Automatic Abstraction in SMT-Based Unbounded Software Model Checking [⋆]

Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract. Software model checkers based on under-approximations and SMT solvers are very successful at verifying safety (*i.e.*, reachability) properties. They combine two key ideas – (a) *concreteness*: a counterexample in an under-approximation is a counterexample in the original program as well, and (b) *generalization*: a proof of safety of an under-approximation, produced by an SMT solver, are generalizable to proofs of safety of the original program. In this paper, we present a combination of *automatic abstraction* with the under-approximation-driven framework. We explore two iterative approaches for obtaining and refining abstractions – *proof based* and *counterexample based* – and show how they can be combined into a unified algorithm. To the best of our knowledge, this is the first application of Proof-Based Abstraction, primarily used to verify hardware, to Software Verification. We have implemented a prototype of the framework using Z3, and evaluate it on many benchmarks from the Software Verification Competition. We show experimentally that our combination is quite effective on hard instances.

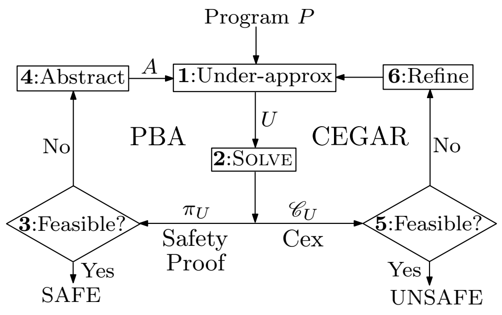
1 Introduction

Algorithms based on generalizing from under-approximations are very successful at verifying safety properties, *i.e.*, absence of bad executions (e.g., [2,10,26]). Those techniques use what we call a *Bounded Model Checking-Based Model Checking* (2BMC). The key idea of 2BMC is to iteratively construct an under-approximation U of the target program P by unwinding its transition relation and check whether U is safe using Bounded Model Checking (BMC) [8]. If U is unsafe, so is P. Otherwise, a proof π_U is produced explaining *why* U is safe. Finally, π_U is generalized (if possible) to a safety proof of P. Notable instances of

[⋆] 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. This material has been approved for public release and unlimited distribution. (DM-0000279).

```
0: x=0; y=0; z=0; w=0;
1: while(*) {
2:   if(*) {x++; y=y+100;}
3:   else if(*)
4:     if (x>=4) {x++; y++;}
5:   else if(y>10*w &&
              z>=100*x)
6:     {y=-y;}
7:   t=1;
8:   w=w+t; z=z+(10*t);
   }
9: assert(!(x>=4 && y<=2));
```

(a)



(b)

Fig. 1: (a) A program P_g by Gulavani et al. [18]; (b) an overview of SPACER.

2BMC are based on interpolation (e.g., [2,26]) or Property Directed Reachability (PDR) [9,13] (e.g., [10,21]).

At the same time, automatic abstraction refinement, such as CounterExample Guided Abstraction Refinement (CEGAR) [11], is very effective [2,7,20]. The idea is to iteratively construct, verify, and refine an abstraction (*i.e.*, an over-approximation) of P based on abstract counterexamples. In this paper, we present SPACER[1], an algorithm that combines abstraction with 2BMC.

For example, consider the safe program P_g by Gulavani et al. [18] shown in Fig. 1(a). P_g is hard for existing 2BMC techniques. For example, μZ engine of Z3 [12] (v4.3.1) that implements Generalized PDR [21] cannot solve it within an hour. However, its abstraction $\hat{P}_g$ obtained by replacing line 7 with a non-deterministic assignment to `t` is solved by the same engine in under a second. Our implementation of SPACER finds a safe abstraction of P_g in under a minute (the transition relation of the abstraction we automatically computed is a non-trivial generalization of that of P_g and does not correspond to $\hat{P}_g$).

SPACER tightly connects *proof-based* (PBA) and *counterexample-based* (CEGAR) abstraction-refinement schemes. An overview of SPACER is shown in Fig. 1(b). The input is a program P with a designated error location *er* and the output is either SAFE with a proof that *er* is unreachable, or UNSAFE with a counterexample to *er*. SPACER is sound, but obviously incomplete, *i.e.*, it is not guaranteed to terminate.

During execution, SPACER maintains an abstraction A of P, and an under-approximation U of A. We require that the safety problem for U is decidable. So, U is obtained by considering finitely many finitary executions of A. Initially, A is any abstraction of P (or P itself) and U is some under-approximation (step 1) of A. In each iteration, the main decision engine, called SOLVE, takes U and outputs either a proof π_U of safety (as an inductive invariant) or a counterexample trace $\mathscr{C}_U$ of U (step 2). In practice, SOLVE is implemented by an interpolating SMT-solver (e.g., [17,23]), or a generalized Horn Clause solver (e.g., [28,16,21]). If U is safe and π_U is also valid for P (step 3), SPACER terminates with SAFE; otherwise, it constructs a new abstraction $\hat{A}$ (step 4) using π_U, picks an under-

[1] Software Proof-based Abstraction with CounterExample-based Refinement.

approximation $\hat{U}$ of $\hat{A}$ (step 1), and goes into the next iteration. If U is unsafe and $\mathscr{C}_U$ is a feasible trace of P (step 5), Spacer terminates with UNSAFE; otherwise, it refines the under-approximation U to refute $\mathscr{C}_U$ (step 6) and goes to the next iteration. Spacer is described in Section 4 and a detailed run of the algorithm on an example is given in Section 2.

Note that the left iteration of Spacer (steps 1, 2, 3, 4) is PBA: in each iteration, an under-approximation is solved, a new abstraction based on the proof is computed and a new under-approximation is constructed. To the best of our knowledge, this is the first application of PBA to Software Model Checking. The right iteration (steps 1, 2, 5, 6) is CEGAR: in each iteration, (an under-approximation of) an abstraction is solved and refined by eliminating spurious counterexamples. Spacer exploits the natural duality between the two.

While Spacer is not complete, each iteration makes progress either by proving safety of a bigger under-approximation, or by refuting a spurious counterexample. Thus, when resources are exhausted, Spacer can provide useful information for other verification attempts and for increasing confidence in the program.

We have implemented Spacer using μZ [21] as Solve (Section 5) and evaluated it on many benchmarks from the 2nd Software Verification Competition[2] (SV-COMP'13). Our experimental results (see Section 6) show that the combination of 2BMC and abstraction outperforms 2BMC on hard benchmarks.

In summary, the paper makes the following contributions: (a) an algorithm, Spacer, that combines abstraction and 2BMC and tightly connects proof- and counterexample-based abstractions, (b) an implementation of Spacer using μZ engine of Z3 and (c) experimental results showing the effectiveness of Spacer.

2 Overview

In this section, we illustrate Spacer on the program P shown in Fig. 2(a). Function `nd()` returns a value non-deterministically and `assume(0)` aborts an execution. Thus, at least one of the updates on lines 3, 4 and 5 must take place in every iteration of the loop on line 2. Note that the variable `c` counts down the number of iterations of the loop to 0, upper bounded by `b`. A restriction to `b` is an under-approximation of P. For example, adding '`assume(b<=0);`' to line 1 corresponds to the under-approximation of P that allows only loop-free executions; adding '`assume(b<=1);`' to line 1 corresponds to the under-approximation that allows at most one execution through the loop, etc. While in this example the *counter variable* `c` is part of P, we synthesize such variables automatically in practice (see Section 5).

