A New Refinement Strategy for CEGAR-Based Industrial Model Checking

Martin Leucker[1], Grigory Markin[1(✉)], and Martin R. Neuhäußer[2]

[1] University of Lübeck, Lübeck, Germany
{leucker,markin}@isp.uni-luebeck.de
[2] Siemens AG, Nuremberg, Germany
martin.neuhaeusser@siemens.com

Abstract. This paper presents a novel refinement strategy in the setting of counterexample-guided abstraction refinement (CEGAR)-based model checking. More specifically, the approach shown builds on lazy abstraction in the context of predicate abstraction. While the concept of interpolants is typically used for refinement, this paper employs unsatisfiability cores together with weakest preconditions. The new refinement technique is especially applicable in the setting where interpolants are hard to compute, such as for McCarthy's theory of arrays or for the theory of fixed-width bit vectors. It is implemented within a model-checking tool developed for the verification of industrial-critical systems and outperforms current refinement strategies on several examples.

1 Introduction

Today's industrial systems are increasingly controlled by software. When these systems are interconnected and control each other, they often form so-called cyber-physical systems acting in our physical environment. For such systems, verification is of major interest as safety and security concerns abound.

In this paper, we are concerned with the verification of industrial software by means of model checking [1]. To this end, we have developed a model checking tool for verifying code for programmable logical controllers (PLC code), which takes a PLC program as well as a reachability property characterizing error states as input and checks whether an error state can be reached from an initial state of our program. For testing and comparison, our model checker also accepts a subset of C programs as input.

As typically no single model checking approach acts as a silver bullet, our model checker comes with different algorithms. In [2], the last author reports about a technique following the idea of IC3 [3]. The current paper, however, describes an approach following the counterexample-guided abstraction refinement (CEGAR) paradigm [4]. Within CEGAR, an abstraction for the system to verify is continuously checked with respect to a correctness property. When the a bstract system satisfies the correctness property, the underlying system is correct as well, while in the case of a counterexample this might be spurious. If so, the counterexample is used to refine the abstract system and the CEGAR

N. Piterman (Ed.): HVC 2015, LNCS 9434, pp. 155–170, 2015.
DOI: 10.1007/978-3-319-26287-1_10

loop is continued. Eventually, the underlying system is either verified as correct or a non-spurious counterexample is found.

Lazy abstraction [5] builds on this scheme, yet refines the abstraction *on demand at locations* for which the current abstraction yields a spurious counterexample. In this way, the overall performance of the model checking process is improved. In [5], the concept of lazy abstraction is introduced in a general setting. In our model checker, we use predicate abstraction [6] as abstract domain. All (symbolic) computations are considered with respect to a given first-order theory, and we use the concept of strongest postconditions as well as that of weakest preconditions to give (partial) semantics to operations, in a way that is suitable for our approach. We assume the availability of a corresponding SMT solver allowing to check formulas in the given theory for satisfiability. In fact, our tool runs with both Microsoft Z3 [7] as well as MathSat [8] as back-end and the theory of fixed-width bit vectors.

One of the most interesting questions when realizing a lazy abstraction approach with predicate abstraction is how to refine the set of predicates when a (non-feasible) counterexample is found. In the original work by Henzinger [5], (negations of) weakest preconditions were used as one example to rule out a counterexample. Another typical approach is to use *interpolants* [9–11].

In this paper, we make use of the intermediate results available from the underlying SMT solver. Whenever a program path is witnessed as spurious, the encoding of the path as an SMT formula is unsatisfiable. We take a subset of the clauses of the formula that is already unsatisfiable. Such a subformula is also called an *unsatisfiability core*, or unsat core, for short. Actually, we take a sequence of unsat cores obtained via the infeasible program path and derive new predicates in this way, forming a new operator called *UCB-refinement operator*. Our approach can, in a way, be understood as an improvement of the original weakest precondition-based approach.

[12–14] also make use of unsat cores in the context of verification, yet none of the approaches uses unsat core-based techniques to improve lazy abstraction.

We have proven our approach as correct, i.e. that the underlying lazy abstraction algorithm always terminates with a correct answer when the underlying theory is bounded. Moreover, we have implemented our model checker and evaluated the performance of our UCB-refinement operator relative to the interpolation-based refinement strategy. Additionally, we evaluated the performance of the Lazy Abstraction algorithm in combination with UCB-predicates and interpolants relative to an implementation of the IC3 model-checking algorithm recently presented in [2]. It turns out that our UCB-refinement strategy is often comparable to the interpolation-based one and in a number of cases, it allows to solve the verification problems which could not be solved by using the interpolation-based approach. The Lazy Abstraction algorithm in combination with UCB-refinement operator typically outperforms the IC3 algorithm and can solve more problems than the latter. As such, the UCB refinement operator is valuable as more verification problems can be solved than before. More importantly, it allows the easy realization of the CEGAR approach for theories and

SMT solvers not supporting interpolants. However, using other theories in our model checker than fixed-width bit vectors is future work.

2 Preliminaries

In this section, we give the gist of the standard concepts of logical theories. For details, we refer the reader to [15].

Throughout this paper let V be a countable set of variables defined over a non-empty domain. The set of all assignments of elements to variables of the underlying domain forms the set of all *data states*, in the following denoted by S. In our setting the set of variables V comes with an associated first-order *theory* $\mathcal{T}$, i.e., a set of closed first-order formulas. We denote by $\mathcal{T}(V)$ the set of formulas that may contain *free* variables from V. We denote by $\top$ and $\bot$ the logical values *true* and *false* respectively. When concerned with satisfiability, we consider such variables as implicitly existentially quantified.

