EUForia: Uninterpreted Functions Abstraction with Incremental Induction

Denis Bueno Karem A. Sakallah
University of Michigan

June 18, 2018

Abstract

We investigate a novel algorithm for an IC3-style checker that operates entirely at the level of equality with uninterpreted functions (EUF). EUF abstraction is efficient to compute from a word-level transition system, whereas predicate abstraction typically requires a (possibly exponential) number of calls to a theorem prover. Data operations are treated as uninterpreted functions and relations as uninterpreted predicates. Our checker, called EUForia, checks a transition system for a given safety property and either (1) discovers an inductive strengthening EUF formula or (2) produces an abstract counterexample which corresponds to zero, one, or many concrete counterexamples. We also present a simple method for computing refinement lemmas that checks the feasibility of the abstract counterexamples. We formalize the EUF transition system, prove our algorithm correct, and demonstrate our results on a subset of benchmarks from the software verification competition (SV-COMP) 2017.

Chapter 1

Preliminaries

1.1 Introduction

Abstraction has become a principal technique for scaling automatic software verification. The predominant abstraction technique for software model checkers is predicate abstraction [1, 2], often combined with counterexample-guided abstraction refinement (CEGAR) [3, 4]. For example, the SLAM project was successful in showing that it is feasible to prove interesting safety properties about a C program using predicate abstraction and refinement [5]. SLAM is now routinely used for API usage verification in Windows drivers. Predicate abstraction casts the state space of a program into one over *predicates* on the original states. The major challenges of predicate abstraction are determining the predicates and constructing the predicate abstraction. Computing the predicate abstraction is exponential in the number of predicates, but later approximations enabled more efficient computation (e.g., Cartesian abstraction [6]). Later improvements to predicate abstraction included lazy abstraction [7] and various refinement schemes [8, 9]. See D'Silva *et al.* for an overview [2].

In this paper, we present an algorithm which structurally abstracts programs using equality with uninterpreted functions (EUF), checks them with an adapted IC3-style algorithm [10], and automatically refines away spurious counterexamples. We employ EUF instead of general predicate abstraction because (1) the EUF abstraction is inexpensive to compute (it is simply a substitution on the concrete program) and (2) we conjecture it is efficient for targeting control-centric properties. Burch and Dill [11] pioneered the use of EUF for pipelined microprocessor verification. Babić and Hu employed EUF for software verification in the style of extended static checking for their tool Calysto [12, 13]. We employ EUF in a fine grained way, at the program statement level, to abstract data computations and concentrate on control relationships.

The aim of this work is to target control-centric properties. Informally, a property is control-centric if only relatively small amounts of reasoning about data are required to prove or disprove it. For example, proving the equivalence of two different programs that calculate the parity of a machine word is *not* control-centric, because it will require reasoning about all possible data values exactly. On the other hand, showing that a protocol state machine never enters an illegal state, or verifying that function calls into an API are called in a particular order, are examples of control-centric properties. The exact data values are not as important—what is important is that the logic around the data, even as it interacts with the data, behaves as it should.

We make the following contributions:

- an integration of incremental induction to the EUF abstraction, which we call EUForia, yielding a potentially faster exploration of a program's data space based on congruence closure,
- the automatic refinement of spurious counterexamples by data lemmas that eliminate infeasible abstract states and transitions; and
- experimental evaluation of EUForia on a subset of benchmarks from SV-COMP '17, including running EUForia in a mode that does not use abstraction.

Section 1.2 gives an overview of our checker's architecture. Sections 2.1 and 2.2 detail the EUF abstraction and transition system encoding and our key algorithmic changes to IC3. Section 3 discusses refinement of spurious counterexamples. We conclude with an evaluation (Section 4), related work (Section 1.3), and then conclude (Section 4.1).

1.2 The EUForia Checker

The overall flow of EUForia is given in Figure 1.1. EUForia processes LLVM bitcode generated from well-formed C programs. LLVM is a compiler infrastructure that provides an intermediate representation (IR) of code in static single assignment (SSA) form and a variety of analyses for this IR [14]. Although in principle using LLVM allows EUForia to support a number of source languages, in practice this is limited to C because we only support a subset of LLVM operations. However, this is not a limitation of our approach, just of our current checker prototype.

From the LLVM control flow graph, EUForia constructs a word-level concrete transition system (CTS) in the style of Manna and Pnueli [15]. The CTS is encoded as formulas in the QF_ABV logic [16] using the Z3 4.5.0 Satisfiability Modulo Theories (SMT) solver [17]. A straightforward syntax-directed rewrite of the CTS produces an abstract transition system (ATS) for checking. Following a typical CEGAR flow, EUForia either produces (1) an inductive invariant for the checked property or (2) an abstract counterexample (ACX). If an abstract counterexample is found, EUForia determines whether the counterexample corresponds to a true error trace; if the counterexample was spurious, EUForia refines the abstract system by adding data lemmas to the ATS. EUForia uses Boolector 2.4.1 [18] for refinement.

EUForia is implemented in 13,700 lines of C++. EUForia runs various LLVM optimizations on the target source code including full inlining, dead code elimination, and promoting memory to registers, but EUForia

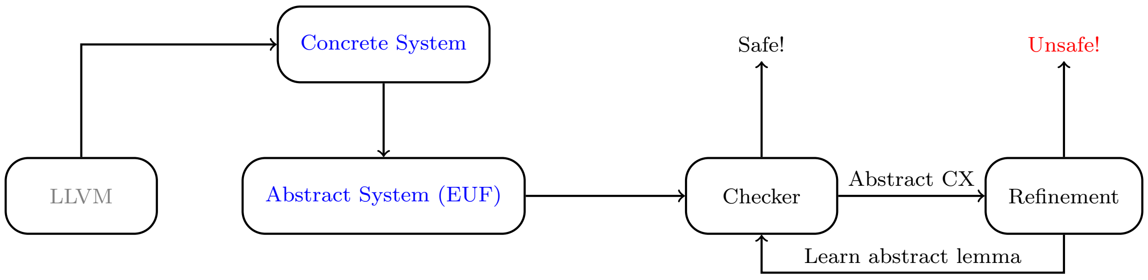

Figure 1.1: The overall flow of EUForia. The grayed boxes are parts of EUForia; LLVM is a third party tool. The abstract transition system (ATS) is explained in Section 2.1 and the concrete transition system (CTS) is covered briefly in Section 2.2.3. Refinement and abstract counterexamples (ACXs) are discussed in Section 3

does no slicing.[1] As EUForia is a prototype tool, it cannot yet process programs with dynamic memory allocation or recursion. EUForia also assumes that C programs do not exhibit undefined behavior (signed overflow, buffer overflow, etc.), and may give incorrect results if the input program is ill-defined.

1.3 Related Work

Incremental Induction EUForia's model checker implementation is inspired by the hardware model checking algorithm called Property Directed Reachability (PDR) of Een *et al.* [19], with some key changes to support software and enabling our abstraction techniques. PDR is an updated implementation of IC3 [20, 10]. Cimatti and Griggio [21] and Hoder and Bjørner [22] presented the first software model checkers built around IC3. Our work uses one Boolean variable per program location (instead of a *pc* variable) and bit-precise bit vector operations (instead of linear arithmetic). Lange *et al.* adapted IC3 to check control flow automata [23] by associated with each program location its own copy of IC3's over-approximate frames. Our work only uses one copy of the frames. Cimatti *et al.* further generalized IC3 to support predicate abstraction [24]. In their work, IC3 operates at the Boolean level of an abstract state space, and their predicates are instantiated over bit vectors and linear rational arithmetic. CTIGAR also generalized IC3 to predicate abstraction with refinement applied during abstract reachability [25]. Our work does not employ predicate abstraction; it operates on an EUF state space directly. Bjørner and Gurfinkel integrated polyhedral abstract interpretation with PDR to compute safe convex polyhedral invariants [26]. Karbyshev *et al.* generalize IC3 to a procedure for inferring quantified universal invariants or proving that none exist [27]. Welp and Kuehlmann use PDR to refine loop invariants [28] as well as a hybrid approach of cooperating IC3/PDR solver instances that are each responsible for disjoint parts of the program to verify [29, 30]. The distinguishing features of our checker are that it uses simple EUF abstract and is capable of checking and refining an EUF transition system; and

[1]Inlining is not required for our approach, but for this paper all function calls were inlined.

that its bit-precise mode uses no specific abstract domains (just the QF_BV theory) for generalizing bit vector constraints.

