Software Model Checking with Uninterpreted Functions

by

Denis Bueno

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Denis Bueno
dlbueno@umich.edu
ORCID iD: 0000-0001-6944-5022

Dedication

For my father and mother,
and the Frangibles

Acknowledgments

I thank the members of my committee: Karem Sakallah, Stephane LaFortune, Westley Weimer, and Arie Gurfinkel.

- Karem was my advisor. Karem always believed in the research, even when I didn't. He helped me sharpen my writing, be curious in evaluations, and to give excited presentations. I learned from his persistence and precision.
- Arie Gurfinkel can take poorly-worded, rambling descriptions of a technical problem and give you two or three memorable remarks (questions/observations/comments) that make your problem plain. Quick conversations with him were worth days or weeks of my thinking time.
- Westley Weimer, besides his prolific research output, is a careful meta-researcher. I learned from his introspection about his approach to research, to novelty, and evaluation.

Pete Manolios gave my first introduction to formal methods, as a professor, and my first introduction to research, as my undergraduate research advisor. Andrew Myers fostered my interest in programming languages. He also taught me the importance of memorable names and mathematical notation. Despite slow progress, both encouraged me in my research.

Except for two years of my PhD, I was full time staff at Sandia Labs. Thanks to LeAnn, In, Nathan, and Brian, I was able to finally complete this journey. Shelley, Geoff, Doug, Todd all supported me consistently. Geoff implemented the first version of `ThreadGlobals`. He was also always up for a technical discussion, whether the area was familiar or not. Shelley has been a boundless source of encouragement, idea-bouncing, and curiousity, and I am deeply thankful for her friendship.

Arlen Cox was instrumental in getting me to graduation. His help has been relentless and invaluable. He has helped me as a colleague, with my own research, and as a shepherd, keeping me on track.

I am grateful to Sean and Marijn for their encouragement, feedback, and friendship.

Bailey provided all kinds of support at a moments' notice. Thanks to the students in his group, who were always welcoming, even though my research was quite different from theirs. I am grateful for my friendships with Kee, David, Mike, and Zakir. Even though I had to teach Kee everything he knows about cooking.

A great deal of thanks is owed to Beth, for getting me out of the rut that was keeping me from graduating.

Believing in me is really hard. Thank you to Lauren, who did so anyway, especially all the times I did not deserve it.

Thanks to my sisters Michelle and Andrea, for putting up with my extended household invasions. My parents always believed in me and supported me, for which I am very grateful. Finally, thank you to my children, who always encouraged me, celebrated with me, and were incredibly patient. I love you.

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.

Table of Contents

List of Figures

List of Appendices

List of Acronyms

ic3	Incremental Construction of Inductive Clauses for Indubitable Correctness.
pdr	Property Directed Reachability.
ACX	Abstract Counterexample.
ATS	Abstract Transition System.
CACX	Concretized Abstract Counterexample.
CTI	Counterexample-to-Induction.
CTS	Concrete Transition System.
EUF	Equality with Uninterpreted Functions.
FOL	First-Order Logic with equality.
SMT	Satisfiability Modulo Theories.
SSA	Static Single Assignment.
UF	Uninterpreted Function.
UP	Uninterpreted Predicate.

Abstract

Software model checkers attempt to algorithmically synthesize an inductive proof that a piece of software is safe. Such proofs are composed of complex logical assertions about program variables and control structures, and are computationally expensive to produce.

Our unifying motivation is to increase the efficiency of verifying software control behavior despite its dependency on data. Control properties include important topics such as mutual exclusion, safe privilege elevation, and proper usage of networking and other APIs. These concerns motivate our techniques and evaluations.

Our approach integrates an efficient abstraction procedure based on the logic of equality with uninterpreted functions (EUF) into the core of a modern model checker. Our checker, called EUFORIA, targets control properties by treating a program's data operations and relations as uninterpreted functions and predicates, respectively. This reduces the cost of building inductive proofs, especially for verifying control relationships in the presence of complex but irrelevant data processing. We show that our method is sound and terminates. We provide a ground-up implementation and evaluate the abstraction on a variety of software verification benchmarks.

