Advancing Software Model Checking Beyond Linear Arithmetic Theories

Ahmed Mahdi[1(✉)], Karsten Scheibler[2], Felix Neubauer[2], Martin Fränzle[1], and Bernd Becker[2]

[1] Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
{mahdi,fraenzle}@informatik.uni-oldenburg.de

[2] Albert-Ludwigs-Universität Freiburg, Freiburg im Breisgau, Germany
{scheibler,neubauef,becker}@informatik.uni-freiburg.de

Abstract. Motivated by the practical need for verifying embedded control programs involving linear, polynomial, and transcendental arithmetics, we demonstrate in this paper a CEGAR technique addressing reachability checking over that rich fragment of arithmetics. In contrast to previous approaches, it is neither based on bit-blasting of floating-point implementations nor confined to decidable fragments of real arithmetic, namely linear or polynomial arithmetic. Its CEGAR loop is based on Craig interpolation within the iSAT3 SMT solver, which employs (abstract) conflict-driven clause learning (CDCL) over interval domains together with interval constraint propagation. As usual, the interpolants thus obtained on spurious counterexamples are used to subsequently refine the abstraction, yet in contrast to manipulating and refining the state set of a discrete-state abstraction, we propose a novel technique for refining the abstraction, where we annotate the abstract model's transitions with side-conditions summarizing their effect. We exploit this for implementing case-based reasoning based on assumption-commitment predicates extracted from the stepwise interpolants in a lazy abstraction mechanism. We implemented our approach within iSAT3 and demonstrate its effectiveness by verifying several benchmarks.

1 Introduction

The wide-spread use of embedded control programs involving linear, polynomial, and transcendental arithmetic provokes a quest for corresponding verification methods. A crucial technique here is the automatic verification of reachability properties in such programs, as many problems can be reduced to it and as it in particular provides a method for detecting unreachable code fragments, a.k.a. *dead code*, in such programs. The latter is an industrial requirement, as various pertinent standards for embedded system development either demand adequate handling of dead code during testing or even bar it altogether, like DO-178C, DO-278A, or ISO/IEC PDTR 24772.

This work was supported by the German Research Council (DFG) as part of SFB/TR 14 AVACS (http://www.avacs.org).

DOI: 10.1007/978-3-319-49052-6_12

The set of verification tools being able to address reachability properties in arithmetic programs involving such a rich fragment of arithmetic is confined. Tools manipulating real-valued rather than machine arithmetic tend to be limited to linear or at most polynomial arithmetic due to the obvious decidability issues arising in richer fragments of real arithmetic; tools resorting to bit-blasting, like C Bounded Model Checking (CBMC) [1], tend to adopt the very same restrictions for complexity reasons. Abstract interpretation [2] could in principle easily go beyond, but then suffers from inexactness since the geometry of sets of numbers representable by its usual lattices and the graphs of the monotonic functions over these can only provide coarse overapproximations.

Our Contributions: Within this paper, **(1)** we are trying to overcome this deficiency by a combination of techniques forming a viable counterexample guided abstraction refinement (CEGAR) loop: we exploit Craig interpolation [3] in the interval constraint-propagation based satisfiability modulo theory (SMT) solving algorithm iSAT [4,5] in order to extract reasons for an abstract counterexample being spurious, leading to goal-directed abstraction refinement as in CEGAR [6]. **(2)** In contrast to the usual scheme manipulating and refining the state set of a discrete-state abstraction by splitting cases [7,8] or splitting paths depending on automata [9], we annotate the abstract model's transitions with side-conditions summarizing their effect. Due to a tight integration of checking the abstraction into the SMT solver iSAT3, we can exploit these annotations for implementing case-based reasoning based on assumption-commitment predicates extracted from the stepwise interpolants in a lazy abstraction mechanism, thereby eliminating all spurious counterexamples that share a local reason of being spurious at one transition by using *one predicative expression.*

We implemented our approach within iSAT3 and demonstrate its effectiveness by verifying several benchmarks. We do in particular compare our approach to a model-checking technique exploiting Craig interpolants over the same fragment of arithmetics as an overapproximation of reachable state sets [5], as originally suggested for the finite-state case by McMillan [10], i.e., implementing approximate reach-set computation rather than CEGAR. The benchmarks indicate superior performance of the new CEGAR approach on non-linear benchmarks.

Related Work: To the authors' best knowledge, this is the first attempt to verify programs which may involve transcendental functions by using CEGAR. Most previous work is confined only to linear arithmetics or polynomials [8,11–15], where our work supports richer arithmetic theories, namely transcendental functions. Although our approach is similar with IMPACT [8], WHALE [16] and Ultimate Automizer [17] solvers regarding the usage of interpolants as necessary predicates in refining the abstraction, there are fundamental differences in the learning procedure. While IMPACT and WHALE explicitly split the states after each learning, Ultimate Automizer which is based on $\omega-$automata in learning reasons of spurious counterexamples [18], applies trace abstraction where interpolants are used to construct an automaton that accepts a whole set of infeasible traces and on the same time overapproximates the set of possible traces of the safe program. In contrast to that, our refinement procedure adds neither transitions nor states to the abstraction, but we do annotate the abstract program transitions with necessary assumption-commitment conditions that eliminate the spurious counterexamples.

2 Preliminaries

We use and suitably adapt several existing concepts. A *control flow graph* (CFG) is a cyclic graph representation of all paths that might be traversed during program execution. In our context, we attach code effect to edges rather than nodes of the CFG. i.e., each edge comes with a set of constraints and assignments pertaining to execution of the edge. Formally, constraints and assignments are defined as follows:

Definition 1 (Assignments and constraints). *Let V be a set of integer and real variables, with typical element v, B be a set of boolean variables, with typical element b, and C be a set of constants over rationals, with typical element c.*

- *The set $\Psi(V,B)$ of assignments over integer, real, and boolean variables with typical element ψ is defined by the following syntax:*

$$\begin{aligned} \psi &::= v := aterm \mid b := bterm \\ aterm &::= uaop\ v \mid v\ baop\ v \mid v\ baop\ c \mid c \mid v \\ bterm &::= ubop\ b \mid b\ bbop\ b \mid b \\ uaop &::= - \mid sin \mid cos \mid exp \mid abs \mid ... \\ baop &::= + \mid - \mid \cdot \mid ... \\ ubop &::= \neg \\ bbop &::= \wedge \mid \vee \mid \oplus \mid ... \end{aligned}$$

  By $\boldsymbol{\psi}$ we denote a finite list of assignments on integer, real, and boolean variables, $\boldsymbol{\psi} = \langle \psi_1, ..., \psi_n \rangle$ where $n \in \mathbb{N}_{\geq 0}$. We use $\Psi(V,B)^*$ to denote the set of lists of assignments and $\langle\ \rangle$ to denote the empty list of assignments.
- *The set $\Phi(V,B)$ of constraints over integer, real, and boolean variables with typical element ϕ is defined by the following syntax:*

$$\begin{aligned} \phi &::= atom \mid ubop\ atom \mid atom\ bbop\ atom \\ atom &::= theory\_atom \mid bool \\ theory\_atom &::= comp \mid simple\_bound \\ comp &::= term\ lop\ c \mid term\ lop\ v \\ simple\_bound &::= v\ lop\ c \\ bool &::= b \mid ubop\ b \mid b\ bbop\ b \\ term &::= uaop\ v \mid v\ baop\ v \mid v\ baop\ c \\ lop &::= < \mid \leq \mid = \mid > \mid \geq \end{aligned}$$

  where uaop, baop, ubop and bbop are defined above.