Semantically, P is given by the transition system shown in Fig 2(b). The control locations *en*, *lp*, and *er* correspond to lines 0, 2, and 8 in P, respectively. An edge from ℓ_1 to ℓ_2 corresponds to all loop-free executions starting at ℓ_1 and ending at ℓ_2. For example, the self-loop on *lp* corresponds to the body of the loop. Finally, every edge is labeled by a formula over current (unprimed)

[2] `http://sv-comp.sosy-lab.org`

```
x=0; y=0; z=0; w=0; c=nd();
b=nd();
while (0<c<=b) {
  if (nd()) {x++; y=y+100;}
  else if (nd() && x>=4) {x++; y++;}
  else if (y>10*w && z>=100*x) {y=-y;}
  else assume (0);
  w++; z=z+10; c--;
}
assert (!(c==0 && x>=4 && y<=2));
```

(a)

$$\begin{tikzcd} en \arrow[r, "I"] & lp \arrow[r, "E"] \arrow[loop below, "T"] & er \end{tikzcd}$$

$$\begin{aligned}
I &\equiv (x' = y' = z' = w' = 0)\\
T &\equiv [(x' = x + 1 \ \wedge\ y' = y + 100) \vee\\
&\quad (x \geq 4 \ \wedge\ x' = x + 1 \ \wedge\ y' = y + 1) \vee\\
&\quad (y > 10w \ \wedge\ z \geq 100x \ \wedge\\
&\qquad y' = -y \ \wedge\ x' = x)] \wedge\\
&\quad w' = w + 1 \ \wedge\ z' = z + 10 \ \wedge\\
&\quad 0 < c \leq b \ \wedge\ c' = c - 1\\
E &\equiv c = 0 \ \wedge\ x \geq 4 \ \wedge\ y \leq 2
\end{aligned}$$

(b)

Fig. 2: (a) A program P and (b) its transition system.

$$\begin{aligned}
\hat{I}_1 &\equiv (x' = y' = z' = w' = 0)\\
\hat{T}_1 &\equiv [(x' = x + 1) \vee\\
&\quad (x \geq 4 \ \wedge\ x' = x + 1) \vee\\
&\quad (y > 10w \ \wedge\ z \geq 100x)] \wedge\\
&\quad 0 < c \leq b \ \wedge\ c' = c - 1\\
\hat{E}_1 &\equiv c = 0 \ \wedge\ x \geq 4
\end{aligned}$$

(a) $\hat{P}_1$

$$\begin{aligned}
\hat{I}_2 &\equiv (x' = y' = z' = w' = 0)\\
\hat{T}_2 &\equiv [(x' = x + 1 \ \wedge\ y' = y + 100) \vee\\
&\quad (x \geq 4 \ \wedge\ x' = x + 1 \ \wedge\ y' = y + 1) \vee\\
&\quad (y > 10w \ \wedge\ z \geq 100x)] \wedge\\
&\quad 0 < c \leq b \ \wedge\ c' = c - 1\\
\hat{E}_2 &\equiv c = 0 \ \wedge\ x \geq 4 \ \wedge\ y \leq 2
\end{aligned}$$

(b) $\hat{P}_2$

Fig. 3: Abstractions $\hat{P}_1$ and $\hat{P}_2$ of P in Fig. 2(b).

and next-state (primed) variables denoting the semantics of the corresponding executions. Hence, I and E denote the initial and error conditions, respectively, and T denotes the loop body. In the rest of the paper, we do not distinguish between semantic and syntactic representations of programs.

Our goal is to find a safety proof for P, *i.e.*, a labeling π of *en*, *lp* and *er* with a set of formulas (called *lemmas*) that satisfies safety, initiation and inductiveness:

$$\bigwedge \pi(er) \Rightarrow \bot, \quad \top \Rightarrow \bigwedge \pi(en), \quad \forall \ell_1, \ell_2 \centerdot \left(\bigwedge \pi(\ell_1) \ \wedge\ \tau(\ell_1, \ell_2)\right) \Rightarrow \bigwedge \pi(\ell_2)'.$$

where $\tau(\ell_1, \ell_2)$ is the label of edge from ℓ_1 to ℓ_2, and for an expression X, X' is obtained from X by priming all variables. In the following, we refer to Fig. 1(b) for the steps of the algorithm.

Steps 1 and 2. Let U_1 be the under-approximation obtained from P by conjoining $(b \leq 2)$ to T. It is safe, and suppose that SOLVE returns the safety proof π_1, shown in Fig. 4(a).

Step 3. To check whether π_1 is also a safety proof of the concrete program P, we extract a *Maximal Inductive Subset* (MIS), $\mathcal{I}_1$ (shown in Fig. 4(b)), of π_1, with respect to P. That is, for every location ℓ, $\mathcal{I}_1(\ell) \subseteq \pi_1(\ell)$, and $\mathcal{I}_1$ satisfies the initiation and inductiveness conditions above. $\mathcal{I}_1$ is an inductive invariant of P, but is not safe (*er* is not labeled with $\bot$). Hence, π_1 does not contain a feasible proof, and another iteration of SPACER is required.

$$\begin{array}{ll} \mathit{en}: & \{\} \\ \mathit{lp}: & \{(z \leq 100x - 90 \vee \\ & \quad y \leq 10w), \\ & z \leq 100x, x \leq 2 \\ & (x \leq 0 \vee c \leq 1) \\ & (x \leq 1 \vee c \leq 0)\} \\ \mathit{er}: & \{\bot\} \end{array}$$

(a) π_1: safety proof of U_1.

$$\begin{array}{ll} \mathit{en}: & \{\} \\ \mathit{lp}: & \{(z \leq 100x - 90 \vee \\ & \quad y \leq 10w), \\ & z \leq 100x\} \\ \mathit{er}: & \{\} \end{array}$$

(b) $\mathcal{I}_1$: invariants of P.

$$\begin{array}{ll} \mathit{en}: & \{\} \\ \mathit{lp}: & \{(z \leq 100x - 90 \vee \\ & \quad y \leq 10w), \\ & z \leq 100x, y \geq 0, \\ & (x \leq 0 \vee y \geq 100)\} \\ \mathit{er}: & \{\bot\} \end{array}$$

(c) π_3: safety proof of U_3.

Fig. 4: Proofs and invariants for the running example in Section 2.

Step 4. We obtain an abstraction $\hat{P}_1$ of P for which, assuming the invariants in $\mathcal{I}_1$, π_1 is a safety proof for the first two iterations of the loop (*i.e.*, when $b \leq 2$). For this example, let $\hat{P}_1$ be as shown in Fig. 3(b). Note that $\hat{T}_1$ has no constraints on the next-state values of z, y and w. This is okay for π_1 as $\mathcal{I}_1(\mathit{lp})$ already captures the necessary relation between these variables. In other words, while $\hat{T}_1$ is a structural (or *syntactic*) abstraction [6], we consider its restriction to the invariants $\mathcal{I}_1$ making it a more expressive, *semantic* abstraction. The next iteration of Spacer is described below.
Steps 1 and 2. Let U_2 be the under-approximation obtained from $\hat{P}_1$ by conjoining $(b \leq 4) \wedge \mathcal{I}_1 \wedge \mathcal{I}_1'$ to $\hat{T}_1$. It is not safe and let Solve return a counterexample $\mathscr{C}_2$ as the pair $\langle \bar{\ell}, \bar{s} \rangle$ of the following sequences of locations and states, corresponding to incrementing x from 0 to 4 with an unconstrained y:

$$\begin{aligned} \bar{\ell} &\equiv \langle \mathit{en}, \mathit{lp}, \mathit{lp}, \mathit{lp}, \mathit{lp}, \mathit{lp}, \mathit{er} \rangle \\ \bar{s} &\equiv \langle (0,0,0,0,0,0), (0,0,0,0,4,4), (1,0,0,0,3,4), (2,0,0,0,2,4), \\ &\quad (3,0,0,0,1,4), (4,3,0,0,0,4), (4,3,0,0,0,4) \rangle \end{aligned} \tag{1}$$

where a state is a valuation to the tuple (x, y, z, w, c, b).
Steps 5 and 6. $\mathscr{C}_2$ is infeasible in P as the last state does not satisfy E. $\hat{P}_1$ is refined to $\hat{P}_2$, say as shown in Fig. 3(b), by adding the missing constraints on y.
Steps 1 and 2. Let U_3 be the under-approximation obtained from $\hat{P}_2$ by conjoining $(b \leq 4) \wedge \mathcal{I}_1 \wedge \mathcal{I}_1'$ to $\hat{T}_2$. It is safe, and let Solve return the proof π_3 shown in Fig. 4(c).
Step 3. π_3 is a MIS of itself, with respect to P. Thus, it is a safety proof for P and Spacer terminates.