We show how to extend this abstraction to memory-manipulating programs. By judicious abstraction of array operations to EUF, we show that we can *directly* reason about array reads and *adaptively* learn lemmas about array writes leading to significant performance improvements over existing approaches. We show that our abstraction of array operations completely eliminates much of the array theory reasoning otherwise required. We report on experiments with and without abstraction and compare our checker to the state of the art.

Programs with procedures pose unique difficulties and opportunities. We show how to retrofit a model checker not supporting procedures so that it supports modular analysis of programs with non-recursive procedures. This technique applies to

EUFORIA as well as other logic-based algorithms. We show that this technique enables logical assertions about procedure bodies to be reused at different call sites. We report on experiments on software benchmarks compared to the alternative of inlining all procedures.

Chapter 1.

The Thesis

EUF abstraction outperforms other model checking techniques on a variety of software analysis problems.

1.1. Introduction

The holy grail of software analysis is to run a "software safety checker" that guarantees that software doesn't do anything unsafe. A checker that works on every software program can't be written — a well-known fact thanks to results from Alan Turing, Alonzo Church, Henry Rice, and others. Nevertheless, history has since repeatedly shown that there are *broadly applicable* safety checking algorithms. In this dissertation, we focus on the technique of automatic software verification by Model Checking.

Model checkers are designed to ensure that all reachable states are safe; this approach often doesn't scale because the number of states is astronomical. Modern model checkers, therefore, usually check an abstraction of a system. Perhaps *the* recurring question in model checking research is: how can we design an abstraction that makes a useful trade-off between fast and precise? Some abstractions are precise but overly expensive to compute. Other abstractions are cheap to compute but lead to too much time spent on false positives. My work provides one answer to this question.

This dissertation describes a model checker, EUFORIA, that checks an equality with uninterpreted functions (EUF) abstraction. While programs expressed in EUF contain infinitely many reachable states, In Chapter 3, I contribute a terminating algorithm called **term projection** that exploits the finiteness of the underlying C programs. I also extend EUFORIA to support programs that manipulate arrays (Chapter 4) and programs with functions (Chapter 5).

EUFORIA checks programs (and other discrete, dynamical systems) using a flexible input format. EUFORIA's targeted EUF abstraction outperforms other model checkers on a variety of software analysis problems. This dissertation describes the checker's implementation and experiments probing its performance.

1.2. Model Checking Methods

Model checker internals can be a bit mysterious, so we're going to review how the research community arrived at the abstraction-based model checkers of today.

Model checking began with a brute force idea. First, define which states are unsafe. Next, explore all the reachable states; if we don't find an unsafe state, conclude that the program can't reach one. Properties proscribing unsafe states are called safety properties.

The code below shows a simple program with an assertion, which is a safety property.

```
$i = 0, j = 0$
**while** $\star \wedge j < 10$
    **if** $i < j$ **then**
        $i = i + 1$
    $j = j + 1$
**assert** $j = i \vee j = i + 1$
```

We use the star ($\star$) syntax to mean an arbitrary value (of appropriate type).

The assertion tests, for every run reaching the assert, "does the state (i, j) satisfy $j = i \vee j = i + 1$?" For brevity, let's call this formula P. The states reachable at the assert are the states that the model checker enumerates:

State		Property		
i	j	$j = i$	$j = i + 1$	$j = i \vee j = i + 1$
0	0	*true*	*false*	*true*
0	1	*false*	*true*	*true*
1	2	*false*	*true*	*true*
2	3	*false*	*true*	*true*
…				
9	10	*false*	*true*	*true*

Since the property is a logical or ($\vee$), I've broken it up into its components. It can be easily seen that every state in fact satisfies P: the first state satisfies $j = i$ and the rest satisfy $j = i + 1$.

All is well until this table becomes too large. The number of states is exponential in the number of if statements in the program; and it only gets worse when there are loops or functions. In practice, this state space exploration method quickly becomes intractable. Even worse, the number of states could be infinite, if i and j are integers.