We assume that there is a well-defined valuation mapping $\nu : V \cup B \rightarrow \mathcal{D}(V) \cup \mathcal{D}(B)$ that assigns to each assigned variable a value from its associated domain. Also, we assume that there is a satisfaction relation $\models \subseteq (V \cup B \rightarrow \mathcal{D}(V) \cup \mathcal{D}(B)) \times \Phi(V, B)$ and in case of arithmetic or boolean variables we write $\nu \models \phi$ iff $\nu|_{V \cup B} \models \phi$. The modification of a valuation ν under a finite list of assignment $\boldsymbol{\psi} = \langle \psi_1, ..., \psi_n \rangle$ denoted by $\nu[\boldsymbol{\psi}] = \nu[\psi_1]...\nu[\psi_n]$, where $\nu[v := \mathit{aterm}](v') = \nu[\mathit{aterm}]$ if $v' = v$, otherwise $\nu[v := \mathit{aterm}](v') = \nu(v')$. The same concept of modification is applied in case of boolean assignments.

Definition 2 (Control Flow Graph (CFG)). *A control flow graph $\gamma = (N, E, i)$ consists of a finite set of nodes N, a set $E \subseteq N \times \Phi \times \Psi \times N$ of directed edges, and an initial node $i \in N$ which has no incoming edges. Each edge $(n, \phi, \boldsymbol{\psi}, n') \in E$ has a source node n, a constraint ϕ, a list $\boldsymbol{\psi}$ of assignments and a destination node n'.*

CFG's operational semantics interprets the edge constraints and assignments:

Definition 3 (Operational Semantics). *The operational semantics $\mathcal{T}$ assigns to each control flow graph $\gamma = (N, E, i)$ a labelled transition system $\mathcal{T}(\gamma) = (\mathit{Conf}(\gamma), \{\xrightarrow{e} \mid e \in E\}, C_{\mathit{init}})$ where $\mathit{Conf}(\gamma) = \{\langle n, \nu \rangle \mid n \in N \wedge \nu : V \cup B \rightarrow \mathcal{D}(V) \cup \mathcal{D}(B)\}$ is the set of configurations of γ, $\xrightarrow{e} \subseteq \mathit{Conf}(\gamma) \times \mathit{Conf}(\gamma)$ are transition relations where $\langle n, \nu \rangle \xrightarrow{e} \langle n', \nu' \rangle$ occurs if there is an edge $e = (n, \phi, \boldsymbol{\psi}, n')$, $\nu \models \phi$ and $\nu' = \nu[\boldsymbol{\psi}]$, and $C_{\mathit{init}} = \{\langle i, \nu_{\mathit{init}} \rangle\} \cap \mathit{Conf}(\gamma)$ is the set of initial configurations of γ.*

A path σ of control flow graph γ is an infinite or finite sequence $\langle n_0, \nu_0 \rangle \xrightarrow{e_1} \langle n_1, \nu_1 \rangle \xrightarrow{e_2} \langle n_2, \nu_2 \rangle \ldots$ of consecutive transitions in the transition system $\mathcal{T}(\gamma)$, which furthermore has to be anchored in the sense of starting in an initial state $\langle n_0, \nu_0 \rangle \in C_{\mathit{init}}$. We denote by $\Sigma(\gamma)$ the set of paths of γ and by $\downarrow \sigma$ the set $\{\langle n_0, \nu_0 \rangle, \langle n_1, \nu_1 \rangle, ...\}$ of configurations visited along a path σ.

As we are interested in determining reachability in control flow graphs, we formally define reachability properties as follows:

Definition 4 (Reachability Property(RP)). *The set $\Theta(N, \Phi)$ of reachability properties (RP) over a control flow graph $\gamma = (N, E, i)$ is given by the syntax*

$$\theta ::= n$$

Given an RP θ and a path σ, we say that σ satisfies θ and write $\sigma \models \theta$ iff σ traverses a configuration $\langle n, \nu \rangle$ that satisfies θ, i.e., $\sigma \models \theta$ iff $\exists \nu : \langle n, \nu \rangle \in \downarrow \sigma$. We say that γ satisfies a reachability property θ iff some path $\sigma \in \Sigma(\gamma)$ satisfies θ. By $\Sigma(\gamma, \theta)$, we denote the set of all paths of γ that satisfy θ.

We analyze CFGs by a counterexample-guided abstraction refinement (CEGAR) scheme [6] employing *lazy abstraction* [11]. As usual, that refinement is based on identifying reasons for an abstract counterexample by means of constructing a *Craig interpolant* [3].

Definition 5 (Craig Interpolation). *Given two propositional logic formulae A and B in an interpreted logics $\mathcal{L}$ such that $\models_{\mathcal{L}} A \rightarrow \neg B$, a* Craig interpolant *for (A, B) is a quantifier-free $\mathcal{L}$-formula $\mathcal{I}$ such that $\models_{\mathcal{L}} A \rightarrow \mathcal{I}$, $\models_{\mathcal{L}} \mathcal{I} \rightarrow \neg B$, and the set of variables of $\mathcal{I}$ is a subset of the set of the free variables shared between A and B, i.e., $Var(\mathcal{I}) \subseteq Var(A) \cap Var(B)$.*

Depending on the logics $\mathcal{L}$, such a Craig interpolant can be computed by various mechanisms. If $\mathcal{L}$ admits quantifier elimination then this can in principle be used; various more efficient schemes have been devised for propositional logic and SAT-modulo theory by exploiting the connection between resolution and variable elimination [13,19].

