Journal of Automated Reasoning (2020) 64:1051–1091
https://doi.org/10.1007/s10817-019-09535-x

Efficient Strategies for CEGAR-Based Model Checking

Ákos Hajdu[1,2] · **Zoltán Micskei**[1]

Received: 28 May 2018 / Accepted: 21 October 2019 / Published online: 11 November 2019

Abstract
Automated formal verification is often based on the Counterexample-Guided Abstraction Refinement (CEGAR) approach. Many variants of CEGAR have been developed over the years as different problem domains usually require different strategies for efficient verification. This has lead to generic and configurable CEGAR frameworks, which can incorporate various algorithms. In our paper we propose six novel improvements to different aspects of the CEGAR approach, including both abstraction and refinement. We implement our new contributions in the THETA framework allowing us to compare them with state-of-the-art algorithms. We conduct an experiment on a diverse set of models to address research questions related to the effectiveness and efficiency of our new strategies. Results show that our new contributions perform well in general. Moreover, we highlight certain cases where performance could not be increased or where a remarkable improvement is achieved.

Keywords Formal verification · Abstraction · CEGAR · Experimental evaluation

1 Introduction

Counterexample-Guided Abstraction Refinement (CEGAR) [31] is a widely used technique for the automated formal verification of different systems, including both software [15,39,53,54,56] and hardware [31,34]. CEGAR works by iteratively constructing and refining abstractions until a proper precision is reached. It starts with computing an abstraction of the system with respect to some abstract domain and a given initial—usually coarse —precision. The abstraction over-approximates [32] the possible behaviors (i.e., the state space) of the original system. Thus, if no erroneous behavior can be found in the abstract state space then the original system is also safe. However, abstract counterexamples corresponding to erroneous behaviors must be checked whether they are reproducible (feasible) in the original system. A feasible counterexample indicates that the original system is unsafe. Otherwise, the

✉ Ákos Hajdu
hajdua@mit.bme.hu

Zoltán Micskei
micskeiz@mit.bme.hu

[1] Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

[2] MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

counterexample is spurious and it is excluded in the next iteration by adjusting the precision to build a finer abstraction. The algorithm iterates between abstraction and refinement until the abstract system is proved safe, or a feasible counterexample is found.

CEGAR is a generic approach with many variants developed over the past two decades, improving both applicability and performance. There are different abstract domains, including predicates [41] and explicit values [15] and various refinement strategies, including ones based on interpolation [55,63]. However, there is usually no single best variant: different algorithms are suitable for different verification tasks [43]. Therefore, generic frameworks are also emerging, which provide configurability [14], combinations of different strategies for abstraction and refinement [2,45], and support for various kind of models [49,60].

Contributions In our paper, we make the following novel contributions. (1) We propose six new strategies improving various aspects of the CEGAR algorithm, including abstraction and refinement as well. (2) We conduct an experimental evaluation on models from diverse domains, including both software and hardware.

Algorithmic improvements We propose various novel improvements and variations of existing strategies to different aspects of the CEGAR approach.

- We generalize explicit-value abstraction to be able to enumerate a predefined, configurable number of successor states, improving its precision while still avoiding state space explosion.
- We adapted a search strategy to the context of CEGAR that estimates the distance from the erroneous state in the abstract state space based on the structure of the original system.
- We study different splitting techniques applied to complex predicates in order to generalize the result of refinement.
- We introduce an interpolation strategy based on backward reachability, which traces back the reason of infeasibility to the earliest point.
- We describe an approach for refinement based on multiple counterexamples, which provides better quality refinement since more information is available.
- We present combinations of different interpolation strategies that enable selection from different refinements.

We implement our new contributions in Theta [60], an open source, generic and configurable framework. In this paper, we illustrate our new approaches on programs (control flow automata), but most of them can be generalized to other formalisms supported in Theta, such as hardware (transition systems). Theta already includes many of the state-of-the-art algorithms, which allows us to use them as a baseline to evaluate our new contributions.

Experimental evaluation We conduct an experimental evaluation on roughly 800 input models from diverse sources, including the Competition on Software Verification [9], the Hardware Model Checking Competition [25] and industrial PLC software from CERN [40]. The advantage of using a diverse set of models is that we can identify the most suitable application areas. Furthermore, we compare lower lever parameters of CEGAR as opposed to most experiments in the literature [11,19,36,37], where different algorithms or tools are compared. We formulate and address a research question related to the effectiveness and efficiency of each of our contributions.

The results show that our new improvements perform well in general compared to the state of the art. In some cases the differences are subtle, but there are certain subcategories of the models for which a new algorithm yields a remarkable improvement. We also show negative results, i.e., models where a new algorithm is less effective—we believe that such results are also important: in a different domain these algorithms can still be successful.

Outline of the paper The rest of the paper is organized as follows. We first introduce the preliminaries of our work in Sect. 2. Then we describe our new contributions in Sect. 3 and evaluate them in Sect. 4. Finally, we present related work in Sect. 5 and conclude our paper in Sect. 6.

2 Background

This section introduces the preliminaries of our paper. First, we present control flow automata as the modeling formalism used in our work (Sect. 2.1). Then we describe the abstraction and CEGAR-based framework (Sect. 2.2), in which we formalize our new algorithms (Sect. 3).

We use the following notations from first-order logic (FOL) throughout our paper. Given a set of variables $V = \{v_1, v_2, \ldots\}$ let $V' = \{v_1', v_2', \ldots\}$ and $V^{\langle i \rangle} = \{v_1^{\langle i \rangle}, v_2^{\langle i \rangle} \ldots\}$ represent the *primed* and *indexed* version of the variables. We use V' to refer to successor states and $V^{\langle i \rangle}$ for paths. Given an expression φ over $V \cup V'$, let $\varphi^{\langle i \rangle}$ denote the indexed expression obtained by replacing V and V' with $V^{\langle i \rangle}$ and $V^{\langle i+1 \rangle}$ respectively in φ. For example, $(x < y)^{\langle 2 \rangle} \equiv x^{\langle 2 \rangle} < y^{\langle 2 \rangle}$ and $(x' = x + 1)^{\langle 2 \rangle} \equiv x^{\langle 3 \rangle} = x^{\langle 2 \rangle} + 1$. Given an expression φ let $\mathsf{var}(\varphi)$ denote the set of variables appearing in φ, e.g., $\mathsf{var}(x < y + 2) = \{x, y\}$.

2.1 Control Flow Automata

In our work we describe programs using *control flow automata* (CFA) [13], a formalism based on FOL variables and expressions.

Definition 1 (*Control flow automata*) A control flow automaton is a tuple $CFA = (V, L, l_0, E)$ where

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of variables with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$,
- L is a set of program *locations* modeling the program counter,
- $l_0 \in L$ is the *initial* program location,
- $E \subseteq L \times Ops \times L$ is a set of directed *edges* representing the *operations* that are executed when control flows from the source location to the target.

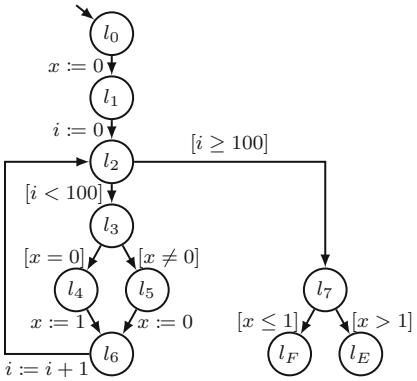
Operations $op \in Ops$ are either *assignments* or *assumptions* over the variables of the CFA. Assignments have the form $v{:=}\varphi$, where $v \in V$, φ is an expression of type D_v and $\mathsf{var}(\varphi) \subseteq V$. Assumptions have the form $[\psi]$, where ψ is a predicate with $\mathsf{var}(\psi) \subseteq V$. An operation $op \in Ops$ can also be regarded as a transition formula $\mathsf{tran}(op)$ over $V \cup V'$ defining its semantics. For an assignment operation, the transition formula is defined as $\mathsf{tran}(v{:=}\varphi) \equiv v' = \varphi \wedge \bigwedge_{v_i \in V \setminus \{v\}} v_i' = v_i$ and for an assume operation it is $\mathsf{tran}([\psi]) \equiv \psi \wedge \bigwedge_{v \in V} v' = v$. In other words, assignments change a single variable and assumptions check a condition.[1] By abusing the notation, we allow operations $op \in Ops$ to appear as FOL expressions by automatically replacing them with their semantics, i.e., $\mathsf{tran}(op)$.

A *concrete data state* $c \in D_{v_1} \times \ldots \times D_{v_n}$ is a (many sorted) interpretation that assigns a value $c(v) = d \in D_v$ to each variable $v \in V$ of its domain D_v. States with a prime (c') or an index ($c^{\langle i \rangle}$) assign values to V' or $V^{\langle i \rangle}$ respectively. A *concrete state* (l, c) is a pair of a location $l \in L$ and a concrete data state. The set of initial states is $\{(l, c) \mid l = l_0\}$

[1] Equality constraints do not appear in the implementation, but a single static assignment form is used where a new symbol is only introduced when a variable is assigned to.

```
1 int x = 0;
2 int i = 0;
3 while (i < 100) {
4   if (x == 0) x = 1;
5   else x = 0;
6   i++;
7 }
8 assert(x <= 1);
```

(a) Example program with various elements of structured programming.



(b) CFA representation of the program. The distinguished location l_E corresponds to an assertion failure.

Fig. 1 A simple program and its corresponding CFA, illustrating the correspondence between elements of structured programming (sequence, selection, repetition) and the structure of the CFA

and a transition exists between states (l, c) and (l', c') if an edge $(l, op, l') \in E$ exists with $(c, c') \models op$.

A *concrete path* is a finite, alternating sequence of concrete states and operations $\sigma = ((l_1, c_1), op_1, \ldots, op_{n-1}, (l_n, c_n))$ if $(l_i, op_i, l_{i+1}) \in E$ for every $1 \leq i < n$ and $(c_1^{\langle 1 \rangle}, c_2^{\langle 2 \rangle}, \ldots, c_n^{\langle n \rangle}) \models \bigwedge_{1 \leq i < n} op_i^{\langle i \rangle}$, i.e., there is a sequence of edges starting from the initial location and the interpretations satisfy the semantics of the operations. A concrete state (l, c) is *reachable* if a path $\sigma = ((l_1, c_1), op_1, \ldots, op_{n-1}, (l_n, c_n))$ exists with $l = l_n$ and $c = c_n$ for some n.

A *verification task* is a pair (CFA, l_E) of a CFA and a distinguished error location $l_E \in L$. A verification task is *safe* if (l_E, c) is not reachable for any c, otherwise it is *unsafe*.

Example A simple program and its corresponding CFA can be seen in Fig. 1. Basic elements of structured programming (sequence, selection, repetition) are represented by the structure of the automaton. The assertion in line 8 is mapped as a selection at location l_7. If the assertion holds, the program ends normally in the final location l_F.[2] Otherwise, failure is indicated with the error location l_E.

2.2 Counterexample-Guided Abstraction Refinement (CEGAR)

Counterexample-Guided Abstraction Refinement (CEGAR) [31] is a verification algorithm that automatically constructs and refines abstractions for a given model (Fig. 2). First, an *abstraction* algorithm computes an *abstract reachability graph* (ARG) [12] over some abstract domain with respect to a given initial precision. The ARG is an over-approximation of the original state space, therefore if no abstract state with the error location is reachable then the original model is also safe [32]. However, if an abstract counterexample (a path to an abstract state with the error location) is found, the *refinement* algorithm checks whether it

[2] Note, that currently we are not considering termination, i.e., the final location l_F does not carry any special meaning.

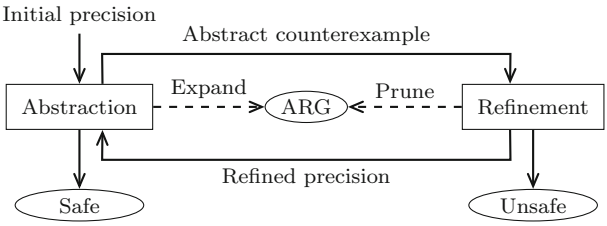


Fig. 2 Overview of a generic Counterexample-Guided Abstraction Refinement (CEGAR) algorithm

is feasible in the original model. A feasible counterexample indicates that the original model is unsafe. Otherwise, the counterexample is spurious, the precision is adjusted and the ARG is pruned so that the same counterexample is not encountered in the next iteration of the abstraction.

2.2.1 Abstraction

We define abstraction based on an *abstract domain* D, a set of *precisions* Π and a *transfer function* T [13].

Definition 2 (*Abstract domain*) An abstract domain is a tuple $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$ where

- S is a (possibly infinite) lattice of abstract states,
- $\top \in S$ is the top element,
- $\bot \in S$ is the bottom element,
- $\sqsubseteq \subseteq S \times S$ is a partial order conforming to the lattice and
- $\mathsf{expr}\colon S \mapsto \mathit{FOL}$ is the expression function that maps an abstract state to its meaning (the concrete data states it represents) using a FOL formula.

By abusing the notation we will allow abstract states $s \in S$ to appear as FOL expressions by automatically replacing them with their meaning, i.e., $\mathsf{expr}(s)$.

Elements $\pi \in \Pi$ in the set of precisions define the current precision of the abstraction. The transfer function $T\colon S \times \mathit{Ops} \times \Pi \mapsto 2^S$ calculates the successors of an abstract state with respect to an operation and a target precision.

In the following, we introduce two domains, namely predicate abstraction and explicit-value abstraction, and their extension with the locations of the CFA.

Predicate abstraction In *Boolean predicate abstraction* [5,41] an abstract state $s \in S$ is a Boolean combination of FOL predicates. The top and bottom elements are $\top \equiv \mathit{true}$ and $\bot \equiv \mathit{false}$ respectively. The partial order corresponds to implication, i.e., $s_1 \sqsubseteq s_2$ if $s_1 \Rightarrow s_2$ for $s_1, s_2 \in S$. The expression function is the identity function as abstract states are formulas themselves, i.e., $\mathsf{expr}(s) = s$.

A precision $\pi \in \Pi$ is a set of FOL predicates that are currently tracked by the algorithm. The result of the transfer function $T(s, op, \pi)$ is the strongest Boolean combination of predicates in the precision that is entailed by the source state s and the operation op. This can be calculated by assigning a fresh propositional variable v_i to each predicate $p_i \in \pi$ and enumerating all satisfying assignments of the variables v_i in the formula $s \wedge op \wedge \bigwedge_{p_i \in \pi}(v_i \leftrightarrow p_i')$. For each assignment, a conjunction of predicates is formed by taking predicates with positive variables and the negations of predicates with negative variables. The disjunction of all such conjunctions is the successor state s'.

In *Cartesian predicate abstraction* [5] an abstract state $s \in S$ is a conjunction of FOL predicates. Only the transfer function is defined differently than in Boolean predicate abstraction. The transfer function yields the strongest conjunction of predicates from the precision π that is entailed by the source state s and the operation op, i.e., $T(s, op, \pi) = \bigwedge_{p_i \in \pi} \{p_i \mid s \wedge op \Rightarrow p_i'\} \wedge \bigwedge_{p_i \in \pi} \{\neg p_i \mid s \wedge op \Rightarrow \neg p_i'\}$.

Note, that when the precision is empty ($\pi = \emptyset$) the transfer function reduces to a feasibility checking of the formula $s \wedge op$, resulting in *true* or *false* (for Boolean and Cartesian predicate abstraction as well).