For the rest of this paper, we assume the theory $\mathcal{T}$ to be decidable, in practice by means of an SMT solver. We use the standard definition of *conjunctive normal form* (CNF) and we silently assume that every formula is in conjunctive normal form. We denote the set of all clauses of a CNF formula φ as $cl(\varphi)$ and use formulas in CNF and sets of clauses interchangeably. An *unsatisfiable core* of an unsatisfiable CNF formula φ is a non-empty subset of clauses of φ that is unsatisfiable. A *minimal unsatisfiable core* is an unsatisfiability core such that removing any one of its elements makes the remaining set of clauses satisfiable. The set of all unsatisfiability cores of φ is denoted by $UC(\varphi)$.

An *ascending chain* is a sequence of formulas $\varphi_1, \ldots$ such that for all $0 \leq i < j$, $\varphi_j \not\Rightarrow \varphi_i$. The ascending chain is called *finite*, if the sequence is finite. A theory $\mathcal{T}$ is called *bounded* if all ascending chains from $\mathcal{T}$ are finite.

This paper addresses the verification of imperative programs, ranging over program locations from a set L and performing operations from some given set Ops. For the sake of generality, we assume the *concrete semantics* of an operation $op \in Ops$ to be given by its *strongest postcondition* [16] $sp(\varphi, op) \in \mathcal{T}(V)$ for a given precondition $\varphi \in \mathcal{T}(V)$. In practice, we assume the availability of an sp-operator that computes for a given precondition φ, denoting a set of current states, the set of successor states. We denote by $wp(\varphi, op)$ the *weakest liberal precondition* [16,17] of φ with respect to op, representing the largest set of states from which φ is reachable by executing op, if it terminates. Similarly, we assume the existence of a wp-operator computing the set of predecessor states for a given φ. More specifically we also assume that corresponding sets of successor and predecessor states can be expressed by formulas of $\mathcal{T}(V)$. See [16–18] for details on weakest preconditions and [9,19,20] for the use of strongest post- and weakest preconditions in model checking.

Throughout this paper, we follow [11] and use the concept of *control flow automata* (CFA) for encoding imperative programs. Hereby, a CFA A is a directed connected graph where the set of vertices L represents the program locations and the set of edges $E \subseteq L \times Ops \times L$ models the execution of operations from the set Ops (see Fig. 1a for an example).

We focus on the verification of *reachability properties* of programs. Hence, we assume a program to be given by its control flow automaton, together with an initial location and an error location. More precisely, a $\mathcal{T}$*-program* is tuple $\mathcal{P} = (V, A, l_I, l_\epsilon)$, where assignments of the variables V form the set of data states S, $A = (L, E)$ is a CFA that models the control flow of the program and $l_I, l_\epsilon \in L$ model the initial and the error locations, respectively.

A set of data states is called a *data region* and we restrict to data regions that can be encoded by a formula $\varphi \in \mathcal{T}(V)$ as a set of all data states $s \in S$ that entail φ. A *concrete state* of a program is a pair (l, s), where $l \in L$ is a program location and $s \in S$ is a data state. A *region* (l, φ) is a set of concrete states $\{(l, s) \mid s \in S\}$ such that $s \models \varphi$.

A *program path* π is a sequence of program operations $\pi = l_0 \xrightarrow{op_1} l_1 \xrightarrow{op_2} \cdots \xrightarrow{op_n} l_n$, where $l_i \in L$, $i \in \{0, \ldots, n\}$ and $op_i \in Ops$, $i \in [n]$, where $[n]$ is a shorthand for $\{1, \ldots, n\}$. The *concrete semantics for a program path* π is defined as $sp(\varphi, \pi) = sp(sp_{n-1}, op_n)$ where $sp_i = sp(sp_{i-1}, op_i)$ and $sp_0 = \varphi$. A program path π is *feasible* when starting from a data region φ if $sp(\varphi, \pi)$ is satisfiable, otherwise it is *infeasible*. For brevity, we sometimes only talk of a (in)feasible program path when the starting region is clear from the context.

Throughout the paper, let $\{\top, \bot\} \subseteq P \subseteq \mathcal{T}(V)$ be a finite set of predicates. A conjunction over the set of predicates P, denoted by $\bigwedge P$, is the conjunction of all predicates from P. The conjunction over an empty set $\bigwedge \emptyset$ is identified by $\top$. The *Cartesian abstraction* [21] of a formula $\varphi \in \mathcal{T}(V)$ with respect to P, denoted by φ^P, is the conjunction of all predicates from P that are (individually) implied by φ, i.e. $\varphi^P = \bigwedge\{p \mid p \in P, \varphi \rightarrow p\}$. We also denote by $C(P)$ a conjunction of predicates that arises from an element of the power set of P. An *abstract state* is a region (l, φ^P) computed by the Cartesian abstraction for a region (l, φ) with respect to a set of predicates P.

We define the *abstract semantics* of an operation $op \in Ops$ by the *abstract strongest postoperator* sp^P defined as $sp^P(\varphi, op) = (sp(\varphi, op))^P$. Similar to previously, we extend the abstract semantics to program paths by $sp^P(\varphi, \pi) = sp^P(sp^P_{n-1}, op_n)$ where $sp^P_i = sp^P(sp^P_{i-1}, op_i)$ and $sp^P_0 = \varphi$.

