EUFORIA: Complete Software Model Checking with Uninterpreted Functions

Denis Bueno and Karem A. Sakallah

{dlbueno,karem}@umich.edu
University of Michigan

Abstract. We introduce and evaluate an algorithm for an IC3-style software model checker that operates entirely at the level of equality with uninterpreted functions (EUF). Our checker, called EUFORIA, targets control properties by treating a program's data operations/relations as uninterpreted functions/predicates. This results in an EUF abstract transition system that EUFORIA analyzes to either (1) discover an inductive strengthening EUF formula that proves the property or (2) produce an abstract counterexample that corresponds to zero, one, or many concrete counterexamples. Infeasible counterexamples are eliminated by an efficient refinement method that constrains the EUF abstraction until the property is proved or a feasible counterexample is produced. 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.

1 Introduction

Control properties are an integral part of software verification. The 2014 Apple Secure Transport "goto fail" bug [1] provides a compelling illustration:

```
extern int f();
int g() {
   int ret = 0;
   /* ... */
   goto out; /* this line was inadvertently added */
   ret = f();
out:
   return ret;
}
```

In this simplified version of the bug, the function `f()` implements a security check that returns 0 on success. `g()` is supposed to call `f()`; however, `f()` is never called because there is an (inadvertent) jump directly to `g()`'s return statement. To prove the absence of this bug, one would like to verify the property that every success path actually calls `f()` (i.e., that `f()` is called whenever `g()` returns 0). This property does not require reasoning precisely about what `f()` does with data; it only requires reasoning about control paths. Consequently, this property is a *control property*.

A variety of important properties are control properties. For instance, many operating systems require that secure programs drop elevated privileges as soon as those privileges are no longer needed. Such a rule is a control property because it has little to do with details about particular privileged operations. Instead, the rule only requires reasoning about when privilege drops occur relative to the unprivileged parts of a program [2]. Similarly, verifying a locking discipline does not require reasoning about the data being protected; it only requires reasoning about when locking and unlocking occurs relative to when data is accessed or modified [3]. Typestate properties [4] are also control properties.

The typical approach for verifying control properties is predicate abstraction [5, 6], which casts the state space of a program into a Boolean space defined by a set of predicates over program variables. The primary challenge with predicate abstraction lies in the selection of predicates. All of the necessary information about data and control must be inferred using a finite set of predicates. Searching the predicate space has an exponential cost because adding a new predicate doubles the size of abstract state space. To make matters worse, predicate abstraction does not directly abstract operations, which can lead to time-consuming solver queries for complex operations – even though many complex operations are irrelevant for control properties.

Instead, we propose a more direct abstraction. Rather than projecting program state onto an interpreted predicate space, we syntactically abstract it into a set of constraints over the theory of equality with uninterpreted functions (EUF). This means that our abstraction can happen at the operation level (e.g., addition, subtraction, comparison, etc.) reducing the complexity of queries sent to the solver. Moreover, EUF reduces the number of bits in the search space (by abstracting bit vector terms), and has efficient implementations. The Averroes verifier [7] showed that such an approach works well for checking control properties in hardware designs.

This paper adapts IC3-style model checking with EUF abstraction to software. We find this gives performance benefits by reducing the number of refinement iterations in a counter-example-guided abstraction refinement (CEGAR) [8, 9] loop, while keeping the Boolean state space smaller. We make the following contributions:

- EUFORIA, a ground-up implementation of a complete software model checking algorithm inspired by Averroes (Section 3);
- detailed descriptions of EUFORIA's novel cube expansion method (Section 3.1) and refinement (Section 3.2), including new proofs of correctness and termination for finite state systems (Section 3.3),
- experimental evaluation on 752 from SV-COMP '17 (Section 4), showing that EUFORIA outperforms a related predicate abstraction algorithm, IC3IA [10], on control property benchmarks.

2 Software Data Abstraction

Our goal is safety verification: showing that all reachable states of a program are safe, or producing a counterexample test case. Kesten and Pnueli [11] made a distinction between control abstraction and data abstraction: while the former abstracts observations of computation sequences, the latter abstracts data values. We are targeting properties that involve verifying the control flow of a program, not its data, and thus we focus on abstracting data values using EUF theory.

This section describes the logic of EUF, how we represent a program (precisely) as a concrete transition system, and how we create an (over-approximate) abstract transition system from that concrete transition system.

2.1 Background

Equality with Uninterpreted Functions Our setting is standard quantifier-free, first-order logic (FOL) with the standard notions of theory, satisfiability, validity, entailment, and models. Inspired by Kroening's presentation in [12], we begin with a review of the EUF logic. The EUF logic grammar is presented here:

non-terminal	production	explanation
term ::=	$\mathsf{x} \mid \mathsf{y} \mid \mathsf{z} \mid \cdots$	0-arity term, sans serif face
|	$\mathsf{F}(term_1, term_2, \ldots, term_n)$	uninterpreted function (UF)
|	$\text{ITE}(formula, term_1, term_2)$	if-then-else
atom ::=	$term_1 = term_2$	equality atom
|	$x \mid y \mid z \mid \cdots$	Boolean atom
|	$\mathsf{P}(term_1, term_2, \ldots, term_n)$	uninterpreted predicate (UP)
formula ::=	*atom*	
|	$\neg atom$	negation
|	$formula_1 \wedge formula_2$	conjunction
|	$formula_1 \vee formula_2$	disjunction

Atomic formulas (atoms) are made up of Boolean identifiers, uninterpreted predicates (UPs), and (possibly-negated) equalities between terms. Formulas are made up of terms combined with arbitrary Boolean structure. For simplicity, but without loss of generality, we only consider formulas in negation normal form. A *literal* is a (possibly-negated) atom containing no occurrences of ITE. A *clause* is a disjunction of literals. A *cube* is a conjunction of literals. $a \models b$ means that a entails b. We write uninterpreted objects – terms x, functions F, and predicates P – in sans serif face. The semantics of these formulas is standard.

Transition Systems The front-end of our checker EUFORIA translates a C program into a bit-precise transition system. A *transition system* [13, 14] is a tuple (X, Y, I, T) consisting of a (non-empty) set of *state variables* $X = \{x_1, \ldots, x_n\}$, a (possibly empty) set of *input variables* $Y = \{y_1, \ldots, y_m\}$, and two formulas: I, the *initial states*, and T, the transition relation. Formulas over state variables are identified with the sets of states they denote; for example, the formula $(x_1 = x_2)$ denotes all states where x_1 and x_2 are equal, and other variables may have any value.

The set of *next-state variables* is $X' = \{x'_1, x'_2, \dots, x'_n\}$. For a formula σ, the set $\mathrm{Vars}(\sigma)$ denotes the set of state variables free in σ (respectively, $\mathrm{Vars}'(\sigma)$ denotes set of next-state variables in σ). We may write σ as $\sigma(X)$ when we wish to emphasize that the free variables in σ are drawn solely from the set X, i.e., $\mathrm{Vars}(\sigma(X)) \subseteq X$. Any formula $\sigma(X')$ (also written σ') refers to the result of substituting for the current-state variables in $\sigma(X)$ with the corresponding next-state variables from X', e.g., $(x_1 = x_2)'$ is $(x'_1 = x'_2)$. The system's *transition relation* T is a formula

$$T\,(X, Y, X') \equiv \bigwedge_{1 \leqslant i \leqslant n} (x'_i = f_i\,(X, Y)) \tag{1}$$

where $f_i\,(X, Y)$ is a term denoting the next-state function for $x_i \in X$.