We represent abstract states (in both kind of abstractions) as SMT formulas. However, a possible optimization would be to use binary decision diagrams (BDDs) for compact representation of states and cheaper coverage checks [28].

Explicit-value abstraction In explicit-value abstraction [15] an abstract state $s \in S$ is an abstract variable assignment, mapping each variable $v \in V$ to an element from its domain extended with top and bottom values, i.e., $D_v \cup \{\top_{d_v}, \bot_{d_v}\}$. The top element $\top$ with $\top(v) = \top_{v_d}$ holds no specific value for any $v \in V$ (i.e., it represents an unknown value). The bottom element $\bot$ with $\bot(v) = \bot_{v_d}$ means that no assignment is possible for any $v \in V$. The partial order $\sqsubseteq$ is defined as $s_1 \sqsubseteq s_2$ if $s_1(v) = s_2(v)$ or $s_1(v) = \bot_{d_v}$ or $s_2(v) = \top_{v_d}$ for each $v \in V$. The expression function is $\mathsf{expr}(s) \equiv true$ if $s = \top$, $\mathsf{expr}(s) \equiv false$ if $s(v) = \bot_{d_v}$ for any $v \in V$, otherwise $\mathsf{expr}(s) \equiv \bigwedge_{v \in V, s(v) \neq \top_{d_v}} v = s(v)$.

A precision $\pi \in \Pi$ is a subset of the variables $\pi \subseteq V$ that is currently tracked by the analysis. The transfer function is given based on the *strongest post-operator* $\mathsf{sp}\colon S \times Ops \mapsto S$, defining the semantics of operations under abstract variable assignments. Given an abstract variable assignment $s \in S$ and an operation $op \in Ops$, let the abstract variable assignment $\hat{s} = \mathsf{sp}(s, op)$ denote the result of executing op from s.

If op is an assumption $[\psi]$ then for all $v \in V$

$$\hat{s}(v) = \begin{cases} \bot_{d_v} & \text{if } s(u) = \bot_{d_u} \text{ for any } u \in V \text{ or } \psi_{/s} \text{ evaluates to } false, \\ s(v) & \text{otherwise,} \end{cases} \tag{1}$$

where $\psi_{/s}$ denotes the expression obtained by substituting all variables in ψ with values from s, except top and bottom values.

Note, that if $[\psi]$ is only satisfiable with a single value for a variable v then the successor could be made more precise by setting $\hat{s}(v)$ to this value [15]. This could be implemented with heuristics[3] for a few simple cases (e.g., $[v = 1]$), but a general solution requires a solver. In our current paper we use a simple heuristic that can detect if an equality constraint has a variable on one side and a constant on the other (e.g., $[v = 1]$) and later we also present a general, configurable solution using a solver in Sect. 3.1.1.

If op is an assignment $w{:=}\varphi$ then for all $v \in V$

$$\hat{s}(v) = \begin{cases} \bot_{d_v} & \text{if } s(u) = \bot_{d_u} \text{ for any } u \in V, \\ s(v) & \text{if } v \neq w, \\ c & \text{if } v = w \text{ and } \varphi_{/s} \text{ evaluates to a literal } c, \\ \top_{d_v} & \text{otherwise.} \end{cases} \tag{2}$$

The transfer function $T(s, op, \pi) = s'$ is defined based on the strongest post-operator sp as follows. Let $\hat{s} = \mathsf{sp}(s, op)$, then $s'(v) = \hat{s}(v)$ if $v \in \pi$ and $s'(v) = \top_{d_v}$ otherwise, for each $v \in V$, i.e., variables not included in the precision are omitted.

Locations Locations of the CFA are usually tracked explicitly regardless of the abstract domain used [13]. Given an abstract domain $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$ (e.g., predicate or

[3] The original paper [15] does not exactly mention such heuristics.

explicit-value abstraction), let $D_L = (S_L, \bot_L, \sqsubseteq_L, \mathsf{expr}_L)$ denote its extension with locations.[4] Abstract states $S_L = L \times S$ are pairs of a location $l \in L$ and a state $s \in S$. The bottom element becomes a set $\bot_L = \{(l, \bot) \mid l \in L\}$ with each location and the bottom element $\bot$ of D. The partial order is defined as $(l_1, s_1) \sqsubseteq (l_2, s_2)$ if $l_1 = l_2$ and $s_1 \sqsubseteq s_2$. The expression function is $\mathsf{expr}_L \equiv \mathsf{expr}$, i.e., the location is not required in the expression as it is encoded in the structure of the CFA.

The precisions Π are also extended with a location, becoming a function $\Pi_L : L \mapsto \Pi$ that maps each location to its precision. Algorithms can be configured to use a *global* precision, which maps each location to the same precision, or a *local* precision, which can map different locations to different precisions.[5]

The extended transfer function $T_L : S_L \times \Pi_L \mapsto 2^{S_L}$ is defined as $T_L((l, s), \pi_L) = \{(l', s') \mid (l, op, l') \in E,\ s' \in T(s, op, \pi_L(l'))\}$, i.e., (l', s') is a successor of (l, s) if there is an edge between l and l' with op and s' is a successor of s with respect to the inner transfer function T and the precision assigned to l'.

Abstract Reachability Graph We represent the abstract state space using an *abstract reachability graph* (ARG) [12].

Definition 3 (*Abstract Reachability Graph*) An abstract reachability graph is a tuple $ARG = (N, E, C)$ where

- N is the set of *nodes*, each corresponding to an abstract state in some domain with locations D_L.
- $E \subseteq N \times Ops \times N$ is a set of directed *edges* labeled with operations. An edge $(l_1, s_1, op, l_2, s_2) \in E$ is present if (l_2, s_2) is a successor of (l_1, s_1) with op.
- $C \subseteq S \times S$ is the set of *covered-by edges*. A covered-by edge $(l_1, s_1, l_2, s_2) \in C$ is present if $(l_1, s_1) \sqsubseteq (l_2, s_2)$.

A node $(l, s) \in N$ is *expanded* if all of its successors are included in the ARG with respect to the transfer function; *covered* if it has an outgoing covered-by edge $(l, s, l', s') \in C$ for some $(l', s') \in N$; and *unsafe* if $l = l_E$. A node that is not expanded, covered or unsafe is called *unmarked*. An ARG is *unsafe* if there is at least one unsafe node and *complete* if no nodes are unmarked.

An *abstract path* $\sigma = ((l_1, s_1), op_1, (l_2, s_2), op_2, \ldots, op_{n-1}, (l_n, s_n))$ is an alternating sequence of abstract states and operations. An abstract path is *feasible* if a corresponding concrete path $((l_1, c_1), op_1, (l_2, c_2), op_2, \ldots, op_{n-1}, (l_n, c_n))$ exists, where each c_i is mapped to s_i, i.e., $c_i \models \mathsf{expr}(s_i)$. In practice, this can be decided by querying an SMT solver [20] with the formula[6] $s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge s_2^{\langle 2 \rangle} \wedge op_2^{\langle 2 \rangle} \wedge \ldots \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$. A satisfying assignment to this formula corresponds to a concrete path.

Abstraction algorithm Based on the concepts defined above, Algorithm 1 presents a basic procedure for abstraction (based on the CPA concept [13]). The input of the abstraction is a partially constructed ARG (with possibly unmarked states), an error location l_E, an abstract domain D_L with locations, a current precision π_L and a transfer function T_L. In the first

[4] Note, that technically D_L is not a domain as for example it has no top element. While it is possible to define a generic product domain with locations [13], we rather use locations as a "wrapper" to make our presentation simpler.

[5] In lazy abstraction [47] the precision can be different even for different instances of the same location in the ARG.

[6] In software model checking s_1 is usually the top element because the program starts with all variables uninitialized. However, in a more general setting, transition systems can have an arbitrary formula describing the initial states [43].

iteration, the ARG only contains the initial state $S_0 = \{(l_0, \top)\}$ and the precision π_L is usually empty, i.e., no predicates or variables are tracked.

Algorithm 1: Abstraction algorithm.

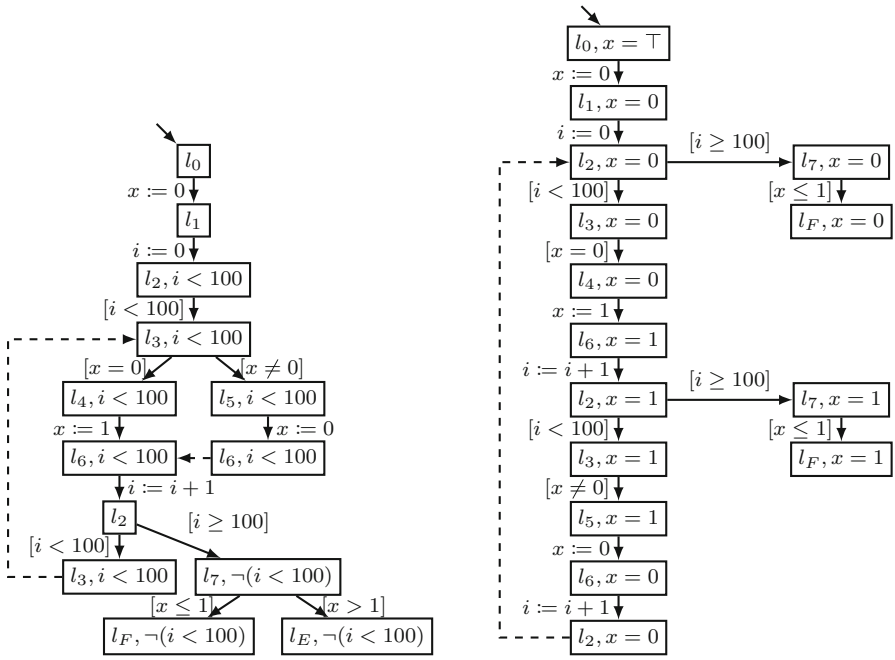
```
Input  : ARG = (N, E, C): partially constructed abstract reachability graph
         l_E: error location
         D_L = (S_L, ⊥_L, ⊑_L, expr_L): abstract domain with locations
         π_L: current precision
         T_L: transfer function with locations
Output : (safe or unsafe, ARG)
1  reached ← N
2  waitlist ← unmarked nodes from N
3  while waitlist ≠ ∅ do
4      (l, s) ← remove from waitlist
5      if l = l_E then
6          // (l, s) is unsafe
7          return (unsafe, ARG)
8      end
9      if (l, s) ⊑ (l', s') for some (l', s') ∈ reached then
10         // (l, s) is covered
11         C ← C ∪ {(l, s, l', s')} // Add covered-by edge
12     else
13         // (l, s) is expanded
14         foreach (l', s') ∈ T_L((l, s), π_L)\⊥_L do
15             reached ← reached ∪ {(l', s')}
16             waitlist ← waitlist ∪ {(l', s')}
17             N ← N ∪ {(l', s')} // Add new node
18             E ← E ∪ {(l, s, l', s')} // Add successor edge
19         end
20     end
21 end
22 return (safe, ARG)
```

The algorithm initializes the reached set with all states from the ARG and the waitlist with all unmarked states. The algorithm removes and processes states from the waitlist based on some search strategy (e.g., BFS or DFS). If the current state corresponds to the error location, the abstraction terminates with an unsafe result and an unsafe ARG. Otherwise, we check if some already reached state covers the current with respect to the partial order. If not, we calculate successors with the transfer function making the node expanded.

If there are no more nodes to explore and the error location was not found, the abstraction concludes with a safe result and a complete ARG. Note that due to its construction, the ARG without covered-by edges is actually a tree.

Example Figure 3a shows the ARG for the program in Fig. 1 using predicate abstraction with a single predicate $\pi_L(l) = \{i < 100\}$ for each location $l \in L$. Nodes are annotated with the location and the predicate (or its negation). Edges are marked with the operations from the CFA. Dashed arrows represent covered-by edges. It can be seen that an abstract state with the error location l_E is reachable, thus abstraction concludes with an unsafe result. However, using a different set of predicates, e.g., $\pi'_L(l) = \{x \leq 1\}$ would be able to prove the safety of the program.

Figure 3b shows the ARG for the same program using explicit-value abstraction with only tracking the variable x, i.e., $\pi_L(l) = \{x\}$ for all $l \in L$. Nodes are annotated with the location



(a) ARG for predicate abstraction with precision $\pi_L(l) = \{i < 100\}$ for each $l \in L$. Using this precision the ARG is unsafe because a state with l_E is reachable.

(b) ARG for explicit-value abstraction with precision $\pi_L(l) = \{x\}$ for each $l \in L$. Using this precision, the ARG is safe as no state with l_E is reachable.

Fig. 3 Example ARGs for the program in Fig. 1. Nodes are represented by rectangles, successors by solid arrows and coverage by dashed arrows

and the value of x. It can be seen that no abstract state is reachable in the ARG with the error location l_E, therefore the original program is safe. Also note that tracking the loop variable i is not necessary, hence reducing the size of the ARG.

2.2.2 Refinement

Feasibility check Algorithm 2 presents the refinement procedure. The input is an unsafe ARG and the current precision π_L. Refinement starts with extracting a path $\sigma = ((l_1, s_1), op_1, (l_2, s_2), op_2, \ldots, op_{n-1}, (l_n, s_n))$ to the unsafe state (i.e., $l_n = l_E$) for feasibility checking. A feasible path corresponds to a concrete path (in the original program) leading to the error location, which terminates refinement with an unsafe result. In this case the precision and the ARG is returned unmodified. Otherwise, an *interpolant* [55] is calculated from the infeasible path σ that holds information for the further steps of refinement.

Definition 4 (*Binary interpolant*) For a pair of inconsistent formulas A and B, an interpolant I is a formula such that

- A implies I,
- $I \wedge B$ is unsatisfiable,
- $\mathsf{var}(I) \subseteq \mathsf{var}(A) \cap \mathsf{var}(B)$.

A binary interpolant for an infeasible path σ can be calculated by defining $A \equiv s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge \ldots \wedge op_{i-1}^{\langle i-1 \rangle} \wedge s_i^{\langle i \rangle}$ and $B \equiv op_i^{\langle i \rangle} \wedge s_{i+1}^{\langle i+1 \rangle}$, where i corresponds to the longest prefix of σ that is still feasible.

Binary interpolants can be generalized to *sequence interpolants* [63] in the following way.

Definition 5 (*Sequence interpolant*) For a sequence of inconsistent formulas $A_1, \ldots, A_n$, a sequence interpolant $I_0, \ldots, I_n$ is a sequence of formulas such that

- $I_0 = true$, $I_n = false$,
- $I_i \wedge A_{i+1}$ implies I_{i+1} for $0 \leq i < n$,
- $\mathsf{var}(I_i) \subseteq (\mathsf{var}(A_1) \cup \ldots \cup \mathsf{var}(A_i)) \cap (\mathsf{var}(A_{i+1} \cup \ldots \cup \mathsf{var}(A_n)))$ for $1 \leq i < n$.

A sequence interpolant for a path σ can be calculated by defining $A_1 \equiv s_1^{\langle 1 \rangle}$ and $A_i \equiv op_i^{\langle i \rangle} \wedge s_{i+1}^{\langle i+1 \rangle}$ for $1 \leq i < n$. A binary interpolant I_k corresponding to a feasible prefix with length k can also be written as a sequence interpolant where $I_i \equiv true$ for $i < k$, $I_i \equiv I_k$ for $i = k$ and $I_i \equiv false$ for $i > k$. Note that each element I_i of the sequence corresponds to a single state (l_i, s_i) in the counterexample σ, except I_0. Therefore, I_0 is dropped and variables $V^{\langle i \rangle}$ are replaced with V before using the formulas for refinement.