In our work we also use the concept of an *abstract reachability tree* (ART) to encode abstract models of programs. An ART is a rooted directed tree whose nodes are labeled by triples (l, φ, P) where the set of concrete states (l, φ) represents an abstract state and the set of predicates P is the local *precision*. For each ART node (l, φ, P), the child nodes are labeled with the abstraction precision P' and successor abstract nodes, computed according to the abstract strongest postoperator sp^P. A node $n = (l, \varphi, P)$ is called *covered* if there is a node $n' = (l, \varphi', P')$ such that $\varphi \Rightarrow \varphi'$. An ART tree is called *complete* if every leaf node is either covered or all possible abstract successor states are present in the ART as children of the node. In simple words, an ART is the unwinding of the CFA. If it is complete, the ART comprises a symbolic representation of all reachable states of the underlying program.

Algorithm 1. LazyAbstraction($\mathcal{P}, \Phi$)

Require: A program $\mathcal{P} = (V, A, l_I, l_\epsilon)$ with a CFA $A = (L, E)$ and a refinement operator $\Phi : \mathcal{T}(V) \times E^+ \to 2^{\mathcal{T}(V)}$.

create an ART with a root node $n_0 = (l_0, \top, \{\top, \bot\})$
while there are unmarked nodes in ART **do**
  pick an unmarked node $n = (l, \varphi, P)$
  if $l = l_\epsilon$ **then**
    let $n' = (l', \varphi', P')$ be the oldest ancestor of n s.t. $n' \xrightarrow{\sigma} n$ and $wp(\top, \sigma) \wedge \varphi' \not\equiv \bot$
    if $l' = l_I$ **then**
      return "error trace"σ
    else
      let $n'' = (l'', \varphi'', P'')$ s.t. $(l'', op, l') \in E$
      let τ denote the time stamp of n''
      relabel n'' by $w = (l'', \varphi'', P'' \cup \Phi(\varphi'', op \cdot \sigma))$
      remove the sub-trees starting from n''
      for all covered leaf m that was marked after τ **do**
        unmark m
  else if there exists $m = (l, \varphi', P')$ s.t. $\varphi \to \varphi'$ **then**
    mark m as *covered*
  else
    for all $l' \in L$ s.t. $(l, op, l') \in E$ **do**
      $\varphi' \leftarrow sp^P(\varphi, op)$
      if $\varphi' \not\models \bot$ **then**
        add a child node $n' = (l', \varphi', P)$ to the ART
    mark n as *uncovered*
return ART

3 Lazy Abstraction

The traditional flow for CEGAR-based model checking [4] consists of the following steps: building an abstract model of the program using a chosen set of predicates; verification of the abstract model; checking the feasibility of the abstract counterexample, i.e. whether it can be executed in the original program; counterexample-driven refinement of the set of predicates. All steps are repeated until either no counterexample can be found in the abstract model, i.e. the original program is error-free, or an abstract counterexample is feasible, i.e. an error state is reachable from an initial state.

As the explicit construction of an abstract model and its verification are generally time-consuming operations, the *lazy abstraction* approach optimizes the CEGAR loop in that it continuously constructs an abstract model and checks whether an error state is reachable at the same time.

In our work, we consider the lazy abstraction algorithm (Algorithm 1), which is a slightly modified version of the one originally presented by Henzinger et al. in [5]. The algorithm constructs an ART which either is complete or contains a feasible abstract counter example. To this end, it starts by creating a new

reachability tree containing one node: a root node corresponding to the initial node in the CFA. At any time, each node of the tree is either unmarked, i.e. not processed, or is marked as *covered* or *uncovered*. After the initialization step, the algorithm iteratively picks an unmarked node and checks whether it is labeled by an error location. If this is not the case, then the algorithm checks whether there is already another node that represents a superset of the data region represented by the current node. If so, it marks the node as covered, and, if not, it marks it as uncovered and adds its children into the tree.

If the picked node is labeled by an error location, then the algorithm checks whether this node is reachable from the initial node (in the concrete program). If the node is reachable, then an error trace is found and the algorithm terminates. Otherwise the algorithm searches backwards for the node, which abstracts concrete states from which an error state can be reached but which are unreachable from all concrete initial states. Such a node is also called a *pivot node* (n'' on line 9). In this refinement step (line 11) the algorithm uses a refinement operator, denoted by Φ, to increase the abstraction precision of the pivot node by adding further predicates. After that, the sub-tree rooted at the pivot node is discarded and all nodes that were covered after processing the pivot node are unmarked, so that it will (potentially) be reconstructed using the enriched set of predicates in later steps of the algorithm.

If an error trace is found or if the constructed ART is complete, i.e. all leaf nodes are covered, the algorithm terminates.

Our version of the algorithm is in fact a concretization of the more generic one given in [5]. Henzinger's algorithm operates on so-called *symbolic abstraction structures*, which comprise abstract domains and corresponding abstract operations (see [5] for details). These operations allow computation of abstract successor states, and predecessors of concrete data regions.

In our setting, we use predicate abstraction as abstract domain and the abstract strongest postoperator to compute abstract states. We use the weakest precondition operator to compute sets of concrete states that are reachable from a given region. As these operators and the abstract domain are built into the program, our algorithm only takes a program $\mathcal{P}$ and the refinement operator Φ as its input rather than a symbolic abstraction structure.[1]

In our version of the algorithm we also always remove the sub-tree rooted at the pivot node after the refinement step. The original algorithm uses an additional heuristic to determine whether the sub-tree can be kept. The heuristic does not affect the correctness but it may affect termination. And because, we

[1] Actually, a symbolic abstraction structure also contains an operator for computation of abstract states that can reach a given abstract state and an operator for computation of concrete states reachable from a given concrete state. These operators are required for a backward search, i.e. when the lazy abstraction algorithm constructs an ART starting from the error region and iteratively checks whether an initial region is reachable. In this paper we only consider lazy abstraction in combination with the forward search (Algorithm 1) and refer the reader to [5] for more details on the use of lazy abstraction in combination with backward search.

are interested in refinement techniques toward termination, we always discard the sub-tree rooted at a pivot node, to be on the safe side.