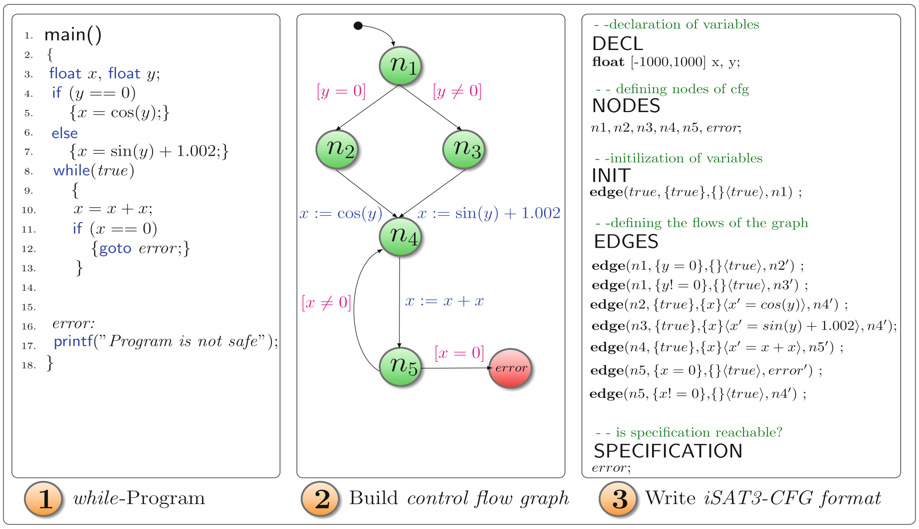
3 Description of the SMT Solver iSAT3

We build our CEGAR loop on the iSAT3 solver, which is an SMT solver accepting formulas containing arbitrary boolean combinations of theory atoms involving linear, polynomial and transcendental functions (as explained in Definition 1). In classical SMT solving a given SMT formula is split into a boolean skeleton and a set of theory atoms. The boolean skeleton (which represents the truth values of the theory atoms) is processed by a SAT solver in order to search for a satisfying assignment. If such an assignment is found, a separate theory solver is used to check the consistency of the theory atoms under the truth values determined by the SAT solver. In case of an inconsistency the theory solver determines an infeasible sub-set of the theory-atoms which is then encoded into a clause and added to the boolean skeleton. This scheme is called CDCL(T).

In contrast to CDCL(T), there is no such separation between the SAT and the theory part in the family of iSAT solvers [4]; instead interval constraint propagation (ICP) [20] is tightly integrated into the CDCL framework in order to dynamically build the boolean abstraction by deriving new facts from theory atoms. Similarly to SAT solvers, which usually operate on a *conjunctive normal form* (CNF), iSAT3 operates on a CNF as well, but a CNF additionally containing the decomposed theory atoms (so-called *primitive constraints*). We apply a definitional translation akin to the Tseitin-transformation [21] in order to rewrite a given formula into a CNF with primitive constraints.

iSAT3 solves the resulting CNF through a tight integration of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [22] in its conflict-driven clause learning (CDCL) variant and interval constraint propagation [20]. Details of the algorithm, which operates on interval valuations for both the boolean and the numeric variables and alternates between choice steps splitting such intervals and deduction steps narrowing them based on logical deductions computed through ICP or boolean constraint propagation (BCP), can be found in [4]. Implementing branch-and-prune search in interval lattices and conflict-driven clause learning of clauses comprising irreducible atoms in those lattices, it can be classified as an early implementation of abstract conflict-driven clause learning (ACDCL) [15].

iSAT3 is also able to generate Craig interpolants. Here we exploit the similarities between iSAT3 and a CDCL SAT solver with respect to the conflict resolution. As atoms occurring as pivot variables in resolution steps are always simple bounds mentioning a single variable only, we are able to straightforwardly generalize the technique employed in propositional SAT solvers to generate partial interpolants [5].



```
main()
{
 float x, float y;
  if (y == 0)
     {x = cos(y);}
  else
     {x = sin(y) + 1.002;}
  while(true)
     {
      x = x + x;
      if (x == 0)
         {goto error;}
      }


  error:
  printf("Program is not safe");
}
```

```
- -declaration of variables
DECL
float [-1000,1000] x, y;

- - defining nodes of cfg
NODES
n1, n2, n3, n4, n5, error;

- -initilization of variables
INIT
edge(true, {true},{}⟨true⟩, n1) ;

- -defining the flows of the graph
EDGES
edge(n1, {y = 0},{}⟨true⟩, n2′) ;
edge(n1, {y! = 0},{}⟨true⟩, n3′) ;
edge(n2, {true},{x}⟨x′ = cos(y)⟩, n4′) ;
edge(n3, {true},{x}⟨x′ = sin(y) + 1.002⟩, n4′);
edge(n4, {true},{x}⟨x′ = x + x⟩, n5′) ;
edge(n5, {x = 0},{}⟨true⟩, error′) ;
edge(n5, {x! = 0},{}⟨true⟩, n4′) ;

- - is specification reachable?
SPECIFICATION
error;
```

Fig. 1. Left: an arithmetic program, middle: corresponding control flow graph, right: encoding in iSAT3 CFG format.

3.1 Encoding Control Flow Graphs in iSAT3

In order to encode control flow graphs in the iSAT3 input language, we extend the syntax of iSAT3 as shown in Fig. 1. A control flow graph file in iSAT3 (iSAT3-CFG) contains five parts, namely the *declaration*, *nodes*, *initialization*, *edges*, and *specification* sections, which are all started by the respective keywords.

As in iSAT3, the declaration part defines all variables and constants to be used subsequently. Non-boolean variables must have an assigned initial range over which a solution is sought. The second part is the newly introduced nodes part, which defines the set of control flow graph nodes to be used as *source* or *destination* locations of transitions. The initialization part then defines both the initial edge of the CFG and the permissible initial values of all program variables. The latter is achieved by stating a predicate confining the possible values. Its counterpart is the reachability specification, which may name the destination node or define a set of variable valuations to be reached.

The edges part, introduced by the keyword EDGES, represents the control flows in the graph. This part contains a list of edges as defined in Definition 2, each defined by a source node, a list of guards, a list of assigned variables that are changed, a list of assignments where the assigned variable has to be primed, and a destination node which has to be primed as well. In case that the list of assigned variables is empty, it means all previous values of variables are propagated. In contrast to the iSAT3 tradition, a framing rule is applied such that all unspecified assignments and behaviors during unrolling are not considered to be nondeterministic choices, but values are maintained by implicit equations $x' = x$ for all unassigned variables.

4 Interpolation-Based Abstraction Refinement in iSAT3

The basic steps in counterexample guided abstraction refinement (CEGAR) are to, first, compute an initial abstraction, then model-check it, thereafter terminating if no counterexample is found or trying to concretize the counterexample otherwise. If concretization succeeds then the counterexample is real, else spurious. In the latter case, a reason for the occurrence of the spurious counterexample is extracted and subsequently used for refining the abstraction, after which model checking is repeated.