Algorithm 2: Refinement algorithm.

```
   Input  : ARG = (N, E, C): unsafe abstract reachability graph
            l_E: error location
            π_L: current precision
   Output : (unsafe or spurious, π'_L, ARG)
1  σ = ((l_1, s_1), op_1, ..., op_{n-1}, (l_n, s_n)) ← path to unsafe node (with l_E) from ARG
2  if s_1^⟨1⟩ ∧ op_1^⟨1⟩ ∧ ... ∧ op_{n-1}^⟨n-1⟩ ∧ s_n^⟨n⟩ is feasible then return (unsafe, π_L, ARG);
3  else
4      (I_1, ..., I_n) ← get interpolant for σ
5      (π_1, ..., π_n) ← map interpolant (I_1, ..., I_n) to precisions
6      if π_L is local then
7          π'_L(l_i) = π_L(l_i) ∪ π_i if l_i is in σ and π'_L(l_i) = π_L(l_i) otherwise
8      else
9          π'_L(l) = π_L(l) ∪ ⋃_{1≤i≤n} π_i for each l ∈ L
10     end
11     i ← lowest index for which I_i ∉ {true, false}
12     N_i ← all nodes in the subtree rooted at (l_i, s_i)
13     remove nodes in N_i from N
14     remove edges connected to any node in N_i from E and from C
15     return (spurious, π'_L, ARG)
16 end
```

Precision adjustment The precision is adjusted by first mapping the formulas of the interpolant $I_1, I_2, \ldots, I_n$ to a sequence of new precisions $\pi_1, \pi_2, \ldots, \pi_n$ (in line 5). In predicate abstraction the formulas in the interpolant can simply be used as new predicates, i.e., $\pi_i = I_i$, whereas in the explicit domain variables of these formulas are extracted,[7] i.e., $\pi_i = \mathsf{var}(I_i)$. Then, the new precision π'_L is updated in the following way (in lines 6–10). If π_L is local, then $\pi'_L(l_i)$ is calculated by joining the new precision for each location l_i in the counterexample

[7] Explicit-value analysis [15] originally performs interpolation with the strongest post operator and constraint sequences. We use an SMT-based approach to generalize our algorithms for transition systems [43], where the transition relation is not limited to assignments and assumptions.

to its previous precision. Otherwise if π_L is global, then $\pi'_L(l)$ is a union of the old and new precisions for each location $l \in L$.

Pruning The final step of the refinement is to prune the ARG back until the earliest state where actual refinement occurred, i.e., where the precision changed (lazy abstraction [47]). Formally, this is the node (l_i, s_i) with lowest index $1 \leq i < n$, for which $I_i \notin \{true, false\}$. Pruning is done by removing the subtree rooted at (l_i, s_i), including all the successor and covered-by edges associated with the nodes of the subtree. Note, that during this process the parent of (l_i, s_i) becomes unmarked (not expanded anymore) and nodes might also get unmarked due to the removal of covered-by edges. Thus, the abstraction algorithm can continue constructing the ARG in the next iteration.

2.2.3 CEGAR Loop

Algorithm 3 connects the abstraction (Algorithm 1) and refinement (Algorithm 2) methods into a CEGAR loop (Fig. 2). The input of the algorithm is an initial location l_0, an error location l_E, an abstract domain D_L with locations, an initial (usually empty) precision π_{L_0} and a transfer function T_L.

Algorithm 3: CEGAR loop.

Input : l_0: initial location
l_E: error location
$D_L = (S_L, \bot_L, \sqsubseteq_L, \mathsf{expr}_L)$: abstract domain with locations
π_{L_0}: initial precision
T_L: transfer function with locations
Output : safe or unsafe

```
1  ARG ← (N ← (l_0, ⊤), E ← ∅, C ← ∅)
2  π_L ← π_{L_0}
3  while true do
4      result, ARG ← ABSTRACTION(ARG, l_E, D_L, π_L, T_L)
5      if result = safe then return safe;
6      else
7          result, π_L, ARG ← REFINEMENT(ARG, l_E, π_L)
8          if result = unsafe then return unsafe
9      end
10 end
```

First, an ARG is initialized with a single node corresponding to the initial location l_0 and the top element of the domain. The current precision π_L is also set to the initial precision π_{L_0}. Then the algorithm iterates between performing abstraction and refinement until abstraction concludes with a safe result, or refinement confirms a real counterexample.

3 Algorithmic Improvements

In this section we introduce several improvements both related to the abstraction (Sect. 3.1) and the refinement phase of the algorithm (Sect. 3.2). For abstraction, we define a modified version of the explicit domain where a configurable number of successors can be enumerated (Sect. 3.1.1). We also propose a new search strategy based on the syntactical distance from the error location (Sect. 3.1.2). Furthermore, we describe different ways of splitting predicates to control the granularity of predicate abstraction (Sect. 3.1.3).

For refinement, we present a novel interpolation strategy based on backward reachability (Sect. 3.2.1). We also introduce a method to use multiple counterexamples for refinement (Sect. 3.2.2). Finally, we define an approach to select from multiple refinements for a single counterexample (Sect. 3.2.3).

3.1 Abstraction

3.1.1 Configurable Explicit Domain

Motivation If an expression cannot be evaluated during successor computation in explicit-value abstraction [15] (e.g., due to top elements in abstract states), it is treated and propagated as the top element (i.e., an arbitrary value). In many cases, this is a desirable behavior, which can for example, avoid explicitly enumerating all possibilities for input variables that can indeed take any value from their domain. However, it is also possible that this behavior prevents successful verification.

Example Consider the program on the left side of Fig. 4. The program is safe, because $0 < x \wedge x < 5$ and $x = 0$ cannot hold at the same time. However, explicit-value abstraction fails to prove safety of this program. Even if x is tracked by the analysis, its value is unknown ($x = \top$) due to the nondeterministic assignment in line 1. The assumption in line 2 is satisfiable, but with multiple values for x. Therefore, the algorithm continues to line 3 with $x = \top$, where the assumption is again satisfiable (with $x = 0$), reaching the assertion violation. At this point, refinement returns the same precision (there are no more variables to be tracked), thus the same abstraction is built again and the algorithm fails to prove the safety of the program.

The problem is that this kind of abstraction can only learn the fact $(0 < x \wedge x < 5)$ by enumerating all possibilities for x. This is actually feasible in this case since there are only 4 different values (successors) for x and from each of them, the assumption $x = 0$ is unsatisfiable, proving the safety of the program. Similar examples include variables with finite domains (e.g., Booleans) or modulo operations (e.g., $x{:=}y \ \% \ 3$).

However, explicitly enumerating all values for a variable is often impractical or even impossible due to the large (or infinite) number of possible values. As an example, consider now the program on the right side of Fig. 4. This program is also safe, because $x \neq 0$ and $x = 0$ cannot hold at the same time. In this case however, enumerating all values for x such that $x \neq 0$ is clearly impractical.

Proposed approach Motivated by the examples above, we propose an extension of the explicit-value domain [15], where in case of a non-deterministic expression we allow a limited number of successors to be enumerated explicitly. If the limit is exceeded, the algorithm works as previously (treating the result as unknown). This way we can still avoid state space

```
1 int x = nondet();
2 if (0 < x && x < 5) {
3   if (x == 0) {
4     assert(false);
5   }
6 }
```

```
1 int x = nondet();
2 if (x != 0) {
3   if (x == 0) {
4     assert(false);
5   }
6 }
```

Fig. 4 Example programs where safety cannot be proven with explicit-value abstraction due to unknown (top) values

explosion, but can also solve certain problems that could not be handled previously with traditional explicit-value analysis.

First, we define a modified version of the strongest post-operator (denoted by sp'), which distinguishes unknown evaluation results from top elements (introduced deliberately by the abstraction). Given an abstract variable assignment $s \in S$ and an operation $op \in Ops$, let the resulting abstract variable assignment $\hat{s} = \mathsf{sp}'(s, op)$ be defined as follows. If op is an assumption $[\psi]$ then for all $v \in V$

$$\hat{s}(v) = \begin{cases} \bot_{d_v} & \text{if } s(u) = \bot_{d_u} \text{ for any } u \in V, \\ \bot_{d_v} & \text{if } \psi_{/s} \text{ evaluates to } \mathit{false}, \\ s(v) & \text{if } \psi_{/s} \text{ evaluates to } \mathit{true}, \\ \text{unknown} & \text{otherwise.} \end{cases} \tag{3}$$

If op is an assignment $w{:=}\varphi$ then for all $v \in V$

$$\hat{s}(v) = \begin{cases} \bot_{d_v} & \text{if } s(u) = \bot_{d_u} \text{ for any } u \in V, \\ s(v) & \text{if } v \neq w, \\ c & \text{if } v = w \text{ and } \varphi_{/s} \text{ evaluates to a literal } c, \\ \text{unknown} & \text{otherwise.} \end{cases} \tag{4}$$

The difference between sp and sp' is that if sp' cannot evaluate an assumption or an assignment to a literal then it is treated as a special unknown value.

Our extended, configurable transfer function $T_k(s, op, \pi)$ works as follows (Algorithm 4). It first uses sp' to compute the successor abstract variable assignment of s with respect to op. If an unknown value is encountered, we use an SMT solver to query satisfying assignments of the primed version of variables in π for the expression $s \wedge op$ with the given limit k. This is done with a feedback loop in the following way. We first query a satisfying assignment for the formula $s \wedge op$ and project it to only include variables in π'. Then we add the negation of the assignment as a formula to the solver and repeat this process until the formula becomes unsatisfiable or we exceed k. Note, that if there are multiple variables in π', the limit k corresponds to all possible combinations and not to each individual variable separately (which would allow $|\pi'|^k$ total assignments). For example, $\{(x = 1, y = 5), (x = 1, y = 6), (x = 2, y = 6)\}$ counts as 3 assignments, even though both x and y can only take 2 different values.

If there are no more than k possible assignments, we treat all of them as a new successor state as if it was returned by sp'. Otherwise, if there are more than k assignments, we stop enumerating them and fall back to using sp instead.

Finally, we perform abstraction by setting the non-tracked variables $v \notin \pi$ to top elements in the successors (as it is done in plain explicit-value abstraction). Note that as a special case $k = 1$ is similar to traditional explicit-value analysis because each state has at most one successor. However, if an expression cannot be evaluated (even using heuristics), we use an SMT solver which makes the analysis more expensive, but also more precise.

Discussion The advantage of this method is that k can be tuned to reduce the number of unknown values while still avoiding state space explosion. For the example on the left side of Fig. 4, any k with $k \geq 4$ would work. Currently we experimented with different values for k from a fixed set of values (Sect. 4.2.1). However, it would also be possible to use heuristics for automatically selecting or even dynamically adjusting k during the analysis. Such heuristics could be based on the domain of variables (e.g., Booleans, bounded integers) or the operations (e.g., modulo arithmetic). Furthermore, different k values could be assigned to different locations $l \in L$ in the CFA similarly to a local precision.

Algorithm 4: Configurable transfer function $T_k(s, op, \pi)$.

Input : k: bound for explicitly enumerating successors
s: source state
op: operation
π: target precision
Output : $S' \subseteq 2^S$: set of successor states

$\hat{s} \leftarrow \mathsf{sp}'(s, op)$
if $\hat{s}$ *contains any unknown value* **then**
  $S' \leftarrow$ query at most k assignments of variables in π' for the formula $s \wedge op$
  if *more than k assignments are possible* **then** $S' \leftarrow \{\mathsf{sp}(s, op)\}$
else
  $S' \leftarrow \{\hat{s}\}$
end
foreach $s' \in S'$ **do**
  $s'(v) = \top_{D_v}$ for each $v \in V \backslash \pi$
end
return S'

Note, that since we are enumerating k successors in each step, after n steps there could be k^n states in the worst case. However, this can only happen if there is a non-deterministic assignment for the variables in each step. Otherwise, we know the exact values of each variable after the first step and we can evaluate every expression in the following steps in exactly one way.

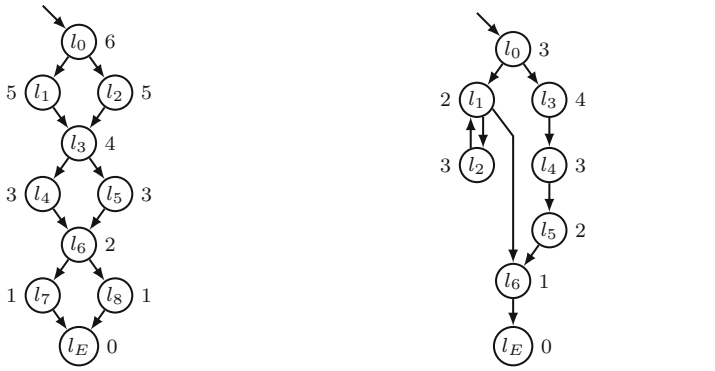
Operations in the CFA have their corresponding FOL expressions, therefore an SMT solver can be used out-of-the box to enumerate successors. However, our algorithm can work with other strategies (known e.g., from explicit model checkers [49]) as long as they can enumerate successors for a source state and an operation. Furthermore, since we only need the actual successors if there are no more than k of them, as an optimization, heuristics could be developed that can tell if an expression has more than k satisfying assignments without actually enumerating them.

3.1.2 Error Location-Based Search

Motivation Recall that the abstract state space can be explored using different search strategies, depending on how the ARG nodes in the waitlist are ordered (Algorithm 1). For example, breadth and depth-first search (BFS and DFS) orders nodes based on their depth ascending and descending respectively. These basic strategies however, use no information from the input verification task.

Proposed approach To focus the search more effectively, we propose a strategy based on the syntactical distance from the error location in the control flow automaton. Given a verification task (CFA, l_E) we define the distance $d_E : L \mapsto \mathbb{N}$ of each location $l \in L$ to the error location l_E as the length of the shortest directed path from l to l_E without considering the operations. Note that $d_E(l)$ is an under-approximation of the actual distance between l and l_E in the ARG since shorter paths are not possible, but some operations might be infeasible, making the actual (feasible) distance longer. The distances can be calculated (and stored for later queries) at the beginning of the analysis using a backward breadth-first search from the error location.[8] Then from each node (l, s) on the waitlist, we simply remove one where

[8] Locations that are not reachable backward from the error location have a distance of infinity. However, using backward slicing [59] as a preprocessing step removes such locations.



(a) Example where BFS would need an exponential number of steps to reach l_E.

(b) Example where DFS may unfold the loop l_1, l_2 many times.

Fig. 5 Examples for error location-based search. Numbers next to the locations denote their distance form l_E

$d_E(l)$ is minimal. However, some examples highlight that loops might trick this approach as well. Therefore, we also experiment with metrics based on a weighted sum of the distance to the error location and the depth of the current node in the ARG.

Example Consider the CFA in Fig. 5a. The distance to the error location l_E is written next to each location. For simplicity, operations are omitted from the edges. Furthermore, suppose that most of the paths are actually feasible at the current level of abstraction, as otherwise all search strategies perform similarly. It can be seen that the number of paths to the error location scales exponentially with the number of branches (if this diamond-shaped pattern is repeated). Therefore, a traditional BFS approach would cause an exponential execution time. DFS would however, find the first path to l_E quickly for example by exploring $l_0, l_1, l_3, l_4, l_6, l_7, l_E$ in this order. The error location-based approach would act similarly, as it first starts with l_0, discovering its successors l_1 and l_2 both with a distance of 5. Then, by picking for example l_1, its only successor is l_3 with a distance of 4. Therefore, the algorithm will pick l_3 (with $d_E(l_3) = 4$) next instead of l_2 (with $d_E(l_2) = 5$), similarly to DFS.