It was shown in [5] that the algorithm terminates with a correct result for finite-state systems, or, more generally, when (i) the language $\mathcal{T}(V)$ is bounded, i.e. does not contain infinite ascending chains, and (ii) the refinement step yields semantically equivalent data regions (see explanation below) as well as a superset of predicates, and (iii) the set of predicates returned in the refinement step is precise enough to rule out the path's suffix starting from the pivot node. The second constraint on the refinement step is needed for correctness. It means, that the region in node n'' (line 9), given by φ'', is not changed by the refinement step. This is clear in our setting, as the new node keeps φ'' (line 11) without any modification. Hence, correctness is immediate, regardless of which refinement operator is considered.[2] The first and last constraint entail termination of the algorithm.

The main objective of this paper is to introduce a new refinement operator. Hence, we concentrate in the next section on refinement methods and their properties toward termination of the algorithm LazyAbstraction$(\mathcal{P}, \Phi)$ shown in Algorithm 1. More precisely, we introduce the notion of a progressive refinement operator and show the termination of the algorithm when applied with such an operator. We then introduce our concept of an unsatisfiability-core-based refinement operator and show that it is progressive and hence assure termination of the lazy abstraction algorithm.

4 Abstraction Refinement

When Algorithm 1 hits an error node, it checks whether the path from the initial node to the error node represents a valid counterexample, i.e. the path is feasible. If this is not the case, Algorithm 1 searches for the pivot node along that path that is the furthermost node from the initial node, representing a set of concrete states reachable from the set of initial states. As the path is an infeasible path, no error state is reachable from any concrete state represented by the pivot node. The algorithm constructs such path in the ART, however, due to the low abstraction precision of the pivot node. Hence, the goal of the refinement step is to increase the abstraction precision of the pivot node by adding new predicates in such a way that the algorithm will not be able to construct the same path's suffix leading to the error node in the next iteration.

We introduce the notion of *abstract infeasibility* of program paths to indicate whether a path can be constructed by the algorithm.

Definition 1 (Abstract (in)feasibility). *Let π be a program path that is infeasible when starting from a region φ. The program path π is* abstractly infeasible *with respect to a set of predicates P when starting from φ if $sp^P(\varphi, \pi)$ is unsatisfiable. Otherwise π is called* abstractly feasible.

[2] Due to page limitations, we refrain from formulating the correctness criteria precisely here but refer the reader to [5] for more details.

When Algorithm 1 constructs an infeasible program path, it is divided by the pivot node $n = (l, \varphi, P)$ into the feasible path prefix (feasible when starting from $\top$) and the path suffix that is infeasible when starting from the data region φ represented by the pivot node. The path suffix is, however, abstractly feasible with respect to the abstraction precision P when starting from φ. Thus, in order to avoid discovering the same path's suffix by the algorithm again, the goal for the refinement operator is to find a set of predicates P' such that the path suffix becomes abstractly infeasible with respect to P' when starting from φ.

Definition 2 (Progressive refinement operator). *Given a refinement operator $\Phi : \mathcal{T}(V) \times E^+ \to 2^{\mathcal{T}(V)}$. We call the refinement operator Φ* progressive *if for every program path π which is infeasible when starting from a data region φ, π is abstractly infeasible with respect to $\Phi(\varphi, \pi)$ when starting from φ.*

Definitions 1 and 2 reflect the constraint on the refinement step as it relates to the termination of the lazy abstraction algorithm presented in [5]. Lemma 1 formally states this condition in term of the progressive refinement.

Lemma 1 (Termination with progressive refinement operator). *Let $\mathcal{P} = (V, A, l_I, l_\epsilon)$ be a $\mathcal{T}$-program with a CFA $A = (L, E)$ and let $\Phi : \mathcal{T}(V) \times E^+ \to 2^{\mathcal{T}(V)}$ be a refinement operator. The execution of* LazyAbstraction*$(\mathcal{P}, \Phi)$ terminates if $\mathcal{T}(V)$ is bounded and Φ is progressive.*

The proof is, on one hand, immediate from the developments in [5], as the concepts introduced so far are just an instantiation of Henzinger's more general approach. Nevertheless, termination can also be shown in a straightforward fashion, using König's lemma: The algorithm cannot produce an infinite tree in the limit, as a finitely-branching, infinite tree would contain an infinite path (according to König's lemma), which would be an infinite ascending chain violating the boundedness of $\mathcal{T}(V)$. Progressiveness ensures that the algorithm does not end up in an infinite loop producing the very same ART again and again.

For each ART node, Algorithm 1 computes a corresponding data region using the abstract strongest postoperator. If we consider a path in the ART, then the data regions represented by the nodes along that path form a sequence. In the following we introduce the notions of an *approximating sequence* and an *infeasibility witness* for a path, which are sequences of data regions of the path's length, where each element over-approximates the corresponding data region computed by the algorithm. The idea behind it is that if a refinement operator constructs a sequence of data regions for an infeasible path such that it is also an infeasibility witness, then the elements of that sequence can be used as new predicates and it will guarantee abstract infeasibility of that path.