As concretization of the abstract counterexample involves solving its concrete path condition, which is a conjunctive constraint system in the logical theory corresponding to the data domain of the program analyzed, SAT-modulo-theory solving often is the method of choice for concretization and Craig interpolation consequently a natural candidate for the extraction of reasons. It has been suggested by Henzinger et al. [12]. Of these classical approaches, we do in particular adopt lazy abstraction [8,11] and inductive interpolants in [14], yet lift them to the analysis of programs featuring arithmetic beyond decidable fragments. While CEGAR on such rich fragments of arithmetic has been pursued within the field of hybrid-system verification, in particular by Ratschan et al. [23], refinement there has not been directed by Craig interpolation and, using explicit-state techniques, the targets where relatively small control skeletons rather possibly unwieldy CFGs. By a tight integration of checking the abstraction and SMT including CI, we are trying to overcome such limitations.

4.1 The Algorithm

This section presents the four main steps of CEGAR in iSAT3; namely abstraction, abstract model verification, predicate extraction during counterexample validation, and refinement.

Initial Abstraction. The first step of applying CEGAR is to extract an initial abstraction from the concrete model by a well-defined abstraction function. The first abstraction represents just the graph structure of the CFG without considering edge interpretations by assignments and guards. It is introduced as follows:

Definition 6 (Initial Abstraction Function). *Given a control flow graph $\gamma = (N, E, i) \in \Gamma$, its initial abstraction mediated by the abstraction function α is the CFG $\alpha(\gamma) = (N, E', i)$, where $E' = \{(n, true, \langle\rangle, n') \mid (n, \phi, \psi, n') \in E\}$.*

Verifying the Abstraction. In the model checking community it is common to verify reachability problems in the abstract model by using finite-state model-checkers, like BDD-based approaches [24]. In this work, we verify reachability properties in the abstract models by SMT solving together with interpolation [10] in order to verify reachability for unbounded depths. The individual runs of thus unbounded SMT-based model-checking are bound to terminate, as the initial abstraction is equivalent to a finite-state problem and as the predicates that are added to enrich the abstraction during refinement are just logical formula over simple bounds $x \sim c$ which are bounds on boolean propositions; i.e., *literals*, thus keeping the model finite-state. By this idea, we can pursue model-checking of the abstraction and the concretizability test of abstract counterexamples within the same tool, thus avoiding back and forth translation between different tools and checking technologies.

Path-Condition Generation and Extraction of Reasons. Given that the abstract model $\alpha(\gamma)$ is a CFG, it induces a set of paths. We call any path $\sigma_{abs} \in \Sigma(\alpha(\gamma))$ an *abstract path*. As the abstraction function just relaxes edge conditions, we can build a corresponding concrete path—if existent—by just reintroducing the missing constraints and assignments as follows.

Definition 7 (Path-Conditions Generation Function). *Given a control flow graph $\gamma = (N, E, i)$ and its abstraction $\alpha(\gamma) = (N, E', i) \in \Gamma$ and a finite abstract path $\sigma_{abs} : \langle i, \nu'_{init}\rangle \xrightarrow{e'_1} \langle n_1, \nu'_1\rangle \xrightarrow{e'_2} ... \xrightarrow{e'_m} \langle n_m, \nu'_m\rangle \in \Sigma(\alpha(\gamma))$, the path-conditions generation function $\kappa : \Gamma \times \Sigma \to \Sigma$ that builds a concrete path semantically by completing its conditions, is defined as follows:*

$$\kappa(\gamma, \sigma_{abs}) = \sigma \ \textit{where, } \sigma : \langle i, \nu_{init}\rangle \xrightarrow{e_1} \langle n_1, \nu_1\rangle \xrightarrow{e_2} ... \xrightarrow{e_m} \langle n_m, \nu_m\rangle,$$
$$\{e_1, ..., e_m\} \subseteq E \ \textit{and} \ \{n_1, ..., n_m\} \subseteq N$$

We say that σ is a real path if and only if its generated path condition, i.e., $\nu_{ini} \wedge \bigwedge_{i=1}^{m} \phi_i \wedge \psi_i$ is satisfiable, else it is spurious.

The crucial step in the CEGAR loop is to extract a reason for counterexamples being spurious such that case splitting on that reason would exclude the particular (and similar) counterexamples. Several previous works used different approaches and schemes to capture such reasons, like state splitting [23], word matching by using ω-automata [18], or interpolants [8,11–14]. In our work, we exploit stepwise interpolants as in [13,14] in order to obtain predicates capturing the reasons, where the first and last interpolants during refining any spurious counterexample are always *true* and *false* respectively [13]. This can be carried out as follows: When encountering a spurious counterexample $\sigma_{sp} = \langle i, \nu'_{init}\rangle \xrightarrow{e'_1} ... \xrightarrow{e'_m} \langle n_m, \nu'_m\rangle \in \Sigma(\gamma')$, where γ' is an abstraction, $\{e'_1, .., e'_m\} \subseteq E'$ – primed edges denote abstract ones –, $m > 0$ and $\theta = n_m$ is the goal to be reached,

– we complete the abstract path σ_{sp} in the original model γ semantically by using the path-conditions generation function κ as in Definition 7.

- as σ_{sp} is spurious, we obtain an unsatisfiable path formula $\kappa(\gamma, \sigma_{sp}) \notin \Sigma(\gamma)$, i.e., $\nu_{init} \wedge \bigwedge_{i=1}^{m} \phi_i \wedge \boldsymbol{\psi}_i \models \mathsf{False}$.
- by using CI in order to extract reasonable predicates as in lazy abstraction [12], one computes a reason of unsatisfiability at each control point (node) of γ. For example, consider that $\kappa(\gamma, \sigma_{sp}) = A \wedge B$, where $A = \nu_{init} \wedge \bigwedge_{j=1}^{k} \phi_j \wedge \boldsymbol{\psi}_j$, $B = \bigwedge_{j=k+1}^{m} \phi_j \wedge \boldsymbol{\psi}_j$ and $0 \leq k \leq m$. If we run the iSAT3 solver iteratively for all possible values of k, we obtain $m+1$ interpolants, where interpolant I_k is an adequate reason at edge e_k justifying the spuriousness of σ_{sp}.
- in case of using *inductive interpolants*, one uses the interpolant of iteration k, i.e., $\mathcal{I}_k$ as A-formula while interpolating against the above formula B in order to obtain interpolant $\mathcal{I}_{k+1}$. As $\mathcal{I}_k$ overapproximates the prefix path formula till k, we compute the next interpolant $\mathcal{I}_{k+1}$ that overapproximates $\mathcal{I}_k \wedge \phi_{k+1} \wedge \boldsymbol{\psi}_{k+1}$. This step assures that the interpolant at step k implies the interpolant at step $k+1$.