Consider now the CFA in Fig. 5b. DFS can easily fail for this case if it is feasible to unfold the loop $l_0, l_1, l_2, l_1, l_2, l_1, l_2 \ldots$ many times. However, the error location-based search may also fail if the edge from l_1 to l_6 is not feasible. In this case, the algorithm would also iterate between l_1 and l_2 (as long as possible), since l_3 on the other path has a greater distance. A possible way to overcome this problem is to use a combined metric based on the depth of the current node in the ARG (denoted by d_D) and the distance to the error location.

Discussion Simply summing the distance and the depth causes each node corresponding to Fig. 5a to be equal in the ordering. Hence, it is reasonable to use a weighted metric $w_D \cdot d_D(s, l) + w_E \cdot d_E(l)$. Assigning a greater weight to d_E can guide the search effectively based on the CFA, while a nonzero weight for w_D can help to avoid unfolding loops too many times. Currently, we experimented with the following five different configurations for the weights (Sect. 4.2.2).

- $(w_E = 0, w_D = 1)$ is a traditional breadth-first search.
- $(w_E = 0, w_D = -1)$ is a traditional depth-first search.
- $(w_E = 1, w_D = 0)$ considers only the distance from the error location.
- $(w_E = 2, w_D = 1)$ combines the distance from the error with the depth (BFS), but with less weight.

– $(w_E = 1, w_D = 2)$ also uses depth and the distance from the error but is biased towards depth.

The first three configurations serve as a baseline, while the last two demonstrate combinations. A possible future work could be to experiment with further values for the weights or with iteration strategies known from abstract interpretation [21].

Remark One might wonder about the usefulness of this approach on safe verification tasks (where no concrete state with l_E is reachable). Even for such tasks, the intermediate iterations of CEGAR still encounter (spurious) counterexamples. In this case the error location-based search can help to find these counterexamples and converge faster.

3.1.3 Splitting Predicates

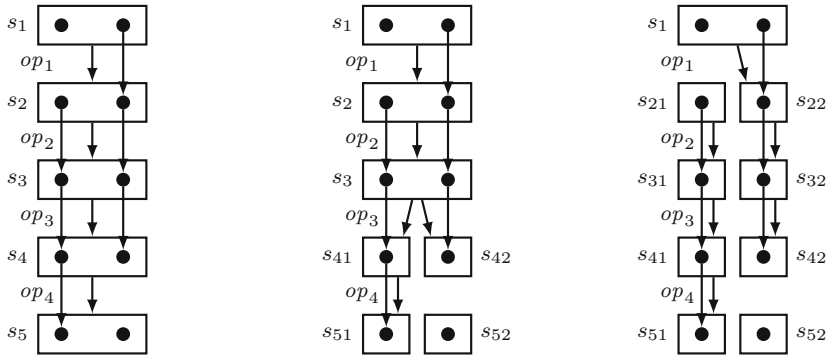
Motivation Predicates are the atomic units of predicate abstraction, i.e., each abstract state is labeled with some Boolean combination of predicates $p_i \in \pi$ from the current precision π. Cartesian predicate abstraction yields a conjunction (e.g., $p_1 \wedge \neg p_2 \wedge p_3$), whereas Boolean predicate abstraction can give any combination (using a disjunction over conjunctions, e.g., $p_1 \wedge \neg p_2 \vee p_2 \wedge p_3$). However, predicates themselves can also correspond to an arbitrary formula over some atoms, e.g., $p \equiv (0 < x) \wedge (y < 5) \vee (x < 5)$. In such cases we can treat a complex predicate both as a whole [19] or we can also split it into smaller parts such as its atoms [46]. This can influence both the precision of abstraction and the performance of the algorithm. For example, suppose that we want to represent a state $a \wedge \neg b \vee \neg a \wedge b$, where a and b are some atoms. If we only consider the atoms $\{a, b\}$ as the precision, their strongest conjunction implied by the state is *true*, i.e., Cartesian abstraction might not be precise enough. While Boolean predicate abstraction is able to faithfully reconstruct the original state, the number of possibly enumerated models grow exponentially with the number of atomic predicates. In contrast, keeping predicates as a whole may yield a slower convergence as subformulas cannot be reused.

Proposed approach New predicates are introduced during abstraction refinement using interpolation. However, interpolation procedures may return complex formulas, which are specific to a single counterexample. A possible way to generalize such formulas is to split complex predicates into smaller parts before adding them to the refined precision. Formally, we define different *splitting functions* $\mathsf{split}\colon FOL \mapsto 2^{FOL}$ that map a FOL formula to a set of formulas.

We experimented with the following configurations, which give different granularities for the precision (Sect. 4.2.3).

– $\mathsf{atoms}(\varphi)$ splits predicates to their *atoms*, which is the finest granularity that can be achieved syntactically.[9] For example, $\mathsf{atoms}(p_1 \wedge (p_2 \vee \neg p_3)) = \{p_1, p_2, p_3\}$.
– $\mathsf{conjuncts}(\varphi)$ is a middle ground that splits predicates to their *conjuncts*. For example, $\mathsf{conjuncts}(p_1 \wedge (p_2 \vee \neg p_3)) = \{p_1, (p_2 \vee \neg p_3)\}$.
– $\mathsf{whole}(\varphi) \equiv \varphi$, i.e., the identity function keeps predicates as a *whole*, which is the coarsest granularity. It is motivated by Boolean variables, where the atoms are the variables themselves and the valuable information learned by the interpolation procedure lies in the logical connections.

[9] Even finer granularity can be achieved by deriving equivalent predicates, e.g., splitting $x = 0$ to $x \leq 0$ and $x \geq 0$.



(a) Spurious counterexample. **(b)** After one step of forward binary interpolation. **(c)** Final result.

Fig. 6 Spurious counterexample and its refinement

A similar idea can be applied to generalize Boolean predicate abstraction, where the Boolean combination of predicates is represented by a single state as a disjunction over conjunctions of predicates. We define a modified version of Boolean predicate abstraction called *splitting abstraction* where this disjunction is split into its elements, which are then treated as separate abstract states (separate nodes in the ARG). This allows us to represent successor and coverage relations in a finer way. For example, the abstract state $s = (p_1 \wedge \neg p_2) \vee (p_2 \wedge p_3)$ is split into $s_1 = (p_1 \wedge \neg p_2)$ and $s_2 = (p_2 \wedge p_3)$. Then it can be possible that although s cannot be covered, but s_1 can be and we only have to continue with s_2.

3.2 Refinement

3.2.1 Backward Binary Interpolation

Motivation The binary interpolation algorithm presented in Sect. 2.2.2 defines the two formulas A and B based on the longest feasible prefix. This yields an interpolant that refines the last abstract state on the counterexample that can still be reached in the concrete program (starting from the initial state). Therefore, from this point on we will refer to this strategy as *forward binary interpolation*. We observed that this strategy gives poor performance in many cases (Sect. 4.2.4).

Example Consider the abstract counterexample in Fig. 6a. Rectangles are abstract states, with dots representing concrete states mapped to them. The initial state is s_1 and the erroneous state is s_5. Edges denote transitions in the concrete and abstract state space. Due to the existential property of abstraction, an abstract transition exists between two abstract states if at least one concrete transition exists between concrete states mapped to them [32].

It can be seen that the longest feasible prefix is $(s_1, op_1, s_2, op_2, s_3, op_3, s_4)$. Forward binary interpolation would therefore set $A \equiv s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge \ldots \wedge op_3^{\langle 3 \rangle} \wedge s_4^{\langle 4 \rangle}$ and $B \equiv op_4^{\langle 4 \rangle} \wedge s_5^{\langle 5 \rangle}$. This gives an interpolant corresponding to s_4, pruning the ARG back until s_3. Continuing from s_3 with the new precision yields s_{41}, s_{42}, s_{51} and s_{52} (instead of s_4 and s_5), as seen in Fig. 6b. However, s_{51} is still reachable in the abstract state space (via $s_1, s_2, s_3, s_{41}, s_{51}$), but the counterexample is only feasible until s_3 now. The algorithm needs

to perform two additional refinements until s_3 and s_2 is refined, and the ARG is pruned back until s_1 (Fig. 6c). All spurious behavior is now eliminated as neither s_{51} nor s_{52} is reachable. However, this requires many iterations for the same counterexample, and a potentially larger abstract state space in each round due to the increasing precision.

We observed such situations when a variable is assigned at a certain point of the path (e.g, $op_1 \equiv x{:=}0$), but only contradicts a guard later (e.g. $op_4 \equiv [x > 5]$). Although the path is feasible until the guard, in these cases the root cause of the counterexample being spurious traces back to the assignment of the variable.

Proposed approach To alleviate the previous problems we define a novel refinement strategy that is based on the longest feasible *suffix* of the counterexample. We call this strategy *backward binary interpolation* as it starts with the erroneous state and progresses backward as long as the suffix is feasible. Formally, let $\sigma = (s_1, op_1, \ldots, op_{n-1}, s_n)$ be an abstract counterexample and let $1 < i \leq n$ be the lowest index for which the suffix $(s_i, op_i, \ldots, op_{n-1}, s_n)$ is feasible. Then we define a backward binary interpolant as $A \equiv s_i^{\langle i \rangle} \wedge op_i^{\langle i \rangle} \wedge \ldots \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$ and $B \equiv s_{i-1}^{\langle i-1 \rangle} \wedge op_{i-1}^{\langle i-1 \rangle}$. In other words, A encodes the feasible suffix and B encodes the preceding transition that makes it infeasible. The formula $A \wedge B$ is unsatisfiable, otherwise a longer feasible suffix would exist. Similarly to forward binary interpolation, the only common variables in A and B correspond to s_i. Therefore, indexes can be removed from the interpolant I.

As an example, consider Fig. 6a again. The longest feasible suffix is $(s_2, op_2, s_3, op_3, s_4, op_4, s_5)$. Thus, the interpolation formulas are $A \equiv s_2^{\langle 2 \rangle} \wedge op_2^{\langle 2 \rangle} \wedge \ldots \wedge op_4^{\langle 4 \rangle} \wedge s_5^{\langle 5 \rangle}$ and $B \equiv s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle}$. The resulting interpolant I corresponds to s_2 and the ARG is pruned back until s_1 (Fig. 6c) in a single step (assuming a global precision).

Discussion We motivated backward binary interpolation by comparing it to forward interpolation and showing that it can trace back the root cause in fewer steps. In software model checking however, sequence interpolation is the standard technique. Hence we also compare our backward interpolation approach to sequence interpolation (Sect. 4.2.4). A potential advantage of backward interpolation is that it can be more compact than sequence interpolation (which could yield a formula for each location along the counterexample, making the algorithm prune a larger portion of the state space). Backward search-based strategies also proved themselves efficient in the context of other algorithms, such as IMPACT [3] or NEWTON [38].

3.2.2 Multiple Counterexamples for Refinement

Motivation Most approaches in the literature stop exploring the abstract state space and apply refinement as soon as the first counterexample is encountered. Although collecting more counterexamples adds an overhead to abstraction, better refinements may be possible as more information is available. Altogether, this could reduce the number of iterations and increase the efficiency of the algorithm.

Proposed approach We modified the abstraction algorithm (Algorithm 1) so that it does not return the first counterexample (by removing line 7), but keeps exploring the state space. The algorithm can be configured (by adding a condition to the loop header in line 3) to stop after a given number of erroneous states or to explore all of them.

If at least one of the counterexamples is feasible, then the algorithm can terminate with an unsafe result. However, if all of them are infeasible, there are many possible ways to use the information for refinement. We propose a technique where we first calculate a refinement for

each counterexample and derive a minimal set required to eliminate all spurious behavior. Then, we update the precision and apply pruning based on this minimal set.

Our approach is formalized in Algorithm 5. First, we extract paths Σ leading to states with the error location l_E from the ARG. If any path $\sigma_i \in \Sigma$ is feasible, then the algorithm terminates with an unsafe result. Otherwise, we calculate an interpolant Itp_i for each path σ_i. Given a path σ_i and its corresponding interpolant Itp_i, we can determine the first state $s_{r_i} \in \sigma_i$ of the path that actually needs refinement (i.e., the first state where the interpolant is not *true* or *false*). These states correspond to pruning points in the ARG.

Algorithm 5: Refinement algorithm for multiple counterexamples.

```
Input  : ARG: abstract reachability graph with multiple counterexamples
         π_L: current precision
Output : unsafe or (spurious, π'_L)
Σ = (σ_1, ..., σ_n) ← extract paths to states with l_E from ARG
if any path σ_i ∈ Σ is feasible then return unsafe;
else
    π'_L = π_L
    Itps = (Itp_1, ..., Itp_n) ← get interpolant for each σ_i ∈ Σ
    S_r = (s_{r_1}, ..., s_{r_n}) ← calculate first refined state for each σ_i ∈ Σ
    foreach σ_i ∈ Σ do
        if no state in S_r is a proper ancestor of s_{r_i} in the ARG then
            (π_{i_1}, ..., π_{i_k}) ← map interpolant Itp_i = (I_{i_1}, ..., I_{i_k}) to precisions
            if π_L is local then
                π'_L(l_{i_j}) = π'_L(l_{i_j}) ∪ π_{i_j} for each location l_{i_j} in σ_i
            else
                π'_L(l) = π'_L(l) ∪ ⋃_{1≤j≤k} π_{i_j} for each l ∈ L
            end
            prune ARG back to s_{r_i}
        end
    end
    return (spurious, π'_L)
end
```

Then, we determine the minimal set of counterexamples to be refined in the following way. For each path σ_i with its first state to be refined s_{r_i}, we check if any other state in S_r is a proper ancestor[10] of s_{r_i} in the ARG.[11] If such state exists, it means that the other path shares its prefix with the currently examined path, and will need refinement earlier. That refinement will add new predicates and prune the ARG earlier, possibly eliminating the current counterexample as well. Therefore, the current path is skipped for now (lazy refinement).

For each path that is not skipped, we map the interpolant to a new precision and join it to the old one, taking into account whether the precision is local or global. Finally, we return a spurious result and the new precision π'_L.

Example Consider the ARG (without covered-by edges) in Fig. 7. There are four counterexamples $\sigma_1, \ldots, \sigma_4$ in the ARG leading to the abstract states $(s_1, l_E), \ldots, (s_4, l_E)$. The first states to be refined are denoted with a gray background. In this example the minimal set of counterexamples is $\{\sigma_2, \sigma_3\}$, because s_{r_2} and s_{r_3} are proper ancestors of s_{r_1} and s_{r_4} respectively. Refining σ_2 and σ_3 will therefore, eliminate all spurious behavior from the current ARG. Note, that in the next iteration (s_1, l_E) and (s_4, l_E) might still be reached again

[10] Proper ancestors of a node are its ancestors excluding the node itself.

[11] Recall that without the covered-by edges, the ARG is a tree.

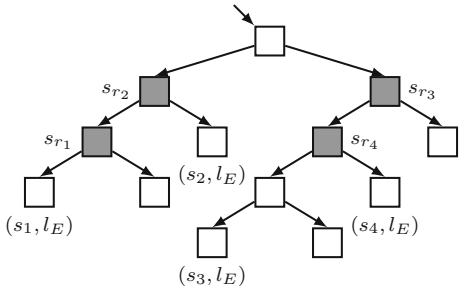


Fig. 7 Example of refining multiple counterexamples

if the predicates for σ_2 and σ_3 were not sufficient. In this case these counterexamples are eliminated in the next iteration.