We write $\sigma(X) \xrightarrow{T} \omega(X)$ if each state in σ transitions to some state in ω under T, i.e., $\sigma \wedge T \models \omega'$. An *execution* of a transition system is a (possibly-infinite) sequence of transitions $\sigma_0(X) \xrightarrow{T} \sigma_1(X) \xrightarrow{T} \sigma_2(X), \dots$ such that $\sigma_0(X) \models I(X)$.

A *safety property* is specified by a predicate, $P(\mathrm{X})$. The *model checking problem* is to check whether any state satisfying $\neg P(X)$ is reachable through an execution of T. A counterexample to a safety property $P(X)$ is a k-step execution such that $\sigma_k(X) \models \neg P(X)$.

A *concrete transition system* (CTS) is defined over bit vector state variables and operations in the quantifier-free logic of bit vectors (QF_BV from SMT-LIB [15]). EUFORIA encodes a C program into a CTS using standard methods [16, 17].

2.2 EUF Transition Systems

Inspired by the work of Burch & Dill [18] for microprocessor verification, our approach is to abstract a program's concrete operations (resp. conditions) by uninterpreted functions (resp. predicates), and to replace constants by 0-arity terms (Kroening also gives a detailed overview of EUF abstraction [12], pp. 61ff). Concrete constants (e.g., 1, −3) are represented as unique uninterpreted 0-arity terms (K1, K-3); data operations such as addition, division, and bit-extraction are represented with correspondingly-named UFs; relational operators are represented as UPs; and bit-vector variables x are represented by 0-arity terms $\widehat{\mathsf{x}}$, and given a hat to distinguish them from constants. Boolean variables are represented directly in EUF. We abstract P into $\widehat{P}$ and I into $\widehat{I}$ in the same way as other formulas. For example, using state variables $X = \{x, a\}$, we represent the transition relation $T(X, \varnothing, X') \equiv (x' = \mathsf{ITE}(x > a, x, 1 + a)) \wedge (a' = x)$ as $\widehat{T}(\widehat{X}, \varnothing, \widehat{X}') \equiv (\widehat{\mathsf{x}}' = \mathsf{ITE}(\mathsf{GT}(\widehat{\mathsf{x}}, \widehat{\mathsf{a}}), \widehat{\mathsf{x}}, \mathsf{ADD}(\mathsf{K1}, \widehat{\mathsf{a}}))) \wedge (\widehat{\mathsf{a}}' = \widehat{\mathsf{x}})$, over state variables $\widehat{X} = \{\widehat{x}, \widehat{a}\}$.

This abstraction can be formally defined by an *abstraction function* $\mathcal{A}[\![\cdot]\!]$ that performs a linear-time, syntax-directed, structure-preserving transformation of the CTS (described in [12]). The resulting *abstract transition system* (ATS) $(\widehat{X}, \widehat{Y}, \widehat{I}, \widehat{T})$ consists of state variables $\widehat{X} = \{\widehat{\mathsf{x}}_1, \widehat{\mathsf{x}}_2, \dots, \widehat{\mathsf{x}}_n\}$, input variables $\widehat{Y} = \{\widehat{\mathsf{y}}_1, \widehat{\mathsf{y}}_2, \dots, \widehat{\mathsf{y}}_m\}$, initial state $\widehat{I}$, and transition relation $\widehat{T}$ defined by n next-state terms $\widehat{f}_1(\widehat{X}, \widehat{Y}), \dots, \widehat{f}_n(\widehat{X}, \widehat{Y})$ according to:

$$\widehat{T}(\widehat{X}, \widehat{Y}, \widehat{X}') \equiv \bigwedge_{1 \leqslant i \leqslant n} \left(\widehat{\mathsf{x}}'_i = \widehat{f}_i(\widehat{X}, \widehat{Y})\right) \tag{2}$$

Abstract formulas over-approximate their concrete counterparts. Recovering the concrete formulas is easy: 0-arity terms (which stand for concrete constants and variables) are mapped to their concrete countererparts; UFs and UPs are mapped to their concrete operations by name. Consider a concrete formula $\sigma(X)$ and its EUF abstraction $\widehat{\sigma}(\widehat{X})$. The relation of the concrete and abstract systems is $\models \widehat{\sigma} \implies \models \sigma$: the concretization σ of any valid EUF formula $\widehat{\sigma}$ is valid [12]. Therefore, if the abstract system cannot reach an unsafe state, then the concrete system will also never reach it. A concrete state is a complete assignment to bit vector and Boolean variables. An abstract state is a pair $\langle \pi, A \rangle$ where π is a partition of all the terms in the ATS and A is a complete assignment to the UPs and Boolean variables.

The EUF abstraction partitions the set of all concrete states. Each concrete state is represented by a single abstract state but abstract states may represent zero, one, or many concrete states. For instance, given a transition system with one 32-bit integer state variable, x, and a single transition equation,

$$x' = 1 + x \qquad \text{concrete transitions}$$
$$\widehat{\mathsf{x}}' = \mathsf{ADD}(\mathsf{K1}, \widehat{\mathsf{x}}) \qquad \text{abstract transitions}$$

the abstract state space is defined over the term set $\{\widehat{\mathsf{x}}, \mathsf{K1}, \mathsf{ADD}(\mathsf{K1}, \widehat{\mathsf{x}})\}$ and consists of the following 5 states and their corresponding concrete states:[1]

Abstract state/partition	Concrete state(s)
$\pi_1 = \{\widehat{\mathsf{x}} \mid \mathsf{K1} \mid \mathsf{ADD}(\mathsf{K1}, \widehat{\mathsf{x}})\}$	$x \neq 1$ and $x \neq 0$
$\pi_2 = \{\widehat{\mathsf{x}}, \mathsf{ADD}(\mathsf{K1}, \widehat{\mathsf{x}}) \mid \mathsf{K1}\}$	$\varnothing$ (infeasible)
$\pi_3 = \{\widehat{\mathsf{x}} \mid \mathsf{K1}, \mathsf{ADD}(\mathsf{K1}, \widehat{\mathsf{x}})\}$	$x = 0$
$\pi_4 = \{\widehat{\mathsf{x}}, \mathsf{K1} \mid \mathsf{ADD}(\mathsf{K1}, \widehat{\mathsf{x}})\}$	$x = 1$
$\pi_5 = \{\widehat{\mathsf{x}}, \mathsf{K1}, \mathsf{ADD}(\mathsf{K1}, \widehat{\mathsf{x}})\}$	$\varnothing$ (infeasible)

We should note that while the CTS is deterministic, the abstraction causes the ATS to be non-deterministic.

3 EUFORIA: Model Checking EUF Transition Systems

EUFORIA builds on the model checker IC3 [19] by extending it to EUF and wrapping it inside a CEGAR loop that refines the abstract transition system. The algorithm's main novelties are that it checks an entirely uninterpreted transition system, is guaranteed to terminate, and refines spurious counterexamples automatically. Our implementation is most closely related to PDR (Property Directed Reachability) [20], a popular variant of IC3.