Structural Abstraction Babić and Hu [12, 13] implemented Calysto, a CEGAR abstraction that uses EUF to abstract at function boundaries. Calysto computes verification conditions (VCs) and function summaries for all the functions in the program. If the abstraction is too coarse to establish the property, then Calysto finds abstract summaries that are responsible for the spurious counterexample, and refines them by removing EUF terms and make them bit-precise. Our refinement differs in that refinement lemmas are lifted to EUF instead of certain EUF terms becoming bit-precise; moreover, we do not unroll loops, as Calysto does. EUF abstraction has been studied extensively, especially for translation validation and equivalence check, but not for IC3/PDR applied to checking safety properties; see [31] for further discussion of EUF abstraction. Similar techniques to ours have been developed by Lee and Sakallah [32] for hardware verification, particularly using uninterpreted functions for abstracting wide datapaths. Our work applies directly to software.

Predicate Abstraction Our work is similar to work in predicate abstraction [1], e.g., in the SLAM [5], BLAST [7], and CPAChecker [33] tools. SLAM's approach is to abstract the program into a program on Boolean variables alone, which preserves control and abstracts data with respect to a set of predicates. SLAM checks its Boolean program with pushdown techniques using Binary Decision Diagrams (BDDs). BLAST improves of the SLAM scheme; it uses interpolants to discover relevant predicates locally and these predicates are only kept track of in the parts of the abstract state space where spurious counterexamples occurred. SLAM requires an exponential number of calls to the theorem prover in the worst case (or an approximation to the abstraction [6]). EUF abstraction is nearly "free" in that it does not require any calls to a theorem prover and preserves the structure of the transition relation.

Abstraction in general has been employed extensively to address verification complexity [4, 34, 35, 36]. Counterexample-Guided Abstraction Refinement (CEGAR) was introduced by Kurshan [3] and refined and generalized by Clarke *et al.* [4]. Jhala and Majumdar [37] give a survey of model checking techniques, including CEGAR and modeling programs as state machines.

Transition System Encoding Our program encoding is mostly standard, with the exception that where control resides is encoded with a set of one-hot Boolean variables. Standard approaches include using an explicit *pc* variable [21] or hierarchical Boolean variables [15].

Chapter 2

Abstract PDR

2.1 Abstract Transition System Encoding

Program Representation by Transition Systems The template for a transition system [38, 20] consists of a (non-empty) set of state variables $X = \{x_1, \cdots, x_n\}$, a (possibly empty) set of input variables $Y = \{y_1, \cdots, y_m\}$, and a set of n next-state formulas $\{f_1(X, Y), \ldots, f_n(X, Y)\}$ that define the system's transition behavior. The system's transition relation is expressed as a conjunction of constraints

$$T\left(X, Y, X^+\right) \doteq \bigwedge_{1 \leqslant i \leqslant n} \left(x_i^+ = f_i(X, Y)\right) \tag{2.1}$$

where the '+' superscript denotes the next-state version of a state variable. We write expressions as $E(X[, Y, X^+])$ when we wish to emphasize that the only free variables in the expression E are drawn from the sets $X[$, possibly including Y and $X^+]$. This template is quite general, allowing the state variables to have different types such as single bits, bit vectors of various widths, unbounded integers, and terms in FOL.

Two additional sets of constraints are needed to completely define a model checking instance: a formula $I(X)$ specifying a set of initial states, and a formula $P(X)$ characterizing "safe" states. The model checking problem is to check if states that violate P are reachable from I or to prove that all states reachable from I are safe. For simplicity, unless otherwise specified, we assume I and P are cubes.

Much previous work has explored checking these kinds of transition systems [15, 39, 20, 21, 24, 40]. IC3, the algorithm on which our checker is based, checks transition systems in which all state variables are Boolean [20]. Other relevant checkers analyze transition systems with linear rational arithmetic [21], bit vectors [24, 40], etc. Our paper focuses on transition systems where (1) each state variable is either Boolean or a 0-arity term, and (2) each next-state function f_i is an EUF formula. EUF is covered in the next section.

The Logic of Formulas Our setting is standard quantifier-free, first-order logic (FOL) with the standard notions of theory, atom, term, predicate, formula, satisfiability, validity, entailment, and models. A literal is an atom. A clause is a disjunction of literals. A cube is a conjunction of literals. $a \models b$ means that a entails b.

Following the presentation in [31], we begin with a review of the EUF logic. Its syntax is simple: it is made up of atoms, terms, and formulas. Terms may be passed to uninterpreted functions (UFs) and uninterpreted predicates (UPs) and returned from UFs. Atomic formulas (atoms) are made up of Boolean identifiers, UPs, and equalities and disequalities between terms. Formulas are made of atoms in arbitrary Boolean combinations:

0-arity term	*term*	$::=$	$\mathsf{identifier}$
UF		$\vert$	$\mathsf{F}(term_1, term_2, \ldots, term_n)$
Equality	*atom*	$::=$	$term_1 = term_2$
Boolean		$\vert$	$identifier$
UP		$\vert$	$\mathsf{P}(term_1, term_2, \ldots, term_n)$
Atom	*formula*	$::=$	$atom$
Not		$\vert$	$\neg formula$
And		$\vert$	$formula_1 \wedge formula_2$

We write uninterpreted terms T and functions (and predicates) $\mathsf{F}(\mathsf{X})$ in sans serif face. The semantics of these expressions is standard. In order to represent programs, we use the well-known formalism of a transition system.

Our abstraction approach is to substitute 0-arity terms for the corresponding values and UFs and UPs for the corresponding concrete operations (see [31], pp. 61ff, for a thorough discussion of EUF abstraction). To summarize: concrete constants $1, 2, \ldots, n$ are represented as unique uninterpreted 0-arity terms $\mathsf{K1}, \mathsf{K2}, \ldots, \mathsf{Kn}$; data operations such as addition, division, bit-extraction, etc. are represented with UFs; relational operators are represented as UPs; non-Boolean variables x are represented by 0-arity terms $\widehat{\mathsf{x}}$, and given a hat to distinguish them from constants. Boolean variables can be represented directly in EUF. For example, we would represent $x^+ = a + 3$ as $\widehat{\mathsf{x}}^+ = \mathsf{ADD}(\widehat{\mathsf{a}}, \mathsf{K3})$. The abstraction function $\mathcal{A}[\![.]\!]$ is formally defined in the appendix.

Formally, our *abstract transition system* consists of state variables $\widehat{X} = \{\widehat{\mathsf{x}}_1, \widehat{\mathsf{x}}_2, \ldots, \widehat{\mathsf{x}}_n\}$, input variables $\widehat{Y} = \{\widehat{\mathsf{y}}_1, \widehat{\mathsf{y}}_2, \ldots, \widehat{\mathsf{y}}_m\}$, and a set of next-state formulas $\{\widehat{f}_1(\widehat{X}, \widehat{Y}), \ldots, \widehat{f}_n(\widehat{X}, \widehat{Y})\}$. The transition relation is:

$$\widehat{T}\left(\widehat{X}, \widehat{Y}, \widehat{X}^+\right) \doteq \bigwedge_{1 \leqslant i \leqslant n} \left(\widehat{\mathsf{x}}_i^+ = \widehat{f}_i(\widehat{X}, \widehat{Y})\right) \tag{2.2}$$

where $\widehat{f}_i(\widehat{X}, \widehat{Y}) = \mathcal{A}[\![f_i(X, Y)]\!]$.