While we have carefully chosen the under-approximations to save space, the abstractions, lemmas and invariants shown above were all computed automatically by our prototype implementation starting with the initial under-approximation of $b \leq 0$ and incrementing the upper bound by 1, each iteration. Even on this small example, our prototype, built using μZ, is five times faster than μZ by itself.

3 Preliminaries

This section defines the terms and notation used in the rest of the paper.

Definition 1 (Program). *A program P is a tuple $\langle L, \ell^o, \ell^e, V, \tau\rangle$ where*

1. *L is the set of control locations,*
2. *$\ell^o \in L$ and $\ell^e \in L$ are the unique initial and error locations,*
3. *V is the set of all program variables (Boolean or Rational), and*
4. *$\tau : L \times L \to BExpr(V \cup V')$ is a map from pairs of locations to Boolean expressions over $V \cup V'$ in propositional Linear Rational Arithmetic.*

Intuitively, $\tau(\ell_i, \ell_j)$ is the relation between the current values of V at ℓ_i and the next values of V at ℓ_j on a transition from ℓ_i to ℓ_j. We refer to τ as the transition relation. Without loss of generality, we assume that $\forall \ell \in L \centerdot \tau(\ell, \ell^o) = \bot \wedge \tau(\ell^e, \ell) = \bot$. We refer to the components of P by a subscript, e.g., L_P.

Fig. 2(b) shows an example program with $L = \{en, lp, er\}$, $\ell^o = en$, $\ell^e = er$, $V = \{x, y, z, w, c, b\}$, $\tau(en, lp) = I$, $\tau(lp, lp) = T$, $\tau(lp, er) = E$.

Let $P = \langle L, \ell^o, \ell^e, V, \tau\rangle$ be a program. A *control path* of P is a finite[3] sequence of control locations $\langle \ell^o = \ell_0, \ell_1, \ldots, \ell_k\rangle$, beginning with the initial location ℓ^o, such that $\tau(\ell_i, \ell_{i+1}) \neq \bot$ for $0 \leq i < k$. A *state* of P is a valuation to all the variables in V. A control path $\langle \ell^o = \ell_0, \ell_1, \ldots, \ell_k\rangle$ is called *feasible* iff there is a sequence of states $\langle s_0, s_1, \ldots, s_k\rangle$ such that

$$\forall 0 \leq i < k \centerdot \tau(\ell_i, \ell_{i+1})[V \leftarrow s_i, V' \leftarrow s_{i+1}] = \top \tag{2}$$

i.e., each successive and corresponding pair of locations and states satisfy τ.

For example, $\langle en, lp, lp, lp\rangle$ is a feasible control path of the program in Fig. 2(b) as the sequence of states $\langle (0,0,0,0,0,0), (0,0,0,0,2,2), (1,100,1,10,1,2), (2,200,2,20,0,2)\rangle$ satisfies (2).

A location ℓ is *reachable* iff there exists a feasible control path ending with ℓ. P is *safe* iff ℓ^e is *not* reachable. For example, the program in Fig. 2(b) is safe. P is *decidable*, when the safety problem of P is decidable. For example, the program U obtained from P in Fig. 2(b) by replacing b with 5 is decidable because (a) U has finitely many feasible control paths, each of finite length and (b) Linear Arithmetic is decidable.

Definition 2 (Safety Proof). *A* safety proof *for P is a map $\pi : L \to 2^{BExpr(V)}$ such that π is safe and inductive,* i.e.*,*

$$\bigwedge \pi(\ell^e) \Rightarrow \bot, \quad \top \Rightarrow \bigwedge \pi(\ell^o), \quad \forall \ell_i, \ell_j \in L \centerdot \Big(\bigwedge \pi(\ell_i) \ \wedge \ \tau(\ell_i, \ell_j)\Big) \Rightarrow \bigwedge \pi(\ell_j)'.$$

For example, Fig. 4(c) shows a safety proof for the program in Fig. 2(b). Note that whenever P has a safety proof, P is safe.

A *counterexample to safety* is a pair $\langle \bar{\ell}, \bar{s}\rangle$ such that $\bar{\ell}$ is a feasible control path in P ending with ℓ^e and $\bar{s}$ is a corresponding sequence of states satisfying τ along $\bar{\ell}$. For example, $\hat{P}_2$ in Fig. 3(b) admits the counterexample $\mathscr{C}_2$ shown in (1) in Section 2.

[3] In this paper, we deal with safety properties only.

Definition 3 (Abstraction Relation). *Given two programs, $P_1 = \langle L_1,\ \ell_1^o,\ \ell_1^e,\ V_1,\ \tau_1\rangle$ and $P_2 = \langle L_2,\ \ell_2^o,\ \ell_2^e,\ V_2,\ \tau_2\rangle$, P_2 is an* abstraction (i.e.*, an over-approximation) of P_1 via a surjection $\sigma : L_1 \to L_2$, denoted $P_1 \preceq_\sigma P_2$, iff*

$$V_1 = V_2, \quad \sigma(\ell_1^o) = \ell_2^o, \quad \sigma(\ell_1^e) = \ell_2^e, \quad \forall \ell_i, \ell_j \in L_1 \centerdot \tau_1(\ell_i, \ell_j) \Rightarrow \tau_2(\sigma(\ell_i), \sigma(\ell_j)).$$

P_1 is called a refinement (i.e.*, an under-approximation) of P_2. We say that P_2 strictly abstracts P_1 via σ, denoted $P_1 \prec_\sigma P_2$, iff $(P_1 \preceq_\sigma P_2) \wedge \neg\exists\nu \centerdot (P_2 \preceq_\nu P_1)$. When σ is not important, we drop the subscript.*

That is, P_2 abstracts P_1 iff there is a surjective map σ from L_1 to L_2 such that every feasible transition of P_1 corresponds (via σ) to a feasible transition of P_2. For example, if P_1 is a finite unrolling of P_2, then σ maps the locations of P_1 to the corresponding ones in P_2. P_2 *strictly abstracts* P_1 iff $P_1 \preceq P_2$ and there is no surjection ν for which $P_2 \preceq_\nu P_1$. For example, $P \prec_{id} \hat{P}_1$, where P is in Fig. 2(b) and $\hat{P}_1$ is in Fig. 3(a).

We extend $\sigma : L_1 \to L_2$ from locations to control paths in the straightforward way. For a counterexample $\mathscr{C} = \langle \bar{\ell}, \bar{s}\rangle$, we define $\sigma(\mathscr{C}) \equiv \langle \sigma(\bar{\ell}), \bar{s}\rangle$. For a transition relation τ on L_2, we write $\sigma(\tau)$ to denote an embedding of τ via σ, defined as follows: $\sigma(\tau)(\ell_1, \ell_2) = \tau(\sigma(\ell_1), \sigma(\ell_2))$. For example, in the definition above, if $P_1 \preceq_\sigma P_2$, then $\tau_1 \Rightarrow \sigma(\tau_2)$.