EUFORIA's entry point is given in Figure 1. We highlight algorithm components that EUFORIA introduces. As in IC3, the central object in EUFORIA is an iteratively-deepened sequence of reachable sets, R_i, each denoting an over-approximation of the set of states reachable in i transitions. The algorithm maintains the following

[1] Vertical bars delineate the cells of a partition

```
EUFORIA(I, T, P):
Globals:
   N                                   current depth
   F_i                                 set of cubes, i ∈ {0, 1, ..., N, N+1} (F_{N+1} = F_∞)
   R_i ≡ ⋀_{j=i}^{N+1} ⋀_{ĉ∈F_j} ¬ĉ     reachable set (over-approximate)

Î, T̂, P̂ ← abstract(I, T, P)                         ▷ construct abstract transition system
N = 0                                                ▷ initialize global variables
push F_∞ = true, push F_0 = {Î(X̂)}                   ▷ assume I is a cube
while true do
    if ∃ŝ ⊨ R_N ∧ ¬P̂ and BACKWARDREACHABILITY(ŝ) is true then
        if REFINECOUNTEREXAMPLE() is true then        ▷ found counterexample
            return BUILDCOUNTEREXAMPLE()
    else
        N ← N + 1, add new frame F_N = true
        if PROPAGATE() is true then                    ▷ found inductive invariant
            return true
```

Fig. 1. Entry point to EUFORIA. I, T, and P define a model checking problem. Backward reachability is performed until it converges or discovers an abstract counterexample, which may trigger a refinement. BUILDCOUNTEREXAMPLE() constructs a concrete program trace from a feasible abstract counterexample. R_i is a global definition in terms of the individual frames, stored in F.

invariants:

$$R_0 = \widehat{I}(\widehat{X}) \tag{3}$$

$$R_i \models R_{i+1} \tag{4}$$

$$R_i \models \widehat{P}(\widehat{X}) \qquad (i < N) \tag{5}$$

$$R_{i+1} \text{ over-approximates the image of } R_i \tag{6}$$

Initially EUFORIA abstracts the concrete transition system and then loops over three distinct phases: backward reachability (Figure 2), forward propagation (Figure 3), and refinement (Figure 6). This section will discuss the first two phases; refinement is discussed in Section 3.2.

Backward reachability (Figure 2) attempts to prove that the property holds for N transitions or to construct a counterexample. It manages a queue of proof obligations that represent potential executions to $\neg\widehat{P}$. At each iteration, it chooses a proof obligation pair $\langle \widehat{s}, i \rangle$ and performs a counterexample-to-induction (CTI) query to see if cube $\widehat{s}'$ is reachable from the current i-step over-approximation (lines 2–6). If so, our new procedure EXPANDPREIMAGE (Section 3.1) generalizes the pre-state and adds it to the queue (lines 6–9). Otherwise, it generalizes the unreachable cube $\widehat{s}$ to refine the reachability frames (lines 11–14). Note that this over-approximation and refinement is a standard part of IC3 and is independent of our EUF abstraction and refinement.

BACKWARDREACHABILITY($\widehat{s}$):
Precondition: cube $\widehat{s} \models \neg\widehat{P}$

push $\langle \widehat{s}, N \rangle$ onto Q
while $\langle \widehat{s}, i \rangle \leftarrow$ pop from Q **do** ▷ states $\widehat{s}$ reach bad state
  if $i = 0$ **then**
    return true ▷ found abstract counterexample
  if $\widehat{s} \wedge R_i$ is SAT **then** ▷ $\widehat{s}$ might be reached in i transitions
    if $\neg\widehat{s} \wedge R_{i-1} \wedge \widehat{T} \wedge \widehat{s}'$ has model M **then**
      $\widehat{z} \leftarrow$ EXPANDPREIMAGE($\widehat{s}'$, M) ▷ $\widehat{z}$ reaches $\widehat{s}$ in one step
      push $\langle \widehat{z}, i-1 \rangle$ onto Q ▷ new part of partial counterexample
      push $\langle \widehat{s}, i \rangle$ onto Q ▷ may still be reachable
    else ▷ $\widehat{s}$ is inductive relative to $\neg\widehat{s} \wedge \widehat{R}_{i-1}$
      $\langle \widehat{z}, m \rangle \leftarrow$ GENERALIZEBLOCKEDCUBE($\langle \widehat{s}, i \rangle$) ▷ $m \geq i$
      while $m < N-1$ and $\neg\widehat{z} \wedge R_{m-1} \wedge \widehat{T} \wedge \widehat{z}'$ is UNSAT **do**
        $\langle \widehat{z}, m \rangle \leftarrow$ GENERALIZEBLOCKEDCUBE($\langle \widehat{z}, m \rangle$) ▷ attempt to block at later frame
      ADDBLOCKEDCUBE($\langle \widehat{z}, m \rangle$)
      if $m < N$ **then**
        push $\langle \widehat{z}, m+1 \rangle$ onto Q ▷ may still be reachable at $m+1$
return false

ADDBLOCKEDCUBE($\langle \widehat{s}, i \rangle$):

for $j \in \{1, 2, \ldots, i\}$ **do** ▷ test whether $\widehat{s}$ subsumes a cube in an earlier frame
  if $\widehat{s} \subseteq \widehat{c}$ for any $\widehat{c} \in F_j$ **then**
    $F_j \leftarrow F_j \setminus \{\widehat{c}\}$
$F_i \leftarrow F_i \cup \{\widehat{s}\}$ ▷ record that $\widehat{s}$ is unreachable in i steps

Fig. 2. Proof obligations are represented as an abstract cube and frame index pair, $\langle \widehat{s}, i \rangle$. The proof obligation queue, Q, is a priority queue that orders cubes by frame index (earliest first) and breaks ties arbitrarily.

PROPAGATE():

1: **for** $i \in \{1, 2, \ldots, N-1\}$ **do** ▷ Propagate at level i
2:   **for** $\widehat{s} \in F_i$ **do**
3:     **if** $R_i \wedge \widehat{T} \wedge \widehat{s}'$ is UNSAT **then** ▷ $\widehat{s}$ is blocked at F_{i+1} or later
4:       $m \leftarrow$ maximum in $\{i+1, i+2, \ldots, N+1\}$ at which $\widehat{s}$ is blocked
5:       ADDBLOCKEDCUBE($\langle \widehat{s}, m \rangle$) ▷ propagate cube $\widehat{s}$ to F_m
6:   **if** F_i is empty **then**
7:     **return** true ▷ invariant found
8: **return** false

Fig. 3. Just prior to this phase of EUFORIA, $R_N \models \widehat{P}$. N is incremented and then PROPAGATE is called. In line 4, it is possible that a cube is blocked *beyond* the next frame $(i+1)$. EUFORIA examines the unsat core given by the solver to see which frames were used in order to calculate m.

Forward propagation (Figure 3) pushes unreachable cubes forward, attempting to extend them over more transitions (lines 1–5). On line 6, if two (over-approximate) reachable sets become identical $R_i = R_{i+1}$ ($i < N$), the algorithm terminates having discovered an inductive invariant that proves the property by equation (5).

Generalizing Unsatisfiable CTI Queries If the CTI query (line 6 of Figure 2) is unsatisfiable, then state $\widehat{s}$ is unreachable in i transitions. We want to generalize $\widehat{s}$ by finding a set of states (a cube) $\widehat{m} \supseteq \widehat{s}$ that is unreachable and covers more states than $\widehat{s}$, if possible. We use a simple greedy scheme for finding a minimal unsatisfiable set that is given in Figure 4.