This guarantees that the interpolants at the different locations achieve the goal of providing a reason eliminating the infeasible error path from further exploration.

Abstraction Refinement. After finding a spurious counterexample and extracting adequate predicates from the path, we need to refine the abstract model in a way such that at least this counterexample is excluded from the abstract model behavior. This refinement step can be performed in different ways. The first way is a global refinement procedure which is the earliest traditional approach, where the whole abstract model is refined after adding a new predicate [25]. The second way is a *lazy abstraction* [8,11,26] where instead of iteratively refining an abstraction, it refines the abstract model on demand, as it is constructed. This refinement has been based on predicate abstraction [11] or on interpolants derived from refuting program paths [8]. The common theme, however, has been to refine and thus generally enlarge the discrete state-space of the abstraction on demand such that the abstract transition relation could locally disambiguate post-states (or pre-states) in a way eliminating the spurious counterexample.

Our approach of checking the abstraction within an SMT solver (by using interpolation based model checking) rather than a finite-state model-checker facilitates a subtly different solution. **Instead of explicitly splitting states in the abstraction**, i.e., refining the nodes of the initial abstraction, we stay with the initial abstraction **and just add adequate pre-post-relations to its edges**. These pre-post-relations are akin to the ones analyzed when locally determining the transitions in a classical abstraction refinement, yet play a different role here in that they are not mapped to transition arcs in a state-enriched finite-state model, but rather added merely syntactically to the existing edges, whereby they only refine the transition effect on an unaltered state space. It is only during path search on the (refined) abstraction that the SMT solver may actually pursue an implicit state refinement by means of case splitting; being a tool for proof search, it would, however, only do so on demand, i.e., only when the particular case distinction happens to be instrumental to reasoning.

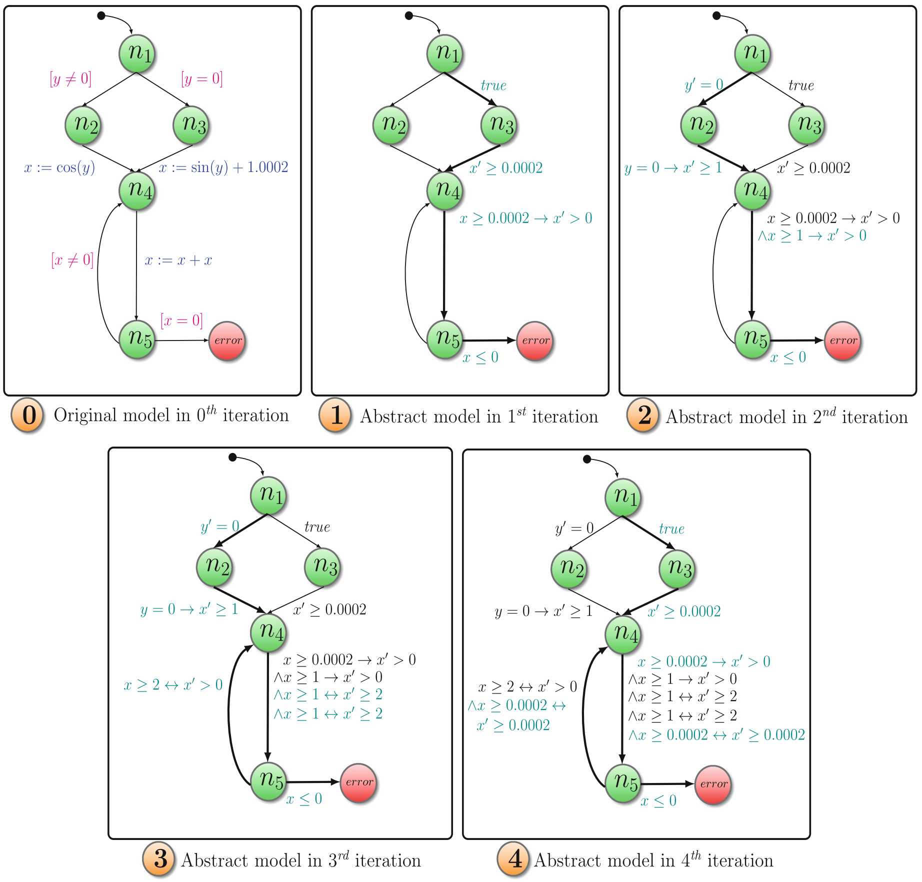

n_1
$[y \neq 0]$
$[y = 0]$
n_2
n_3
$x := \cos(y)$
$x := \sin(y) + 1.0002$
n_4
$[x \neq 0]$
$x := x + x$
$[x = 0]$
n_5
error
0 Original model in 0th iteration
true
$x' \geq 0.0002$
$x \geq 0.0002 \to x' > 0$
$x \leq 0$
1 Abstract model in 1st iteration
$y' = 0$
$y = 0 \to x' \geq 1$
$x \geq 0.0002 \to x' > 0$
$\wedge x \geq 1 \to x' > 0$
2 Abstract model in 2nd iteration
$x \geq 2 \leftrightarrow x' > 0$
$\wedge x \geq 1 \leftrightarrow x' \geq 2$
$\wedge x \geq 1 \leftrightarrow x' \geq 2$
3 Abstract model in 3rd iteration
$\wedge x \geq 0.0002 \leftrightarrow$
$x' \geq 0.0002$
$\wedge x \geq 0.0002 \leftrightarrow x' \geq 0.0002$
4 Abstract model in 4th iteration

Fig. 2. CEGAR iterations where bold **paths** and cyan predicates represent the current counterexample and added constraints in each iteration after refinement. (Color figure online)

We support this implicit refinement technique for both lazy abstraction (with inductive interpolants as optional configuration) and global refinement.