Discussion Our approach for multiple counterexamples can work with any refinement strategy. In our current experiment (Sect. 4.2.5) we use sequence interpolation. However, it would even be possible to use different strategies for the different counterexamples as opposed to existing approaches that use multiple counterexamples (e.g., DAG interpolation [1] or global refinement [54]).

Currently we have a single error location in the CFA so each counterexample leads to the same location on a different path. However, our approach does not rely on this, and would work the same way even if the collected counterexamples lead to different locations.

The presented algorithm handles all counterexamples in the solver separately by reusing existing interpolation modules. A possible optimization would be to use the incremental API of SMT solvers by pushing the first counterexample, performing the check and interpolation and then popping only back to the common prefix of the current and next counterexample, and so on.

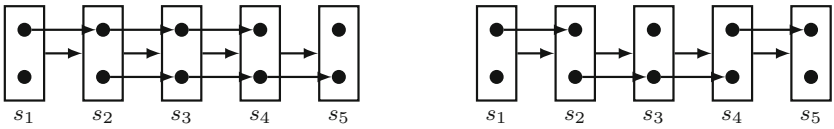
3.2.3 Multiple Refinements for a Counterexample

Motivation In Sect. 3.2.1 we presented a novel interpolation approach based on backward search, which performs better than the traditional forward search method according to our experiments (Sect. 4.2.4). Using a portfolio of refinements can combine the advantages of different methods [16,45]. Therefore, in this section we suggest strategies that calculate both forward and backward interpolants and pick the "better" one based on certain heuristics.

Proposed approach The heuristics that we currently introduce are based on the index of pruning. Recall that given an interpolant in its general form $I_0, \ldots, I_n$, the ARG is pruned back until actual refinement occurred, i.e., until the lowest index $1 \leq i < n$ with $I_i \notin \{true, false\}$. This corresponds to the longest feasible prefix and suffix for forward and backward binary interpolants respectively.

Two basic heuristics that we experiment with (Sect. 4.2.6) are to select the interpolant with the minimal or maximal prune index. These heuristics prune the ARG as close as possible to the initial state or the error state respectively.

Example Consider Fig. 8 with two possible abstract counterexamples. In case of Fig. 8a forward and backward interpolation would prune until s_4 and s_2 respectively. For the counterexample in Fig. 8b pruning would be the other way around. However, the minimal and maximal prune index strategies would prune until s_2 and s_4 respectively in both cases.



(a) Example where forward and backward interpolation would prune at s_4 and s_2 respectively.

(b) Example where forward and backward interpolation would prune at s_2 and s_4 respectively.

Fig. 8 Examples for minimal and maximal prune indexes

4 Evaluation

In this section, we evaluate the effectiveness and efficiency of our algorithmic contributions presented before (Sect. 3) by conducting an experiment. First, we introduce our experiment plans along with the research questions to be addressed (Sect. 4.1). Then, we present and discuss our results and analyses for each research question in a separate subsection (Sect. 4.2). Finally, we compare our implementation to other tools in order to provide a baseline for the research questions (Sect. 4.3). The design and terminology of the experiment are based on the book of Wohlin et al. [64]. The raw data, a detailed report and instructions to reproduce our experiment are available in a supplementary material [42].

4.1 Experiment Planning

The goal of our experiment is to evaluate our new contributions on a broad set of verification tasks from diverse sources. In our experiment we execute various *configurations* of the CEGAR algorithm on several *input models*.

4.1.1 Research Questions

We formulate a research question for the performance of each algorithmic contribution presented in Sect. 3. We are mainly interested in two performance aspects: the number of verification tasks solved within a given time limit per task (effectiveness) and the total execution time required (efficiency). Other measured aspects include a more refined categorization of unsolved tasks (timeout, out-of-memory, exception) and the peak memory consumption.

- RQ1 How does the configurable explicit domain perform for increasing values of k compared to traditional explicit-value analysis?
- RQ2 How does the error location-based search perform for different weights (w_D, w_E) compared to breadth and depth-first search?
- RQ3 How do splitting predicates (into conjuncts or atoms) and splitting states perform compared to predicate abstraction without splitting?
- RQ4 How does backward binary interpolation perform compared to forward binary and sequence interpolation?
- RQ5 How does refinement based on multiple counterexamples perform compared to using only a single one?
- RQ6 How do the combined refinement strategies (based on the minimal/maximal prune index) perform compared to backward and forward binary interpolation?

Table 1 Overview of the input verification tasks with the number of variables, locations, edges and the cyclomatic complexity (CC)

Source	Category	Models	Tasks	Vars	Locs	Edges	CC
SV-COMP	Locks	13	143	4–32	9–40	10–57	3–23
	Loops	59	105	1–11	4–26	3–33	2–19
	ECA	3	180	9–30	302–1301	375–1516	73–231
	ssh-simpl.	12	17	64–81	187–267	262–375	87–121
CERN	PLC	6	90	1–596	8–4614	7–4782	4–188
HWMCC	HWMCC	300	300	0–245278 inputs, 0–460501 latches, 0–4806245 gates			
	Total	393	835				

Ranges denote minimal and maximal numbers

4.1.2 Subjects and Objects

We implemented both the existing algorithms presented in the background (Sect. 2) and our new contributions (Sect. 3) in the open source THETA tool[12] [60]. THETA is a generic, modular and configurable framework, supporting the development and evaluation of abstraction-based algorithms in a common environment.

One of the distinguishing features of THETA is that it supports different kind of models (e.g., control flow automata, transition systems, timed automata). An interpreter hides the differences between these formalisms so the algorithms presented in this paper work for verification tasks from different domains (e.g., software, hardware). There are some exceptions though: the configurable explicit domain (Sect. 3.1.1) requires statements and the error location-based search (Sect. 3.1.2) requires locations. Therefore, these algorithms do not work for hardware models since those are encoded as transition systems.

For the objects of the experiment, we use C programs from the Competition on Software Verification (SV-COMP) [9], hardware models from the Hardware Model Checking Competition (HWMCC) [25] and industrial Programmable Logic Controller (PLC) software models from CERN [40].

Table 1 gives an overview of the number of input models and verification tasks along with size and complexity metrics. We selected models from four categories of the 2018 edition[13] of SV-COMP that are currently supported by the limited[14] C front-end of THETA [59]. By applying backward slicing [59] we generate a separate verification task for each assertion. The category *locks* consists of small (94-234 LoC) locking mechanisms with several assertions per model. The collection *loops* includes small (9-70 LoC) programs focusing on loops. The *ECA* (event-condition-action) task set contains larger (591-1669 LoC) event-driven reactive systems. The tasks in *ssh-simplified* describe larger (557-713 LoC) client-server systems.

We also experimented with industrial PLC software modules from CERN. These modules operate in an infinite loop, where a formula (the requirement) is always checked at the end of the loop. It can be seen that the size of these models is greatly varying from a few dozens of locations to a couple of thousands.

[12] http://github.com/FTSRG/theta (commit `f32d3f9`).

[13] https://sv-comp.sosy-lab.org/2018/.

[14] Currently THETA does not support arrays, pointers, structs, and function inlining is limited to simple cases.

Table 2 Variables of the experiment

Category	Name	Type	Description
Model (indep.)	Model	String	Unique name of the model (i.e., verification task)
	Category	Enum.	Category of the model. Possible values: eca, hwmcc, locks, loops, plc, ssh
Config. (indep.)	Domain	Enum.	Domain of the abstraction. Possible values: EXPL, PRED_BOOL, PRED_CART, PRED_SPLIT
	MaxEnum	Integer	Maximal number of successors to enumerate in the explicit domain (k). Only applicable if Domain is EXPL
	PrecGranularity	Enum.	Granularity of the precision. Possible values: GLOBAL, LOCAL
	PredSplit	Enum.	Predicate splitting method. Possible values: ATOMS, CONJUNCTS, WHOLE. Only valid if Domain is PRED_*
	Refinement	Enum.	Refinement strategy. Possible values: BW_BIN_ITP, FW_BIN_ITP, MAX_PRUNE, MIN_PRUNE, MULTI_SEQ, SEQ_ITP
	Search	Enum.	Search strategy. Possible values: BFS, DFS, ERR, DFS_ERR, ERR_DFS
Metrics (dep.)	Succ	Boolean	Indicates whether the algorithm successfully provided a correct result within the given resource limits
	Termination	Enum.	Indicates the termination reason. Possible values: success, time, memory, exception
	Result	Boolean	Result of the algorithm, indicates whether the model is safe according to the algorithm
	TimeMs	Integer	CPU time used by the algorithm (in milliseconds)
	Memory	Integer	Peak memory consumption of the algorithm (in bytes)

Furthermore, we picked all 300 models from the 2017 edition[15] of HWMCC. These tasks are encoded as transition systems, describing circuits with inputs, logical gates and latches. The metrics reported in the table for the hardware models are after applying cone of influence (COI) reduction [33].

The majority of the CFA tasks (442) is expected to be safe, while the rest is unsafe (93). To the best of our knowledge, the (300) hardware models do not have an expected result.

Due to slicing [59] it is possible that different tasks corresponding to the same program will have different models (i.e., CFA). Hence, we encode each task in a separate file including the model (CFA) and the property and treat them as if they were different models. Therefore, from now on we use the terms "model" and "verification task" interchangeably.

4.1.3 Variables

Variables of our experiment are listed in Table 2, grouped into three main categories. Properties of the model and parameters of the algorithm configuration are independent variables, whereas output metrics of the algorithm are dependent.

Properties of the model

[15] http://fmv.jku.at/hwmcc17/.

– The variable Model represents the unique name of each model (verification task).
– Furthermore, models have a Category based on their origin.

Parameters of the algorithm

– The variable Domain represents the abstract domain used. The values PRED_BOOL and PRED_CART stand for Boolean and Cartesian predicate abstraction, while EXPL stands for explicit-value analysis. Furthermore, our Boolean predicate abstraction with state splitting (Sect. 3.1.3) is encoded by PRED_SPLIT.
– The integer variable MaxEnum corresponds to the maximal number of successors allowed to be enumerated (denoted by k) in our configurable explicit domain (Sect. 3.1.1). The value 0 represents $k = \infty$, i.e., there is no limit on the number of successors. Furthermore, the value 1^* enumerates at most one successor *without* using an SMT solver (corresponding to traditional explicit-value analysis [15]).
– The variable PrecGranularity represents the granularity of the precision. When the granularity is LOCAL, a different precision can be assigned to each location, whereas GLOBAL granularity means that the precision is the same for each location.
– The variable PredSplit defines the way complex predicates are split into smaller parts before introducing them in the refined precision (Sect. 3.1.3). Possible values are ATOMS, CONJUNCTS and WHOLE (no splitting).
– The variable Refinement corresponds to the refinement strategy used. The values FW_BIN_ITP and SEQ_ITP represent traditional binary and sequence interpolation (Sect. 2.2.2). The value BW_BIN_ITP is our novel backward search-based binary interpolation strategy (Sect. 3.2.1), whereas MAX_PRUNE and MIN_PRUNE refer to combined refinements with maximal and minimal prune index (Sect. 3.2.3). The value MULTI_SEQ uses sequence interpolation and our approach of multiple counterexamples (Sect. 3.2.2).
– The variable Search represents the search strategy in the abstract state space. Values BFS and DFS denote breadth and depth first search. Other values correspond to our error location-based strategy (Sect. 3.1.2) with different weights w_D and w_E. The strategy ERR only takes into account the error location, i.e., $w_D = 0$ and $w_E = 1$. The values ERR_DFS and DFS_ERR use both weights but are biased towards one or the other ($w_D = 2$, $w_E = 1$ and $w_D = 1$, $w_E = 2$ respectively).

Metrics

– The dependent variable Succ indicates whether the algorithm terminated and provided a correct result (no false negative/positive) successfully within the given CPU time and memory limits (effectiveness).
– The variable Termination indicates the reason for termination (success, timeout, out-of-memory, exception) in a finer way than Succ. It is only used in the detailed plots of the supplementary report [42].
– The variable Result denotes whether the model is safe or unsafe according to the algorithm. We check that the result matches the expected (if available) and that all configurations agree.
– The variable TimeMs holds the execution time (CPU time) of the algorithm in milliseconds (efficiency).
– The variable Memory measures the peak (maximal) memory consumption during the execution of the algorithm in bytes (efficiency).

Table 3 Overview of the experiment

Parameter	RQ1	RQ2	RQ3	RQ4	RQ5	RQ6
Domain	EXPL	EXPL PRED_CART PRED_BOOL	PRED_CART PRED_BOOL PRED_SPLIT	EXPL PRED_CART PRED_BOOL	EXPL PRED_CART PRED_BOOL	EXPL PRED_CART PRED_BOOL
MaxEnum	0 1 1* 5 10 50	plc: 0 sv-comp: 1 hwmcc: NA	NA	plc: 0 sv-comp: 1 hwmcc: NA	plc: 0 sv-comp: 1 hwmcc: NA	plc: 0 sv-comp: 1 hwmcc: NA
PredSplit		WHOLE	ATOMS CONJUNCTS WHOLE	WHOLE	WHOLE	WHOLE
Refinement	SEQ_ITP	SEQ_ITP	SEQ_ITP	BW_BIN_ITP FW_BIN_ITP SEQ_ITP	MULTI_SEQ SEQ_ITP	BW_BIN_ITP FW_BIN_ITP MAX_PRUNE MIN_PRUNE
Search	BFS DFS	BFS DFS ERR DFS_ERR ERR_DFS	BFS	BFS	BFS	BFS
PrecGran.	plc: LOCAL, sv-comp: GLOBAL, hwmcc: NA					

Factors and blocking factors are marked with darker and lighter gray background respectively

4.1.4 Experiment Design

The experiment design is summarized in Table 3. It would be possible to execute each configuration on every model (crossover design) and then select the relevant subsets of data for each research question. However, due to the high number of parameters and their possible values, it would yield hundreds of configurations. Instead, for each research question we identify and manipulate one or two parameters that correspond to our new contributions. These parameters are called *factors*, for which each value (level) is executed on every model and the output is observed.

Based on our previous experience and the literature, the domain of the abstraction is a prominent parameter of CEGAR. Therefore, we also include it in the experiments as a *blocking factor* to systematically eliminate its undesired effect. RQ1 forms an exception, where only the explicit domain is applicable, therefore we use the search strategy for blocking.

The rest of the independent variables are kept at a *fixed level* that usually performed well in our previous experiments. These fixed levels however, can be different based on the type of the model, e.g., a local precision granularity is used for PLC models, while SV-COMP models perform better with global precision. Furthermore, certain parameters might not be applicable (NA) to hardware models since they are represented as transition systems instead of CFA.

To illustrate our design with an example, in RQ1 we evaluate 6 levels for MaxEnum and 2 levels for Search, while keeping other parameters at a fixed level. This yields a total number of 12 configurations.

4.1.5 Measurement Procedure

Measurements were executed on physical machines with 4 core (2.50 GHz) Intel Xeon L5420 CPUs and 32 GB of RAM, running Ubuntu 18.04.1 LTS and Oracle JDK 1.8.0_191 (THETA is implemented in Java). We used Z3 version 4.5.0 [57] for SMT solving.[16] To ensure reliable and accurate measurements, we used the RUNEXEC tool from the BENCHEXEC suite [18], which is a state-of-the-art benchmarking framework (also used at SV-COMP). Each measurement was executed with CPU time limit of 300 s[17] and a memory limit of 4 GB. The results were collected into CSV files for further analysis. Each measurement was repeated 2 times. Instructions to reproduce our experiment can be found in the supplementary material [42].