This is known as the *state explosion problem*. One way this is addressed is by representing states implicitly instead of explicitly. Symbolic model checkers [1] use mathematical formulas to represent sets of states. The following formula represents the set of states from the table above:

$$(i = 0 \wedge j = 0) \vee (0 < j \leq 10 \wedge j - 1 = i)\ . \tag{1.1}$$

The states in the table, and only those states, satisfy this formula. This formula description is shorter than the tabular description, in terms of the number of symbols each requires. If we increase the bound on program variable j from 10 to a billion, for instance, the formula description becomes much shorter than the state enumeration.

Without explicitly enumerating states, the property can no longer be checked against each state. Instead, symbolic algorithms check whether the formula (1.1) forms an inductive proof for P. P is satisfied inductively if

1. it is true initially, and

2. it is preserved by every program transition.

To illustrate, let's treat $(i = 0, j = 0)$ as the initial program state and say a program transition corresponds to executing the while loop and exiting to the **assert**. (We discuss alternate notions of "transition" shortly.) Here is an inductive proof of P:

1. Is P true initially? It is, because $i = j = 0$ implies P.
2. Is P preserved by executing the while loop (and exiting)? Formula (1.1) describes exactly the states that exit the while loop. Let's break the formula into two cases, one for each side of the $\vee$, and show that each case implies P:
   a) Case $(i = 0 \wedge j = 0)$: Then $j = i$, so P holds. (This case corresponds to executing the while loop body 0 times.)
   b) Case $(0 < j \leq 10 \wedge j - 1 = i)$: $j - 1 = i$ can be rewritten $j = i + 1$, implying P holds in this case, too. (This case corresponds to executing the while loop body up to 10 times.)

These steps form an inductive proof, relying on the exact formula (1.1) describing the set of reachable states.

But how do we find such formula descriptions? We don't know how to search across arbitrary symbolic descriptions well enough to scale symbolic algorithms, except in restricted cases. This reality led to the idea of approximating the set of reachable states. Approximations can be designed to facilitate automatic exploration. They also usually make for more concise formulas. A superset approximation in particular has the desirable property that if nothing in a superset violates P, the program is safe. Abstraction-based model checkers implement this kind of abstraction.

To illustrate abstraction using the program above, consider a different property, $P_2(i) = (i \geq 0)$. The program satisfies this property (we can prove this by appealing to the property P which the program also satisfies). But now we will form an inductive proof for P_2 by constructing a superset of the reachable states.

For this proof, we define program transitions differently: a transition corresponds to executing one statement. How a transition is defined isn't baked into the inductive proof method, so the model checker has some flexibility. Defining a transition in this particular way also appears to put us at a disadvantage, since there are more transitions than before. But with the power of abstraction, our proof is even simpler!

There are only two transitions (i.e., statements) in the program that change i: $i = 0$ and $i = i + 1$. Let's define the prime'd version i' of i as "the value of i after any one transition." This is a powerful definition as it allows us to relate the next value of i to the previous value of i; formulas written in terms of variables and prime'd variables are called transition relations. Considering every statement, the program does one of two things to i:

1. It increments i: we write this as $i' = i + 1$.

2. It doesn't change i: $i' = i$.

These cases provide the foundation for an *approximate* representation of the program. The full approximating formula is:

$$i = 0 \text{ initially} \qquad \text{and} \qquad \begin{array}{l}[(i' = i + 1) \vee (i' = i)] \wedge \\ (j' = \star)\end{array} \tag{1.2}$$

All the reachable states are contained in this formula. But this formula also contains these states (i, j) (and many others):

- $(0, 20)$ since j can be assigned *anything*.

- $(10, 111)$ since $(9, 10)$ is a reachable state.

This approximation is terribly useful because it can be used to form a proof for P_2. We want to show that (1) initially i is at least zero and (2) the next value of i is at least zero. We write the latter as $P_2'(i) = (i' \geq 0)$. We want to show, using formula (1.2), that $P_2'(i)$ holds after every program transition.

1. Initially $i = 0$: Surely $0 \geq 0$, so $P_2(0)$ holds initially.

2. Separating the two possible transitions for i into cases:

   a) $i' = i + 1$: By induction hypothesis we assume that $P_2(i)$ holds, so $i \geq 0$. Incrementing a non-negative number yields a non-negative number, so $i' \geq 0$, hence $P_2'(i)$ holds.

   b) $i' = i$: Also by induction hypothesis.