In the following we concisely state how the (novel) implicit refinement is performed by attaching pre-post-conditions to edges. Given a spurious counterexample $\sigma_{sp} = \langle i, \nu_{init}\rangle \xrightarrow{e'_1} \dots \xrightarrow{e'_m} \langle n_m, \nu_m\rangle \in \Sigma(\gamma')$ with $\theta = n_m$ as shown in the previous subsection, we obtain $m + 1$ (optionally inductive) interpolants, where I_k and I_{k+1} are consecutive interpolants at edges e_k and e_{k+1}, respectively, and $0 < k < m$. We continue as follows:

1. if $I_k \wedge \phi_{k+1} \wedge \boldsymbol{\psi}_{k+1} \to I_{k+1}$ holds, then we add $I \to I'$ to e'_{k+1},
2. if $I_k \wedge \phi_{k+1} \wedge \boldsymbol{\psi}_{k+1} \to \neg I_{k+1}$ holds, then we add $I \to \neg I'$ to e'_{k+1},
3. if $\neg I_k \wedge \phi_{k+1} \wedge \boldsymbol{\psi}_{k+1} \to \neg I_{k+1}$ holds, then we add $\neg I \to \neg I'$ to e'_{k+1},
4. if $\neg I_k \wedge \phi_{k+1} \wedge \boldsymbol{\psi}_{k+1} \to I_{k+1}$ holds, then we add $\neg I \to I'$ to e'_{k+1},

where I is I_k with all its indexed variable instances x_k replaced by undecorated base names x and I' is I_{k+1} with all its indexed variable instances x_{k+1}

replaced by primed base names x'. These checks capture all possible sound relations between the predecessor and successor interpolants. For example, consider the abstract model in the first iteration as in Fig. 2. Interpolation on the path condition of the spurious counterexample yields $I_1 := \mathit{true}$ and $I_2 := x \geq 0.0002$. By performing the previous four checks, we obtain only one valid check, namely

$$\underbrace{\mathit{true}}_{I_1} \wedge \underbrace{\mathit{true}}_{\phi_1} \wedge \underbrace{x_2 = \mathit{sin}(y_1) + 1.0002 \wedge y_2 = y_1}_{\boldsymbol{\psi}_1} \rightarrow \underbrace{x_2 \geq 0.0002}_{I_2}$$

We consequently construct the pre-post-predicate $\mathit{true} \rightarrow x' \geq 0.0002$ as shown on the arc from n_3 to n_4 of Image 1 of Fig. 2. We can derive that the pre-post-predicate thus obtained is a sufficient predicate to refine not only the abstract model at edge e'_{k+1} for eliminating the current spurious counterexample, but also for any other spurious counterexample that (1) has a stronger or the same precondition before traversing edge e'_{k+1} and (2) has a stronger or the same postcondition after traversing edge e'_{k+1}.

Lemma 1. *Given a control flow graph $\gamma \in \Gamma$, its abstraction $\alpha(\gamma)$ and a spurious counterexample $\sigma_{sp} \in \Sigma(\alpha(\gamma)$ over the sequence of edges $e_1, ...e_m$, adding side-conditions is sufficient to eliminate the spurious counterexample.*

Proof. (sketch): by using stepwise interpolants, we get a sequence of interpolants $I_0, ..., I_m$ attributing the previous (spurious) abstract counterexample with the path condition $\bigwedge_{i=0}^{m-1}(I_i \rightarrow I_{i+1})$,[1] where "$I_i \rightarrow I_{i+1}$" is obtained since $I_i \wedge \phi_{i+1} \wedge \boldsymbol{\psi}_{i+1} \rightarrow I_{i+1}$ is a tautology. As the first and – at least – the last interpolants are true and false respectively, the path formula $(\bigwedge_{i=0}^{m-1} I_i \rightarrow I_{i+1})$ becomes contradictory. Thus the current spurious counterexample is eliminated. □

Due to their implicational pre-post-style, we can simply conjoin all discovered predicates at an edge, regardless on which path and after how many refinement steps they are discovered. Such incremental refinement of the symbolically represented pre-post-relation attached to edges by means of successively conjoining new cases proceeds until finally we can prove the safety of the model by proving that the bad state is disconnected from all reachable states of the abstract model, or until an eventual counterexample gets real in the sense of its concretization succeeding. To prove unreachability of a node in the new abstraction, we use Craig interpolation for computing a safe overapproximation of the reachable state space as proposed by McMillan [10]. The computation of the overapproximating CI exploits the pre-post conditions added.

In the following, we illustrate how the program in Fig. 1 is proven to be safe; i.e., that location error is unreachable. The arithmetic program, the corresponding control flow graph, and the encoding of the control flow graph in iSAT3 are stated in the Fig. 1. In the first iteration, we get the initial coarse abstraction according to Definition 6. In case of finding spurious counterexample, which is the case in the first four iterations, we refine the model as shown in Fig. 2. After

[1] The proof considers the first type of implication check, the others hold analogously.

that, the solver proves that the error is not reachable in the abstract model. Additionally, the third and fourth counterexamples have a common suffix, but differ in the prefix formula, therefore both are needed for refining the abstraction in the third and fourth iterations. However, as all following paths from loop unwinding share the prefix formula with the previous two counterexamples, yet have stronger suffix formulas, the already added pre-post predicates are sufficient to eliminate all further counterexamples.

5 Experiments

We have implemented our approach, in particular the control flow graph encoding and the interpolation-based CEGAR verification, within the iSAT3 solver. We verified reachability in several linear and non-linear arithmetic programs and CFG encodings of hybrid systems. The following tests are mostly C-programs modified from [25] or hybrid models discussed in [5,27]. As automatic translation into CFG format is not yet implemented, the C benchmarks are currently mostly of moderate size (as encoding of problems is done manually), but challenging; e.g., hénon map and logistic map [5]. We compared our approach with interpolant-based model checking implemented in both CPAchecker [28] (IMPACT configuration [8]), version 1.6.1, and iSAT3,[2] where the interpolants are used as overapproximations of reachable state sets [5]. Also, we compared with CBMC [1] as it can verify linear and polynomial arithmetic programs. Comparison on programs involving transcendental functions could, however, only be performed with interpolant-based model checking in iSAT3 as CBMC does not support these functions and CPAchecker treats them as uninterpreted functions.

CBMC, version 4.9, was used in its native bounded model-checking mode with an adequate unwinding depth, which represents a logically simpler problem, as the k-inductor [34] built on top of CBMC requires different parameters to be given in advance for each benchmark, in particular for loops, such that it offers a different level of automation. We limited solving time for each problem to five minutes and memory to 4 GB. The benchmarks were run on an Intel(R) Core(TM) i7 M 620@2.67GHz with 8 GB RAM.

5.1 Verifying Reachability in Arithmetic Programs

Table 1 summaries the results of our experimental evaluation. It comprises five groups of columns. The first includes the name of the benchmark, type of the problem (whether it includes non-linear constraints or loops), number of control points, and number of edges. The second group shows the result of verifying the benchmarks when using iSAT3 CEGAR (lazy abstraction), thereby stating the verification time in seconds, memory usage in kilobytes, number of abstraction refinements, and the final verdict. The third group has the same structure, yet

[2] Although we contacted the authors of *dReal* [29] which supports unbounded model checking for non-linear constraints [30], they referred us to the latest version which does not support unbounded model checking, thus it is excluded.