4 The Algorithm

In this section, we describe Spacer at a high-level. Low-level details of our implementation are described in Section 5. The pseudo-code of Spacer is shown in Fig. 5. The top level routine Spacer decides whether an input program P (passed through the global variable) is safe. It maintains (a) invariants $\mathcal{I}$ such that $\mathcal{I}(\ell)$ is a set of constraints satisfied by all the reachable states at location ℓ of P (b) an abstraction A of P, (c) a decidable under-approximation U of A and (d) a surjection σ such that $U \preceq_\sigma A$. Spacer ensures that $P \preceq_{id} A$, *i.e.*, A differs from P only in its transition relation. Let $A_\mathcal{I}$ denote the restriction of A to the invariants in $\mathcal{I}$ by strengthening τ_A to $\lambda\ell_1, \ell_2 \centerdot \mathcal{I}(\ell_1) \wedge \tau_A(\ell_1, \ell_2) \wedge \mathcal{I}(\ell_2)'$. Similarly, let $U_\mathcal{I}$ denote the strengthening of τ_U to $\lambda\ell_1, \ell_2 \centerdot \mathcal{I}(\sigma(\ell_1)) \wedge \tau_U(\ell_1, \ell_2) \wedge \mathcal{I}(\sigma(\ell_2))'$. Spacer assumes the existence of an oracle, Solve, that decides whether $U_\mathcal{I}$ is safe and returns either a safety proof or a counterexample.

Spacer initializes A to P and $\mathcal{I}$ to the empty map (line 1), calls InitU(A) to initialize U and σ (line 2) and enters the main loop (line 3). In each iteration, safety of $U_\mathcal{I}$ is checked with Solve (line 4). If $U_\mathcal{I}$ is safe, the safety proof π is checked for feasibility w.r.t. the original program P, as follows. First, π is mined for new invariants of P using ExtractInvs (line 6). Then, if the invariants at ℓ_P^e are unsatisfiable (line 7), the error location is unreachable and Spacer returns Safe (line 8). Otherwise, A is updated to a new proof-based abstraction via Abstract (line 9), and a new under-approximation is constructed using NextU (line 10). If, on the other hand, $U_\mathcal{I}$ is unsafe at line 4, the counterexample

```
    global($P$ : $prog$)
    global($\mathcal{I} : L_P \rightarrow 2^{BExpr(V_P)}$)

    Spacer ( )
    begin
1       $A := P, \mathcal{I} := \emptyset$
2       $(U, \sigma) :=$ InitU$(A)$
3       while true do
4           $(result, \pi, \mathscr{C}) :=$ Solve $(U_{\mathcal{I}})$
5           if result is Safe then
6               $\mathcal{I} = \mathcal{I} \cup$ ExtractInvs$(A, U, \pi)$
7               if $\bigwedge \mathcal{I}(\ell^e_P) \Rightarrow \bot$ then
8                   return Safe
9               $(A, U) :=$ Abstract$(A, U, \pi)$
10              $(U, \sigma) :=$ NextU$(A, U)$
            else
11              $(feas, A, U) :=$ Refine$(A, U, \mathscr{C})$
12              if $feas$ then
13                  return Unsafe

    Adapt($U$ : prog, $\tau$ : trans, $\sigma : L_U \rightarrow L_P$)
      requires($\tau$ : transition relation on $L_P$)
    begin
14      return $U[\tau_U \leftarrow (\tau_U \wedge \sigma(\tau))]$

    NextU($A$ : prog, $U$ : prog)
      requires($U \preceq_\sigma A$)
    begin
15      return $(\hat{U}, \sigma_2)$ s.t. $U \prec_{\sigma_1} \hat{U} \preceq_{\sigma_2} A$,
          $\sigma = \sigma_2 \circ \sigma_1$ and
          Adapt$(U, \tau_P, \sigma) \prec$ Adapt$(\hat{U}, \tau_P, \sigma_2)$

    Abstract($A, U$ : prog, $\pi$ : proof of $U$)
      requires($U \preceq_\sigma A, \tau_U = \sigma(\tau_A) \wedge \rho$)
    begin
16      let $\hat{U}$ be s.t. $L_{\hat{U}} = L_U$,
          $\tau_{\hat{U}} \equiv \sigma(\hat{\tau}_P) \wedge \hat{\rho}$ with $\tau_P \Rightarrow \hat{\tau}_P$,
          $\rho \Rightarrow \hat{\rho}$, and $\pi$ is a safety proof of $\hat{U}_{\mathcal{I}}$
17      return $(A[\tau_A \leftarrow \hat{\tau}_P], \hat{U})$

    Refine($\hat{A}, \hat{U}$ : prog, $\mathscr{C}$ : cex of $\hat{U}$)
      requires($\hat{U} \preceq_\sigma \hat{A}$)
    begin
18      $feas :=$ IsFeasible$(\sigma(\mathscr{C}), P)$
19      if $\neg feas$ then
20          let $A \prec_{id} \hat{A}$ s.t. $\neg$IsFeasible$(\sigma(\mathscr{C}), A_{\mathcal{I}})$
21          $U :=$ Adapt$(\hat{U}, \tau_A, \sigma)$
22          return (false, $A$, $U$)
23      return (true, None, None)

    ExtractInvs($A, U$ : prog, $\pi$ : proof of $U$)
      requires($U \preceq_\sigma A$)
    begin
24      $\mathcal{R} : L_P \rightarrow 2^{\mathrm{BExpr}(V_P)} := \emptyset$
25      for $\ell \in L_U$ do
26          add $\bigwedge \pi(\ell)$ to $\mathcal{R}(\sigma(\ell))$
27      for $\ell \in L_P$ do
28          $\mathcal{R}(\ell) := conjuncts(\bigvee \mathcal{R}(\ell))$
29      while $\exists \ell_i, \ell_j \in L_P, \varphi \in \mathcal{R}(\ell_j)$ s.t.
          $\neg\left(\mathcal{R}(\ell_i) \wedge \mathcal{I}(\ell_i) \wedge \tau_P(\ell_i, \ell_j) \Rightarrow \varphi'\right)$
        do
30          $\mathcal{R}(\ell_j) := \mathcal{R}(\ell_j) \setminus \{\varphi\}$
31      return $\mathcal{R}$
```

Fig. 5: Pseudo-code of Spacer.

$\mathscr{C}$ is validated using Refine (line 11). If $\mathscr{C}$ is feasible, Spacer returns Unsafe (line 13), otherwise, both A and U are refined (lines 20 and 21).

Next, we describe these routines in detail. Throughout, fix U, σ and A such that $U \preceq_\sigma A$.

ExtractInvs. For every $\ell \in L$, the lemmas of all locations in L_U which map to ℓ, via the surjection $\sigma : L_U \rightarrow L_P (= L_A)$, are first collected into $\mathcal{R}(\ell)$ (lines 25–26). The disjunction of $\mathcal{R}(\ell)$ is then broken down into conjuncts and stored back in $\mathcal{R}(\ell)$ (lines 27–28). For e.g., if $\mathcal{R}(\ell) = \{\phi_1, \phi_2\}$, obtain $\phi_1 \vee \phi_2 \equiv \bigwedge_j \psi_j$ and update $\mathcal{R}(\ell)$ to $\{\psi_j\}_j$. Then, the invariants are extracted as the maximal subset of $\mathcal{R}(\ell)$ that is mutually inductive, relative to $\mathcal{I}$, w.r.t. the concrete transition relation τ_P. This step uses the iterative algorithm on lines 29–30 and is similar to Houdini [15].