Consider a concrete formula ϕ and its EUF abstraction $\widehat{\phi}$; the relation of the concrete and abstract systems is give by the following relation [31]:

$$\models \widehat{\phi} \Longrightarrow \models \phi$$

That is, the concretization ϕ of any valid EUF formula $\widehat{\phi}$ is valid. Therefore, if the abstract system cannot reach an unsafe state, then the concrete system will also never reach it.

Deriving the abstract system takes linear time on the original concrete system. We now turn to the topic of checking the abstract system once it is created.

2.2 Abstract Reachability

EUForia implements an IC3-style reachability computation on the abstract transition system described in Section 2.1. The implementation's main novelty is that it checks an entirely uninterpreted transition system. Our implementation is most closely related to PDR (an abbreviation of Property Directed Reachability), which is a popular variant of IC3 [19]. IC3 and PDR were developed for hardware verification and operate on purely Boolean transition systems. While the basic flow of our adaptation is the same as PDR, there are some novelties that are specific to handling EUF transition systems. We begin with a review of PDR.

Briefly, PDR constructs an iteratively-deepened sequence of k over-approximate frames, each representing a set of states reachable in at most k steps from the initial state. The basic computation step in PDR is a query of the form $\phi(\text{Source}, \text{Target}) = \text{Source}(X) \wedge T(X, Y, X^+) \wedge \text{Target}(X^+)$ where Source and Target denote sets of current and next states. A satisfiable (SAT) query indicates the existence of a counterexample-to-induction (CTI), a state $s \in \text{Source}$ that has a transition to state $t^+ \in \text{Target}$ under some input assignment. This state s can now be used as a target cube and built on to find a path backward, eventually finding a counterexample. On the other hand, an unsatisfiable (UNSAT) query indicates the absence of transitions between Source and Target under any possible input assignment. This means Target is *blocked* and cannot be used to construct a counterexample.

In order to obtain a practical algorithm, satisfiable queries and unsatisfiable queries must be generalized. Our generalization procedure for satisfiable queries is specific to the EUF ATS, and is covered next.

2.2.1 Generalizing Satisfiable Queries

PDR performs generalization using ternary simulation at the bit level, which is not possible for the EUF abstract transition system. Other research has explored generalization specific to linear arithmetic [21, 22] and polyhedra [40], as well as weakest preconditions [32]. EUForia's generalization is a model-based simplification of the weakest precondition of the target state with respect to T.

Let $M = (\pi, \delta)$ be a model for a satisfiable one-step query. Consider a four-element set $\{a, b, c, d\}$ and the following partition of this set: $\{\{a, b\}, \{c\}, \{d\}\}$. We write this partition $\pi = \{a, b \mid c \mid d\}$. This partition states that the elements a and b are equal to each other, but c and d are distinct from one another and from a and b. The assignment δ is an assignment to each Boolean variable; we write $\delta(z) = 1$ (resp. 0) to indicate that under the assignment δ, variable z is true (resp. false).

The model M produced by the solver includes a (complete) partition π on all the terms in the formula as well as a Boolean assignment δ to all the Boolean variables and uninterpreted predicates. It is straightforward to encode this model as a first-order cube C_M: the partition is expressed as a conjunction of equalities and disequalities and the Boolean assignment by definition is a cube. The goal of cube generalization is to extract the cube containing the most extensions. One way to do this is to drop literals one at a time from C_M while the resulting formula still reaches the target cube.

This procedure can be improved by a cone-of-influence reduction: literals in M that do not occur in the next-state cones of the variables in $\text{Target}(X^+)$ may be dropped.

Example It can be difficult to see exactly what kinds of pre-states this procedure would produce, so we provide an example. Consider the target $t^+ = \mathsf{GT}(x_1^+, x_2^+)$ and the model $M = \{x_1 \mapsto 1, x_2 \mapsto 1, \mathsf{K1} \mapsto 2, \mathsf{ADD}[(\cdot,\cdot) \mapsto 2], \mathsf{GT}[(\cdot,\cdot) \mapsto \text{true}\}$. The syntax $x \mapsto v$ denotes that the model maps term x to the domain value v. $\mathsf{F}[(v_i) \mapsto v_r]$ means that the UF F evaluates to v_r on all terms mapped to domain value v_i; if $v_i = \cdot$, F unconditionally evaluates to v_r (this notation is straightforward but tedious to extend to multiple-argument functions and predicates, so we elide those details). Consider the following transition system:[1]

$$x_1^+ = \text{ITE}(x_1 = x_2, \mathsf{ADD}(x_1, \mathsf{K1}), x_2) \qquad x_2^+ = x_1$$

The $\text{WP}(T, t^+) = \mathsf{GT}(\text{ITE}(x_1 = x_2, \mathsf{ADD}(x_1, \mathsf{K1}), x_2), x_1)$. Simplify produces:

$$\phi_{simp} \equiv \mathsf{GT}(\mathsf{ADD}(x_1, \mathsf{K1}), x_1) \qquad \mathbb{A} \equiv \{(x_1 = x_2)\}$$

Literal ϕ_{simp} is excluded since it contains a UF. The generalized pre-state is then, $\tilde{s} \equiv (x_1 = x_2)$.

2.2.2 Generalizing Unsatisfiable Queries

If the query ϕ is unsatisfiable, then no $s \in \text{Source}$ transitions to any $t^+ \in \text{Target}$. In this case, we want to generalize by finding a cube c^* that represents a set of states $T \subseteq \text{Target}$ such that $\forall t \in T : \phi(\text{Source}, t^+)$ is unsatisfiable. We use a simple greedy scheme for finding a minimal unsatisfiable set. Recall that Target is encoded as a cube c. We drop literals l_i one at a time from c, each time querying $\phi(\text{Source}, c \setminus \{l_i\})$. If the query is still unsatisfiable, l_i is dropped permanently; otherwise l_i is put back and we proceed to the next literal in c.

2.2.3 Concrete Transition System Encoding

The concrete transition system is encoded using standard methods, but we summarize our approach here. Program variables – locals and globals – are represented as bit vectors of the appropriate bit widths. Location variables are used as labels of program statements in order to capture the program's control flow and can be encoded in a variety of ways (see, e.g., [15, 21, 23]). We choose to label each program statement with a single Boolean location variable that becomes true when that statement is reached and executed. This is different from other checkers which introduce a dedicated *pc* variable; our encoding is similar to [15].

[1]The motivation is that GT stands for the greater-than operation, but its concrete semantics are not required for this example.

Briefly, the transition equation for a location l_i expresses that the location becames active when there is a transfer of control from a predecessor location:

$$l_i^+ = \bigvee_{j \in \text{pred}(i)} (l_j \wedge c_{ji}) \tag{2.3}$$

where c_{ji} is the predicate on the transition from each predecessor location j to location i (pred(i) is the set of predecessors of location i).

The transition constraints for program variables are slighly more complex. Let Modify(v) be the set of locations where program variable v is on the left-hand-side of an assignment. Let $\langle \text{expr}_i \rangle$ denote the translation of the right-hand-side expression into the QF_BV logic. Let l_i range over Modify(v). Variable updates are represented with the following constraints:

$$\bigvee_{j \in \text{pred}(i)} (l_j \wedge c_{ji}) \rightarrow \big(v^+ = \langle \text{expr}_i \rangle\big) \text{ [Update]} \tag{2.4}$$

$$\left(\neg \bigvee_{j \in \text{pred}(i)} (l_j \wedge c_{ji})\right) \rightarrow \big(v^+ = v\big) \text{ [Preserve]} \tag{2.5}$$

Note that in equation (2.3), l_i^+ is defined as the disjunction in the antecedent of equations (2.4) and (2.5), so we could use l_i^+ in place of the disjunction.

Equations (2.3)-(2.5) can easily be adapted to produce small and large block encodings [41]. Our implementation uses a large-step encoding that is similar to the adjustable-block encoding [42].