Table 1. Verification results of linear/non-linear hybrid models. Bold lines refer to best results w.r.t. best verification time.

Program features \ Approach						iSAT3 CEGAR, lazy abstraction				iSAT3 Interpolation-based MC				CBMC maximum depth 250				CPAchecker ITP + lazy abstraction			
No	**Name**	**Non-linear**	**Loops**	**#Nodes**	**#Edges**	**Time(s)**	**Memory(KB)**	**Iteration**	**Result**	**Time(s)**	**Memory(KB)**	**Depth**	**Result**	**Time(s)**	**Memory(KB)**	**unwinding loop**	**Result**	**Time(s)**	**Memory(KB)**	**#Refinements**	**Result**
1	cfa_test0001 [25]	✗	✓	11	13	**1.962**	**17256**	**14**	**SAFE**	TO	6038428	20	UNKNOWN	98.396	151028	56	SAFE	2.782	150984	2	SAFE
2	cfa_test0002 [25]	✗	✓	11	13	**0.173**	**6352**	**5**	**SAFE**	TO	168240	801	UNKNOWN	9.406	141152	56	SAFE	2.242	143948	1	SAFE
3	cfa_test0003 [25]	✗	✓	11	13	**0.127**	**5716**	**5**	**SAFE**	TO	169072	800	UNKNOWN	8.160	140996	56	SAFE	2.202	128216	1	SAFE
4	cfa_test0004 [25]	✗	✓	11	13	**0.174**	**6568**	**8**	**UNSAFE**	55.653	5883156	15	UNSAFE	3.801	140936	56	UNSAFE	2.818	149652	2	UNSAFE
5	cfa_test0005 [25]	✓	✓	15	18	0.455	8812	9	CAND.	1.657	30840	16	CAND.	**0.150**	**22972**	**6**	**UNSAFE**	3.690	158588	3	UNSAFE
6	cfa_test0006 [25]	✓	✗	13	17	**0.043**	**5196**	**2**	**SAFE**	0.070	6500	7	SAFE	0.137	22320	0	SAFE	2.561	141876	2	SAFE
7	cfa_test0007 [25]	✓	✗	7	8	**0.047**	**4856**	**2**	**SAFE**	TO	4541444	3	UNKNOWN	unsupported functions				2.424	127384	1	UNSAFE
8	cfa_test0008 [25]	✗	✗	3	3	**0.017**	**4180**	**1**	**UNSAFE**	0.023	4112	1	UNSAFE	0.140	22600	0	UNSAFE	2.456	145936	2	UNSAFE
9	cfa_test0009 [31]	✗	✓	6	8	**0.054**	**5048**	**3**	**SAFE**	TO	157748	864	UNKNOWN	7.702	50248	56	SAFE	2.510	145684	1	SAFE
10	cfa_test0010	✓	✓	6	8	**0.075**	**5268**	**3**	**SAFE**	TO	775032	2	UNKNOWN	unsupported functions				2.229	128328	1	UNSAFE
11	control flow [5]	✗	✗	7	8	**0.035**	**4904**	**1**	**SAFE**	0.039	4716	5	SAFE	0.303	26820	116	SAFE	2.330	127968	1	SAFE
12	cruise control [5]	✓	✓	8	15	**0.103**	**5724**	**8**	**SAFE**	TO	3196492	130	UNKNOWN	0.147	22528	18	SAFE	2.819	146284	3	UNSAFE
13	frontier_01 [32]	✓	✓	3	4	**0.050**	**4744**	**1**	**CAND.**	0.056	4824	2	CAND.	3.367	101844	32	UNSAFE	2.650	145524	2	UNSAFE
14	frontier_02 [32]	✓	✓	3	4	**0.173**	**6148**	**3**	**SAFE**	TO	94200	721	UNKNOWN	1.276	102124	32	SAFE	3.046	149128	2	UNSAFE
15	frontier_03 [32]	✓	✓	3	4	**0.141**	**5332**	**3**	**SAFE**	TO	97580	796	UNKNOWN	1.284	102056	32	SAFE	2.868	148964	2	UNSAFE
16	hénon map [5]	✓	✓	3	4	**0.033**	**4628**	**3**	**CAND.**	0.041	4628	3	CAND.	0.694	24496	25	SAFE	2.216	129908	1	UNSAFE
17	logistic map [5]	✓	✓	3	4	**0.149**	**7380**	**3**	**SAFE**	0.205	5932	12	CAND	4.759	188928	38	SAFE	2.142	124620	1	UNSAFE
18	two circles_01	✓	✓	6	7	**0.067**	**4608**	**1**	**SAFE**	48.938	13848	8	SAFE	0.163	22452	54	SAFE	2.359	145832	1	UNSAFE
19	two circles_02	✓	✓	6	7	**0.033**	**4584**	**1**	SAFE	0.144	5260	8	SAFE	0.155	22204	55	SAFE	2.383	145204	1	UNSAFE
20	tank_controller [27]	✗	✓	5	13	7.822	159708	24	SAFE	**0.107**	**6784**	**20**	**SAFE**	0.149	22344	69	SAFE	2.446	143688	1	UNSAFE
21	gas_burner [27]	✗	✓	4	8	3.776	12720	42	SAFE	**0.361**	**7260**	**33**	**SAFE**	27.843	43580	1282	SAFE	4.511	151460	3	UNSAFE
22	cfa_test0022 [33]	✗	✓	8	19	2.264	18468	23	SAFE	0.845	126680	6	SAFE	**0.151**	**22312**	**25**	**SAFE**	5.358	202840	2	SAFE
23	cfa_test0023 [33]	✗	✓	3	4	8.189	42620	21	SAFE	0.143	6716	22	SAFE	**0.264**	**26880**	**56**	**SAFE**	3.025	145360	2	UNSAFE

reports results for using iSAT3 with interpolation-based reach-set overapproximation used for model checking.