The over-approximation (1.2) is simpler than the exact formula (1.1): (1.2) completely ignores how the program updates j! Abstractions can radically reduce the complexity of implicit state representations and are considered essential today.

In short, the goal of a symbolic model checker is to construct a proof (by induction) that properties embedded in the program, represented by P, are valid. In symbols,

$$I \implies P \tag{1.3}$$

$$P \wedge T \implies P' \tag{1.4}$$

where I stands for the initial states and T represents the program's transition relation. The model checker employs abstraction for two reasons: to organize the search for formulas and to bias itself toward simpler descriptions. The next section discusses our abstraction.

1.3. Equality with Uninterpreted Functions Abstraction

EUF is a quantifier-free logical theory in which formulas are composed of Boolean connectives, uninterpreted function symbols $f(x)$, and uninterpreted predicate symbols $P(x)$. $f(x)$ may evaluate to any value under the constraint that equal arguments produce equal results; this constraint is called *functional consistency*. $P(x)$ may evaluate to true or false under the same constraint. For instance, the formula

$$f(x) \neq f(y) \wedge (x = y)$$

is not satisfiable in EUF because the arguments x and y are equal, but the results are not.

Burch and Dill [2] introduced the use of EUF for pipelined microprocessor verification. Their task was to ensure that a single-cycle instruction execution produced the same result as a multi-cycle pipelined execution of the same instruction. The instruction set architecture mandated the input-output behavior of each instruction — this was the specification. The microprocessor implemented an instruction pipeline, allowing multiple instructions to be in the process of executing at once. They observed that "the differences between the specification and implementation behaviors are entirely in the timing of operations and the transfer of values" (p. 69). EUF now seems an obvious match to this task: uninterpreted functions abstract functional units (such as adders) in both the instruction set specification and in the implementation; uninterpreted predicates abstract relational operators (like less-than). Once abstracted with EUF, the specification is checked for equivalence with the implementation.

Because this dissertation concerns software verification, I'll give an example of *translation validation* [3] where the goal is to verify that the result of evaluating an expression is equivalent to a particular sequential order of evaluation. A compiler has the freedom to pick among all sequential evaluation orders for nested expressions (one order may be better than another because of pipelining, in fact). Here's a nested expression assigned to a variable z_1 followed by a particular evaluation order, assigned to z_2 [4]:

$$z_1 = (x_1 + y_1) \cdot (x_2 + y_2) \tag{1.5}$$

$$u_1 = x_1 + y_1; u_2 = x_2 + y_2; z_2 = u_1 \cdot u_2 \tag{1.6}$$

A compiler will translate equation (1.5) into the three statements (1.6). To prove that $z_1 = z_2$, we check a verification condition (VC). Verification conditions [5] are formulas that encode correctness properties; if the condition is always true (valid), the property holds. We check whether the following VC is valid:

$$\begin{aligned}(u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z_2 = u_1 \cdot u_2)\\ \implies z_2 = (x_1 + y_1) \cdot (x_2 + y_2)\end{aligned} \tag{1.7}$$

The VC (1.7) encodes the property that z_2 is equal to that produced by evaluating the expression (1.5), for any x_1, y_2, x_2, y_2.

To prove the VC above one could use a theorem prover that supports reasoning about arithmetic. But the VC includes non-linear multiplication, which is an undecidable theory in general, so the proof might not be straightforward.[1]

Alternatively, this translation can by validated using EUF abstraction. EUF verification proceeds by abstracting the addition operations with an uninterpreted function

[1]I'm sure that every theorem prover can prove this faster than you could write it down. Nevertheless, as the expressions get longer and more more complex, the complexity of a proof is likely to skyrocket.

A, abstracting the multiplication with M, which results in an abstract VC:

$$\begin{aligned}(u_1 = A(x_1, y_1) \wedge u_2 = A(x_2, y_2) \wedge z_2 = M(u_1, u_2)) \\ \implies z_2 = M(A(x_1, y_1), A(x_2, y_2))\end{aligned} \tag{1.8}$$

If the VC is valid in EUF (it is), then the original translation is valid. It is crucial that the uninterpreted function symbols used in the antecedent and consequent are the same (A and M). When one uses EUF for a translation validation, there is an expectation that translation *preserves the sequence of data flow* through the functions. EUF verification checks that the data flow is preserved, since its core axiom is functional consistency.