Definition 3 (Approximating sequence and infeasibility witness). *Given a program path $\pi = l_0 \xrightarrow{op_1} l_1 \xrightarrow{op_2} \cdots \xrightarrow{op_n} l_n$, a data region φ_0 and a set of predicates P, we call a sequence of formulas $\varphi_1, \ldots, \varphi_n$, $\varphi_i \in C(P)$ for $i \in [n]$, a* P-approximating sequence *for π if $sp(\varphi_{i-1}, op_i) \models \varphi_i$ for all $i \in [n]$. We call a P-approximating sequence a* P-infeasibility witness *if $\varphi_n \equiv \bot$.*

As is evident, an approximating sequence for a path forms a sequence of data regions such that each data region φ_i over-approximates a data region that is reachable from the predecessor data region φ_{i-1} by executing the operation op_i. The first element of each sequence over-approximates the data region that is reachable from the given data region φ_0 by executing the first operation of the path. If the last data region of the sequence is empty, i.e. $\varphi_n \equiv \bot$, the approximating sequence forms an infeasibility witness.

We define an approximating sequence and an infeasibility witness with respect to a set of predicates P in order to determine whether a set of predicates is precise enough to show that a path is abstractly infeasible.

First we show that each element of a P-approximating sequence for a path over-approximates the corresponding data region constructed by Algorithm 1.

Lemma 2. *Given a program path* $\pi = l_0 \xrightarrow{op_1} l_1 \xrightarrow{op_2} \cdots \xrightarrow{op_n} l_n$*, a data region* φ_0 *and a set of predicates* P*. For every* P*-approximating sequence* $\varphi_1, \ldots, \varphi_n$ *it holds that* $sp^P(\varphi_0, op_1 \ldots op_i) \models \varphi_i$ *for all* $i \in [n]$*.*

Now we prove the necessary and sufficient condition of abstract infeasibility of an infeasible program path with respect to a set of predicates in terms of an infeasibility witness.

Lemma 3. *Given a set of predicates* P *and a program path* π*, which is infeasible when starting from a data region* φ*. The program path* π *is abstractly infeasible with respect to* P *when starting from* φ *iff there is a* P*-infeasibility witness for* π*.*

Corollary 1 (Termination and infeasibility witness). *Given a* $\mathcal{T}$*-program* $\mathcal{P}$ *and a refinement operator* Φ*, whereby* $\mathcal{T}$ *is bounded. If, for every program path* π *that is infeasible when starting from a data region* φ*, the refinement operator* Φ *yields an infeasibility witness, then* LazyAbstraction$(\mathcal{P}, \Phi)$ *terminates.*

One of the examples of a set of predicates that forms an infeasibility witness for an infeasible path is the set of negated weakest preconditions computed on that path. Consider a path $\pi = l_0 \xrightarrow{op_1} l_1 \xrightarrow{op_2} \cdots \xrightarrow{op_n} l_n$, which is infeasible when starting from a data region φ. Let $wp_i = wp(\top, op_{i+1} \ldots op_n)$, $i \in [n]$ and $wp_n = \top$ to denote a weakest precondition for location l_i. Each wp_i represents the largest data region from which an error state can be reached by executing $op_{i+1} \ldots op_n$. But as the considered path is infeasible when starting from φ, wp_i cannot be reached from φ by executing $op_1, \ldots, op_{i-1}$. Thus, each $\neg wp_i$ represents the largest data region from which no error state can be reached by executing $op_1, \ldots, op_{i-1}$. Moreover, $\neg wp_1$ is reachable from φ and each $\neg wp_i$ is reachable from $\neg wp_{i-1}$ by executing op_1 and op_i respectively. It follows that the sequence $W = (\neg wp_1, \ldots, \neg wp_{n-1}, \neg wp_n = \bot)$ is a W-infeasibility witness. Hence, a refinement operator yielding weakest preconditions for an infeasible path would be progressive and would fulfill the termination criterion for refinement operators.

Weakest preconditions are used for the refinement operator in the original work to lazy abstraction [5]. Though the use of weakest preconditions are sufficient to guarantee the termination of the algorithm, they often encode too much

information and hence are represented by quite complex formulas. Each formula encoding a weakest precondition would contain all variables from a path suffix, regardless of whether a variable has any impact on infeasibility of that path.

In the following, we present a novel method for computation of new predicates from infeasible paths. It is also based on weakest preconditions but tries to discard as much information as possible that does not affect infeasibility. It results in much simpler formulas as well as, in some cases, in generation smaller number of predicates needed to prove or disprove a property using lazy abstraction algorithm. All these can significantly improve overall performance of the lazy abstraction algorithm. The computation of new predicates makes use of unsatisfiability cores and weakest preconditions.

