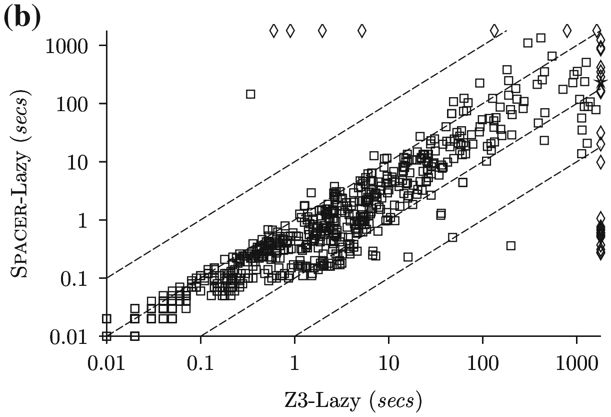


Fig. 19 Effect of changing the order of query creation in QUERY in (**a**) SPACER and (**b**) the corresponding change in Z3, on SVCOMP-2 encodings. Axes are logarithmically scaled

conjuncts of π when applying the rule and the choice of the procedure R is not deterministic. Our current implementation in SPACER can order the conjuncts either in the given order or in the reversed order and for lack of good heuristics, we do not consider other permutations. These two orderings correspond to top-down and bottom-up feasibility analyses. In particular, the plot shown in Fig. 17 corresponds to a bottom-up analysis.

As mentioned in the beginning, the SVCOMP-2 encodings are obtained by taking the while-program encodings in SVCOMP-1 and factoring out maximal loop-free fragments into new loop-free, recursion-free procedures. Furthermore, as also mentioned in the beginning, loops are encoded in SVCOMP-1 by introducing new predicate symbols that denote loop invariants and by encoding the corresponding verification conditions. So, a path in a procedure in the resulting logical encoding (see Sect. 3) contains at most two calls, one corresponding to an invariant at a control location and the other corresponding to a newly introduced procedure for a loop-free fragment. Thus, a top-down analysis refines the may summaries of the new procedures only when necessary, similar to a CEGAR-style reasoning where the may summaries of the new procedures abstract the loop-free fragments. We call this a *lazy* refinement strategy. In contrast, a bottom-up analysis on these encodings corresponds to an *eager* refinement strategy which is shown in Fig. 17. Figure 19a shows a scatter plot of

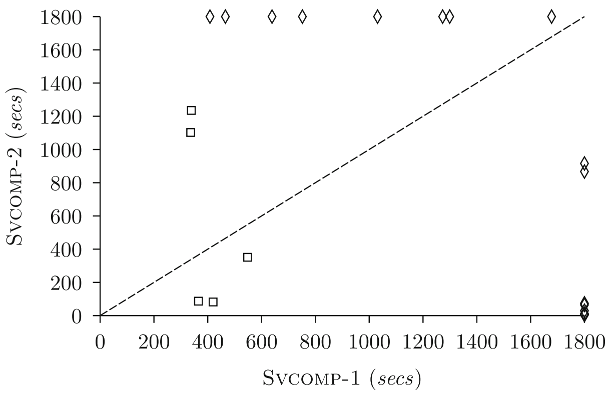


Fig. 20 Comparison of SPACER's behavior on the encodings SVCOMP-1 and SVCOMP-2 of the same SVCOMP benchmarks. This plot only includes data for the benchmarks where SPACER takes more than 5 min on the SVCOMP-1 encodings. We use SPACER in the *lazy* mode for SVCOMP-2 encodings

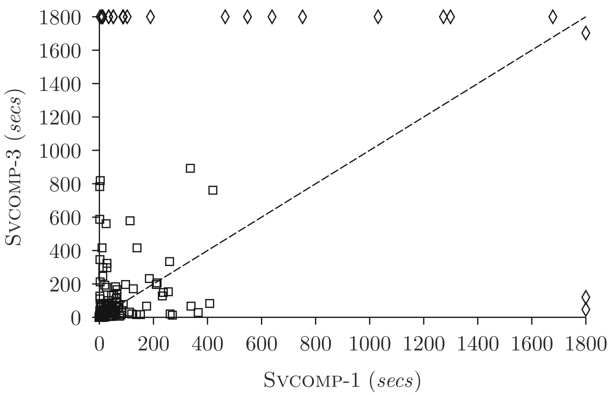


Fig. 21 Comparison of SPACER's behavior on the encodings SVCOMP-1 and SVCOMP-3 of the same SVCOMP benchmarks

runtimes on SVCOMP-2 comparing the behavior of SPACER for the two orderings. While it is unclear from the figure which ordering is better, SPACER continues to outperform Z3 even with the lazy strategy, as shown in Fig. 19b.