4.1.6 Threats to Validity

In this subsection we discuss threats to *construct*, *internal* and *external* validity [64] of our experiment. We are not concerned with *conclusion* validity, as we do not use statistical tests [64].

Construct validity can be ensured by using proper metrics to describe the "goodness" of algorithms. We use the number of solved instances for effectiveness, and the total execution time and peak memory consumption for efficiency. These metrics are widely used to characterize model checking algorithms [9,25,50].

Internal validity is concerned with identifying the proper relationship between the treatments and the outcome. We use dedicated hardware machines and repeated executions to reduce noise from the environment. Accuracy of the results is ensured by BENCHEXEC [18], a state-of-the-art benchmarking tool. We also apply blocking factors to eliminate undesired effects from known factors systematically. Nevertheless, internal validity could still be improved using a full, crossover design (executing all configurations on all models).

External validity is increased by using models from different and diverse sources, including standard benchmark suites (SV-COMP [9] and HWMCC [25]) and industrial models [40]. We compared our new contributions with various state-of-the-art algorithms implemented within the same framework. Furthermore, we also compare our implementation to other tools to provide a baseline (Sect. 4.3). However, external validity would benefit from using additional models (for example from other categories of SV-COMP) and from comparing related algorithms as well. Describing models with additional variables (e.g., size or complexity) besides their category would also further generalize our results.

4.2 Results and Analysis

We present the results and analyses for each research question in a separate subsection. Analyses were performed using the R software environment [58] version 3.4.3. We only present the most important results in the paper, but the raw data, the R script and a detailed report can be found in the supplementary material [42].

In each analysis, we first merge the repeated executions of the same measurement (same configuration on the same model) into a single data point in the following way. We consider

[16] Z3 dropped support for interpolation since version 4.8.1, but still works with version 4.5.0 that we used for the measurements. However, in order to use more recent versions, we are considering to use a separate SMT solver for interpolation, e.g., SMTInterpol [29].

[17] RUNEXEC also puts a limit on the wall time, which is CPU time limit plus 30 s by default.

a measurement successful if at least one of the repeated executions is successful. This is a reasonable choice as in most cases either all executions are successful or none of them are. Then, we calculate the execution time of a measurement by taking the mean time of its successful repetitions. The relative standard deviation[18] between the repeated executions was usually around 1% to 2%, allowing us to represent them with their mean. In a few cases, the repeated executions terminated due to a different reason (e.g., timeout first, then out-of-memory). In these cases we counted the first reason during aggregation.

4.2.1 RQ1: Configurable Explicit Domain

Results In this question we analyze 6 different levels for MaxEnum with respect to 2 levels for the blocking factor Search. This algorithm is applicable only to the 535 CFA models, giving a total number of $(6 \cdot 2) \cdot 535 = 6420$ measurements, from which 3928 (61%) are successful.

The heatmap in Fig. 9 presents an overview of the results. Configurations are described by the levels of Search and MaxEnum. Categories are given by their name and the number of models, and the rightmost column is a summary of all categories. Each cell represents the number of successful measurements in a given category, along with the total execution time and peak memory consumption for successful measurements (rounded to 3 significant digits [18]). The background color of the cell indicates the success rate of the configurations, i.e., the percentage of successful measurements. The last row is the virtual best configuration, i.e., taking the result of the best configuration for each model individually.

Discussion It can be seen that traditional explicit-value analysis, i.e., configurations BFS_01* and DFS_01* perform well for the SV-COMP categories (locks, eca, ssh), but give poor performance on PLC models.

On the other end of the spectrum, configurations BFS_0 and DFS_0 enumerate all possible successors ($k = \infty$). This gives a poor success rate on certain SV-COMP categories, having integer variables with a theoretically infinite[19] domain. Note, that these configurations can still solve certain problems as they represent non-deterministic variables with the top value initially and only start enumerating possible values as soon as they appear in some expression (and are tracked explicitly). These configurations are more suitable for PLC models than traditional explicit-value analysis, because PLCs usually contain many Boolean input variables and it is often feasible to enumerate all possibilities to increase precision.

The advantage of our configurable approach is demonstrated by the configurations having 5, 10 or 50 for MaxEnum. These configurations give a good performance overall and a remarkably better success rate on category plc compared to traditional explicit-value analysis. Moreover, with $k >= 10$, configurations can solve a few more plc instances than with enumerating all possibilities. It can also be observed, that using an SMT solver for expressions that cannot be evaluated with simple heuristics (01) can improve success rate compared to not using a solver (01*) with 13 and 17 models for DFS and BFS respectively. Furthermore, it can be seen that BFS is consistently more effective than DFS for the same MaxEnum value. The overall best configuration in this analysis is BFS_50, but BFS_05 and BFS_10 closely follows.

An interesting further research direction would be to determine the optimal value for MaxEnum in advance, based on static properties of the input model or to adjust it dynamically during analysis.

[18] The relative standard deviation (also called the coefficient of variation) is the ratio of the standard deviation to the mean.

[19] SV-COMP contains C programs where integers have a fixed bit-width. However, in our current implementation we use SMT integers having an infinite domain. From a practical point of view, enumerating 2^{32} or 2^{64} states can be considered as infinite.

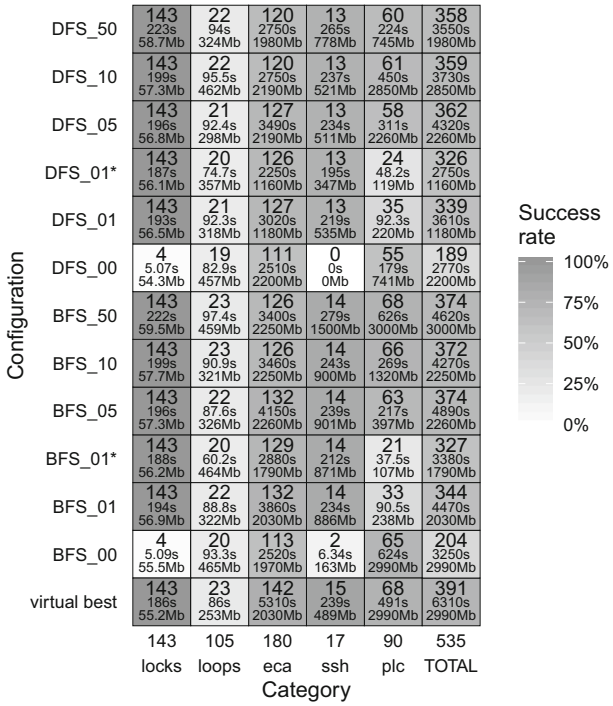


Fig. 9 Overview of the success rates, total execution time and peak memory consumption for RQ1

Summary. *Our configurable explicit domain can combine the advantages of traditional explicit-value analysis and explicit enumeration of successor states, giving a good performance overall in each category. Furthermore, although using an SMT solver requires more time, it increases precision and achieves a slightly higher success rate.*

4.2.2 RQ2: Error Location-Based Search

Results In this question we analyze 5 different levels for Search with respect to 3 levels for the blocking factor Domain. This algorithm is applicable only to the 535 CFA models, giving a total number of $(5 \cdot 3) \cdot 535 = 8025$ measurements, from which 6242 (78%) are successful. The heatmap in Fig. 10 presents an overview of the results. Configurations are described by the levels of Domain and Search.

Discussion The overall performance of configurations is similar, ranging from 416 to 447 successful measurements for PRED_* and 357 to 389 for EXPL. However, there are some interesting patterns in certain categories. The blocking factor (Domain) is dominant for the loops, ssh and plc categories: configurations with EXPL perform better for ssh and PRED_* is more effective for loops and plc.

The success rates for different search strategies within the same domain is quite similar with a few notable examples. Our purely error location-based strategy (ERR) yields a higher success rate in general compared to others. In contrast, our ERR_DFS combined strategy has a poor performance for eca models in the predicate domain. The supplementary report [42] includes separate plots for safe and unsafe benchmarks. This confirms that the advantage of

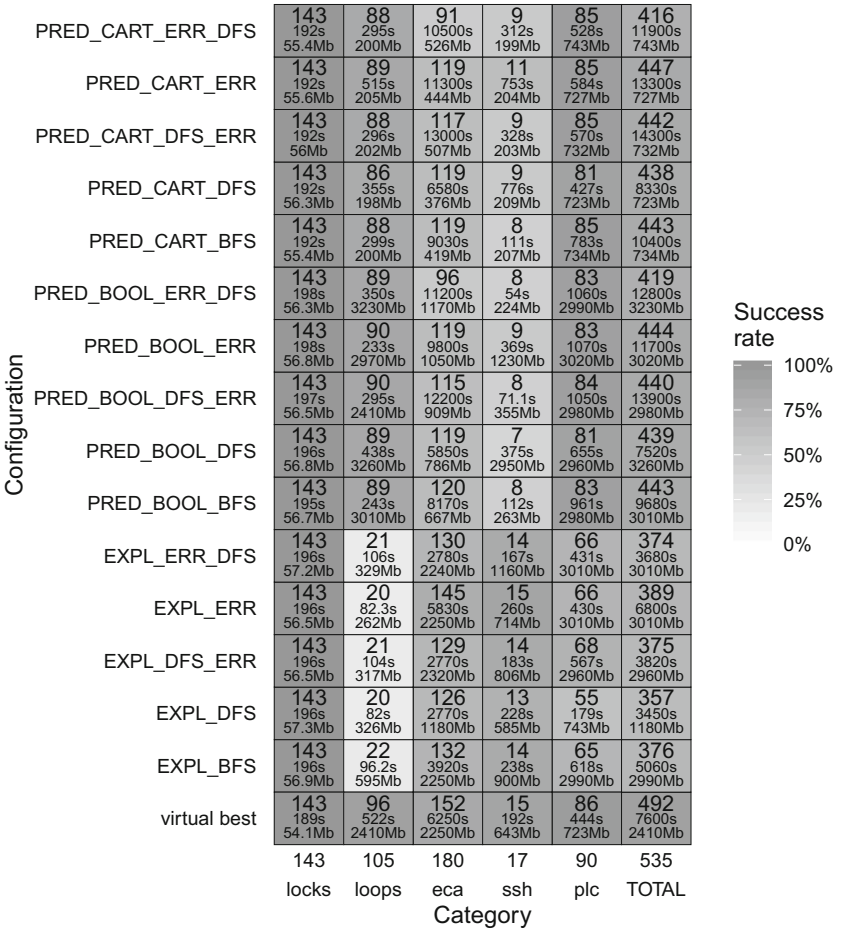


Fig. 10 Overview of the success rates, total execution time and peak memory consumption for RQ2

ERR strategies is more prominent for unsafe models and they are similar to others for safe instances.

A possible future research direction is to experiment with different combinations and weights for the strategies, possibly based on domain knowledge about the input models.

Summary. *Our error location-based search can yield improvement for certain models. However, our combined strategies that are efficient for artificial examples (Fig. 5) provide no remarkable improvement for real-world models.*

4.2.3 RQ3: Splitting Predicates

Results In this question we analyze 3 different levels for PredSplit and 3 levels for Domain. The levels of PredSplit determine how complex predicates obtained during refinement are treated, whereas the levels of Domain correspond to the way abstract states are formed from these predicates. These algorithms are applicable to all 835 models, giving a total number

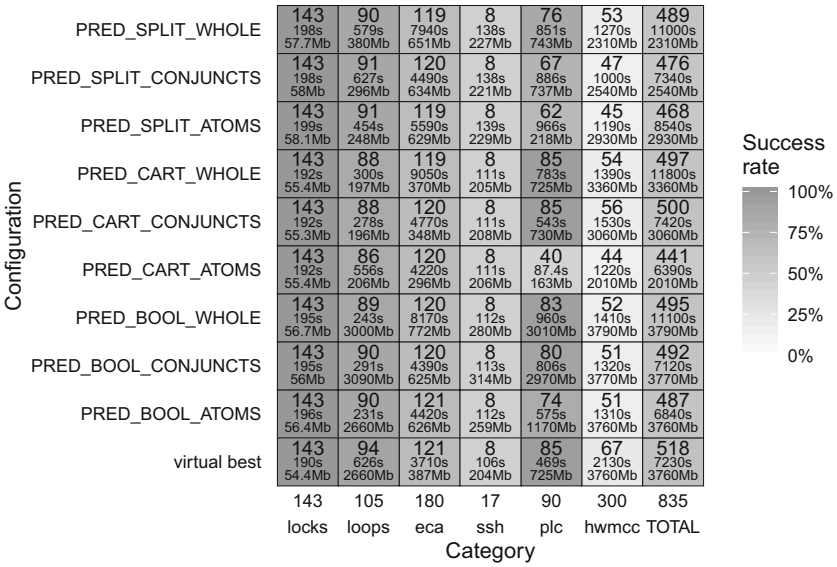


Fig. 11 Overview of the success rates, total execution time and peak memory consumption for RQ3

of $(3 \cdot 3) \cdot 835 = 7515$ measurements, from which 4345 (58%) are successful. The heatmap in Fig. 11 presents an overview of the results. Configurations are described by the levels of Domain and PredSplit.

Discussion It can be seen that the overall performance of the configurations mainly ranges from 468 to 500 successful measurements. An exception is the configuration PRED_CART_ATOMS having a remarkably poor performance (due to plc models). This can be attributed to the fact that if we split complex formulas to their atoms and use Cartesian abstraction, we will only be able to represent conjunctions of atoms, but no disjunctions. Similarly for hardware models, splitting to atoms only works well with Boolean abstraction.

On the other hand, splitting into conjuncts is especially favorable with Cartesian abstraction, making PRED_CART_CONJUNCTS the most successful configuration. Although it can only solve 3 more models than the second best, the total execution time is much lower (7420 s compared to 11,800 s).

It can also be observed that our PRED_SPLIT domain has a slightly worse performance than PRED_BOOL. Hence, it is not worth splitting disjuncts of Boolean predicate abstraction into separate states. A possible reason is that a large disjunction might be simplified to a simpler formula, e.g., $(A \wedge B) \vee (A \wedge \neg B)$ is only A. When the disjunction is kept as a whole, modern SMT solvers might be able to perform such reductions. However, disjuncts alone usually cannot be simplified.

The differences between configurations could be of greater magnitude if more disjunctions occurred (e.g., due to pointer-aliasing encoding or large-block encoding [10]). In our case we observed that disjunctions mostly come from the encoding itself in hardware models and PLC programs or from the interpolants in SV-COMP programs.

Currently, we do not normalize the formulas to CNF before splitting. An equivalent formula in CNF can have an exponential size compared to the original [22], but an interesting further direction would be to experiment with equisatisfiable encodings [62].