EUForia provides a mode to generate this CTS and perform a concrete reachability check in a similar way to the abstract reachibility check. In fact, the concrete pre-state generalization is almost the same as the one used in the ATS, except that there are no UFs to handle in the CTS. MUS generalization is done identically in the concrete check. Our CTS checking algorithm is, to the best of our knowledge, unique among the bit vector adaptations of IC3/PDR. Other algorithms have used predicate abstraction (over BV predicates) [24], polytope domains [40], and unadorned weakest preconditions [23] (with a implicit encoding of program locations). This close correspondence between our concrete checker and our abstract checker makes our evaluation nearly "apples to apples."

The CTS must be used for refining the abstraction, in case the abstraction is too coarse to prove the property. We discuss refinement next.

Large Block Encoding Implementation

- Associated with every location l_i are two relations In_i and Out_i, as is typical for a data flow analysis.
- Our intent is that getting from l_j to l_i using edge e_{ji} is expressed as $\text{SP}(e_{ji}, \text{In}_i)$.

1. For all locations l_i, initially $\text{In}_i = \text{Out}_i = \mathit{true}$.
2. For the entry location l_0, there is no In_0 and $\text{Out}_0 \doteq l_0$.

3. For an error or loop header location l_i:

$$\begin{aligned}\mathrm{In}_i &\doteq \bigvee_{j\in \mathrm{pred}(i)} \mathrm{Out}_j \\ \mathrm{Out}_i &\doteq l_i^+\end{aligned} \tag{2.6}$$

This expresses the intent that these types of locations begin a new transition, so they do not propagate constraints on Out.

4. For any other location l_i:

$$\begin{aligned}\mathrm{In}_i &\doteq \bigvee_{j\in \mathrm{pred}(i)} \mathrm{Out}_j \\ \mathrm{Out}_i &\doteq \bigvee_{j\in \mathrm{pred}(i)} \wedge \mathrm{SP}(e_{ji}, \mathrm{In}_i)\end{aligned} \tag{2.7}$$

5. Flow these around the graph until no P or S changes.

For a single edge from l_1 to l_2. Assume edge says $i = i + 1$ and that In_1 is ϕ as shown below.

Iteration	Location	In	Out
1	1	ϕ_1	$l_1^+ = \phi_1$
	2	$l_1^+ = \phi_1$	$(l_2^+ = l_1) \wedge (i^+ = i + 1)$
2	1	T	$l_1^+ = \phi_1$
	2	$l_1^+ = \phi_1$	$(l_2^+ = l_1) \wedge (i^+ = i + 1)$

The transition relation for these locations is Out_i.

$$l_1^+ = \phi_1$$
$$(l_2^+ = l_1) \wedge (i^+ = i + 1)$$

Consider adding a predecessor $l_3 \to l_2$ to the graph, labeled with $i = i - 1$.

Iteration	Location	In	Out
1	1	ϕ_1	$l_1^+ = \phi_1$
	3	ϕ_3	$l_3^+ = \phi_3$
	2	$l_1^+ = \phi_1 \vee l_3^+ = \phi_3$	$(l_2^+ = l_1 \wedge i^+ = i + 1)$
			$\vee (l_2^+ = l_3 \wedge i^+ = i - 1)$
2	1	ϕ_1	$l_1^+ = \phi_1$
	3	ϕ_3	$l_3^+ = \phi_3$
	2	$l_1^+ = \phi_1 \vee l_3^+ = \phi_3$	$(l_2^+ = l_1 \wedge i^+ = i + 1)$
			$\vee (l_2^+ = l_3 \wedge i^+ = i - 1)$

The transition relation is then:

$$l_1^+ = \phi_1$$

$$l_3^+ = \phi_3$$

$$(l_2^+ = l_1 \wedge i^+ = i + 1) \vee (l_2^+ = l_3 \wedge i^+ = i - 1)$$

Consider $l_1 \to l_2 \to l_3$ with $e_{12} = (i = i + 1)$ and $e_{13} = (i = i + 2)$:

Iteration	Location	In	Out
1	2	T	$(l_2^+ = l_1) \wedge (i^+ = i + 1)$
	3	T	$(l_3^+ = l_2) \wedge (i^+ = (i + 1) + 2)$

Chapter 3

Refinement

When an abstract counterexample is found, it must be checked for feasibility against the concrete transition system. An n-step abstract counterexample is a sequence $\widehat{A}_1 \xrightarrow{\widehat{Y_1}} \widehat{A}_2 \xrightarrow{\widehat{Y_2}} \cdots \widehat{A}_{n-1} \xrightarrow{\widehat{Y}_{n-1}} \widehat{A}_n$where each $\widehat{A}_k$ is an abstract state cube ($1 \leq k \leq n$) and $\widehat{Y}_k$ is an abstract input predicate ($1 \leq k < n$). The abstract reachability algorithm guarantees that this is a continuous counterexample – that $\widehat{A}_{k-1} \wedge \widehat{Y}_{k-1} \wedge \widehat{T} \Rightarrow \widehat{A}_k^+$. However, it is possible that the abstract counterexample is not concretely feasible. Let A_k stand for the concretized cube corresponding to $\widehat{A}_k$; and similarly for Y_k. The abstract counterexample is spurious if:

1. A_k is unsatisfiable for some k, i.e., there are no concrete states that correspond to the abstract state cube; or

2. $A_{k-1} \wedge Y_{k-1} \wedge T \wedge A_k$ is unsatisfiable for some k, i.e., there are no concrete transitions that correspond to the abstract state transition; or

3. the concretized counterexample is discontinuous. This will happen if all concretized cubes and transitions are feasible but the transitions "land" on distinct concrete states in a concretized cube (see Figure 3.1).

There are many options for performing these feasibility checks and deriving suitable refinements from them if one or more of them fail (e.g., [43, 9, 7]). Our prototype currently performs a very simple procedure

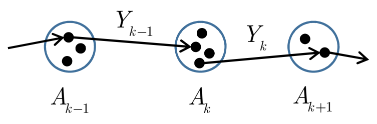

Figure 3.1: Illustration of a spurious abstract counterexample when all corresponding concrete cubes and transitions are feasible. The circles represent concrete cubes each containing a set of concrete states represented by the small filled circles.

for checking feasibility and deriving refinement lemmas:

1. Check SAT$(I \wedge A_0)$, i.e., check if the concrete counterexample is consistent with the initial state(s):

   a. If UNSAT, obtain from the concrete solver an unsatisfiable core $\{a_1, \ldots, a_m\}$ which is an unsatisfiable subset of the literals in I and A_0. EUForia adds the following (global) lemma to the abstract transition system, $\widehat{T}$: $\neg(\widehat{a_1} \wedge \cdots \wedge \widehat{a_m})$. Return immediately to abstract reachability.

   b. If SAT, call the satisfying assignment s_0, which is a single concrete state. A concrete program simulation is now begun using s_0 as the initial state.

2. For each $1 \leq k \leq n$, check SAT$(s_{k-1} \wedge T \wedge Y_{k-1} \wedge A_k)$, i.e., check that the current concrete state reaches a next state from the counterexample:

   a. If UNSAT, form a lemma in the same way as in step 1, add it to the abstraction, and return to abstract reachability. Otherwise, we continue.

   b. If all steps of the abstract counterexample are feasible, we have found a true counterexample, and verified it with the concrete transition system. Hence we return to the user with a property violation.

We chose this refinement procedure in our prototype for its simplicity and because our initial focus was on assessing the suitability and effectiveness of EUF abstraction for the class of control-centric properties we target. For the set of benchmarks we tested, most did not require even a single refinement, and for many that did require refinement, the number of refinement lemmas was modest. This suggests that EUF abstraction is a plausible over-approximation of program behavior for these benchmarks and associated properties. That said, the refinement procedure can still be improved significantly and we plan to explore several optimizations including:

1. Performing the feasibility checks on state cubes and transitions in parallel since they are largely independent.

2. Using the abstract counterexample to guide a concrete symbolic execution that can potentially rule out not just one but several infeasible concrete executions

3. Interleaving abstract reachability with feasibility checking: rather than wait for a complete abstract counterexample to be produced before checking its feasibility, concrete checks of intermediate abstract cubes can be done concurrently with the abstract reachability. Many options exist for orchestrating such an approach.