Previous work has applied EUF abstraction to hardware equivalence checking (e.g., [2], [4], [6], [7]). The two circuits under test use the *same* functions (e.g., for addition and multiplication) and this is what makes EUF verification effective, because it allows applications of functional consistency.

1.4. Beyond Equivalence Checking with EUF

EUFORIA applies EUF abstraction beyond equivalence and to safety properties in general. We have developed an EUF abstraction that is effective for reasoning about lock discipline, state machines, and array programs. More generally, our abstraction is effective for control properties.

Control properties are an integral part of software verification. The 2014 Apple Secure Transport "goto fail" bug [8] provides a compelling illustration:

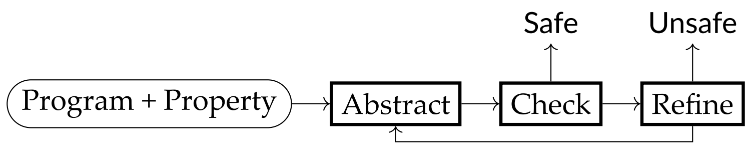
```
extern int f();
int g() {
    int ret = 0;
    if ((ret = /* ... */) != 0)
      goto out;
      goto out; /* this line was inadvertently added */
    ret = f();
out:
    return ret;
}
```

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 should ensure 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 [9]. 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 [10]. Data-independent programs, in which the only allowed operation on data is equality testing, are programs for which entire classes of properties are control properties [11]–[13]. Typestate properties [14] are also control properties.

1.5. Model Checking Architectures

In modern checkers, abstractions aren't fixed. Instead, abstractions are refined until they are strong enough to prove a property. Many model checkers follow the pattern below, at a high level:



Such checkers roughly use the following steps:

1. Compute an initial abstraction, usually guided by a pile of heuristics.

2. Check if the program abstraction is strong enough to prove the property.

3. If so, success! The program is safe. If not,

4. If the abstraction was insufficient because the program is unsafe, success! A bug is identified. Otherwise, use the spurious bug trace to increase the fidelity of the abstraction and go to step 2.

The spurious bug trace is called a counterexample. This general pattern is known as *counterexample-guided abstraction refinement*, or CEGAR [15]. Each part of the algorithm has myriad possible embodiments but the core of "guess, check, improve guess" remains the same.

1.6. Incremental Induction

The last algorithmic method underlying EUFORIA is incremental induction. Why not simply induction?

Symbolic model checkers search for an exact or over-approximate formula R characterizing the reachable states to complete an inductive proof. Specifically, R is called an inductive strengthening when P is not inductive but $R \wedge P$ is:

$$I \implies R \wedge P \tag{1.9}$$

$$R \wedge P \wedge T \implies R' \wedge P' \tag{1.10}$$

In general, one unrolls the program — that is, one looks at many transitions in a row — to figure out how to come up with this formula. Unrolling a program and determining what needs to be added to the proof quickly becomes a complex endeavor.

Enter incremental induction [16], whose key insight is to find facts that are inductive *relative to* other facts. A fact Q is inductive if it is true initially, and it is *preserved* by the program: if Q is true before any program transition, it is true afterward. For instance, $Q_1 = (i \geq 0)$ is inductive for the transition relation $i' = i + 1$. Let's call this transition relation T.

A fact Q is inductive relative to R if Q is true initially, and Q is preserved by the program *under the assumption* of R: if Q and R hold before any program transition, then Q holds afterward. For instance $Q_2 = (i \neq 0)$ is inductive relative to $R_2 = (5 \leq$

$i < 8$) for T, even though neither is inductive on its own. To see why, consider Q_2 and R_2 in isolation:

- Q_2 by itself is not inductive for a. Suppose $i = -1$ which is consistent with Q_2. After T, $i' = 0$, which is inconsistent with Q_2, so the transition doesn't preserve Q_2. Q_2 is not inductive for a.
- R_2 isn't inductive for T either, because $i = 7$ implies $i' = 8$, which no longer satisfies R_2.