Abstract first constructs an abstraction $\hat{U}$ of U, such that π is a safety proof for $\hat{U}_{\mathcal{I}}$ and then, uses the transition relation of $\hat{U}$ to get the new abstraction. W.l.o.g., assume that τ_U is of the form $\sigma(\tau_A) \wedge \rho$. That is, τ_U is an embedding of τ_A via σ strengthened with ρ. An abstraction $\hat{U}$ of U is constructed such that $\tau_{\hat{U}} = \sigma(\hat{\tau}_P) \wedge \hat{\rho}$, where $\hat{\tau}_P$ abstracts the concrete transition relation τ_P, $\hat{\rho}$

abstracts ρ and π proves $\hat{U}_{\mathcal{I}}$ (line 16). The new abstraction is then obtained from A by replacing the transition relation by $\hat{\tau}_P$ (line 17).
NextU returns the next under-approximation $\hat{U}$ to be solved. It ensures that $U \prec \hat{U}$ (line 15), and that the surjections between U, $\hat{U}$ and A compose so that the corresponding transitions in U and $\hat{U}$ map to the same transitions of the common abstraction A. Furthermore, to ensure progress, NextU ensures that $\hat{U}$ contains *more concrete* behaviors than U (the last condition on line 15). The helper routine Adapt strengthens the transition relation of an under-approximation by an embedding (line 14).
Refine checks if the counterexample $\mathscr{C}$, via σ, is feasible in the original program P using IsFeasible (line 18). If $\mathscr{C}$ is feasible, Refine returns saying so (line 23). Otherwise, $\hat{A}$ is refined to A to (at least) eliminate $\mathscr{C}$ (line 20). Thus, $A \prec_{id} \hat{A}$. Finally, $\hat{U}$ is strengthened with the refined transition relation via Adapt (line 21).

The following statements show that Spacer is sound and maintains progress.

Lemma 1 (Inductive Invariants). *In every iteration of* Spacer, *$\mathcal{I}$ is inductive with respect to τ_P.*

Theorem 1 (Soundness). *P is safe (unsafe) if* Spacer *returns* Safe (Unsafe).

Theorem 2 (Progress). *Let A_i, U_i, and $\mathscr{C}_i$ be the values of A, U, and $\mathscr{C}$ in the i^{th} iteration of* Spacer *with $U_i \preceq_{\sigma_i} A_i$ and let $\dot{U}_i$ denote the concretization of U_i,* i.e., *result of* Adapt(U_i, τ_P, σ_i). *Then, if U_{i+1} exists,*

1. *if U_i is safe, U_{i+1} has strictly more concrete behaviors,* i.e., *$\dot{U}_i \prec \dot{U}_{i+1}$,*
2. *if U_i is unsafe, U_{i+1} has the same concrete behaviors,* i.e., *$\dot{U}_i \preceq_{id} \dot{U}_{i+1}$ and $\dot{U}_{i+1} \preceq_{id} \dot{U}_i$, and*
3. *if U_i is unsafe, $\mathscr{C}_i$ does not repeat in future,* i.e., *$\forall j > i \centerdot \sigma_j(\mathscr{C}_j) \neq \sigma_i(\mathscr{C}_i)$.*

In this section, we presented the high-level structure of Spacer. Many routines (InitU, ExtractInvs, Abstract, NextU, Refine, IsFeasible) are only presented by their interfaces with their implementation left open. In the next section, we complete the picture by describing the implementation used in our prototype.

5 Implementation

Let $P = \langle L, \ell^o, \ell^e, V, \tau \rangle$ be the input program. First, we transform P to $\tilde{P}$ by adding new *counter* variables for the loops of P and adding extra constraints to the transitions to count the number of iterations. Specifically, for each location ℓ we introduce a counter variable c_ℓ and a bounding variable b_ℓ. Let C and B be the sets of all counter and bounding variables, respectively, and *bound* : $C \to B$ be the bijection defined as $bound(c_\ell) = b_\ell$. We define $\tilde{P} \equiv \langle L, \ell^o, \ell^e, V \cup C \cup B, \tau \wedge \tau_B \rangle$, where $\tau_B(\ell_1, \ell_2) = \bigwedge X(\ell_1, \ell_2)$ and $X(\ell_1, \ell_2)$ is the smallest set satisfying the following conditions: (a) if $\ell_1 \to \ell_2$ is a back-edge, then $\left(0 \le c'_{\ell_2} \wedge c'_{\ell_2} = c_{\ell_2} - 1 \wedge c_{\ell_2} \le b_{\ell_2}\right) \in X(\ell_1, \ell_2)$, (b) else, if $\ell_1 \to \ell_2$ exits

Global	*Trans*	$E_{i,j} \Rightarrow \tau_\Sigma(\ell_i, \ell_j) \wedge \tau_B(\ell_i, \ell_j), \quad \ell_i, \ell_j \in L$	(1)
		$N_i \Rightarrow \bigvee_j E_{j,i}, \quad \ell_i \in L$	(2)
	Invars	$\left(\bigvee_j E_{i,j}\right) \Rightarrow \varphi, \quad \ell_i \in L, \varphi \in \mathcal{I}(\ell_i)$	(3)
		$N_i \Rightarrow \varphi', \quad \ell_i \in L, \varphi \in \mathcal{I}(\ell_i)$	(4)
Local	*Lemmas*	$\bigwedge_{\ell_i \in L, \varphi \in \pi(\ell_i)} \left(\mathcal{A}_{\ell_i,\varphi} \Rightarrow \left(\left(\bigvee_j E_{i,j}\right) \Rightarrow \varphi\right)\right)$	(5)
		$\neg \bigwedge_{\ell_i \in L, \varphi \in \pi(\ell_i)} (\mathcal{B}_{\ell_i,\varphi} \Rightarrow (N_i \Rightarrow \varphi'))$	(6)
	Assump. Lits	$\mathcal{A}_{\ell,\varphi}, \quad \ell \in L, \varphi \in \pi(\ell)$	(7)
		$\neg\mathcal{B}_{\ell,\varphi}, \quad \ell \in L, \varphi \in \pi(\ell)$	(8)
	Concrete	Σ	(9)
	Bound Vals	$b \leq bvals(b), \quad b \in B$	(10)

Fig. 6: Constraints used in our implementation of Spacer.

the loop headed by ℓ_k, then $(c_{\ell_k} = 0) \in X(\ell_1, \ell_2)$ and (c) otherwise, if $\ell_1 \to \ell_2$ is a transition inside the loop headed by ℓ_k, then $\left(c'_{\ell_k} = c_{\ell_k}\right) \in X(\ell_1, \ell_2)$. In practice, we use optimizations to reduce the number of variables and constraints.

This transformation preserves safety as shown below.

Lemma 2. *P is safe iff $\tilde{P}$ is safe,* i.e., *if $\tilde{\mathscr{C}} = \langle \bar{\ell}, \bar{s} \rangle$ is a counterexample to $\tilde{P}$, projecting $\bar{s}$ onto V gives a counterexample to P; if $\tilde{\pi}$ is a proof of $\tilde{P}$, then $\pi = \lambda\ell \cdot \{\forall B \geq 0, C \geq 0 \,\text{.}\, \varphi \mid \varphi \in \tilde{\pi}(\ell)\}$ is a safety proof for P.*

In the rest of this section, we define our abstractions and under-approximations of $\tilde{P}$ and describe our implementation of the different routines in Fig. 5.