4. In any of the above schemes for feasibility checking, derive not just one but several refinement lemmas.

Several of these optimizations have been mentioned in the literature (e.g., [9, 25]). One element of future work is to evaluate them, as well as other variants, in the context of EUF abstract reachability.

Chapter 4

Evaluation

We evaluate euforia on 753 benchmarks from the SVCOMP'17 competition [44]. All the benchmarks are C programs in the ReachSafety-ControlFlow, ReachSafety-Loops, and ReachSafety-ECA sets. The benchmarks contain assertions and the checker's job is to find a path to that assertion or an invariant that proves safety. 516 are safe and 236 are unsafe. We elided all the benchmarks that use pointers or arrays, as well as those which took greater than 30 seconds to preprocess (typical instances take just a second or two in preprocessing). We ran all the benchmarks on 2.6 GHz Intel Sandy Bridge (Xeon E5-2670) machines with 2 sockets, 8 cores with 64GB RAM. Each benchmark was assigned to one socket during execution. Each benchmark was given a one hour timeout.

It is well known that the encoding technique one uses can drastically alter checking time. In order to make sure both checkers are using the same encoding of the transition system, we dump euforia's model checking problem (transition system and property encoding) into a `.vmt` file, which is readable by ic3ia. More information on the `vmt` format is available online[1]. It is a straightforward text format, based on SMT-LIB, that describes a model checking problem.

Our evaluation desires to answer the following questions:

1. How does euforia perform overall compared to ic3ia?
2. Where is the time spent during checking?
3. When euforia performs relatively well, why?
4. When euforia performs relatively poorly, why?
5. Does euforia require more cubes than ic3ia to accomplish verification? (unanswered, no added_cubes data)
6. How does convergence depth compare?

[1]`https://es-static.fbk.eu/tools/nuxmv/index.php?n=Languages.VMT`

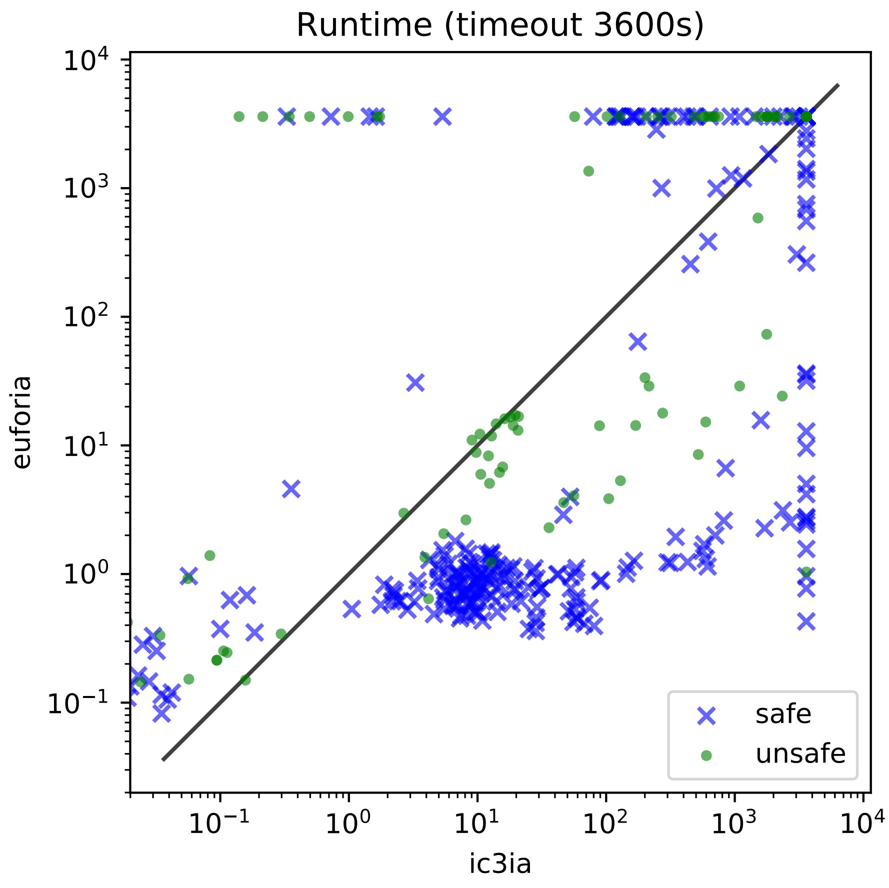

Figure 4.1: Plot of overall runtime in seconds

	euforia	ic3ia
avg	12.98	766.57
med	0.11	135.95

(a) Refinement time, euforia terminated (26 benchmarks)

	euforia	ic3ia
avg	937.65	154.27
med	975.41	81.59

(b) Refinement time, ic3ia terminated (61 benchmarks)

Figure 4.2: Summary statistics for timeout benchmarks (avg = average, med = median). Figure 4.2a is for the set of benchmarks on which ic3ia timed out but euforia solved. Figure 4.2b is the opposite: it's for the set of benchmarks on which euforia timed out but ic3ia solved.

Figure 4.1 shows our overall results on all benchmarks compared with ic3ia. euforia seems to win quickly or not at all. Due to the many timeouts of benchmarks completed by the other solver, euforia and ic3ia are to a certain extent complementary in what they are able to solve.

When euforia completes but ic3ia times out (26 cases), it is because the abstraction is matching the code, and euforia is able to spend very little time in refinement. Indeed, euforia is on average spending only 13 seconds in refinement on these benchmarks (see figure 4.2a), compared to 767 for ic3ia. Moreover, all but one of the benchmarks are safe, suggesting that euforia is able to quickly find inductive invariants for some hard instances (but it is less easy to find a counterexample). euforia solved 11 benchmarks without any refinements; the rest required up to 161 refinements.

ic3ia solves 61 of euforia's timeout cases. In all but a few of these the number of predicates is under 20, meaning interpolation is able to efficiently discover a small and sufficient set of predicates. euforia required dozens or hundreds of refinement steps for each and every benchmark, and spent much of its time in refinement (see figure 4.2b).

euforia seems to be better and finding invariants than it is at finding counterexamples (since the blue in figure 4.1 is lower than the green, and there are a bunch of green timeouts). [Why?]

Both solvers have three main phases: backward reachability, forward propagation, and refinement. The following table shows the average time spent in each of the phases for both checkers:

	Backward Reachability	Forward Propagation	Refinement
euforia	71%	3%	22%
ic3ia	54%	20%	24%

These averages were obtained on benchmarks where both checkers terminated within the timeout and euforia took at least two seconds.[2] The rest of the time is spent in preprocessing and optimizing the instance. Notably, the checkers' refinement times are similar but ic3ia spends relatively more in propagation and less in reachability. We conjecture that this is due to the fact that euforia tends to produce cubes with more literals than ic3ia, leading to more time spent minimizing cubes (see figure 4.3).