Definition 4 (Unsat-core-based (UCB) predicates). *Let* $\pi = l_0 \xrightarrow{op_1} l_1 \xrightarrow{op_2} \dots \xrightarrow{op_n} l_n$ *be a program path that is infeasible with respect to a data region* φ*. We call a sequence of predicates* $p_1, \dots, p_n$ unsat-core based predicates *if* $p_i = \neg(\varphi_i \setminus cl(\psi_i))$ *where* $\varphi_i \in UC(\psi_i \wedge wp_i)$*,* $\psi_i = sp\,(p_{i-1}, op_i)$*,* $p_0 = \varphi$*, and* $wp_n = \top$ *and* $wp_i = wp(\top, op_{i+1} \dots op_n)$*.*

Definition 4 can be seen as an algorithm for computing new predicates. In the first pass, the algorithm computes the weakest preconditions $wp_1, \dots, wp_n$ for program locations $l_1, \dots, l_n$. Then, starting from the location l_1, it iteratively computes for each location l_i the data region ψ_i that is reachable from the data region p_{i-1} by executing corresponding operation op_i. The data region ψ_i over-approximates the set of states that are reachable from φ, and thus has no common states with the weakest precondition because the given path is infeasible with respect to φ. Using this fact, we compute an unsatisfiability core of the conjunction of ψ_i and wp_i and select all clauses from the unsatisfiability core that are only included in the weakest precondition. The resulting formula over-approximates the weakest precondition wp_i, which represents the largest set of states from which an error state can be reached by executing the path's suffix $op_i \dots op_n$. Hence, we take the negation of that formula as a new predicate p_i, which represents an under-approximation of $\neg wp_i$, which is the largest set of states from which no error state can be reached by executing the path's suffix. At the same time, p_i represents an over-approximation of the data region that is reachable from the given data region φ by executing the path's prefix $op_1 \dots op_{i-1}$.

Intuitively, the properties of UCB predicates described above allow us to use them for the abstraction refinement. Though each UCB predicate p_i represents in general a smaller data region than the corresponding $\neg wp_i$, they contain much more precise information about the reason of unsatisfiability than the corresponding weakest precondition.

Let us sketch how to prove that the use of UCB predicates computed according to Definition 4 in the refinement step guarantees termination of Algorithm 1.

First one shows that each UCB-predicate p_i as well as ψ_i represent the sets of states that are disjointed from the set of states represented by wp_i. As wp_i represents the set of all states from which the path suffix $op_{i+1} \dots op_n$ can be

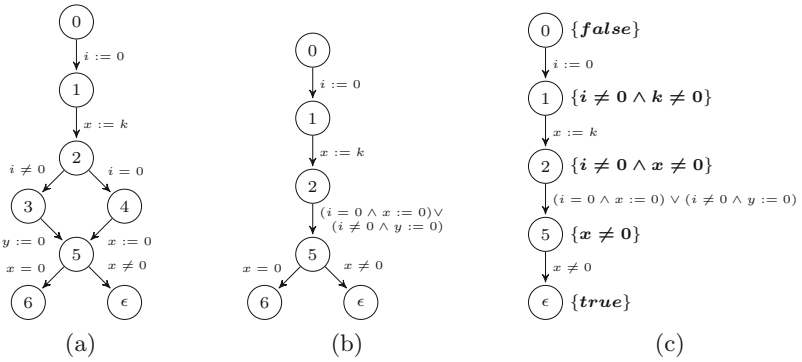


Fig. 1. A CFA of a simple imperative program (a). The optimized CFA (b) after merging "if-then-else" block. An infeasible path (c) from the optimized CFA, where nodes are labeled by the corresponding weakest preconditions (curly brackets).

executed, we show that p_i and ψ_i under-approximate the set of states represented by $\neg wp_i$: (i) $p_i \rightarrow \neg wp_i$ and $\psi_i \rightarrow \neg wp_i$ for all $i \in [n]$. Now we show that p_i over-approximates the data region reachable from p_{i-1}, which we use to show that the sequence $p_1, \ldots, p_n$ forms an approximating sequence: (ii) $sp(p_{i-1}, op_i) \rightarrow p_i$ for all $i \in [n]$. Now we can show that a sequence of UCB predicates computed according to Definition 4 forms an infeasibility witness. (iii) Given a path π that is infeasible when starting from a data region φ then a sequence of UCB predicates U computed for π is an U-infeasibility witness for π.

We sum up our developments with the following theorem, which follows easily from the items (i)–(iii) stated above.

Theorem 1 (Termination with UCB predicates). *Let $\mathcal{P}$ be a $\mathcal{T}$-program and let Φ be the refinement operator yielding the UCB predicates for an infeasible path. If $\mathcal{T}$ is bounded, then* LazyAbstraction*$(\mathcal{P}, \Phi)$ terminates.*

4.1 Path Projection

One of the commonly used techniques to simplify the refinement based on unsatisfiability cores, is the refinement of a path formula based on an unsatisfiability core and the following predicate extraction on the refined path. As a path formula of an infeasible path is unsatisfiable, one can construct a path "projection", which will only contain the information included in the unsatisfiability core and then apply some predicate extraction technique, e.g. weakest precondition-based or interpolation-based techniques, on that path. The path projection is achieved by removing clauses from the encoding of program operations that are not included in the unsatisfiability core.

Unfortunately, the choice of the predicate extraction techniques in such approach strongly depends on the encoding of program operations as well as on other components of the Lazy Abstraction algorithm, such as the abstract post-operator.

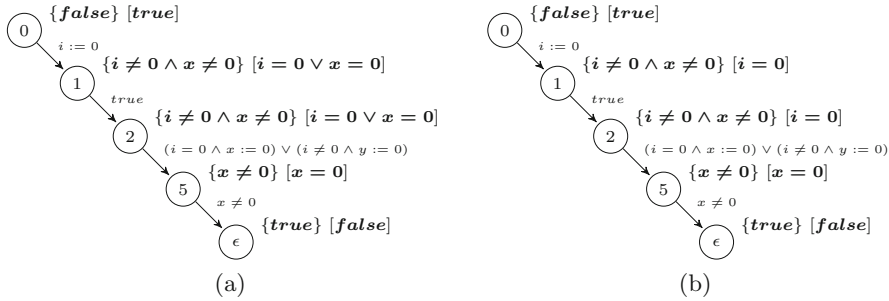


Fig. 2. The projected paths, where nodes are labeled by weakest preconditions (curly brackets) as well as by extracted predicates (square brackets): negated weakest preconditions (a) and UCB predicates (b).