Abstractions. Recall that $\tau(\tilde{P}) = \tau \wedge \tau_B$. W.l.o.g., assume that τ is transformed to $\exists \Sigma \,\text{.}\, (\tau_\Sigma \wedge \bigwedge \Sigma)$ for a finite set of fresh Boolean variables Σ that only appear negatively in τ_Σ. We refer to Σ as *assumptions* following SAT terminology [14]. Dropping some assumptions from $\bigwedge \Sigma$ results in an abstract transition relation, *i.e.*, $\exists \Sigma \,\text{.}\, \left(\tau_\Sigma \wedge \bigwedge \hat{\Sigma}\right)$ is an abstraction of τ for $\hat{\Sigma} \subseteq \Sigma$, denoted $\hat{\tau}(\hat{\Sigma})$. Note that $\hat{\tau}(\hat{\Sigma}) = \tau_\Sigma[\hat{\Sigma} \leftarrow \top, \Sigma \setminus \hat{\Sigma} \leftarrow \bot]$. The only abstractions of $\tilde{P}$ we consider are the ones which abstract τ and keep τ_B unchanged. That is, every abstraction $\hat{P}$ of $\tilde{P}$ is such that $\tilde{P} \preceq_{id} \hat{P}$ with $\tau(\hat{P}) = \hat{\tau}(\hat{\Sigma}) \wedge \tau_B$ for some $\hat{\Sigma} \subseteq \Sigma$. Moreover, a subset $\hat{\Sigma}$ of Σ induces an abstraction of $\tilde{P}$, denoted $\tilde{P}(\hat{\Sigma})$.

Under-approximations. An under-approximation is induced by a subset of assumptions $\hat{\Sigma} \subseteq \Sigma$, which identifies the abstraction $\tilde{P}(\hat{\Sigma})$, and a mapping $bvals : B \to \mathbb{N}$ from B to natural numbers, which bounds the number of iterations of every loop in $\tilde{P}$. The under-approximation, denoted $U(\hat{\Sigma}, bvals)$, satisfies $U(\hat{\Sigma}, bvals) \prec_{id} \tilde{P}(\hat{\Sigma})$, with $\tau(U(\hat{\Sigma}, bvals)) = \hat{\tau}(\hat{\Sigma}) \wedge \tau_B(bvals)$ where $\tau_B(bvals)$ is obtained from τ_B by strengthening all transitions with $\bigwedge_{b \in B} b \leq bvals(b)$.

Solve. We implement Solve (see Fig. 5) by transforming the decidable under-approximation U, after restricting by the invariants to $U_{\mathcal{I}}$, to Horn-SMT [21] (the input format of μZ) and passing the result to μZ. Note that this intentionally limits the power of μZ to solve only decidable problems. In Section 6, we compare Spacer with unrestricted μZ.

```
ExtractInvsImpl($\mathcal{C}$, $\{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi}$, $\{\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi}$)
begin
1    $M := \emptyset$, $X := \{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi}$, $Y := \{\neg\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi}$
2    $T := X$
3    while ($S :=$ Mus($\mathcal{C}, T, Y$)) $\neq \emptyset$ do
4        $M := M \cup S$, $Y := Y \setminus M$
5        $X := \{\mathcal{A}_{\ell,\varphi} \mid \neg\mathcal{B}_{\ell,\varphi} \in Y\}$
6        $T := X \cup M$
7    return $X$
```

```
Mus($\mathcal{C}, T, V$)
begin
8    $R := \emptyset$
9    while Sat($\mathcal{C}, T \cup R$) do
10       $m :=$ GetModel($\mathcal{C}, T \cup R$)
11       $R := R \cup \{v \in V \mid m(\neg v)\}$
12   return $R$
```

Fig. 7: Our implementation of ExtractInvs of Fig. 5.

We implement the routines of Spacer in Fig. 5 by maintaining a set of constraints $\mathcal{C}$ as shown in Fig. 6. Initially, $\mathcal{C}$ is *Global*. *Trans* encodes the transition relation of $\tilde{P}$, using fresh Boolean variables for transitions and locations ($E_{i,j}$, N_i, respectively) enforcing that a location is reachable only via one of its (incoming) edges. Choosing an abstract or concrete transition relation is done by adding a subset of Σ as additional constraints. *Invars* encodes currently known invariants. They approximate the reachable states by adding constraints for every invariant at a location in terms of current-state variables (3) and next-state variables (4). The antecedent in (3) specifies that at least one transition from ℓ_i has been taken implying that the current location is ℓ_i and the antecedent in (4) specifies that the next location is ℓ_i.

$\mathcal{C}$ is modified by each routine as needed by adding and retracting some of the *Local* constraints (see Fig. 6) as discussed below.

For a set of *assumption literals* $\mathcal{A}$, let Sat($\mathcal{C}, \mathcal{A}$) be a function that checks whether $\mathcal{C} \cup \mathcal{A}$ is satisfiable, and if not, returns an *unsat core* $\hat{\mathcal{A}} \subseteq \mathcal{A}$ such that $\mathcal{C} \cup \hat{\mathcal{A}}$ is unsatisfiable.

In the rest of the section, we assume that π is a safety proof of $U_{\mathcal{I}}(\hat{\Sigma}, \mathit{bvals})$.

InitU. The initial under-approximation is $U(\Sigma, \lambda b \in B \centerdot 0)$.

ExtractInvs is implemented by ExtractInvsImpl shown in Fig. 7. It extracts a *Maximal Inductive Subset* (MIS) of the lemmas in π w.r.t. the concrete transition relation $\tau \wedge \tau_B$ of $\tilde{P}$. First, the constraints *Concrete* in Fig. 6 are added to $\mathcal{C}$, including all of Σ. Second, the constraints *Lemmas* in Fig. 6 are added to $\mathcal{C}$, where fresh Boolean variables $\mathcal{A}_{\ell,\varphi}$ and $\mathcal{B}_{\ell,\varphi}$ are used to mark every lemma φ at every location $\ell \in L$. This encodes the negation of the inductiveness condition of a safety proof (see Def. 2).

The MIS of π corresponds to the *maximal* subset $I \subseteq \{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi}$ such that $\mathcal{C} \cup I \cup \{\neg\mathcal{B}_{\ell,\varphi} \mid \mathcal{A}_{\ell,\varphi} \notin I\}$ is unsatisfiable. I is computed by ExtractInvsImpl in Fig. 7. Each iteration of ExtractInvsImpl computes a *Minimal Unsatisfiable Subset* (MUS) to identify (a minimal set of) more non-inductive lemmas (lines 3–6). M, on line 4, indicates the cumulative set of non-inductive lemmas and X, on line 5, indicates all the other lemmas. Mus($\mathcal{C}, T, V$) in Fig. 7 iteratively computes a minimal subset, R, of V such that $\mathcal{C} \cup T \cup R$ is unsatisfiable.

Abstract finds a $\hat{\Sigma}_1 \subseteq \Sigma$ such that $U_{\mathcal{I}}(\hat{\Sigma}_1, \mathit{bvals})$ is safe with proof π. The constraints *Lemmas* in Fig. 6 are added to $\mathcal{C}$ to encode the negation of the conditions in Definition 2. Then, the constraints in *Bound Vals* in Fig. 6 are added to $\mathcal{C}$ to encode the under-approximation. This reduces the check for π

to be a safety proof to that of unsatisfiability of a formula. Finally, $\textsc{Sat}(\mathcal{C}, \Sigma \cup \{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi} \cup \{\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi})$ is invoked. As $\mathcal{C}$ is unsatisfiable assuming Σ and using all the lemmas (since π proves $U_{\mathcal{I}}(\hat{\Sigma}, \mathit{bvals})$), it returns an unsat core. Projecting the core onto Σ gives us $\hat{\Sigma}_1 \subseteq \Sigma$ which identifies the new abstraction and, together with *bvals*, the corresponding new under-approximation. The minimality of $\hat{\Sigma}_1$ depends on the algorithm for extracting an unsat core, which is part of the SMT engine of Z3 in our case. In practice, we use a *Minimal Unsatisfiable Subset* (MUS) algorithm to find a minimal $\hat{\Sigma}_1$. As we treat $\{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi}$ and $\{\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi}$ as assumption literals, this also corresponds to using only the necessary lemmas during abstraction.