GENERALIZEBLOCKEDCUBE($\langle \widehat{s}, i \rangle$):
1: $\widehat{t} \leftarrow \widehat{s}$, $j \leftarrow i$
2: **for** $\widehat{l} \in \widehat{s}$ **do**
3: $\quad \widehat{m} \leftarrow \widehat{t} \setminus \widehat{l}$ $\qquad \triangleright$ test if $\widehat{m}$ unreachable if literal $\widehat{l}$ removed
4: $\quad$ **if** $\widehat{m} \not\models I(\widehat{X})$ and $\neg\widehat{m} \wedge R_{j-1} \wedge \widehat{T} \wedge \widehat{m}'$ is UNSAT **then**
5: $\qquad j \leftarrow$ frame $\geq j$ at which $\widehat{m}$ is still blocked
6: $\qquad \widehat{t} \leftarrow \widehat{m}$ $\qquad \triangleright$ literal $\widehat{l}$ was not necessary
7: **return** $\langle \widehat{t}, j \rangle$

Fig. 4. Generalized blocked cube procedure. EUFORIA, like PDR, examines the unsat core of the query on line 4 in order to implement line 5.

3.1 Generalizing Satisfiable Counterexample-to-induction Queries

If the CTI query (line 6 of Figure 2) is satisfiable, EUFORIA generalizes (expands) the preimage state to a cube that includes many states that satisfy the query. The purpose of generalization is efficiency: a bad state is often reached by many states and it is usually more efficient to find counterexamples if state sets contain as many states as possible.

Example 1. Consider the following transition relation on variables $\widehat{X} = \{\widehat{x}_1, \widehat{x}_2\}$:

$$\widehat{x}_1' = f_1 \text{ where } f_1 = \text{ITE}(\widehat{x}_1 = \widehat{x}_2, \mathsf{ADD}(\widehat{x}_1, \mathsf{K1}), \mathsf{SUB}(\widehat{x}_1, \mathsf{K3})) \tag{7}$$

$$\widehat{x}_2' = f_2 \text{ where } f_2 = \widehat{x}_1 \tag{8}$$

Consider a proof obligation cube $\widehat{s}' \equiv \mathsf{GT}(\widehat{x}_1', \widehat{x}_2')$ and a model consisting of partition $\{\widehat{x}_1, \widehat{x}_2, \widehat{x}_2' \mid \mathsf{K1}, \mathsf{ADD}(\widehat{x}_1, \mathsf{K1}), \widehat{x}_1' \mid \mathsf{K3}, \mathsf{SUB}(\widehat{x}_1, \mathsf{K3})\}$ and assignment $\mathsf{GT}(\widehat{x}_1, \widehat{x}_2) \wedge \mathsf{GT}(\widehat{x}_1', \widehat{x}_2')$. EUFORIA performs a cone-of-influence (COI) traversal on f_1 and f_2 to find relevant constraints, terms, and variables; in this case, it finds the constraint $(\widehat{x}_1 = \widehat{x}_2)$, as well as terms $\mathsf{K1}, \mathsf{ADD}(\widehat{x}_1, \mathsf{K1})$, and variables $\widehat{x}_1, \widehat{x}_2$. It does not find the $\mathsf{SUB}(\cdots)$ term because it only traverses the true branch of the ITE. Relating these constraints, terms, and variables according to the model yields

ExpandPreimage($\widehat{s}'$, M):
1: $C \leftarrow \varnothing$ ▷ set of constraints
2: **for** $\widehat{x}_i' \in \text{Vars}'(\widehat{s}')$ **do**
3: $\quad c \leftarrow \text{COI}(f_i(\widehat{X}, \widehat{Y}), M)$ ▷ traverse f_i to collect M-relevant constraints
4: $\quad C \leftarrow C \cup c$
5: $\widehat{g} \leftarrow$ restrict model M to variables, terms, and predicates in C
6: **return** $\widehat{g}$

Fig. 5. Pre-image generalization procedure. M is the model for the CTI query. COI(f, M) is a model-based cone of influence traversal.

our generalized pre-image cube: $(\widehat{x}_1 = \widehat{x}_2) \wedge (\mathsf{ADD}(\widehat{x}_1, \mathsf{K1}) = \mathsf{K1}) \wedge (\widehat{x}_1 \neq \mathsf{K1})$. This has the effect of generalizing away the predicate $\mathsf{GT}(\widehat{x}_1, \widehat{x}_2)$. We omit the COI traversal details due to space constraints and because it is relatively straightforward: for each variable $\widehat{x}_i' \in \text{Vars}'(\widehat{s}')$, its next-state formula $f_i(X, Y)$ is traversed, collecting constraints required to satisfy the model. Then those constraints are used to form the pre-state cube.

EUFORIA's expansion procedure, given in Figure 5, has two key properties: (1) it projects only onto constraints from $\widehat{T}$ and (2) it exploits the fact that $\widehat{T}$ represents each next-state relation as a function in order to perform a COI traversal on each next-state function $f_i(X, Y)$. This allows us to omit irrelevant state variables and constraints. Property (1) is important for guaranteeing termination and (2) is important for efficiency.

CTI expansion is common to many IC3-style checkers. CTIGAR [21] generalizes by examining the unsatisfiable core of a query that is unsatisfiable by construction: it asks whether a state has, under the same inputs, some other successor than the reached one [21]. EUFORIA can't use this method to generalize because such a query may be satisfiable over EUF (due to the non-deterministic nature of UFs). PDR performs generalization using ternary simulation at the bit level, which is not suitable for the word-level EUF abstract transition system. Other checkers have explored theory-specific generalization methods, such as for linear arithmetic [22, 23] and for polyhedra [24]. Yet other checkers generalize by calculating the weakest precondition for the proof obligation [25, 7]. Weakest preconditions (WP) are particularly problematic for EUF, as iterated applications of WP can cause EUF terms to grow arbitrarily large, leading to potential non-termination of EUF abstract reachability.

3.2 Refinement

When BackwardReachability finds an abstract counterexample, it must be checked for feasibility, potentially refining the abstract state space. An n-step abstract counterexample (ACX) is an execution $\widehat{A}_0 \xrightarrow{\widehat{T} \wedge \widehat{Y}_0} \widehat{A}_1 \xrightarrow{\widehat{T} \wedge \widehat{Y}_1} \cdots \xrightarrow{\widehat{T} \wedge \widehat{Y}_{n-2}} \widehat{A}_{n-1} \xrightarrow{\widehat{T} \wedge \widehat{Y}_{n-1}} \widehat{A}_n$ where each $\widehat{A}_i$ $(0 \leq i \leq n)$ is a state cube and $\widehat{Y}_i$ $(0 \leq i < n)$

is a cube constraining input variables. An abstract formula $\widehat{\sigma}$ is *feasible* if its concretization σ is satisfiable over `QF_BV`. The ACX is spurious for any of the following reasons:

1. A_i is infeasible for some i, i.e., there are no concrete states that correspond to the abstract state cube $\widehat{A}_i$; or
2. $A_{i-1} \wedge Y_{i-1} \wedge T \wedge A_i$ is unsatisfiable for some i, 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. Below, the circles represent concrete cubes and the dots represent concrete states:

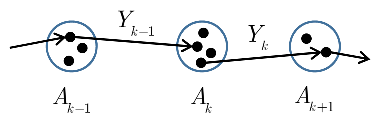


Figure 6 shows EUFORIA's refinement algorithm. RefineCounterexample first performs feasibility checks on individual transitions to address reasons 1 and 2 (Figure 6a, lines 1–8), afterward performing symbolic simulation on the counterexample path to address reason 3 (Figure 6b). If the counterexample is spurious, one of these feasibility checks will find an unsatisfiable subset of constraints. LearnLemma creates a *refinement lemma* by abstracting the unsatisfiable subset and asserting its negation in $\widehat{T}$.