The fourth part provides figures for CBMC with a maximum unwinding depth of 250. CBMC could not address the benchmarks 7 and 10 as they contain unsupported transcendental functions. The fifth part provides the figures for CPAchecker while using the default IMPACT configuration where the red lines refer to false alarms (*for comparison, CPAchecker was run with different configurations, yet this didn't affect the presence of false alarms.*) reported by IMPACT due to non-linearity or non-deterministic behaviour of the program. For each benchmark, we mark in boldface the best results in terms of time. iSAT3-based CEGAR outperforms the others in 18 cases, interpolation-based MC in iSAT3 outperforms the others in 2 cases, and CBMC outperforms the others in 3 cases. Figures 3 and 4 summarize the main findings. The tests demonstrate the efficacy of the new CEGAR approach in comparison to other competitor tools. Concerning verification time, we observe that iSAT3 with CEGAR scores the best results. Namely, iSAT3-based CEGAR needs about 27 s for processing the full set of benchmarks, equivalent to an average verification

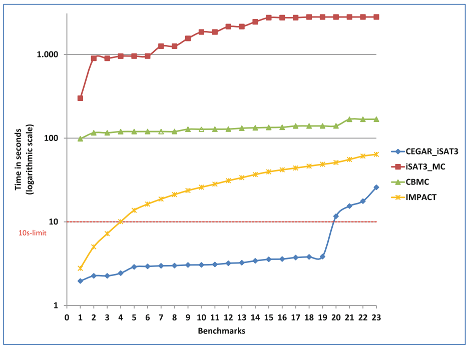


Fig. 3. Accumulated verification times for the first n benchmarks

time of 1.2 s, iSAT3 with the interpolation-based approach needs 2809 s total and 122 s on average, CBMC needs 168 s total and 8 s on average, and IMPACT needs 64 s total and 2.7 s on average.

Concerning memory, we observe that iSAT3 with CEGAR needs about 15 MB on average, iSAT3 with interpolation 906 MB on average, CBMC needs 66 MB on average, and IMPACT needs 141 MB on average. The findings confirm that at least on the current set of benchmarks, the CEGAR approach is by a fair margin the most efficient one.

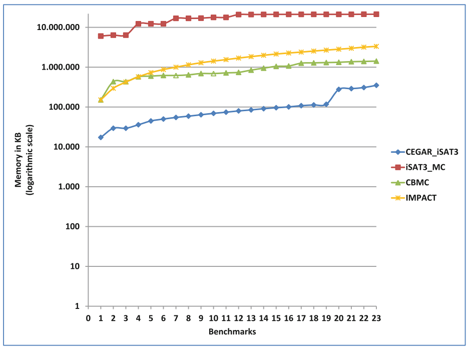


Fig. 4. Memory usage (#benchmarks processed within given memory limit)

The only weakness of both iSAT3-based approaches is that they sometimes report a *candidate solution*, i.e., a very narrow interval box that is hull consistent, rather than a firm satisfiability verdict. This effect is due to the incompleteness of interval reasoning, which here is employed in its outward rounding variant providing safe overapproximation of real arithmetic rather than floating-point arithmetic. It is expected that these deficiencies vanish once floating-point support in iSAT3 is complete, which currently is under development as an alternative theory to real arithmetic. It should, however, be noted that CEGAR with its preoccupation to generating conjunctive constraint systems (the path conditions) already alleviates most of the incompleteness, which arises particularly upon disjunctive reasoning.

6 Conclusion and Future Work

In this paper, we tightly integrated interpolation-based CEGAR with SMT solving based on interval constraint propagation. The use of the very same tool, namely iSAT3, for verifying the abstraction and for concretizing abstract error paths facilitated a novel implicit abstraction-refinement scheme based on attaching symbolic pre-post relations to edges in a structurally fixed abstraction. The resulting tool is able to verify reachability properties in arithmetic programs which may involve transcendental functions, like *sin*, *cos*, and *exp*. With our prototype implementation, we verified several benchmarks and demonstrated the feasibility of interpolation-based CEGAR for non-linear arithmetic programs well beyond the polynomial fragment.

Minimizing the size of interpolants (and thus pre-post relations generated) and finding adequate summaries of loops in case of monotonic functions will be subject of future work.

References

1. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24730-2_15
2. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, pp. 238–252 (1977)
3. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Logic **22**(3), 269–285 (1957)
4. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. JSAT **1**(3–4), 209–236 (2007)
5. Kupferschmid, S., Becker, B.: Craig interpolation in the presence of non-linear constraints. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 240–255. Springer, Heidelberg (2011). doi:10.1007/978-3-642-24310-3_17
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). doi:10.1007/10722167_15
7. Clarke, E.M.: SAT-based counterexample guided abstraction refinement in model checking. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 1–1. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45085-6_1
8. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006). doi:10.1007/11817963_14
9. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03237-0_7
10. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45069-6_1
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16–18, pp. 58–70 (2002)
12. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244 (2004)
13. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. JSAT **5**(1–4), 27–56 (2008)
14. Beyer, D., Löwe, S.: Explicit-value analysis based on CEGAR and interpolation. CoRR abs/1212.6542 (2012)
15. Brain, M., D'Silva, V., Griggio, A., Haller, L., Kroening, D.: Interpolation-based verification of floating-point programs with abstract CDCL. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 412–432. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38856-9_22
16. Albarghouthi, A., Gurfinkel, A., Chechik, M.: WHALE: an interpolation-based algorithm for inter-procedural verification. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 39–55. Springer, Heidelberg (2012). doi:10.1007/978-3-642-27940-9_4

17. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_2
18. Segelken, M.: Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 433–448. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73368-3_46
19. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Logic **62**(3), 981–998 (1997)
20. Benhamou, F., Granvilliers, L.: Combining local consistency, symbolic rewriting and interval methods. In: Calmet, J., Campbell, J.A., Pfalzgraf, J. (eds.) AISMC 1996. LNCS, vol. 1138, pp. 144–159. Springer, Heidelberg (1996). doi:10.1007/3-540-61732-9_55
21. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Stud. Math. Math. Logic Part II, 115–125 (1968)
22. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
23. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. ACM Trans. Embedded Comput. Syst. **6**(1), 8 (2007)
24. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000). doi:10.1007/10722468_7
25. Dinh, N.T.: Dead code analysis using satisfiability checking. Master's thesis, Carl von Ossietzky Universität Oldenburg (2013)
26. Jha, S.K.: Numerical simulation guided lazy abstraction refinement for nonlinear hybrid automata. CoRR abs/cs/0611051 (2006)
27. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**(1), 3–34 (1995)
28. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73368-3_51
29. Gao, S., Kong, S., Clarke, E.M.: dReal: an SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 208–214. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38574-2_14
30. Gao, S., Zufferey, D.: Interpolants in nonlinear theories over the reals. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 625–641. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_41
31. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric bounds analysis with conflict-driven learning. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 48–63. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28756-5_5
32. Kupferschmid, S.: Über Craigsche Interpolation und deren Anwendung in der formalen Modellprüfung. Ph.D. thesis, Albert-Ludwigs-Universität Freiburg im Breisgau (2013)
33. Seghir, M.N.: Abstraction refinement techniques for software model checking. Ph.D. thesis, Albert-Ludwigs-Universität Freiburg im Breisgau (2010)
34. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23702-7_26