[2]If the runtime is less than two seconds, then it is dubious to rely on the runtime breakdown in the different parts of the

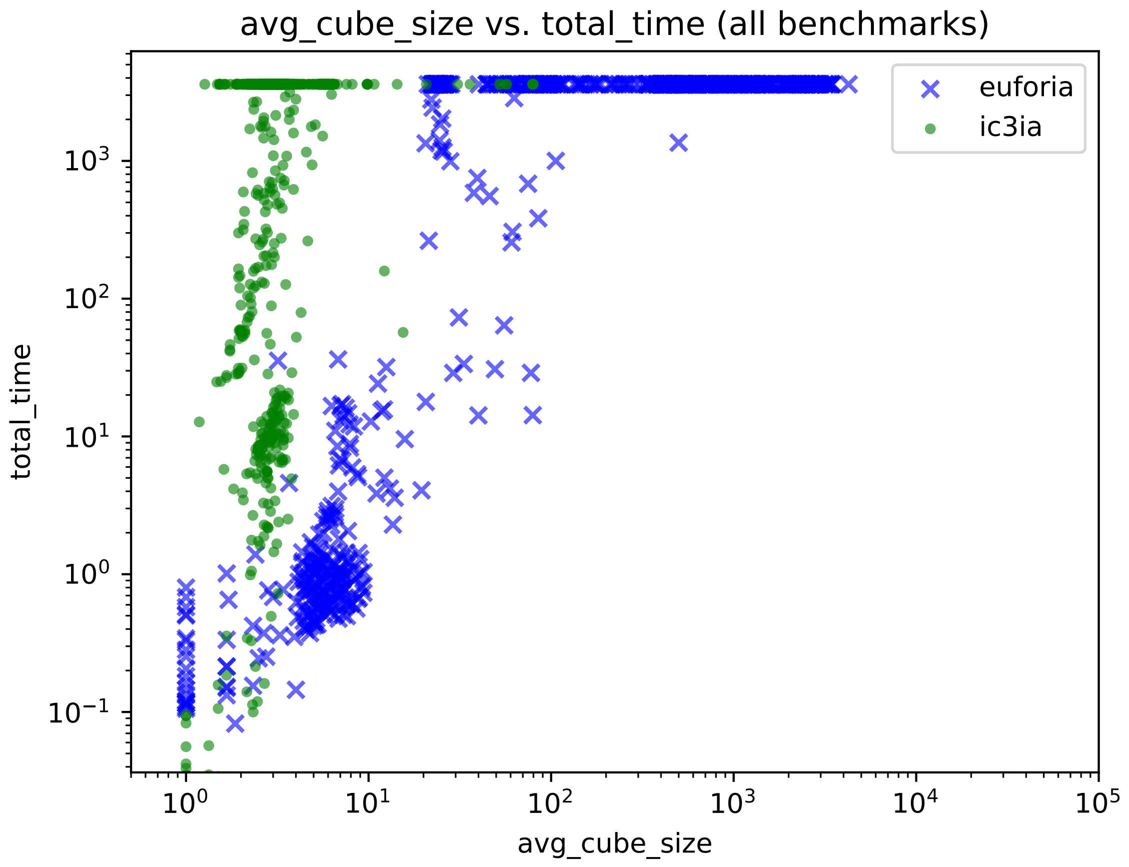

Figure 4.3: Plot showing average cube size compared to runtime. euforia's cubes are on average much larger than ic3ia and harder instances tend to have the largest cubes.

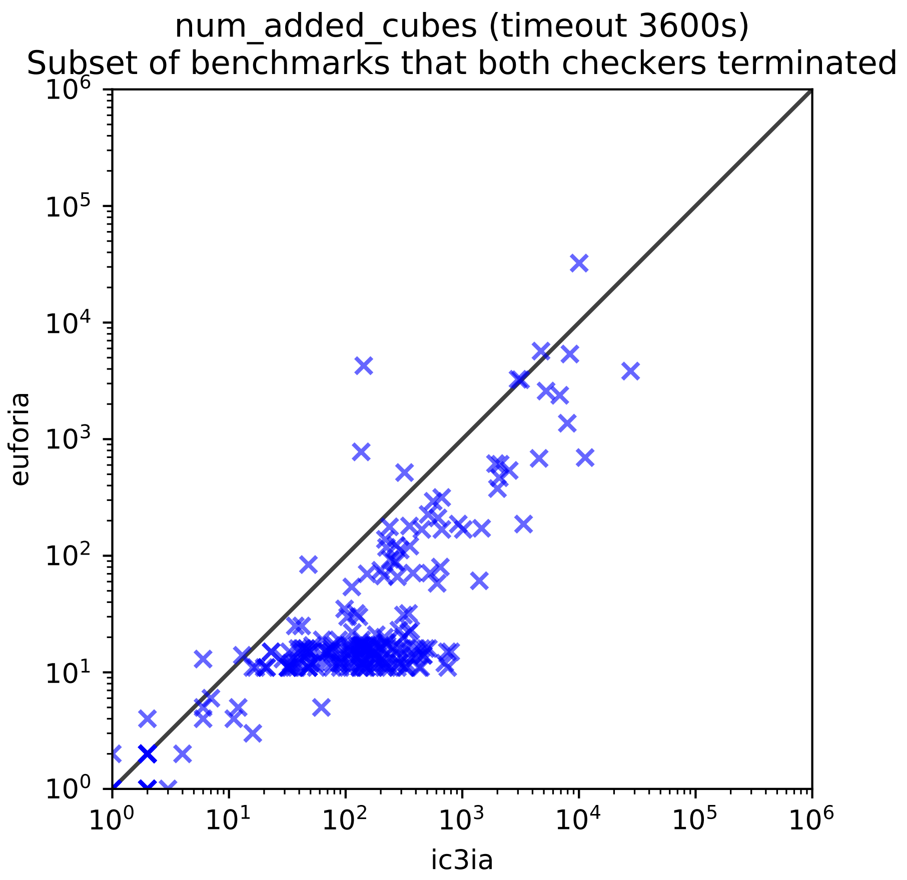

Figure 4.4: Plot showing, for all benchmarks solved by euforia and ic3ia, how many cubes were blocked during solving. Overall, euforia seems to add fewer cubes.

The fact that euforia's cubes contain more literals means they are relatively less general than ic3ia's cubes. Given a cube c in over fixed theory T (such as QF_BV), if we form $c' = c \wedge l$ by adding literal l to c, then (1) l could be redundant (i.e., c is equivalent to c'), or (2) l could further constrain c (i.e., $c' \rightarrow c$ but not vice versa). In case (2), we say that c' is less general than c. euforia is operating on cubes that are less general than perhaps they could be. Given a cube c over QF_BV, euforia is operating on cubes less general than its abstraction, $\mathcal{A}[\![c]\!]$. In future work, we will explore improving the cube expansion to strengthen the abstract cubes.

Figure 4.4 shows the number of cubes blocked during solving. Every time a cube is blocked, for any frame i, the running number is incremented. We note that generally, euforia is able to complete with fewer blocked cubes than ic3ia. Given that euforia's cubes are on average larger (i.e. contain more literals) than ic3ia, one might hypothesize that it would require more cubes to prove the property. This plot shows that euforia can use less general cubes, and fewer of them, while still proving the property.

solver, since the absolute time is so small. The measurements are dominated by noise.

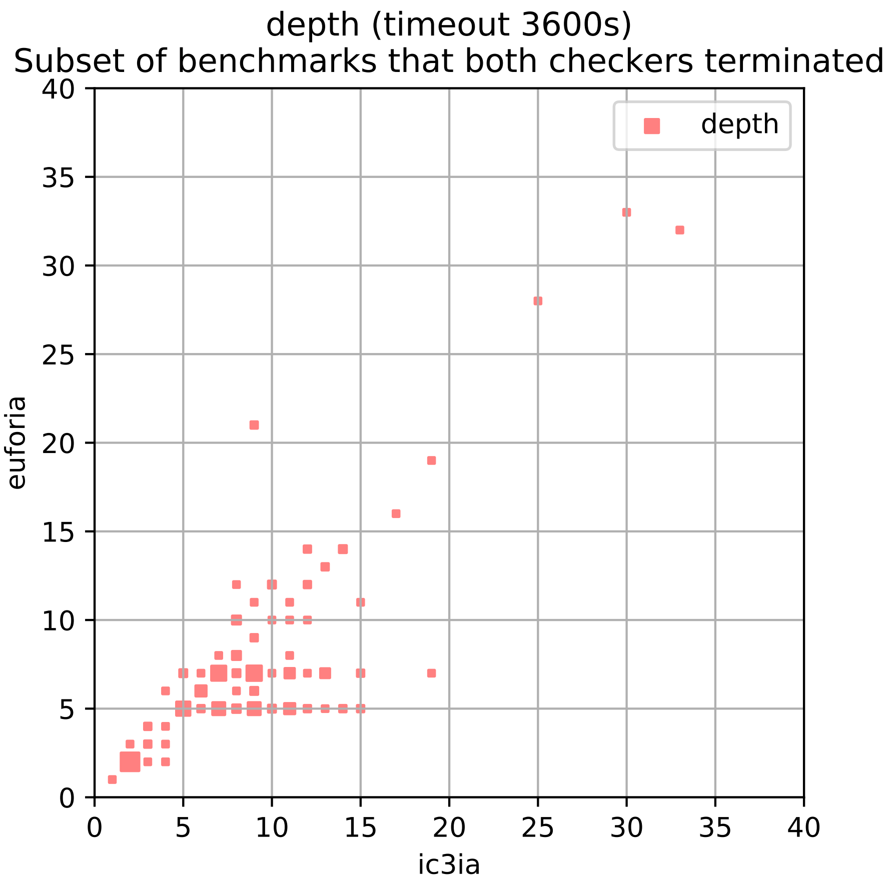

Figure 4.5: Plot showing the depth of IC3 after terminated, for both euforia and ic3ia. The area of the squares is proportional to the number of different benchmarks terminating at the given depths.