The details of forward refinement are fiddly but the idea is simple: to determine if the counterexample is feasible, symbolically simulate the program along the concretized counterexample path. Beginning in the initial state, our implementation iteratively computes the next state in a manner reminiscent of image computation in BDD-based symbolic model checking. Note that there is no path explosion during this process because we only follow the path denoted by the concrete counterexample. If a contradiction is reached, then an unsatisfiable subset is found and used to learn a lemma.

Specifically, RefineForward (Figure 6b) represents a symbolic state s_i as a pair $\langle v_i, pc_i \rangle$ where v_i represents a map of state variables to values, and pc_i is the path constraint represented as a set of cubes. One transition at a time, it asks whether the next transition in the abstract counterexample is concretely feasible. If it is, Simulate (Figure 6c) computes the next state symbolically, in two steps: (1) updating variable assignments by symbolically evaluating each next-state function in T (as was done during cube expansion, Section 3.1), (2) updating the path constraint with any new input constraints, and (3) uniquely renaming all input variables. The notation $f_i[X/v_{i-1}]$ denotes the simultaneous substitution of state variables in X for their values from v_{i-1} in f_i. For a formula g with model M, $g \downarrow M$ simplifies g to a literal (by removing any complex Boolean logic) using the model M, similar to our COI procedure (see Section 3.1).

As we have said, the symbolic formula created by this process represents a single execution path through the program being analyzed, with inputs remaining

```
RefineCounterexample( Â_0 --T̂∧Ŷ_0--> Â_1 --T̂∧Ŷ_1--> ··· --T̂∧Ŷ_{n−2}--> Â_{n−1} --T̂∧Ŷ_{n−1}--> Â_n ):
if n = 1 then
    if A_0 is UNSAT, with unsat core c then        ▷ check for 0-step counterexample
        LearnLemma(c)
        return false
for i ∈ {1, 2, 3, ..., n} do                        ▷ test cubes and transitions
    if A_{i−1} ∧ T ∧ Y_{i−1} ∧ A_i is UNSAT, with unsat core c then
        LearnLemma(c)
        return false
return RefineForward()
```

(a) Refinement entry point

```
RefineForward() :
if I ∧ A_0 is UNSAT, with unsat core c then         ▷ check initial state
    LearnLemma(c)
    return false
s_1 ← ⟨concrete assignment for each state variable, {}⟩
for i ∈ {2, 3, ..., n} do                           ▷ test cubes and transitions
    if v_{i−1} ∧ pc_{i−1} ∧ T ∧ Y_{i−1} ∧ A'_i is UNSAT, with unsat core c then
        LearnLemma(c)
        return false
    s_i ← Simulate(M, s_{i−1}, T, Y_{i−1}, A_i)     ▷ M is the model for the query
return true                                         ▷ feasible counterexample
```

(b) Symbolically simulate counterexample

```
Simulate(M, ⟨v_{i−1}, pc_{i−1}⟩, T, Y_{i−1}, A_i) :
v ← empty map
for x_i ∈ X do
    update v with value f_i[X/v_{i−1}] ↓ M    ▷ substitute last values, simplify with M
pc ← Y_{i−1} ∪ {l[X/v] | l ∈ A_i and l[X/v] contains inputs}
return ⟨RenameInputs(v), RenameInputs(pc)⟩
```

(c) Steps a symbolic state $s_{i-1} = \langle v_{i-1}, pc_{i-1} \rangle$ forward one step by updating values (v) and path constraint (pc) using T

```
LearnLemma(c) :
ĉ ← AbstractAndNormalize(c)          ▷ abstract and eliminate input variables
if c contains no inputs then
    if Vars(c) ⊆ X then              ▷ only present-state vars
        Simplify and add lemma ¬ĉ(X̂')
    if Vars(c) ⊆ X' then             ▷ only next-state vars
        Simplify and add lemma ¬ĉ(X̂)
Simplify and add lemma ¬ĉ
```

(d) Learns a lemma by abstracting the concrete core c and conjoining $\widehat{c}$ to $\widehat{T}$

Fig. 6. EUFORIA's refinement procedure, RefineCounterexample

symbolic. If this formula is found to be unsatisfiable, then it is desirable to find an equivalent formula without symbolic input variables. A full-fledged quantifier elimination procedure is computationally expensive. Instead, LearnLemma (Figure 6d) calls AbstractAndNormalize, which (1) performs some simple equality propagation (which often will eliminate the inputs) and (2) otherwise under-approximates by substituting for each input variable the last concrete value that was assigned during symbolic simulation.

euforia's refinement lemmas fall into two categories: (1) *one-step lemmas* learned during individual transition checks (lines 1–8 in Figure 6a); and (2) *forward lemmas* learned during the symbolic simulation of the concrete counterexample (Figure 6b). The key fact is that one-step lemmas *do not increase* the size of the abstract state space; they merely constrain existing terms, similar to a blocking clause in ic3. One-step lemmas constrain the behavior of uninterpreted objects to be consistent with their concrete semantics, i.e., partially interpreting the uninterpreted operations. Forward lemmas, on the other hand, increase the size of the abstract state space, similar to predicates added by refinement in predicate abstraction.

There are many options for performing feasibility checks and deriving suitable refinements from them if one or more of them fail (e.g., [26–28]). We chose this refinement procedure because our focus is on assessing the suitability of EUF abstraction for control properties, and because it's simple.

3.3 Proof of Correctness

First, we prove that reachability for EUF transition systems terminates. Second, we show that euforia's refinement will increase the fidelity of the abstract system until it represents all concrete states exactly. Since the concrete system is finite, euforia must eventually terminate.

Theorem 1. BackwardReachability *terminates with an answer of* true *or* false*.*

Proof. Our proof relies on two facts: (1) the number of models for an abstract transition system is finite and (2) euforia searches among these models only, eventually blocking all of them or producing an abstract counterexample.

The set of possible models for a given abstract transition system $\hat{T}$ is finite. In fact, if the system has k Boolean state variables and n terms, then the number of Herbrand models is bounded by $2^k \cdot B_n$, where 2^k is the number of possible Boolean assignments to k Boolean variables and $B_n = \sum_{i=0}^{n} S(n,i)$ is the number of ways to partition n objects into disjoint sets (the Bell number). $S(n,i)$ is the number of ways to partition a set of n objects into i non-empty subsets (Stirling number of the second kind).

euforia's preimage generalization procedure, ExpandPreimage (Figure 5), searches only among this bounded set of models, since it explicitly uses only terms from $\widehat{T}$ to construct its preimage cube. If a cube is subsequently blocked by GeneralizeBlockedCube (Figure 4), those models will be infeasible. As there are finitely many models and frames, eventually all cubes will be blocked and BackwardReachability will terminate.

Theorem 2. EUFORIA*'s refinement procedure increases the fidelity of the abstract transition system (ATS), up to expressing all concrete* `QF_BV` *behavior.*

Proof. One-step lemmas do increase the fidelity of the ATS but do not increase the number of terms in the ATS. RefineForward may increase the number of terms in the ATS, resulting in an increased state space. If the state space size could grow without bound, EUFORIA would potentially not terminate.