NextU. Given the current valuation *bvals* and the new abstraction $\hat{\Sigma}$, this routine returns $U(\hat{\Sigma}, \lambda b \in B \centerdot \mathit{bvals}(b) + 1)$.

Refine and IsFeasible. Let $U_{\mathcal{I}}(\hat{\Sigma}, \mathit{bvals})$ be unsafe with a counterexample $\mathscr{C}$. We create a new set of constraints $\mathcal{C}_{\mathscr{C}}$ corresponding to the unrolling of $\tau_\Sigma \wedge \tau_B$ along the control path of $\mathscr{C}$ and check $\textsc{Sat}(\mathcal{C}_{\mathscr{C}}, \Sigma)$. If the path is feasible in $\tilde{P}$, we find a counterexample to safety in $\tilde{P}$. Otherwise, we obtain an unsat core $\hat{\Sigma}_1 \subseteq \Sigma$ and refine the abstraction to $\hat{\Sigma} \cup \hat{\Sigma}_1$. The under-approximation is refined accordingly with the same *bvals*.

We conclude the section with a discussion of the implementation choices. NextU is implemented by incrementing all bounding variables uniformly. An alternative is to increment the bounds only for the loops whose invariants are not inductive (e.g., [2,26]). However, we leave the exploration of such strategies for future. Our use of μZ is sub-optimal since each call to Solve requires constructing a new Horn-SMT problem. This incurs an unnecessary pre-processing overhead that can be eliminated by a tighter integration with μZ. For Abstract and ExtractInvs, we use a single SMT-context with a single copy of the transition relation of the program (without unrolling it). The context is preserved across iterations of Spacer. Constraints specific to an iteration are added and retracted using the incremental solving API of Z3. This is vital for performance. For Refine and IsFeasible, we unroll the transition relation of the program along the control path of the counterexample trace returned by μZ. We experimented with an alternative implementation that instead validates each individual step of the counterexample using the same global context as Abstract. While this made each refinement step faster, it increased the number of refinements, becoming inefficient overall.

6 Experiments

We implemented Spacer in Python using Z3 v4.3.1 (with a few modifications to Z3 API[4]). The implementation and complete experimental results are available at `http://www.cs.cmu.edu/~akomurav/projects/spacer/home.html`.

Benchmarks. We evaluated Spacer on the benchmarks from the *systemc*, *product-lines*, *device-drivers-64* and *control-flow-integers* categories of SV-COMP'13.

[4] Our changes are being incorporated into Z3, and will be available in future versions.

Other categories require bit-vector and heap reasoning that are not supported by SPACER. We used the front-end of UFO [3] to convert the benchmarks from C to the Horn-SMT format of μZ.

Overall, there are 1,990 benchmarks (1,591 SAFE, and 399 UNSAFE); 1,382 are decided by the UFO front-end that uses common compiler optimizations to reduce the problem. This left 608 benchmarks (231 SAFE, and 377 UNSAFE).

For the UNSAFE benchmarks, 369 cases are solved by both μZ and SPACER; in the remaining 8 benchmarks, 6 are solved by neither tool, and 2 are solved by μZ but not by SPACER. In summary, abstraction did not help for UNSAFE benchmarks and, in a few cases, it hurts significantly. Having said that, these benchmarks are easy with SPACER needing at most 3 minutes each, for the 369 cases it solves.

For the SAFE benchmarks, 176 are solved in under a minute by both tools. For them, the difference between SPACER and μZ is not significant to be meaningful. Of the remaining 55 hard benchmarks 42 are solved by either μZ, SPACER or both with a time limit of 15 minutes and 2GB of memory. The rest remain unsolved. All experiments were done on an Intel® Core$^{\text{TM}}$2 Quad CPU of 2.83GHz and 4GB of RAM.

Results. Table 1 shows the experimental results on the 42 solved benchmarks. The t columns under μZ and SPACER show the running times in seconds with 'TO' indicating a time-out and a 'MO' indicating a mem-out. The best times are highlighted in bold. Overall, abstraction helps for *hard* benchmarks. Furthermore, in `elev_13_22`, `elev_13_29` and `elev_13_30`, SPACER is successful even though μZ runs out of memory, showing a clear advantage of abstraction. Note that `gcnr`, under *misc*, in the table is the example from Fig. 1(a).

The B column in the table shows the final values of the loop bounding variables under the mapping *bvals*, *i.e.*, the maximum number of loop iterations (of any loop) that was necessary for the final safety proof. Surprisingly, they are very small in many of the hard instances in *systemc* and *product-lines* categories.

Columns a_f and a_m show the sizes of the final and maximal abstractions, respectively, measured in terms of the number of the original constraints used. Note that this only corresponds to the *syntactic* abstraction (see Section 4). The final abstraction done by SPACER is very aggressive. Many constraints are irrelevant with often, more than 50% of the original constraints abstracted away. Note that this is in addition to the aggressive property-independent abstraction done by the UFO front-end. Finally, the difference between a_f and a_m is insignificant in all of the benchmarks.

Another approach to ABSTRACT is to restrict abstraction to state-variables by making assignments to some next-state variables non-deterministic, as done by Vizel et al. [29] in a similar context. This was especially effective for *ssh* and *ssh-simplified* categories – see the entries marked with '*' under column t.

An alternative implementation of REFINE is to concretize the under-approximation (by refining $\hat{\Sigma}$ to Σ) whenever a spurious counterexample is found. This is analogous to Proof-Based Abstraction (PBA) [27] in hardware verification. Run-time for PBA and the corresponding final values of the bounding variables are shown

Benchmark	μZ	Spacer					
	t (sec)	t (sec)	B	a_f (%)	a_m (%)	t_p (sec)	B_p
systemc							
`pipeline`	224	**120**	4	33	33	249	4
`tk_ring_06`	64	**48**	2	59	59	65	2
`tk_ring_07`	69	120	2	59	59	†**67**	2
`tk_ring_08`	232	**158**	2	57	57	358	2
`tk_ring_09`	817	**241**	2	59	59	266	2
`mem_slave_1`	536	**430**	3	24	34	483	2
`toy`	TO	822	4	32	44	†**460**	4
`pc_sfifo_2`	**73**	137	2	41	41	TO	–
product-lines							
`elev_13_21`	TO	**174**	2	7	7	TO	–
`elev_13_22`	MO	**336**	2	9	9	624	4
`elev_13_23`	TO	**309**	4	6	14	TO	–
`elev_13_24`	TO	**591**	4	9	9	TO	–
`elev_13_29`	MO	**190**	2	6	10	TO	–
`elev_13_30`	MO	**484**	3	11	13	TO	–
`elev_13_31`	TO	**349**	4	8	17	TO	–
`elev_13_32`	TO	**700**	4	9	9	TO	–
`elev_1_21`	**102**	136	11	61	61	161	11
`elev_1_23`	**101**	276	11	61	61	†140	11
`elev_1_29`	92	199	11	61	62	†**77**	11
`elev_1_31`	127	135	11	62	62	†**92**	11
`elev_2_29`	**18**	112	11	56	56	†26	11
`elev_2_31`	**16**	91	11	57	57	†22	11
ssh							
`s3_clnt_3`	109	∗90	12	13	13	**73**	12
`s3_srvr_1`	187	**43**	9	18	18	661	25
`s3_srvr_2`	587	∗**207**	14	3	7	446	15
`s3_srvr_8`	99	**49**	13	18	18	TO	–
`s3_srvr_10`	83	**24**	9	17	17	412	21
`s3_srvr_13`	355	∗**298**	15	8	8	461	15
`s3_clnt_2`	**34**	∗124	13	13	13	†95	13
`s3_srvr_12`	**21**	∗64	13	8	8	54	13
`s3_srvr_14`	**37**	∗141	17	8	8	†91	17
`s3_srvr_6`	**98**	TO	–	–	–	†300	25
`s3_srvr_11`	**270**	896	15	14	18	831	13
`s3_srvr_15`	**309**	TO	–	–	–	TO	–
`s3_srvr_16`	**156**	∗263	21	8	8	†159	21
ssh-simplified							
`s3_srvr_3`	171	130	11	21	21	**116**	12
`s3_clnt_3`	**50**	∗139	12	17	22	†104	13
`s3_clnt_4`	**15**	∗76	12	22	22	56	13
`s3_clnt_2`	**138**	509	13	26	26	†145	13
`s3_srvr_2`	**148**	232	12	16	23	222	15
`s3_srvr_6`	**91**	TO	–	–	–	†272	25
`s3_srvr_7`	**253**	398	10	20	26	764	10
misc							
`gcnr`	TO	56	26	81	95	**50**	25