The termination depths of euforia and ic3ia are compared in figure 4.5. Generally, the termination depths of both checkers are comparable.

4.1 Conclusions and Future Work

We have presented an approach for the abstraction and verification of programs using EUF abstraction that has the following desirable traits.

- Abstraction and refinement are automatic and driven by the program itself.
- EUF abstraction is simple, because it preserves the structure of the transition system and can be computed in linear time over a given transition system.
- We have integrated in a simple way with modern, incremental inductive solving.
- Our initial experiments show that many benchmarks can be solved without any refinement which validates our conjection that EUF abstraction is plausible for control-centric safety properties.

We implemented EUF abstraction over a concrete transition system in a prototype tool, EUForia, and evaluated it on a set of 291 benchmarks from SV-COMP 2017.

For future work, we plan to expand the C language constructs that EUForia can handle: we plan to add static memory allocation, generic memory access, and support for recursion. These features will allow us to evaluate our approach on more, different kinds of software. Since most interesting programs have loops, we plan to explore how to leverage loop identification inside the EUForia algorithm. For example, it should be possible to map abstract counterexamples to much longer concrete counterexamples during feasibility checking; this has the potential to produce better refinement lemmas. We also plan to explore refinement strategies as detailed in Section 3.

Appendix

Encoding of phi variables as state variables Theorem. Within an LLVM function F, it is sufficient to designate left-hand-sides of PHI instructions as state variables and all other local variables as auxiliary variables.

Proof. In order for a variable to be encoded as an auxiliary, its defining expression must be *available* at every use. Consider a variable assignment `%x =` f `a b` at location y, where f may be a `phi` instruction or some other LLVM instruction. By definition, the expression f `a b` is available at the instruction immediately after y; the expression becomes unavailable at any definition of `a` or `b`, because then f `a b` will evaluate to a possibly different value. Thus, for any location z, the expression is available at z if there is no re-definition of `a` or `b` on any path from y to z (that doesn't go through y).

Variables are either defined as SSA temporaries or PHI instructions, so there are two cases to consider.

- If `%x =` f `a b` is a non-PHI variable, then SSA form guarantees two properties: (1) its definition dominates its use and (2) there is only one definition of `%x` in F. Therefore, the expression f `a b` is available at every use of `%x`. Hence, it may be encoded as an auxiliary variable. Logically, this is equivalent to replacing every use of `%x` with f `a b`.

- If `%x = phi ...` is defined by a PHI instruction, then it is possible that there is a use of `%x` in the definition. In this case, the value of `%x` may depend on its current assignment. In this case, the encoding must "remember" the previous value of `%x` in order to define it. Hence, it must be encoded as a state variable.

Therefore, it is sufficient to encode the left-hand-sides of PHI instructions as state variables and all other local variables as auxiliary variables.

4.1.1 EUForia soundness and termination

Lemma. There is a Galois connection for the EUF abstraction We define an abstraction function $\mathcal{A}[\![\cdot]\!]$ which returns the abstraction of a given concrete expression. We write $x^{[n]}$ to indicate that x is a concrete bit vector of bit width n.[3] We write $e^{\langle n\rangle}$ to indicate that the abstract term e is a term of uninterpreted sort $\mathsf{U}n$.

Boolean variables b and constants are translated to themselves. Bit vector variables $x^{[n]}$ and constants $c^{[n]}$ are translated to uninterpreted terms of sort $\mathsf{U}n$.

$$\begin{aligned}\mathcal{A}[\![b]\!] &\doteq b \text{ (for Boolean } b)\\ \mathcal{A}[\![c^{[n]}]\!] &\doteq \widehat{\mathrm{K}c}^{\langle n\rangle}\\ \mathcal{A}[\![x^{[n]}]\!] &\doteq \widehat{x}^{\langle n\rangle}\end{aligned}$$

Equalities are translated into abstract equalities and Boolean structure is preserved.

$$\begin{aligned}\mathcal{A}[\![a^{[n]} = b^{[n]}]\!] &\doteq \mathcal{A}[\![a^{[n]}]\!] = \mathcal{A}[\![b^{[n]}]\!]\\ \mathcal{A}[\![\neg a]\!] &\doteq \neg\mathcal{A}[\![a]\!]\\ \mathcal{A}[\![a \wedge b]\!] &\doteq \mathcal{A}[\![a]\!] \wedge \mathcal{A}[\![b]\!]\\ \mathcal{A}[\![\mathrm{ITE}(c,a,b)]\!] &\doteq \mathrm{ITE}(\mathcal{A}[\![c]\!], \mathcal{A}[\![a]\!], \mathcal{A}[\![b]\!])\end{aligned}$$

Each operation is rewritten to an uninterpreted variant on translated arguments. For example, for the signed greater-than and bitwise-or operators:

$$\mathcal{A}[\![x^{[n]} >_s y^{[n]}]\!] \doteq \mathrm{SGT}_n(\mathcal{A}[\![x]\!], \mathcal{A}[\![y]\!])$$

[3]We treat $n = 1$ as Boolean.

$$\mathcal{A}[\![x^{[n]} \mid y^{[n]}]\!] \doteq \mathrm{BOR}_n(\mathcal{A}[\![x]\!], \mathcal{A}[\![y]\!])$$

The rest of the operators follow this pattern.

This abstraction ensures type safety. LLVM is type safe and by ensuring that each concrete type is translated into its corresponding abstract type, we guarantee that the abstract system respect the types of the underlying values. Therefore the abstract system will never attempt to compare a byte with a 32 bit integer, for example.

Concretization works by performing the reverse mapping. Let F_c range over all the concrete operators implementing LLVM instructions (e.g., `add`); then let $\widehat{\mathsf{F}}$ be its corresponding uninterpreted function.

$$\begin{aligned}
\mathcal{D}[\![\widehat{\mathrm{K}c}^{\langle n\rangle}]\!] &\doteq c^{[n]} \\
\mathcal{D}[\![\widehat{\%\mathrm{x}}^{\langle n\rangle}]\!] &\doteq \%\mathrm{x}^{[n]} \\
\mathcal{D}[\![\widehat{\mathsf{F}}(x_1, x_2, \ldots, x_k)]\!] &\doteq F_c(\mathcal{D}[\![x_1]\!], \mathcal{D}[\![x_2]\!], \ldots, \mathcal{D}[\![x_k]\!]) \\
\mathcal{D}[\![\mathrm{ITE}(c, x, y)]\!] &\doteq \mathrm{ITE}(\mathcal{D}[\![c]\!], \mathcal{D}[\![x]\!], \mathcal{D}[\![y]\!]) \\
\mathcal{D}[\![x = y]\!] &\doteq \mathcal{D}[\![x]\!] = \mathcal{D}[\![y]\!] \\
\mathcal{D}[\![x \wedge y]\!] &\doteq \mathcal{D}[\![x]\!] \wedge \mathcal{D}[\![y]\!] \\
\mathcal{D}[\![\neg x]\!] &\doteq \neg \mathcal{D}[\![x]\!]
\end{aligned}$$

Let $\langle \mathcal{C}, \sqsubseteq \rangle$ denote a partial order on the concrete set of states $\mathcal{C}$ (implication is the ordering) and let $\langle \mathcal{H}, \preccurlyeq \rangle$ denote a partial order on the abstract set of states $\mathcal{H}$. A Galois connection between $\langle \mathcal{C}, \sqsubseteq \rangle$ and $\langle \mathcal{H}, \preccurlyeq \rangle$ is a pair of monotonic functions (α, γ) where $\alpha : \mathcal{P}(\mathcal{C}) \to \mathcal{P}(\mathcal{H})$ and $\gamma : \mathcal{P}(\mathcal{H}) \to \mathcal{P}(\mathcal{C})$ such that $\alpha(S) \preccurlyeq Y \iff S \sqsubseteq \gamma(Y)$. The abstraction function is α and the concretization function is γ.