We first show that we can guarantee termination by using a refinement method simpler than RefineForward. This method learns a lemma from a single concrete path. Recall that an n-step abstract counterexample is an execution $\widehat{A}_0 \xrightarrow{\widehat{T}\wedge\widehat{Y}_0} \widehat{A}_1 \xrightarrow{\widehat{T}\wedge\widehat{Y}_1} \cdots \xrightarrow{\widehat{T}\wedge\widehat{Y}_{n-2}} \widehat{A}_{n-1} \xrightarrow{\widehat{T}\wedge\widehat{Y}_{n-1}} \widehat{A}_n$ where each $\widehat{A}_i$ is an abstract state cube ($0 \leq i \leq n$) and $\widehat{Y}_i$ is an abstract formula constraining input variables ($0 \leq i < n$). Beginning in any single state $\sigma_0 \in A_1 \wedge I$, for all $1 \leq i \leq n$,

1. Check whether $\sigma_{i-1} \wedge T \wedge Y_{i-1} \wedge A'_i$ is satisfiable.
2. If so, form new state σ_i using the concrete assignments to all variables X'
3. If not, call LearnLemma(c) where c is the unsat subset of the query (1.)

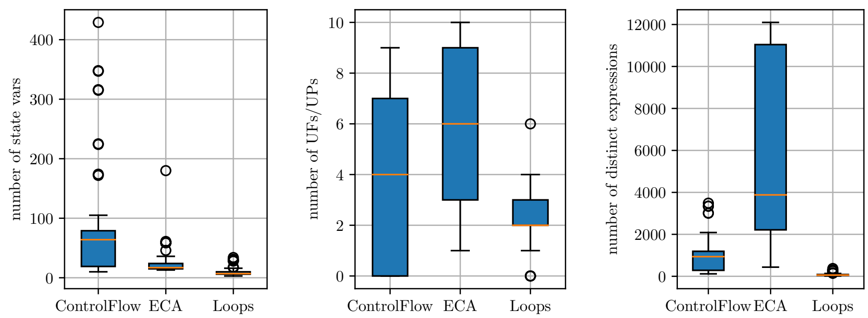
When step 1 is not satisfiable, this procedure will introduce *a new abstract constant* (from state σ_{i-1}) and *a new abstract UF/UP constraint* (due to the transition to A'_i) on that constant. The number of constants is bounded by the size of bit vector words in the concrete transition system and the number of constraints is as well (up to modeling every concrete behavior of every UF/UP in the program).

RefineForward (Section 3.2) is essentially the same as this procedure, except RefineForward attempts to generate stronger lemmas that refute multiple spurious concrete paths at once.

4 Evaluation

EUFORIA is implemented in 13,700 lines of C++. It uses LLVM 5.0.1 as front-end for processing C programs, running various optimizations including inlining, dead code elimination, and promoting memory to registers. It uses Z3 4.5.0 [29] for EUF solving during backward reachability and Boolector 2.0 [30] for `QF_BV` solving during refinement. EUFORIA cannot yet process programs with 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.

We evaluated EUFORIA on 752 benchmarks containing safety property assertions from the SV-COMP'17 competition [31]. 516 are safe and 236 are unsafe. 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 and was given a one hour timeout. All the benchmarks are C programs in the ReachSafety-ControlFlow, ReachSafety-Loops, and ReachSafety-ECA sets. Although these sets contain 1,451 total benchmarks, we elided all the benchmarks that use pointers or arrays, as well as those that took more than 30 seconds to pre-process.[2] Some static characteristics of these benchmarks are presented in Figure 7.



(a) State variables (b) Uninterpreted elements (c) Benchmark size

Fig. 7. Traditional box plots showing quartile ranges and outliers for all benchmark. Plot (a) shows that the ControlFlow class contains the instances with the most state variables. The y axis of plot (c) is the number of distinct expressions in $\widehat{T}$, indicating that the ECA instances can be huge. In particular, the ECA benchmarks are on average the largest-size benchmarks; followed by ControlFlow, followed by Loops.

We evaluated EUFORIA against IC3IA [10], an IC3-based checker that implements implicit predicate abstraction. We chose IC3IA largely because it is similar to EUFORIA, with one essential difference: it uses predicate abstraction instead of EUF abstraction. Moreover, as pointed out by Cimatti *et al.* [10], IC3IA is superior in performance to state-of-the-art bit-level IC3 implementations as well as other IC3-Modulo-Theories implementations; and it can support hundreds of predicates (around an order of magnitude more than what explicit predicate abstraction tools can practically compute). In order to ensure an apples-to-apples comparison, we run IC3IA on the exact same model checking problem as EUFORIA, by dumping the model checking instance (transition system and property encoding) into a `vmt`[3] file, which is readable by IC3IA. Currently, EUFORIA only supports LLVM bitcode as input, so our runtime numbers for EUFORIA include the time it takes to re-encode the transition system and property, but IC3IA does not need to do this; thus EUFORIA's numbers are slightly higher than they could be (up to 30 seconds).

Our evaluation sought answers to the following questions:

1. When EUFORIA performs relatively well, why?

[2] Note that this is *pre-processing* time, which is the time to optimize and encode the instances. The instances that take more than 30 seconds to preprocess are multi-megabyte source files that come from the ECA set. They are so big that they time out on both checkers, so we excluded them from our evaluation.

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

2. When EUFORIA performs relatively poorly, why?
3. Does EUFORIA require more clauses than IC3IA to accomplish verification?
4. How does convergence depth compare?

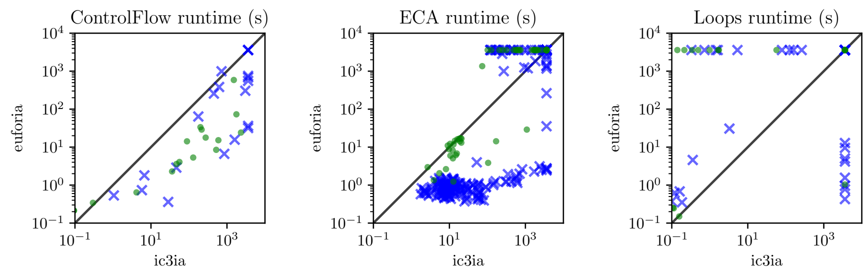


Fig. 8. Scatter plot of runtimes broken down by benchmark set. Timeout was set to one hour. Safe benchmarks show with green dots, unsafe with blue x's.

Figure 8 shows our overall results on all benchmarks compared with IC3IA. EUFORIA and IC3IA are to a certain extent complementary in what they are able to solve within the timeout. IC3IA uniquely solves 62 benchmarks (17 from Loops and 45 from ECA, none from ControlFlow); all of these benchmark properties are about arithmetic and EUFORIA gets stuck inferring weak refinement lemmas. The properties involve things like proving sorting; complex state updates involving division, multiplication, and addition; and invariants involving relationships between addition and signed/unsigned integer comparison. These are benchmarks expected to be tough for EUFORIA, since we have explicitly abstracted these operations in order to target control properties. We believe this weakness can be addressed through a refinement algorithm that infers lemmas related to arithmetic facts, such as commutativity or monotonicity. These benchmarks help address research question 2.

EUFORIA*'s uniquely solved benchmarks* EUFORIA uniquely solves 26 benchmarks; these cut across the benchmark sets: 9 in Loops, 5 ControlFlow, and 12 ECA. EUFORIA is on average spending only 13 seconds in refinement on these benchmarks, compared to 767 for IC3IA:

Refinement times on uniquely solved benchmarks

	EUFORIA	IC3IA (timeout)
average	12.98	766.57
median	0.11	135.95

	EUFORIA (timeout)	IC3IA
average	937.65	154.27
median	975.41	81.59

On the ControlFlow set (which fits our property target best), EUFORIA solves 5 unique benchmarks and IC3IA solved no uniques. The ControlFlow benchmarks have the most state variables, moderate UF/UP use, and are medium-sized. Moreover, EUFORIA requires very little refinement time, supporting our hypothesis that EUFORIA's EUF abstraction provides a decent means for targeting control properties.