In our model-checking tool we use the large-block encoding (LBE) technique [22] and Cartesian abstraction as the abstract postoperator. In the following we show by means of an example that in our configuration the combination of the weakest precondition-based predicate extraction and path projection does not guarantee termination of the Lazy Abstraction algorithm.

Consider a CFA (Fig. 1a) of a simple imperative program assigning a value to either variable x or y according to the value of the variable i assigned in the first step. Figure 1b presents the CFA after applying the optimization step that merges the "if-then-else" block into one transition. If we start the algorithm with an empty set of predicates, it will find a counterexample (Fig. 1c) in the first iteration, which is obviously infeasible. Due to infeasibility, one can simplify it using an unsatisfiability core. The example of such projection is presented in Fig. 2a and b. After simplification step, we can apply a predicate extraction algorithm. Figures 2a and b show the predicates that were constructed by negating the corresponding weakest preconditions (Fig. 2a) and by computing UCB predicates (Fig. 2b) as described in Definition 4. Unfortunately, the predicates, which were constructed using negation of weakest preconditions, do not form an approximating sequence for the original path and thus, do not fulfill the necessary condition for termination of the algorithm. The problem lies on the simplified transition between nodes 1 and 2. On one hand, the transition was simplified by removing the superfluous assignment $x := k$ but on the other hand, the variable x still appears in the predicate. This leads to the problem at program location 2, namely, there is no predicate, which under-approximates the negated weakest precondition $i = 0 \vee k = 0$ from the original path, which represents the largest set of states from which the error state is unreachable. As one can see, there is no such problem when using the UCB predicates, as all superfluous variables are removed from predicates as well.

We have to note that in order to use the UCB predicates in such configuration, one has to ensure that only minimal unsatisfiability cores are used in the computation of UCB predicates. The reason for that is that by applying path projection it is necessary that no superfluous variables will remain in the

resulting predicate, which can be guaranteed by using minimal unsatisfiability cores. Also note, that the interpolation-based technique to predicate construction also has this property and hence can be used in combination with projection. We also consider successfully such an additional projection in our setting (see Sect. 5 (Results)). A formal proof of the previous remarks require a precise definitions of the notion of projection etc. and is, due to space constraints, left to a full version of the paper.

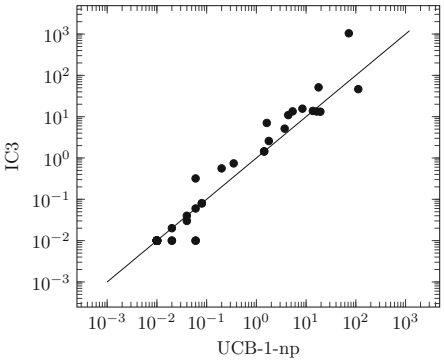
5 Implementation and Experimental Results

Implementation. We implemented the Lazy Abstraction algorithm in combination with the UCB-based as well as the interpolation-based refinement strategies on top of an existing proprietary model-checking framework. This framework makes use of LLVM project to parse C programs and translate them into the intermediate representation (IR). We apply some static analysis and optimization techniques on this IR, e.g. Steensgaard's pointer analysis and model minimization [23–25], as well large block encoding [22]. We use the bit-precise memory model that supports limited pointer operations including array-element and record-field addressing. While the Lazy Abstraction algorithm in combination with the UCB-based refinement strategy is fully theory unaware and can be used for infinite-domain theories, such as linear real arithmetic (LRA) we use the finite-domain theory of bit vectors. Our tool supports the Z3 and MathSAT SMT solvers, but as the Z3 solver does not support computation of interpolants for the theory of interest we executed all benchmarks using the MathSAT solver.

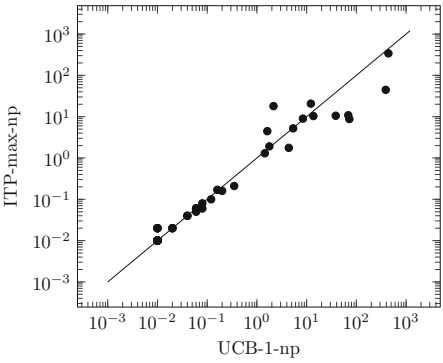
Besides the refinement algorithms we implemented and evaluated two optimizations which may improve performance of the Lazy Abstraction algorithm in some cases. The first one is the path projection described in the previous section and the second one is the limitation of number of extracted predicates. The intention behind it is that the fewer predicates are extracted the less SMT queries has to be done by the abstract postoperator and in some cases not all predicates extracted from an infeasible path are necessary to prove or disprove a property. It does not affect the correctness, as in the worst case, the algorithm will need to refine paths multiple times.

We additionally compared our implementation to an improvement of the IC3 model-checking approach that is currently one of the most actively studied model-checking algorithms. The improved version of the IC3 algorithm was recently presented in [2] and is implemented within the same model-checking framework. Hence, all preprocessing and optimization steps are identical to our implementation.

Experiments. We have evaluated our algorithms on 178 C programs taken from the SV competition and enriched by some of our own programs. For 89 programs, none of the algorithms has terminated within the given time or memory bound. In the following, we discuss the behavior of the algorithms on the remaining 89 examples, for which at least configuration of one algorithm terminated.