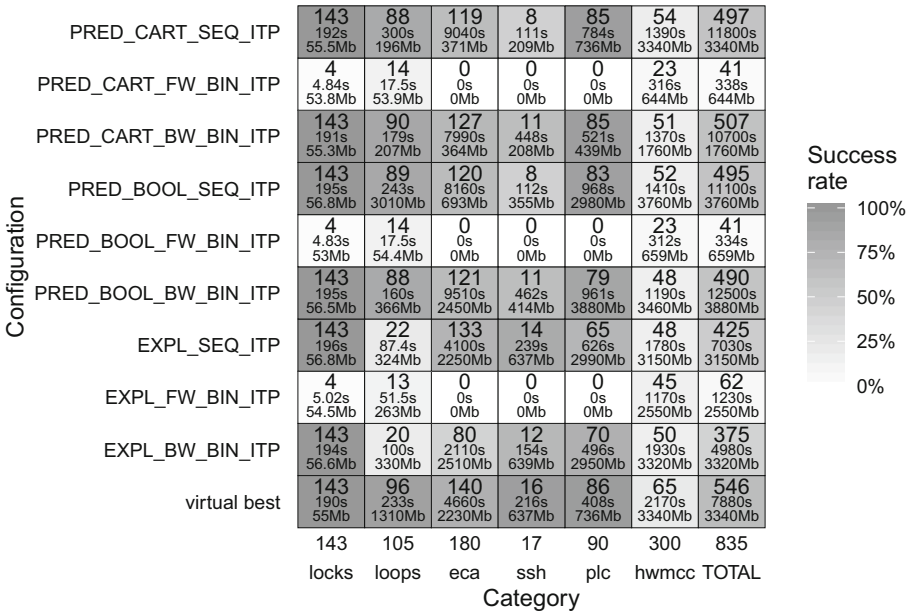


Fig. 12 Overview of the success rates, total execution time and peak memory consumption for RQ4

Summary. *For the best performance, complex predicates should be treated as a whole, or split to conjuncts, but not split to atoms. Furthermore, states in Boolean predicate abstraction should also be kept as a single state.*

4.2.4 RQ4: Backward Binary Interpolation

Results In this question we analyze 3 different levels for Refinement with respect to 3 levels for the blocking factor Domain. These algorithms are applicable to all 835 models, giving a total number of $(3 \cdot 3) \cdot 835 = 7515$ measurements, from which 2933 (39%) are successful. The heatmap in Fig. 12 presents an overview of the results. Configurations are described by the levels of Domain and Refinement.

Discussion It can be seen clearly that forward binary interpolation (FW_BIN_ITP) fails for almost every CFA model, except for a few (mainly unsafe) instances in categories locks and loops. For hardware models, it is slightly more effective in the EXPL domain.

Sequence interpolation (SEQ_ITP) and our backward binary interpolation approach (BW_BIN_ITP) have similar success rates. The former one is more successful in the PRED_BOOL and EXPL domains, while the latter is effective in the PRED_CART domain (making it the best overall configuration). The differences are however, only remarkable in the EXPL domain, where BW_BIN_ITP has a low success rate on eca models. Furthermore, BW_BIN_ITP in the PRED_CART domain has around half the peak memory consumption than SEQ_ITP in any domain.

An interesting further direction would be to involve the granularity of the precision (local/global) as a blocking factor, as for BW_BIN_ITP a local precision could involve more refinement steps.

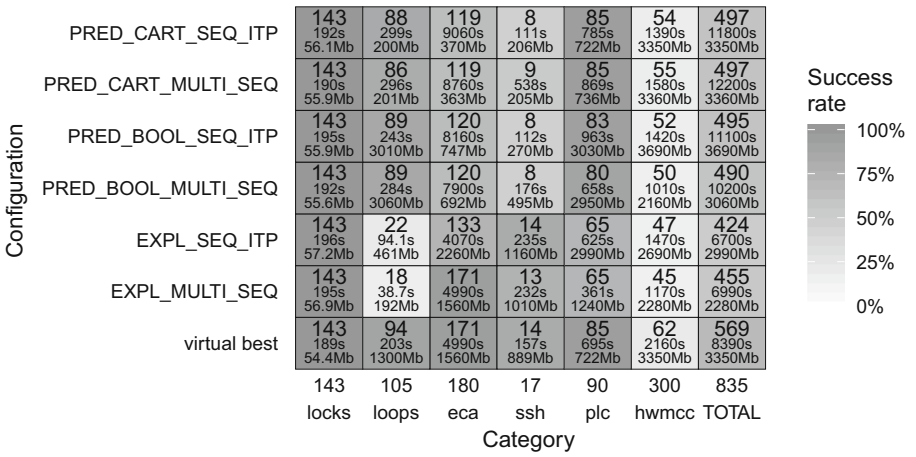


Fig. 13 Overview of the success rates, total execution time and peak memory consumption for RQ5

Summary. *Our backward binary interpolation strategy clearly outperforms forward interpolation and has similar performance to sequence interpolation, in some cases even outperforming it.*

4.2.5 RQ5: Multiple Counterexamples for Refinement

Results In this question we analyze 2 different levels for Refinement with respect to 3 levels for the blocking factor Domain. We are interested whether collecting and refining multiple counterexamples at once (MULTI_SEQ) can yield better performance than using a single path (SEQ_ITP). These algorithms are applicable to all 835 models, giving a total number of $(2 \cdot 3) \cdot 835 = 5010$ measurements, from which 2858 (57%) are successful. The heatmap in Fig. 13 presents an overview of the results. Configurations are described by the levels of Domain and Refinement.

Discussion It can be seen that the blocking factor (Domain) is dominant for the eca, loops, ssh and plc categories. Configurations with PRED_* domain are more successful for loops and plc models, whereas EXPL is more effective for categories eca and ssh.

The difference between using a single or multiple counterexamples within the PRED_* domains is not remarkable, ranging from 490 to 497 verified models. However, using multiple counterexamples is clearly more effective in the EXPL domain due to the eca category. This can be attributed to the fact that these models have the largest cyclomatic complexity, enabling to utilize the full power of our strategy that uses multiple counterexamples. Furthermore, there are 39 models that only EXPL_MULTI_SEQ could verify.

As a possible future direction, it would be interesting to experiment with different refinements strategies (e.g., backward binary). Moreover, information from multiple counterexamples could be utilized in more detail than just simply selecting a minimal set required to eliminate all spurious behavior.

Summary. *Our strategy for using multiple counterexamples can yield a remarkably better performance in the explicit domain for complex models.*

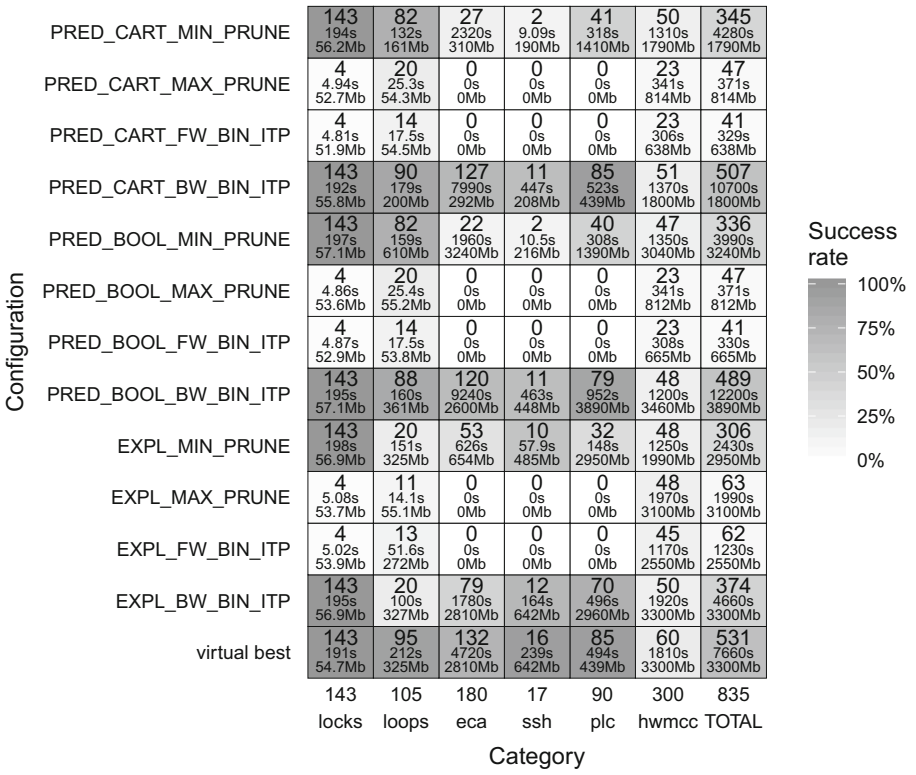


Fig. 14 Overview of the success rates, total execution time and peak memory consumption for RQ6

4.2.6 RQ6: Multiple Refinements for a Counterexample

Results In this question we analyze 4 different levels for Refinement with respect to 3 levels for the blocking factor Domain. We are interested whether combinations of BW_BIN_ITP and FW_BIN_ITP can yield better performance. These algorithms are applicable to all 835 models, giving a total number of $(4 \cdot 3) \cdot 835 = 10020$ measurements, from which 2658 (27%) are successful. The heatmap in Fig. 14 presents an overview of the results. Configurations are described by the levels of Domain and Refinement.

Discussion It can be seen that BW_BIN_ITP is successful overall, while FW_BIN_ITP gives rather poor performance. Interestingly, our combined strategy MAX_PRUNE is close to FW_BIN_ITP, whereas MIN_PRUNE is more successful, but still less effective than BW_BIN_ITP.

A possible further research direction would be to combine BW_BIN_ITP with the effective SEQ_ITP approach. In this case however, combining based on the prune index may not work since sequence interpolation usually refines more states along the counterexample.

Summary. *Combining an effective refinement approach (BW_BIN_ITP) with a rather unsuccessful one (FW_BIN_ITP) based on the prune index could not improve performance.*

Table 4 Configurations of THETA compared against other tools

Configuration name	Domain	MaxEnum	PredSplit	Refinement	Search	PrecGran.
theta-pred-seq	PRED_CART		ATOMS	SEQ_ITP	BFS	GLOBAL
theta-expl-seq	EXPL	1*		SEQ_ITP	BFS	GLOBAL
theta-pred-bw	PRED_CART		WHOLE	BW_BIN_ITP	ERR	GLOBAL
theta-expl-multiseq	EXPL	1		MULTI_SEQ	ERR	GLOBAL

4.3 Comparison to Other Tools

In order to provide a baseline for the research questions in the previous section, we compare THETA to other tools. Unfortunately, we did not have the computing resources to run all measurements in a common environment. Therefore, we took the raw data[20] from SV-COMP 2018 and filtered to the models that THETA can handle. Furthermore, we executed four configurations of THETA in a similar environment to SV-COMP, using 900 s time limit and 15 GB memory limit. Note, that these are larger limits compared to the research questions. The hardware machines we used for THETA had weaker CPUs than the ones at SV-COMP, giving us a slight disadvantage. However, our purpose was not to give an exact comparison, but rather just to show that THETA is competitive with respect to the state of the art. Therefore, we omit time and memory measurements and only indicate the number of successful executions.

Currently, the frontend of THETA produces a different verification task for each assertion in a C program due to slicing. Therefore, we selected those models from loops that contain a single assertion. In category locks, there is also a single assertion, reachable by multiple labels that we used as slicing criteria for the research questions. For the current measurements we create a single task to be able to compare to other tools. The other categories (eca and ssh) contain one assertion per file.

Configurations of THETA are summarized in Table 4. The first two configurations (theta-pred-seq and theta-expl-seq) implement already existing strategies, while the latter two include some of our new approaches that performed well in the research questions. For example, theta-pred-bw performs backward binary interpolation (RQ4, RQ6) and keeps predicates as a whole (RQ3), while theta-expl-multiseq uses an SMT solver to evaluate unknown expressions (up to a limit of one) (RQ1) and performs refinement based on multiple counterexamples (RQ5). Furthermore, the latter two configurations use the error location-based search strategy (RQ2).

Results can be seen in Fig. 15, where each cell indicates the success rate of a tool (or configuration) in a given category. The last column is a summary of all categories. Empty spaces indicate that a tool did not compete in a certain category.

Based on the competition reports [8,9] the tools CPA- BAM- BNB, CPA- BAM- SLICING, CPA- SEQ, INTERPCHECKER and SKINK are the most closely related to THETA as they also work with CEGAR and ARG-based analysis. The tools UAUTOMIZER, UKOJAK and UTAIPAN also employ CEGAR, but their analysis is based on automata [9]. Other tools are mainly based on bounded model checking, k-induction or symbolic execution.

The models represent a small subset of SV-COMP benchmarks and many of them belong to the simpler instances. However, the success rates already have a great variance, ranging from 70 to 248 (out of 259) for those tools that competed in all of the categories. Configurations for THETA perform well in this comparison, verifying 176 to 215 tasks. The takeaway message of this comparison is that although the C frontend of THETA is limited, our implementation is

[20] https://sv-comp.sosy-lab.org/2018/results/results-verified/All-Raw.zip.

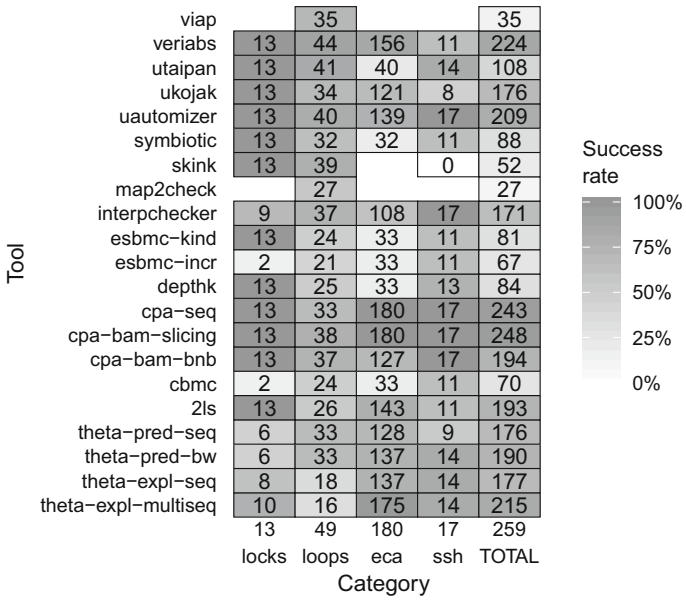


Fig. 15 Overview of the success rates for various THETA configurations and other tools from SV-COMP 2018

still competitive with respect to state-of-the-art tools and can serve as a baseline for evaluating new algorithms.

5 Related Work

In this section, we present related work to our framework in general, to our algorithmic contributions and to our experimental evaluation.

General Abstraction and CEGAR-based methods are widely used for model checking software [9], implemented by several tools, e.g., SLAM [6], BLAST [12], SATABS [35], IMPACT [56], WOLVERINE [51]. The most closely related are the frameworks CPACHECKER [14] and UFO [2] that support configurability based on abstract domains and refinement strategies. These tools however, only target software models in contrast to THETA, which also supports transition systems and timed automata [60,61]. The LTSMIN [49] tool and the ULTIMATE framework[21] also support different kind of models and algorithms, but their primary focus is on symbolic methods and automata respectively instead of abstraction.

Configurable explicit domain The transfer function of our configurable explicit domain (Sect. 3.1.1) can be considered as a generalization of explicit-value analysis [15], which always enumerates at most one successor state. The visible/invisible variables approach [34] is similar to the other end of the spectrum, enumerating all possible successors ($k = \infty$) for transition systems defined by partitioned transition relations.

Error location-based search Our error location-based search (Sect. 3.1.2) is basically an A* search [44], for which we adjusted the cost function to the domain of software model checking. We use the depth as the cost of the current path, and the distance from the error

[21] https://ultimate.informatik.uni-freiburg.de/.

location as the estimated remaining cost. State space traversal strategies have also been discussed in the context of explicit model checking and abstract interpretation [21]. The main focus of these approaches is to reach a fixpoint by identifying widening points and iterations strategies (e.g., based on loops). In contrast, the goal of our method is to guide the search towards an abstract state with a specific location. However, some ideas of the existing approaches could also be combined with our method, e.g., process loops first and then head towards the error location. Considering the syntactical distance in the CFA has also been proven effective for achieving higher coverage in dynamic test generation tools such as CREST [24] and KLEE [27].