Benchmarks both solved Figure 9 shows that, of the 249 benchmarks for which *both* checkers terminated, EUFORIA is able to solve the overwhelming majority faster than IC3IA. Surprisingly, nearly 200 benchmarks among these required no refinements from EUFORIA, as shown in Figure 10. This result is perhaps unexpected because EUFORIA's abstraction removes nearly all behavior from program operators, suggesting that refinement is likely necessary. While much behavior is abstracted, equality, which is critical for verification, is preserved and some benchmarks simply need EUF reasoning (i.e., functional consistency), as we'll see shortly.

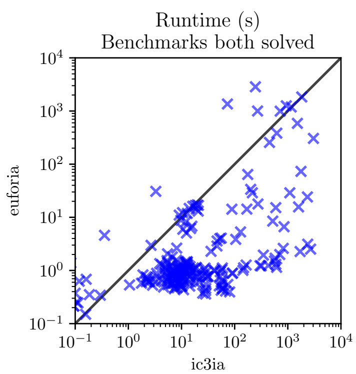


Fig. 9. Runtime of EUFORIA and IC3IA on 249 benchmarks for which both checkers terminated within an hour. EUFORIA solves most instances more quickly than IC3IA.

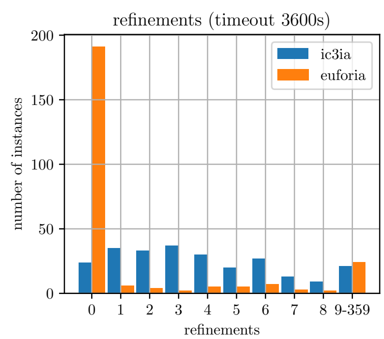


Fig. 10. Number of instances grouped by how many refinements were required to solve them, on benchmarks both checkers finished. The key take away is that EUFORIA is able to solve many instances with very few refinements.

It is interesting that for some relatively simple arithmetic benchmarks, IC3IA diverges and EUFORIA converges. IC3IA begins inferring predicates like $(k = 0), (k = 1), (k = 2), \ldots$ as well as $(1 < j), (2 < j), (3 < j), \ldots$ and will continue this until exhausting all possible values (on 32 bits). A sample program is shown below:

$k = i = 0$
while $i < n$ **do** ▷ $k = i$ is invariant
  $i \leftarrow i + 1;\ k \leftarrow k + 1$
$j \leftarrow n$ ▷ $k = j = n$
while $j > 0$ **do** ▷ $k = j$ is invariant

assert($k > 0$)
$j \leftarrow j - 1; k \leftarrow k - 1$

The second while loop's assertion holds because of the relatively simple property that $(k = j \wedge j > 0) \rightarrow (k > 0)$, which also holds in EUF. IC3IA was unable to discover the relevant predicates, underscoring that choice of predicates is crucial for predicate abstraction. Several other benchmarks follow a similar pattern.

We hypothesize that EUFORIA can take advantage of certain structure from the ControlFlow benchmarks. For example, many of the benchmarks implement a state machine that records its state in an integer state variable. Our abstraction will keep state machine states distinct, since equality is interpreted and integer terms are kept distinct. IC3IA on the other hand must learn predicates such as $(s = 4)$, $(s = 5)$, in order to reason about which state the state machine is in. Indeed, *all* predicates that IC3IA learns on this benchmark set are of the form $(x = y)$ where x is a state variable and y is a constant or a variable; in other words, it learns no predicates besides simple equalities that EUFORIA preserves intrinsically.

There are several other factors contributing to EUFORIA's relatively low runtime on these benchmarks. EUFORIA's SMT queries are roughly an order of magnitude faster than IC3IA's, due to the fact that it is reasoning using EUF and not bit vectors. EUFORIA's effort spent per lemma is consistently lower than IC3IA's effort spent per predicate: the time spent generating each new lemma is up to 10x faster than IC3IA. IC3IA performs bounded model checking on the concrete system to extract an interpolant to generate new predicates, which is more expensive than our approach of examining a single error path and finding an unsatisfiable constraint. For larger transition relations, the difference between query times increases steadily, and the performance advantage of EUFORIA's EUF reasoning becomes more evident. This difference comes out in driver benchmarks which implement several state machines at once. EUFORIA solves these benchmarks one or two orders of magnitude faster than IC3IA and finds smaller invariants. Both checkers refine similarly (i.e., number of refinement lemmas/predicates introduced is comparable) but EUFORIA exploits that information much more effectively, as evidenced by IC3IA requiring roughly an order of magnitude more blocking cubes than EUFORIA.

An interesting outcome of these experiments is that the vast majority of EUFORIA's refinement lemmas are one-step lemmas that merely constrain the behavior of the UFs and UPs in the abstract transition system. In contrast, every new predicate that is introduced by IC3IA doubles the size of the state space (i.e., it goes from size 2^n to 2^{n+1} when increasing the number of predicates from n to $n + 1$).

Figure 11 shows the number of cubes blocked (i.e., clauses added) during solving. Generally, EUFORIA is able to complete with fewer blocked cubes than IC3IA, addressing research question 3.

We hypothesized that EUFORIA, due to its abstraction, may require fewer frames to converge than IC3IA; this is why we asked research question 4. Figure 12 shows the termination depths of EUFORIA and IC3IA. Generally, the termination depths of both checkers are comparable.

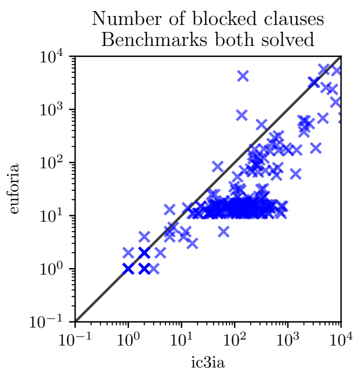


Fig. 11. Number of blocked clauses during solving for all benchmarks solved by EUFORIA and IC3IA. Overall, EUFORIA seems to add fewer cubes.

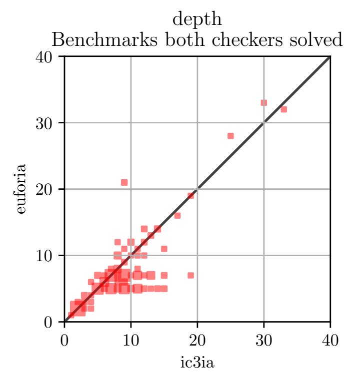


Fig. 12. Frame depth after convergence for both EUFORIA and IC3IA. The area of the squares is proportional to the number of different benchmarks terminating at the given depths.

Overall, EUFORIA performs well on benchmarks testing control properties. In aggregate, EUFORIA solved 275 out of 752 and timed out on 477. IC3IA solved 311 and timed out on 441.

5 Related Work

Since IC3's advent in 2011 [19], applications and extensions of the basic algorithm have flourished. Cimatti and Griggio [22] and Hoder and Bjørner [23] presented the first software model checkers built in IC3 style. More germane for this paper is how abstraction has been applied in IC3-style solvers. SPACER [32] is implemented in IC3 style using a Horn clause solver and linear rational arithmetic. It abstracts programs by dropping elements of the transition relation; it's a kind of generic abstraction support, but expressing EUF abstraction under such a model would require a significant amount of extra constraints (to encode functional consistency). IC3 has been adapted to use predicate abstraction, with a couple of different refinement schemes. CTIGAR's [21] refinement is triggered by individual queries during backward reachability. IC3IA's [10] refinement is triggered whenever an abstract counterexample is found and uses interpolation to derive new predicates. Bjørner and Gurfinkel [33] integrated polyhedral abstract interpretation with IC3 to compute safe convex polyhedral invariants. Our work abstracts using EUF, which is a different mechanism from each of these, and is bit-precise in its concrete representation.