Finally, as an interesting exercise, we compared the behavior of SPACER on various encodings of the SVCOMP benchmarks. For the comparison of runtimes between SVCOMP-2 and SVCOMP-1, we restricted ourselves to the benchmarks where SPACER takes more than 5 min on the SVCOMP-1 encodings and we considered the lazy mode of SPACER for the SVCOMP-2 encodings. Note that checking safety of the SVCOMP-2 encodings in the lazy mode essentially corresponds to abstract reasoning by inferring sufficient summaries of the loop-free fragments. So, the rationale behind restricting to this subset of benchmarks is that we can see how helpful abstraction is on hard benchmarks. Figure 20 shows the runtime comparison for the benchmarks where we see that there is no clear winner. However, as we saw in an earlier work [31], abstraction can be quite powerful for SMT-based model checking and we plan to incorporate those ideas into the framework of RECMC in the future. Then, Fig. 21 shows the runtime comparison for the encoding SVCOMP-3 against SVCOMP-1. Recall that SVCOMP-3 encodings are obtained by factoring out loops into tail-recursive procedures. In other words, we are replacing the inference of loop invariants by that of summaries of the corresponding tail-recursive procedures. Whereas a loop invariant depends on the variables in scope, the signature of the corresponding tail-recursive procedure, and hence its summary, depends on two copies of the variables in scope which denote their values before and after a loop iteration. As the plot shows, this can negatively affect the performance of verification.

Overall, we have shown significant practical benefits of the core ideas behind RecMC using our implementation in Spacer and various realistic benchmarks.

7 Related work

There is a large body of work on interprocedural program analysis. It was pointed out early on that safety verification of recursive programs is reducible to the computation of a fixed-point over relations representing the input-output behavior of each procedure [9]. Such relations are also called *summaries* in the *functional approach* of Sharir and Pnueli [10]. Reps et al. [8] showed that for a large class of finite, interprocedural dataflow problems, the summaries can be computed in time polynomial in the number of dataflow facts and procedures. Ball and Rajamani [1] adapted the RHS algorithm to the verification of Boolean programs as part of the SLAM project.

Following SLAM, other software model checkers – such as BLAST [32] and MAGIC [33] – also implemented the CEGAR loop with predicate abstraction. These approaches do not use under-approximating summaries as we do.

In the context of predicate abstraction, the algorithm Smash also combines over- and under-approximations for analyzing procedural programs [34]. However, the summaries in Smash can have auxiliary variables which differ from one calling context to another, restricting the reusability of the summaries. Smash also under-approximates existential quantification in computing the results of the *post* and *pre* operations, but unlike RecMC, the under-approximations are obtained using concrete values encountered during testing of the program.

As mentioned earlier in the paper, several SMT-based algorithms have been proposed for safety verification of recursive programs, including Whale [14], HSF [6], Duality [18], Ultimate Automizer [16], and Corral [35]. These algorithms share a similar structure – they use SMT-solvers to look for counterexamples and interpolation to compute over-approximating procedure 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 GPDR [15], which extends the hardware model checking algorithm IC3 of Bradley [22] to SMT-supported theories and recursive programs. Unlike RecMC, GPDR does not maintain must-summaries. In the context of Fig. 7, this means that σ_u is always empty and there is no Must rule. Instead, the Query rule is modified to use a model M that satisfies the premises (instead of our use of the entire path π when creating a query). Furthermore, the reachable 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 may-summary). Since there are only finitely many models, the algorithm is guaranteed to terminate. However, in the case of Linear Arithmetic, a formula can have infinitely many models and GPDR might end up applying the Query rule indefinitely (see Appendix). In contrast, RecMC creates only finitely many queries for a given bound on the call-stack depth and is guaranteed to find a counterexample if one exists.

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* [31,36,37] to explore a combination of the approximations of procedure semantics with transition-relation abstraction.

Acknowledgments 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 Aarti Gupta, Cesare Tinelli, and the anonymous reviewers of both the CAV 2014 version of this paper and this FMSD special issue 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.

A divergence of GPDR for bounded call-stack

Consider the program $\langle\langle M, L, G\rangle, M\rangle$ with procedures $M = \langle y_0, y, \Sigma_M, \langle x, n\rangle, \beta_M\rangle$, $L = \langle n, \langle x, y, i\rangle, \Sigma_L, \langle x_0, y_0, i_0\rangle, \beta_L\rangle$, $G = \langle x_0, x_1, \Sigma_G, \emptyset, \beta_G\rangle$ with the following bodies:

$$\begin{aligned}
\beta_M &= \Sigma_L(x, y_0, n, n) \wedge \Sigma_G(x, y) \wedge n > 0\\
\beta_L &= (i = 0 \wedge x = 0 \wedge y = 0) \vee\\
&\quad\ (\Sigma_L(x_0, y_0, i_0, n) \wedge x = x_0 + 1 \wedge y = y_0 + 1 \wedge i = i_0 + 1 \wedge i > 0)\\
\beta_G &= (x = x_0 + 1)
\end{aligned}$$

The GPDR [15] algorithm can be shown to diverge when checking the bounded safety problem $M \models_2 y_0 \leq y$, for e.g., by inferring the diverging sequence of over-approximations of $[\![L]\!]^1$:

$$(x < 2 \implies y \leq 1), (x < 3 \implies y \leq 2), \ldots$$

We also observed this behavior experimentally (Z3 revision `d548c51` at http://z3.codeplex.com). The Horn-SMT file for the example is available at http://www.cs.cmu.edu/~akomurav/projects/spacer/gpdr_diverging.smt2.