Splitting predicates Different variants of the predicate domain (including Cartesian and Boolean) have been studied before [5]. Beyer and Wendler conclude, that while Boolean abstraction is more precise than Cartesian, it is also more expensive (especially with single-block encoding) [19]. There were also works on the compact representation of predicates in Boolean predicate abstraction using SMT techniques [52] and BDDs [28]. Our splitting domain (Sect. 3.1.3) works similarly to the approach of Brückner et al. [23]. However, we do not only split states during refinement but during construction of the abstract state space. The first approach that uses interpolation in the context of predicate abstraction extracts atomic formulas from the interpolant [46], which might lead to a loss of precision when combined with Cartesian abstraction [19]. The lazy abstraction with interpolants (IMPACT) algorithm [56] keeps interpolants as a whole [19], but it is a different approach than the predicate abstraction presented in our paper. To the best of our knowledge, splitting complex interpolants into smaller subsets (Sect. 3.1.3) has not yet been studied systematically in the context of predicate abstraction.

Backward binary interpolation The most closely related to our backward binary interpolation (Sect. 3.2.1) is the approach of Brückner et al. [23]. They first calculate a minimal subpath of the counterexample that is spurious, i.e., it is feasible, but extending it in any direction makes it infeasible. Then, they use a binary interpolant to refine the last state of this subpath. In contrast, our approach can be considered as refining the state before the first state of the subpath. Henzinger et al. [46] also use binary interpolation, but they calculate an interpolant for each location in the counterexample from the same proof. The counterexample minimization approach of Alberti et al. [3] is also similar to ours as they consider the shortest infeasible suffix of the counterexample. However, their approach is defined in the context of lazy abstraction with interpolants (IMPACT [56]) and they compute an interpolant for each location. Moreover they perform a backward unwinding whereas we do a forward search and then proceed backwards only in the counterexample. This also highlights the possibility to experiment with different combinations of forward/backward search and interpolation.

The NEWTON approach [7] performs a forward search, but uses the strongest postcondition operator instead of interpolants. However, counterexamples are generalized with symbolic variables, which could be combined with the forward or backward interpolation strategies. A different variant [38] of the NEWTON approach performs a backward search during refinement, but uses the weakest precondition operator combined with unsatisfiability cores instead of Craig interpolants. The approach of SLAM [4] also performs backward check on a counterexample but only up to a bounded depth. Then, they use Craig interpolation at each step to weaken the predicates coming from the weakest preconditions.

Cabodi et al. [26] compare the iterative application of traditional forward interpolation to sequence interpolation. They come to a similar conclusion as us, namely that while sequence interpolation performs refinement at once, traditional interpolation can sometimes have a better performance (due to convergence at shorter depths in their case).

Multiple counterexamples for refinement Most algorithms in the literature use a single counterexample for refinement. The UFO tool [2] includes DAG interpolants [1] that refine all counterexamples at once. Our approach for multiple counterexamples (Sect. 3.2.2) calculates a separate interpolant for each path and minimizes and merges the results. While computing a DAG interpolant seems more efficient that a series of independent interpolations, our approach could also have various advantages. First, different paths could use different refinement procedures (e.g., backward vs. sequence). Second, it would also be possible to do multiple refinements for each path (e.g., by different interpolation approaches or by multiple prefixes [17]), take the "best" one and merge it with interpolants from the other counterexamples.

The global refinement algorithm from the thesis of Löwe [54] computes a tree of interpolants using a series of interpolations (by reusing common prefixes). Our approach could also gain performance from reusing common prefixes (with the incremental API of solvers). However, our approach has the advantage that each counterexample can use any kind of refinement procedure (e.g., backward interpolation). We believe that this is beneficial in the context of a global precision, where the predicates or variables from the interpolants are merged and used globally.

Multiple refinements for a counterexample Beyer et al. [17] calculate multiple prefixes for the same counterexample to enable selection from different refinements. Furthermore, they also define some basic strategies for selecting the possibly best refinement [16]. Our approach (Sect. 3.2.3) also uses a single counterexample and the heuristic using the prune index is essentially the same as their "depth of pivot location" strategy. However, instead of calculating prefixes, our approach selects from different interpolants for the same counterexample. The two approaches therefore, can be considered orthogonal: it would also be possible to calculate different interpolants for multiple prefixes.

ULTIMATE AUTOMIZER [45] also works with a portfolio of refiners, including Craig interpolation, unsatisfiable cores, various SMT solvers and different ways to abstract a trace. They use a single measure for the quality of an interpolation, namely checking if the interpolant constitutes a Floyd-Hoare annotation. In principle, our approaches could be added as new strategies to the portfolio of ULTIMATE AUTOMIZER, and their methods could also extend the portfolio of THETA.

Combining multiple refinements has also been studied in the context of the IC3/PDR approach. Cimatti and Griggio [30] propose a hybrid IC3 algorithm, that first calculates a proof-based interpolant (similar to sequence interpolants in our work). If this interpolant contains too many clauses, they switch to the interpolation strategy of the original IC3 algorithm, which is more expensive, but yields fewer clauses typically. Hoder and Bjørner [48] also calculate a proof-based interpolant and use it as conflict clauses in order to support linear real arithmetic in IC3.

Experimental evaluation There are many works in the literature that focus on experimental evaluation and comparison of model checking algorithms [11,19,36,37]. However, they usually focus on a certain domain (e.g., SV-COMP). Our framework allows us to experiment with models from different domains, including SV-COMP, HWMCC and PLC codes as well. Furthermore, our experiments compare parameters and configurations of a single algorithm (CEGAR), yielding a finer granularity as opposed to most experiments in the literature, where different tools or different algorithms are compared. This allows us to assess the effectiveness and efficiency of our lower level strategies.

6 Conclusions

In our paper, we presented six new heuristics and variations of existing strategies to improve various aspects of the CEGAR algorithm, including both abstraction and refinement. For abstraction, we introduced a configurable explicit domain, an error location-based search strategy and the splitting of complex predicates. On the side of refinement, we proposed a novel backward reachability-based interpolation strategy, an approach for using multiple counterexamples for refinement, and a selection method from multiple refinements for the same counterexample.

We implemented our new contributions in the open source, configurable model checking framework THETA along with state-of-the-art algorithms. This allowed us to conduct an experiment on various input models from diverse sources, including SV-COMP, HWMCC and CERN.

Our results show that the configurable explicit domain can combine the advantages of traditional explicit-value analysis and the enumeration of states. The error location-based search can yield better results for certain models, but combining it with DFS gives no remarkable improvement. Splitting predicates reveals that complex formulas should be treated as a whole, or split to their conjuncts.

Our backward binary interpolation clearly outperforms forward interpolation and has a similar performance to sequence interpolation. Our strategy for using multiple counterexamples during refinement can yield a remarkable improvement in the explicit domain for the most complex models. Finally, combining the effective backward refinement with forward interpolation based on pruning index cannot improve performance. However, other refinement strategies combined differently could still be successful, which is a topic of further research.

We can conclude that our new contributions perform well in general compared to existing approaches. Furthermore, we highlighted certain domains and categories of models where effectiveness and efficiency of the CEGAR approach remarkably increased.

Acknowledgements We would like to thank the reviewers for their valuable feedback.

Funding Open access funding provided by Budapest University of Technology and Economics (BME). This work was partially supported by the BME-Artificial Intelligence FIKP Grant of EMMI (BME FIKP-MI/SC) and by the National Research, Development and Innovation Fund (TUDFO/51757/2019-ITM, Thematic Excellence Program).

References

1. Albarghouthi, A.: Software verification with program-graph interpolation and abstraction. Ph.D. thesis, University of Toronto (2015)
2. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: a framework for abstraction- and interpolation-based software verification. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 7358, pp. 672–678. Springer (2012)
3. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: An extension of lazy abstraction with interpolation for programs with arrays. Form. Methods Syst. Des. **45**(1), 63–109 (2014). https://doi.org/10.1007/s10703-014-0209-9

4. Ball, T.: Formalizing counterexample-driven refinement with weakest preconditions. Tech. Rep. MSR-TR-2004-134, Microsoft Research (2004)
5. Ball, T., Podelski, A., Rajamani, S.: Boolean and Cartesian abstraction for model checking C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 2031, pp. 268–283. Springer (2001)
6. Ball, T., Rajamani, S.: The SLAM toolkit. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 2102, pp. 260–264. Springer (2001)
7. Ball, T., Rajamani, S.: Generating abstract explanations of spurious counterexamples in C programs. Tech. Rep. MSR-TR-2002-09, Microsoft Research (2002)
8. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 9636, pp. 887–904. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55
9. Beyer, D.: Software verification with validation of results. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 10206, pp. 331–349. Springer (2017)
10. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proceedings of the 2009 Conference on Formal Methods in Computer-Aided Design, pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147
11. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reason. **60**(3), 299–335 (2018)
12. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transf. **9**(5), 505–525 (2007)
13. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 4590, pp. 504–518. Springer (2007)
14. Beyer, D., Keremoglu, M.E.: CPACHECKER: a tool for configurable software verification. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011)
15. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol. 7793, pp. 146–162. Springer (2013)
16. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Model Checking Software, Lecture Notes in Computer Science, vol. 9232, pp. 20–38. Springer (2015)
17. Beyer, D., Löwe, S., Wendler, P.: Sliced path prefixes: an effective method to enable refinement selection. In: Formal Techniques for Distributed Objects, Components, and Systems, Lecture Notes in Computer Science, vol. 9039, pp. 228–243. Springer (2015)
18. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**, 1–29 (2017). Online first
19. Beyer, D., Wendler, P.: Algorithms for software model checking: predicate abstraction vs. Impact. In: Proceedings of the Formal Methods in Computer Aided Design, pp. 106–113. IEEE (2012)
20. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability. IOS press, Amsterdam (2009)
21. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Formal Methods in Programming and Their Applications, Lecture Notes in Computer Science, vol. 735, pp. 128–141. Springer (1993)
22. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, Berlin (2007)
23. Brückner, I., Dräger, K., Finkbeiner, B., Wehrheim, H.: Slicing abstractions. In: International Symposium on Fundamentals of Software Engineering, Lecture Notes in Computer Science, vol. 4767, pp. 17–32. Springer (2007)
24. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 443–446. IEEE (2008). https://doi.org/10.1109/ASE.2008.69
25. Cabodi, G., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S., Vendraminetto, D., Biere, A., Heljanko, K., Baumgartner, J.: Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks. J. Satisf. Boolean Model. Comput. **9**, 135–172 (2016)
26. Cabodi, G., Nocco, S., Quer, S.: Interpolation sequences revisited. In: 2011 Design, Automation and Test in Europe, pp. 1–6. IEEE (2011). https://doi.org/10.1109/DATE.2011.5763056
27. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 209–224. USENIX Association (2008)

28. Cavada, R., Cimatti, A., Franzén, A., Kalyanasundaram, K., Roveri, M., Shyamasundar, R.K.: Computing predicate abstractions by integrating BDDs and SMT solvers. In: Proceedings of the Formal Methods in Computer Aided Design, pp. 69–76. IEEE (2007). https://doi.org/10.1109/FMCAD.2007.18
29. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Model Checking Software, Lecture Notes in Computer Science, vol. 7385, pp. 248–254. Springer (2012)
30. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 10806, pp. 277–293. Springer (2012)
31. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
32. Clarke, E., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
33. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
34. Clarke, E., Gupta, A., Strichman, O.: SAT-based counterexample-guided abstraction refinement. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **23**(7), 1113–1123 (2004)
35. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 3440, pp. 570–574. Springer (2005)
36. Czech, M., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Predicting rankings of software verification tools. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics, pp. 23–26. ACM (2017)
37. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. Form. Methods Syst. Des. **50**(2), 289–316 (2017)
38. Dietsch, D., Heizmann, M., Musa, B., Nutz, A., Podelski, A.: Craig vs. Newton in software model checking. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 487–497. ACM (2017). https://doi.org/10.1145/3106237.3106307
39. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, vol. 7148, pp. 186–201. Springer (2012)
40. Fernández Adiego, B., Darvas, D., Blanco Viñuela, E., Tournier, J.C., Bliudze, S., Blech, J.O., González Suárez, V.M.: Applying model checking to industrial-sized PLC programs. IEEE Trans. Ind. Inform. **11**(6), 1400–1410 (2015)
41. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 1254, pp. 72–83. Springer (1997)
42. Hajdu, Á., Micskei, Z.: Supplementary material for the paper "Efficient strategies for CEGAR-based model checking" (2018). https://doi.org/10.5281/zenodo.1252784
43. Hajdu, Á., Tóth, T., Vörös, A., Majzik, I.: A configurable CEGAR framework with interpolation-based refinements. In: Formal Techniques for Distributed Objects, Components and Systems, Lecture Notes in Computer Science, vol. 9688, pp. 158–174. Springer (2016)
44. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968). https://doi.org/10.1109/TSSC.1968.300136
45. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 10806, pp. 447–451. Springer (2018)
46. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 232–244. ACM (2004)
47. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 58–70. ACM (2002)
48. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Theory and Applications of Satisfiability Testing—SAT 2012, Lecture Notes in Computer Science, vol. 7317, pp. 157–171. Springer (2012)
49. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer (2015)
50. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amparore, E., Beccuti, M., Berthomieu, B., Ciardo, G., Dal Zilio, S., Liebke, T., Li, S., Meijer, J., Miner, A., Srba, J., Thierry-Mieg, Y., van de Pol, J., van Dirk, T., Wolf, K.: Complete results for the 2019 edition of the model checking contest. (2019) http://mcc.lip6.fr/2019/results.php
51. Kroening, D., Weissenbacher, G.: Interpolation-based software verification with Wolverine. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 6806, pp. 573–578. Springer (2011)

52. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 4144, pp. 424–437. Springer (2006)
53. Leucker, M., Markin, G., Neuhäußer, M.: A new refinement strategy for CEGAR-based industrial model checking. In: Hardware and Software: Verification and Testing, Lecture Notes in Computer Science, vol. 9434, pp. 155–170. Springer (2015). https://doi.org/10.1007/978-3-319-26287-1_10
54. Löwe, S.: Effective approaches to abstraction refinement for automatic software verification. Ph.D. thesis, University of Passau (2017)
55. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 3440, pp. 1–12. Springer (2005)
56. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 4144, pp. 123–136. Springer (2006)
57. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
58. R Core Team: R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria (2017). https://www.R-project.org/
59. Sallai, Gy., Hajdu, Á., Tóth, T., Micskei, Z.: Towards evaluating size reduction techniques for software model checking. In: Proceedings of the 5th International Workshop on Verification and Program Transformation, EPTCS, vol. 253, pp. 75–91. Open Publishing Association (2017)
60. Tóth, T., Hajdu, Á., Vörös, A., Micskei, Z., Majzik, I.: THETA: a framework for abstraction refinement-based model checking. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design, pp. 176–179. FMCAD inc. (2017)
61. Tóth, T., Majzik, I.: Lazy reachability checking for timed automata with discrete variables. In: Model Checking Software, Lecture Notes in Computer Science, vol. 10869, pp. 235–254. Springer (2018)
62. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning, Symbolic Computation, pp. 466–483. Springer (1983)
63. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Formal Methods in Computer-Aided Design, pp. 1–8. IEEE (2009)
64. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin (2012)