Let P be a set of concrete states. Consider $\alpha(P)$. In EUF, more terms (values) may be equal than at the concrete level; and uninterpreted functions may have more behaviors than their concrete counterparts. Hence $P \sqsubseteq \gamma(\alpha(P))$.

Let $\widehat{Q}$ be an EUF formula (denoting a set of abstract states). By the same reasoning as above, some of these may be concretely infeasible. Or perhaps $\widehat{Q}$ is concretely infeasible (e.g., $\widehat{x} = K0 \wedge \widehat{x} > K1$). Because $\mathcal{D}[\![\widehat{Q}]\!]$ is defined syntactically, concretizing will produce a concrete formula. Because this formula behaves consistently wrt EUF, $\alpha(\gamma(\widehat{Q}))$ will only contain states in $\widehat{Q}$ (those implied by functional consistency, transitivity of equality, and other things). Therefore $\alpha(\gamma(\widehat{Q})) \preccurlyeq \widehat{Q}$. Hence we have established a Galois connection.

Lemma. The CTS has a finite state space. (Termination of ATS reachability) The ATS state space has finitely many models. Either there is a finite path from the goal back to the initial states (there are finitely many paths??) or there is no path back to the goal. If there is no path, then eventually the frames will be constrained by unreachable cubes so the reachability will terminate.

It is sufficient to refine by simulating the counterexample. If the counterexample is infeasible, this will be witnessed as some concrete transition that is infeasible. Some concrete query $c \wedge T \wedge A^+$ will be UNSAT. The unsatisfiable set of constraints extracted from this will rule out the offending transition from the abstraction.

Therefore, EUForia will always terminate, sooner or later, with either a counterexample to the property or a proof of the property.

Bibliography

[1] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, O. Grumberg, Ed., vol. 1254. Springer, 1997, pp. 72–83.

[2] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[3] R. P. Kurshan, *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, 1994.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 154–169.

[5] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," in *POPL*, J. Launchbury and J. C. Mitchell, Eds. ACM, 2002, pp. 1–3.

[6] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and cartesian abstraction for model checking C programs," ser. Lecture Notes in Computer Science, T. Margaria and W. Yi, Eds., vol. 2031. Springer, 2001, pp. 268–283. [Online]. Available: https://doi.org/10.1007/3-540-45319-9_19

[7] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *SPIN*, ser. Lecture Notes in Computer Science, T. Ball and S. K. Rajamani, Eds., vol. 2648. Springer, 2003, pp. 235–239.

[8] T. Ball and S. Rajamani, "Generating abstract explanations of spurious counterexamples in c programs," Tech. Rep., January 2002.

[9] T. Ball, E. Bounimova, R. Kumar, and V. Levin, "SLAM2: static driver verification with under 4% false alarms," in *Proceedings of International Conference on Formal Methods in Computer-Aided Design*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 35–42.

[10] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538, Springer. Springer, 2011, pp. 70–87.

[11] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Springer, 1994, pp. 68–80.

[12] D. Babic and A. J. Hu, "Structural abstraction of software verification conditions," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 366–378.

[13] ——, "Calysto: scalable and precise extended static checking," in *International Conference on Software Engineering*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008, pp. 211–220.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization*, Palo Alto, California, Mar 2004.

[15] Z. Manna and A. Pnueli, *Temporal verification of reactive systems – safety.* Springer, 1995.

[16] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[17] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[18] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0 system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 53–58, 2014 (published 2015).

[19] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Formal Methods in Computer-Aided Design.* IEEE, 2011, pp. 125–134.

[20] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer-Aided Design.* IEEE Computer Society, 2007, pp. 173–180.

[21] A. Cimatti and A. Griggio, "Software model checking via IC3," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. A. Seshia, Eds., vol. 7358. Springer, 2012, pp. 277–293.

[22] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 157–171.

[23] T. Lange, M. R. Neuhäußer, and T. Noll, "IC3 software model checking on control flow automata," R. Kaivola and T. Wahl, Eds. IEEE, 2015, pp. 97–104.

[24] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "IC3 modulo theories via implicit predicate abstraction," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 46–61.

[25] J. Birgmeier, A. R. Bradley, and G. Weissenbacher, "Counterexample to induction-guided abstraction-refinement (CTIGAR)," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 831–848. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_55

[26] N. Bjørner and A. Gurfinkel, "Property directed polyhedral abstraction," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, D. D'Souza, A. Lal, and K. G. Larsen, Eds., vol. 8931. Springer, 2015, pp. 263–281. [Online]. Available: https://doi.org/10.1007/978-3-662-46081-8_15

[27] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham, "Property-directed inference of universal invariants or proving their absence," *J. ACM*, vol. 64, no. 1, pp. 7:1–7:33, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/3022187

[28] T. Welp and A. Kuehlmann, "Property directed invariant refinement for program verification," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 114:1–114:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616746

[29] T. Welp, "Program verification with property directed reachability," Ph.D. dissertation, University of California, Berkeley, USA, 2013. [Online]. Available: http://www.escholarship.org/uc/item/29k4n5d4

[30] T. Welp and A. Kuehlmann, "Property directed invariant refinement for program verification," in *Design, Automation & Test*, G. P. Fettweis and W. Nebel, Eds. European Design and Automation Association, 2014, pp. 1–6.

[31] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-74105-3

[32] S. Lee and K. A. Sakallah, "Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Springer International Publishing, 2014, vol. 8559, pp. 849–865.

[33] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 184–190.

[34] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Cegar-based formal hardware verification: A case study," *Ann Arbor*, vol. 1001, pp. 48 109–2122, 2008.

[35] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 203–213.

[36] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 2–17.

[37] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys*, vol. 41, no. 4, pp. 21:1–21:54, Oct 2009.

[38] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[39] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking.* MIT Press, 2001. [Online]. Available: http://books.google.de/books?id=Nmc4wEaLXFEC

[40] T. Welp and A. Kuehlmann, "QF BV model checking with property directed reachability," in *Design, Automation & Test*, E. Macii, Ed. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 791–796.

[41] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Formal Methods in Computer-Aided Design.* IEEE, 2009, pp. 25–32.

[42] D. Beyer, M. E. Keremoglu, and P. Wendler, "Predicate abstraction with adjustable-block encoding," in *Proceedings of International Conference on Formal Methods in Computer-Aided Design*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 189–197. [Online]. Available: http://ieeexplore.ieee.org/document/5770949/

[43] D. Kroening, A. Groce, and E. M. Clarke, "Counterexample guided abstraction refinement via program execution," in *International Conference on Formal Engineering Methods*, ser. Lecture Notes in Computer Science, J. Davies, W. Schulte, and M. Barnett, Eds., vol. 3308. Springer, 2004, pp. 224–238. [Online]. Available: https://doi.org/10.1007/978-3-540-30482-1_23

[44] D. Beyer, "Software verification with validation of results - (report on SV-COMP 2017)," ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds., vol. 10206, 2017, pp. 331–349.

[45] R. Bloem and N. Sharygina, Eds., *Formal Methods in Computer-Aided Design.* IEEE, 2010.