Table 1: Comparison of μZ and Spacer. t and t_p are running times in seconds; B and B_p are the final values of the bounding variables; a_f and a_m are the fractions of assumption variables in the final and maximal abstractions, respectively.

in columns t_p and B_p of Table 1, respectively. While this results in more timeouts, it is significantly better in 14 cases (see the entries marked with '†' under column t_p), with 6 of them comparable to μZ and 2 (*viz.*, `toy` and `elev_1_31`) significantly better than μZ.

We conclude this section by comparing our results with UFO [3] — the winner of the 4 categories at SV-COMP'13. The competition version of UFO runs several engines in parallel, including engines based on Abstract Interpretation, Predicate Abstraction and 2BMC with interpolation. UFO outperforms Spacer and μZ in *ssh* and *product-lines* categories by an order of magnitude. They are difficult for 2BMC, but easy for Abstract Interpretation and Predicate Abstraction, respectively. Even so, note that Spacer finds really small abstractions for these categories upon termination. However, in the *systemc* category both Spacer and μZ perform better than UFO by solving hard instances (e.g., `tk_ring_08` and `tk_ring_09`) that are not solved by any tool in the competition. Moreover, Spacer is faster than μZ. Thus, while Spacer itself is not the best tool for all benchmarks, it is a valuable addition to the state-of-the-art verification engines.

7 Related work

There is a large body of work on 2BMC approaches both in hardware and software verification. In this section, we briefly survey the most related work.

The two most prominent approaches to 2BMC combine BMC with interpolation (e.g., [2,25,26]) or with inductive generalization (e.g., [9,10,13,21]). Although our implementation of SPACER is based on inductive generalization (the engine of μZ), it can be implemented on top of an interpolation-based engine as well.

Proof-based Abstraction (PBA) was first introduced in hardware verification to leverage the power of SAT-solvers to focus on relevant facts [19,27]. Over the years, it has been combined with CEGAR [4,5], interpolation [5,24], and PDR [22]. To the best of our knowledge, SPACER is the first application of PBA to software verification.

The work of Vizel et al. [29], in hardware verification, that extends PDR with abstraction is closest to ours. However, SPACER is not tightly coupled with PDR, which makes it more general, but possibly, less efficient. Nonetheless, SPACER allows for a rich space of abstractions, whereas Vizel et al. limit themselves to state variable abstraction.

Finally, UFO [2,1] also combines abstraction with 2BMC, but in an orthogonal way. In UFO, abstraction is used to guess the depth of unrolling (plus useful invariants), BMC to detect counterexamples, and interpolation to synthesize safe inductive invariants. While UFO performs well on many competition benchmarks, combining it with SPACER will benefit on the hard ones.

8 Conclusion

In this paper, we present an algorithm, SPACER, that combines Proof-Based Abstraction (PBA) with CounterExample Guided Abstraction Refinement (CEGAR) for verifying safety properties of sequential programs. To our knowledge, this is the first application of PBA to software verification. Our abstraction technique combines localization with invariants about the program. It is interesting to explore alternatives for such a *semantic* abstraction.

While our presentation is restricted to non-recursive sequential programs, the technique can be adapted to solving the more general Horn Clause Satisfiability problem and extended to verifying recursive and concurrent programs [16].

We have implemented SPACER in Python using Z3 and its GPDR engine μZ. The current implementation is an early prototype. It is not heavily optimized and is not tightly integrated with μZ. Nonetheless, the experimental results on 4 categories of the 2nd Software Verification Competition show that SPACER improves on both μZ and the state-of-the-art.

Acknowledgment. We thank Nikolaj Bjørner for many helpful discussions and help with μZ and the anonymous reviewers for insightful comments.

References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig Interpretation. In *SAS*, 2012.

2. A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. In *TACAS*, 2012.
3. A. Albarghouthi, A. Gurfinkel, Y. Li, S. Chaki, and M. Chechik. UFO: Verification with Interpolants and Abstract Interpretation - (Competition Contribution). In *TACAS*, 2013.
4. N. Amla and K. L. McMillan. A Hybrid of Counterexample-Based and Proof-Based Abstraction. In *FMCAD*, pages 260–274, 2004.
5. N. Amla and K. L. McMillan. Combining Abstraction Refinement and SAT-Based Model Checking. In *TACAS*, 2007.
6. D. Babic and A. J. Hu. Structural Abstraction of Software Verification Conditions. In *CAV*, 2007.
7. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. *SIGPLAN Not.*, 36(5):203–213, 2001.
8. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:117–148, 2003.
9. A. R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, 2011.
10. A. Cimatti and A. Griggio. Software Model Checking via IC3. In *CAV*, 2012.
11. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, 2000.
12. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
13. N. Eén, A. Mishchenko, and R. K. Brayton. Efficient Implementation of Property Directed Reachability. In *FMCAD*, pages 125–134, 2011.
14. N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, 2003.
15. C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME*, pages 500–517, 2001.
16. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing Software Verifiers from Proof Rules. In *PLDI*, pages 405–416, 2012.
17. A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, January 2012.
18. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *TACAS*, 2008.
19. A. Gupta, M. K. Ganai, Z. Yang, and P. Ashar. Iterative Abstraction using SAT-based BMC with Proof Analysis. In *ICCAD*, pages 416–423, 2003.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. *SIGPLAN Not.*, 37(1):58–70, 2002.
21. K. Hoder and N. Bjørner. Generalized Property Directed Reachability. In *SAT*, 2012.
22. A. Ivrii, A. Matsliah, H. Mony, and J. Baumgartner. IC3-Guided Abstraction. In *FMCAD*, 2012.
23. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *TACAS*, 2006.
24. B. Li and F. Somenzi. Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking. In *TACAS*, 2006.
25. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, 2003.
26. K. L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, 2006.
27. K. L. McMillan and N. Amla. Automatic Abstraction without Counterexamples. In *TACAS*, 2003.
28. K. L. McMillan and A. Rybalchenko. Solving Constrained Horn Clauses using Interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.
29. Y. Vizel, O. Grumberg, and S. Shoham. Lazy Abstraction and SAT-based Reachability for Hardware Model Checking. In *FMCAD*, 2012.