But if $Q_2 \wedge R_2$ holds before T, then i is either 5, 6, or 7 and so i' is 6, 7, or 8, respectively—Q_2' holds. Q_2 is inductive relative to R_2.

Relative induction enables building an inductive proof bit by bit, avoiding any program unrolling. A model checker can construct an inductive strengthening R using many relative inductive steps.

This process is embodied in the IC3 algorithm. A complete and precise description is given by Bradley [16] and our extension to EUF is described in Chapter 3. In this section I show how IC3 avoids unrolling the transitions and how that leads to a potential problem for EUF abstraction.

Let T be a program description as a transition relation. Let's presume the existence of a sequence of formulas, F_i ($i \in \{0, \ldots, k\}$ for some integer k), each denoting a set of states, satisfying the following constraints:

$$F_0 = I \tag{1.11}$$

$$F_{i-1} \implies F_i \qquad i \in \{1, \ldots, k\} \tag{1.12}$$

$$F_{i-1} \wedge T \implies F_i' \qquad i \in \{1, \ldots, k\} \tag{1.13}$$

$$F_i \implies P \qquad i \in \{0, \ldots, k\} \tag{1.14}$$

F_i denotes a set of states not known to be unreachable in i transitions. Any state $t \notin F_i$ is known to be unreachable in i transitions. Only states in I are known to be reachable initially. (1.12) means what is known about states after i transitions is no more precise than what is known about states after $i - 1$ transitions. (1.13) means every transition from a state in F_{i-1} goes to a state in F_i (but F_i may contain unreachable states). (1.14) means all states not known to be unreachable in i transitions satisfy the property.

The core of IC3 asks whether there is a counterexample-to-induction (CTI) relative to some F_i. Each CTI check for state s has the form $F_i \wedge \neg s \wedge T \implies \neg s'$, which tests whether $\neg s$ is inductive relative to F_i. When $\neg s$ is not inductive relative to F_i (i.e., the CTI check is not logically valid), then that means there is some *pre-image* state t that transitions to s and t is not (yet) known to be unreachable in i steps. t is called a CTI state.

Assume there is a state s that does not satisfy P and which is reachable from F_k, i.e., $(s \implies P)$ is not valid and $(F_k \wedge T \implies \neg s)$ is not valid. If s is reachable from the initial states it disproves the property. When a CTI check for s on F_k yields a CTI t, we can continue working backward with a subsequent CTI check for t: $F_{k-1} \wedge \neg t \wedge T \implies \neg t'$. If the latter CTI check yields another CTI, we can continue this process, perhaps all the way to F_0. If the last CTI check, relative to F_0, has a CTI, then we have discovered a sequence of transitions that (1) begins in I and (2) takes k steps until in reaches s. In this way, IC3 can find a counterexample to a failing property as a sequence of CTI checks, rather than by unrolling T.

CTI checks are the heart of why IC3 does not unroll the transition relation. How do we represent CTIs in EUF?

1.7. Finite Pre-images in Infinite Spaces

EUF abstraction has a pre-image problem: there are infinitely many descriptions of them. Consider the following program. The program is given on the left and its description in logic and EUF are given on the right. Assume that the variables t and i are word-sized unsigned integers; for example, 32-bit integers.

Program:	Logic:	EUF:
while *true*		
$\quad t \leftarrow i$	$t' = i$	$\mathsf{t}' = \mathsf{i}$
$\quad i \leftarrow i + 1$	$i' = i + 1$	$\mathsf{i}' = \mathsf{succ}(\mathsf{i})$
$\quad$ **assert** $i > t$	$P' = (i' > t')$	$\mathsf{P}' = \mathsf{GT}(\mathsf{i}', \mathsf{t}')$

CTI states are pre-images and IC3 may need to explore every CTI state to prove the property. Since t and i are word sized integers, the state space is finite. States in this program are represented by a pair (t, i) whose elements represent the assignment to the program variables t and i, respectively. Working one step back, the pre-image $(t^{(1)}, i^{(1)})$ for a state (t, i) is described by $i^{(1)} = i + 1, t^{(1)} = i$. Working two steps back, the second pre-image is $i^{(2)} = i + 1 + 1, t^{(2)} = i + 1$. Working backward n states, we can see that if the current state is $(t^{(n)}, i^{(n)})$ then the nth pre-image (t, i) satisfies these equations:

$$i^{(n)} = \underbrace{1 + 1 + \cdots + i}_{n \text{ additions}} \qquad t^{(n)} = \underbrace{1 + \cdots + i}_{n-1 \text{ additions}}$$

Since unsigned addition wraps, all possible assignments to t and i will eventually be enumerated, ensuring termination of the algorithm.