References

1. Ball T, Rajamani SK (2000) Bebop: a symbolic model checker for Boolean programs. In: SPIN, pp 113–130

2. Ball T, Majumdar R, Millstein T, Rajamani SK (2001) Automatic predicate abstraction of C programs. SIGPLAN Not 36(5):203–213
3. Clarke EM, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: TACAS
4. Barnett M, Chang B-YE, DeLine R, Jacobs B, Leino KRM (2005) Boogie: a modular reusable verifier for object-oriented programs. In: FMCO, pp 364–387
5. Albarghouthi A, Gurfinkel A, Chechik M (2012) From under-approximations to over-approximations and back. In: TACAS
6. Grebenshchikov S, Lopes NP, Popeea C, Rybalchenko A (2012) Synthesizing software verifiers from proof rules. In: PLDI, pp 405–416
7. Clarke EM (1979) Programming language constructs for which it is impossible to obtain good Hoare axiom systems. JACM 26(1):129–147
8. Reps TW, Horwitz S, Sagiv S (1995) Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp 49–61
9. Clarke EM (1979) Program invariants as fixed points. Computing 21(4):273–294
10. Sharir M, Pnueli A (1981) Program flow analysis: theory and applications. In: Two approaches to interprocedural data flow analysis. Prentice-Hall, pp 189–233
11. Alur R, Benedikt M, Etessami K, Godefroid P, Reps T, Yannakakis M (2005) Analysis of recursive state machines. TOPLAS 27(4):786–818
12. Esparza J, Hansel D, Rossmanith P, Schwoon S (2000) Efficient algorithms for model checking pushdown systems. In: CAV, pp 232–247
13. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: CAV
14. Albarghouthi A, Gurfinkel A, Chechik M (2012) Whale: an interpolation-based algorithm for interprocedural verification. In: VMCAI, pp 39–55
15. Hoder K, Bjørner N (2012) Generalized property directed reachability. In: SAT
16. Heizmann M, Christ J, Dietsch D, Ermis E, Hoenicke J, Lindenmann M, Nutz A, Schilling C, Podelski A (2013) Ultimate automizer with SMTInterpol—(competition contribution). In: Piterman N, Smolka SA (eds) TACAS, lecture notes in computer science, vol 7795. Springer, Heidelberg, pp 641–643
17. Heizmann M, Hoenicke J, Podelski A (2010) Nested interpolants. SIGPLAN Not 45(1):471–482 1
18. McMillan KL, Rybalchenko A (2013) Solving constrained horn clauses using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research
19. Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y (2003) Bounded model checking. Adv Comput 58:117–148
20. Craig W (1957) Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. Symb Logic 22(3):269–285
21. Alur R, Benedikt M, Etessami K, Godefroid P, Reps T, Yannakakis M (2005) Analysis of recursive state machines. ACM Trans Program Lang Syst 27(4):786–818
22. Bradley AR (2011) SAT-based model checking without unrolling. In: VMCAI
23. Loos R, Weispfenning V (1993) Applying linear quantifier elimination. Computing 36(5):450–462
24. Cooper DC (1972) Theorem proving in arithmetic without multiplication. Mach Intel 7(91—-100):300
25. Nieuwenhuis R, Oliveras A, Tinelli C (2006) Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J ACM 53(6):937–977
26. De Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: TACAS
27. Ganai MK, Gupta A, Ashar P (2004) Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In: ICCAD, pp 510–517
28. Nipkow T (2010) Linear quantifier elimination. J Autom Reason 45(2):189–212
29. Albarghouthi A, Gurfinkel A, Li Y, Chaki S, Chechik M (2013) UFO: verification with interpolants and abstract interpretation—(competition contribution). In: TACAS
30. Software Verification Competition. TACAS, 2014. http://sv-comp.sosy-lab.org
31. Komuravelli A, Gurfinkel A, Chaki S, Clarke EM (2013) Automated abstraction in SMT-based unbounded software model checking. In: CAV, pp 846–862
32. Henzinger TA, Jhala R, Majumdar R, Sutre G (2002) Lazy abstraction. In: Proceedings of the POPL, pp 58–70
33. Chaki S, Clarke EM, Groce A, Jha S, Veith H (2004) Modular verification of software components in C. IEEE Trans Softw Eng 30(6):388–402
34. Godefroid P, Nori AV, Rajamani SK, Tetali S (2010) Compositional may-must program analysis: unleashing the power of alternation. In: POPL, pp 43–56
35. Lal A, Qadeer S, Lahiri SK (2012) A solver for reachability modulo theories. In: Madhusudan P, Seshia SA (eds) CAV, lecture notes in computer science, vol 7358. Springer, Heidelberg, pp 427–443

36. Gupta A, Ganai MK, Yang Z, Ashar P (2003) Iterative abstraction using SAT-based BMC with proof analysis. In: ICCAD, pp 416–423
37. McMillan KL, Amla N (2003) Automatic abstraction without counterexamples. In: TACAS