(a)



(b)

Algorithm	# solve	solve t
UCB-1-np	70	1234
ITP-1-np	77	3471
UCB-max-np	68	1474
ITP-max-np	79	5947
UCB-1-p	58	1777
ITP-1-p	67	2750
UCB-max-p	61	1268
ITP-max-p	70	4356
IC3	63	1249

(c)

Algorithm	# solve
UCB	75
ITP	85
IC3	63

(d)

Fig. 3. Performance comparison of UCB (best configuration) with IC3 and interpolants (best configuration) (a), (b). Number of solved problems by each algorithm in each configuration (1/max denotes the number of extracted predicates and p/np whether the path projection is applied (p) or not (np)) (c). Number of solved problems by algorithms for all configurations (d).

All experiments have been executed on a cluster using a single core running at 2.1 GHz, a memory limit of 4GB per file and a timeout of 3600 s.

We briefly compare the performance of the Lazy Abstraction algorithm in combination with UCB-based and interpolation-based refinement in its best configurations (w.r.t number of solved problems) as well the IC3 implementation from [2]. We also compare the number of solved problems by different approaches.

Results. From the results in Fig. 3 we can come to the following conclusions. First, the best configuration of the interpolation-based approach solved more problems out of 89 than others (Fig. 3c). At the same time, our results which are not presented in tables, shows that there are 4 problems which could only be solved by the UCB-based and IC3 algorithms. The best configuration of the UCB-based algorithm solved 8 problems more than the IC3 algorithm and the latter solved one problem which the best of UCBs could not solve. The best configuration of the interpolation-based approach solved 15 problems more than the best configuration of UCBs but at the same time the latter solved 6 problems which the best interpolation-based could not solve.

Second, application of the path projection and limitation of the number of predicates allow to solve problems which can not be solved otherwise, increasing thereby the total number of solved problems (Fig. 3c) and thus, also play an important role in Lazy Abstraction approach. While interpolation-based approach solved the most problems without any optimization, the UCB-based approach was most efficient by only taking one predicate during the refinement. The difference arises from the fact that for the computation of all UCB-predicates one needs to make as many SMT queries as the path's length, while an SMT-solver computes all interpolants from one query. At the same time, applying path projection may simplify the consequent queries but as we can see, in many cases it introduces the superfluous computation which decreases the overall performance.

Finally, the best configuration of the UCB-based approach outperforms the IC3 approach in most cases (Fig. 3a) and is comparable to the best configuration of the interpolation-based approach (Fig. 3b).

Summarizing our results we can conclude that the best results in our current setting can be achieved by combining the UCB-based and interpolation-based approaches for example by running them in parallel.

6 Conclusion

This paper studied a new refinement technique within the CEGAR-based approach to model checking. More specifically, we build on lazy abstraction, where the refinement step is usually carried out using weakest preconditions on program paths or using techniques like interpolants.

As interpolants are sometimes not available by the underlying SMT solver, we have developed a new refinement step that makes use of unsatisfiable cores to improve refinement and which can be used with any SMT solver (that can compute unsatisfiability cores). We have shown that our refinement step is progressive in the sense that the lazy abstraction approach terminates when verifying finite state systems. Moreover, we have implemented the strategy within our own model-checking tool. We have shown that UCB-based refinement nearly always outperforms one of the implementation of IC3 model-checking algorithms in our setting. Regarding interpolation-based refinement, UCB-based is often comparable but also allows to verify programs which can not be solved by using interpolants. As such, the papers presents a new valuable refinement strategy.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Lange, T., Neuhäußer, M.R., Noll, T.: IC3 software model checking on control flow automata. In: Formal Methods in Computer-Aided Design, FMCAD 2015 (2015)
3. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)

4. 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)
5. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. ACM SIGPLAN Not. **37**, 58–70 (2002). ACM
6. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
9. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. ACM SIGPLAN Not. **39**(1), 232–244 (2004)
10. 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)
11. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. Int. J. Softw. Tools Technol. Transf. (STTT) **9**(5), 505–525 (2007)
12. Jain, H., Kroening, D., Sharygina, N., Clarke, E.M.: Word-level predicate-abstraction and refinement techniques for verifying RTL Verilog. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **27**(2), 366–379 (2008)
13. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Cegar-based formal hardware verification: a case study. Ann Arbor **2007**, 48109–2122 (1001)
14. Yang, Z., Al-Rawi, B., Sakallah, K., Huang, X., Smolka, S., Grosu, R.: Dynamic path reduction for software model checking. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 322–336. Springer, Heidelberg (2009)
15. Kroening, D., Strichman, O.: Decision Procedures, vol. 5. Springer, New York (2008)
16. Gries, D.: The Science of Programming. Springer, Heidelberg (1981)
17. Dijkstra, E.W.: A Discipline of Programming, vol. 1. Prentice-Hall, Englewood Cliffs (1976)
18. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
19. Jager, I., Brumley, D.: Efficient directionless weakest preconditions. Technical report, CMU-CyLab-10-002, Carnegie Mellon University, CyLab (2010)
20. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. **93**(6), 281–288 (2005)
21. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
22. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Formal Methods in Computer-Aided Design, FMCAD 2009, pp. 25–32. IEEE (2009)
23. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)
24. Lange, T., Neuhäußer, M.R., Noll, T.: Speeding up the safety verification of programmable logic controller code. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 44–60. Springer, Heidelberg (2013)
25. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, New York (1999)