Instead, we want to use EUF abstraction. The nth pre-image is analogous:

$$\mathsf{i}^{(n)} = \underbrace{\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\ldots\mathsf{succ}(\mathsf{i}))))}_{n \text{ additions}} \qquad \mathsf{t}^{(n)} = \underbrace{\mathsf{succ}(\mathsf{succ}(\ldots\mathsf{succ}(\mathsf{i})))}_{n-1 \text{ additions}}$$

The drawback of EUF is that we can do this forever and never accomplish our original goal, which was to prove (or disprove) the program's assertion. For every integer n, the state $(\mathsf{t}^{(n)}, \mathsf{i}^{(n)})$ is unique, so the algorithm above risks running indefinitely.

Instead, we introduce a technique to represent pre-images using *only terms from the EUF formula that describes the program*. In our example, only one occurrence of succ is used. This technique, introduced in Chapter 3, is called the **term projection**. Because the formula is finite, there are a finite number of assignments to explore before all pre-images are exhausted.

This solution has introduced another problem: what if we need more than what occurs in the formula? Answering this is the subject of the next section.

1.8. Abstraction Refinement in EUF

Refinement means changing the abstraction so that it more adequately reflects the behavior of the concrete program. Each refinement step produces a new formula which rules out some spurious behavior of the abstraction.

Recall the addition operation from the previous section. Below we show some possible spurious behaviors of an EUF abstraction along with possible abstraction refinements on the right:

Spurious Behavior	Refinement
$\mathsf{succ}(i) = i$	$\mathsf{succ}(i) \neq i$
$i' = 3, i' = \mathsf{succ}(0)$	$\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(0))) = 3$
$i = \texttt{MAX\_INT}, i' \neq 0$	$\mathsf{succ}(\texttt{MAX\_INT}) = 0$

In general, there may be many different formulas that adequately refine the EUF abstraction. EUFORIA produces them by abstracting inconsistent concrete formulas.

1.9. Memory Manipulating Programs

Every program considered so far only uses scalar variables. But real C programs use memory, and they use it all the time. This dissertation contributes, and I describe in this section, a simple **array abstraction** using EUF that avoids large amounts of redundant computation during model checking.

Encoding C programs with memory is a chore, as it typically requires a region-based pointer analysis and is very sensitive to how precise those regions are. Regions denote non-overlapping portions of the program's memory. Each pointer variable is associated with exactly one region; in this way, reads and writes to memory become accesses into a particular region. Associated with a region are all accesses which may overlap with other accesses to the same region.

Once segregated into regions, programs are encoded for EUFORIA by representing pointer accesses in terms of array accesses. Arrays, first axiomatized by McCarthy [17], represent a contiguous set of equal size memory locations. The locations store words and are word-addressed. Arrays support two access operations:

- select(A, i) returns the value stored at array location i in array A.
- store(A, i, x) returns a fresh array in which location i contains x and every other location contains whatever is stored in A.

Regions are represented by arrays; pointers are represented by indices into arrays.

Chapter 4 contributes an EUF abstraction that naturally accommodates this array representation. Select terms are abstracted to uninterpreted select terms, the same way addition is abstracted. Store terms are treated similarly.

This abstraction also requires refinement. In the array theory, selecting from an index that has been stored to retrieves the stored value; but EUF abstraction allows select to return some other value. I contribute a refinement method that effectively refines the array abstraction.

1.10. Functions

So far we have discussed programs with a single function (main). To support multiple functions without modifying the checker, one must somehow get rid of functions. There are a couple of common ways to do this:

1. By inlining. When a function call is inlined, the body of the function is copied to the call site. The function's local variables are renamed to avoid conflicts with the local variables at the call site. If all call sites of the function are inlined, the function itself can be removed from the analysis.