Burch and Dill [18] introduced the use of EUF for pipelined microprocessor verification. For software, Babić and Hu [34, 35] implemented Calysto, a CEGAR abstraction that uses EUF to abstract away internal function bodies. 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, Calysto finds abstract summaries that are responsible for the spurious counterexample, and refines them by removing EUF terms and making 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 checking, but not for IC3/PDR applied to checking safety properties; see [12] for further discussion of EUF abstraction. Similar techniques to ours were developed by Andraus [36] for hardware verification, particularly using uninterpreted functions for abstracting wide datapaths. In the context of hardware model checking, Ho *et al.* [37] abstract difficult operations by turning them into inputs; they then use EUF to perform refinement of these previously-abstracted operations. Our work applies directly to software and abstracts uniformly in order to effectively target control properties.

Predicate abstraction [5] is the dominant technique in control property verification, e.g., as used in the tools SLAM [3], BLAST [28], and IC3IA [10]. 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 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 [38]). IMPACT demonstrated how to implicitly compute the predicate abstraction, to avoid this cost [39]. EUF abstraction is nearly "free" in that it does not require any calls to a theorem prover. Moreover, our approach directly abstracts operations as well as predicates, because we are targeting control properties.

Abstraction in general has been employed extensively to address verification complexity [9, 40–42]. Counterexample-Guided Abstraction Refinement (CEGAR) was introduced by Kurshan [8] and refined and generalized by Clarke *et al.* [9].

6 Conclusions and Future Work

We presented an approach for the automatic verification of safety properties of programs using EUF abstraction. Our approach targets control properties by abstracting operations and predicates but leaving a program's control flow structure intact. EUF abstraction is syntactic; it preserves the structure of the concrete transition system and can be computed in linear time. We have integrated it with modern incremental inductive solving and proved that it terminates by producing a word-level inductive invariant demonstrating safety or a true concrete-level counterexample.

Our evaluation shows that EUFORIA is particularly effective on control-oriented benchmarks. In many cases EUFORIA completes without requiring any refinements even in the presence of arithmetic operations. In cases where refinement is required, most refinement lemmas are simply constraints on the abstract transition system that do not increase the size of the state space. This suggests that EUF abstraction is a natural over-approximation of program behavior when data state is mostly irrelevant to establishing the truth or falsehood of the desired safety property.

Going forward, we plan to demonstrate EUFORIA on larger and more diverse benchmarks. This requires modification to its front-end to add support for program constructs such as pointers and arrays, as well as modification to the backend to support more efficient checking. We also plan to explore how to leverage loop identification inside the EUFORIA algorithm, specifically during refinement to find concrete counterexamples longer than the abstract counterexamples.

Some control properties require reasoning about relatively small amounts of data operations. Often, specific code fragments in a program are critical for verifying the property. It may be beneficial in these situations to modify the refinement procedure so that such fragments are *concretized* to avoid generating a large number of refinement lemmas.

During development, we noticed that the front-end is at times generating code that is sub-optimal for verification. We found a simple example that contains one state variable, and uses only assignments of constants and equality tests against constants. The property requires only equality reasoning and thus should not trigger any refinement. Nevertheless, LLVM's optimizer transforms this into code that uses a subtraction, and verifying the property requires a refinement. Moreover, recent work [43] has elucidated some drawbacks of static single assignment (SSA) form, specifically in its name management and input/output asymmetry. Besides complicating EUFORIA's encoder implementation, our SSA-based encoding introduces more state variables and leads to less understandable verification lemmas. Future work will explore using alternative front-ends tailored for verification.

Acknowledgements We would like to thank Arlen Cox, Shelley Leger, Geoff Reedy, Doug Ghormley, Sean Weaver, Marijn Heule, and the anonymous reviewers for their incisive comments on previous drafts. Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

1. A. Langley, "Apple's SSL/TLS bug," https://www.imperialviolet.org/2014/02/22/applebug.html, 2014, [Online; accessed September 28, 2018].
2. H. Chen and D. A. Wagner, "MOPS: an infrastructure for examining security properties of software," in *Conference on Computer and Communications Security*, V. Atluri, Ed. ACM, 2002, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/586110.586142

3. T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," in *Symposium on Principles of Programming Languages*, J. Launchbury and J. C. Mitchell, Eds. ACM, 2002, pp. 1–3.
4. R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Software Eng.*, vol. 12, no. 1, pp. 157–171, 1986. [Online]. Available: https://doi.org/10.1109/TSE.1986.6312929
5. 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.
6. 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.
7. 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.
8. R. P. Kurshan, *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
9. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 154–169.
10. 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.
11. Y. Kesten and A. Pnueli, "Control and data abstraction: The cornerstones of practical formal verification," *STTT*, vol. 2, no. 4, pp. 328–342, 2000. [Online]. Available: https://doi.org/10.1007/s100090050040
12. 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
13. 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.
14. 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.
15. C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Workshop on Satisfiability Modulo Theories*, A. Gupta and D. Kroening, Eds., 2010.
16. Z. Manna and A. Pnueli, *Temporal verification of reactive systems – safety*. Springer, 1995.
17. 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/
18. J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Springer, 1994, pp. 68–80.
19. 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.

20. 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.
21. 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
22. 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.
23. 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.
24. 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.
25. T. Lange, M. R. Neuhäußer, and T. Noll, "IC3 software model checking on control flow automata," in *Formal Methods in Computer-Aided Design*, R. Kaivola and T. Wahl, Eds. IEEE, 2015, pp. 97–104.
26. 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
27. 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.
28. 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.
29. 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.
30. 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).
31. D. Beyer, "Software verification with validation of results - (report on SV-COMP 2017)," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds., vol. 10206, 2017, pp. 331–349.
32. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, "Automatic abstraction in smt-based unbounded software model checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 846–862. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_59
33. 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
34. 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.
35. ——, "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.
36. Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Reveal: A formal verification tool for verilog designs," in *Logic for Programming, Artificial Intelligence, and Reasoning*, ser. Lecture Notes in Computer Science, I. Cervesato, H. Veith, and A. Voronkov, Eds., vol. 5330. Springer, 2008, pp. 343–352.
37. Y. Ho, A. Mishchenko, and R. K. Brayton, "Property directed reachability with word-level abstraction," in *Formal Methods in Computer Aided Design*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, pp. 132–139. [Online]. Available: https://doi.org/10.23919/FMCAD.2017.8102251
38. T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and cartesian abstraction for model checking C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 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
39. K. L. McMillan, "Lazy abstraction with interpolants," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 123–136. [Online]. Available: https://doi.org/10.1007/11817963_14
40. 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.
41. 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.
42. 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.
43. G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Horn clauses as an intermediate representation for program analysis and transformation," *TPLP*, vol. 15, no. 4-5, pp. 526–542, 2015. [Online]. Available: https://doi.org/10.1017/S1471068415000204
44. R. Bloem and N. Sharygina, Eds., *Formal Methods in Computer-Aided Design*. IEEE, 2010.