   Inlining may lead to exponential blowup in the size of the program. Recursive functions, moreover, can't be inlined in general.

2. By up-front abstraction. A function call $x = f(y)$ can be replaced by an assignment $x = \star$. If the function is irrelevant to the property, then the analysis will return a correct answer. But if it is relevant, then the analysis may result in a false positive: it may claim the program has a bug due to a value stored in x, when in fact $f(x)$ never returns any such value.

EUFORIA addresses functions in a different way, via the program encoding. Chapter 5 contributes a **function encoding** that handles function call & return semantics on a finite stack. As a result, EUFORIA can analyze programs with non-recursive functions without modifications, inlining, and abstraction.

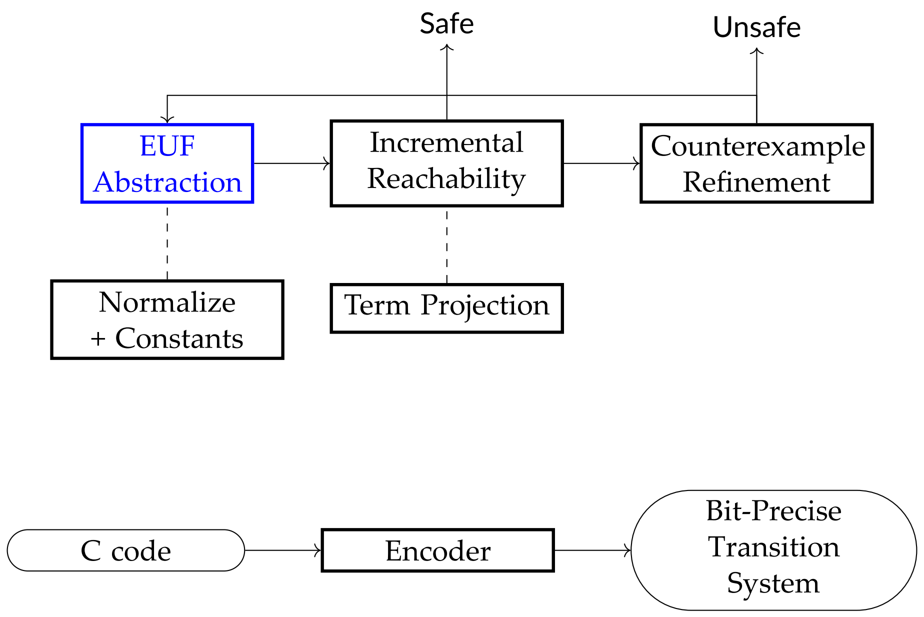


Figure 1.1: EUFORIA architecture.

1.11. EUFORIA Architecture

EUFORIA was implemented from the ground up. At first, its design was soup to nuts. The front-end took a C programand at the end produced an answer: yes, the property holds, or no, it doesn't. Inside, it computed its own precise encoding of the program as well as its abstraction.

The current architecture is shown in Figure 1.1; its entry point is the blue node. The front-end encoding is now separate from model checking. Instead of a front-end for C programs, EUFORIA reads an SMT-LIB file describing the program as a transition system directly using the first-order theories of bit-vectors and arrays. This dissertation discusses encoding methods that can be produce such files, but they can also be hand-written.

EUF Abstraction creates the initial abstraction of the transition system, which includes normalizing mathematical operators and ensuring constants are distinct. Incremental Reachability attempts to find an abstract counterexample to the property or an inductive invariant, using **term projection**. If an invariant is found, the program is Safe. Otherwise, the abstract counterexample is handed off to Counterexample

Refinement, which either deduces that the counterexample is real — the program is Unsafe — or that it should be used to refine the EUF abstraction.

A complete program and its SMT-LIB encoding is shown in Figure 1.2. The program is defined by three Boolean formulas: `:init`, `:trans`, and `:invar-property`. These define the initial state, transition relation, and property, respectively. A transition relation is a formula, similar to those discussed earlier in this chapter, which relates next-state variables i' to current-state variables i. In SMT-LIB syntax, we use a plus (+) instead of prime.

Variables are declared with `declare-fun` in SMT-LIB. The integers i and j are treated as 32-bit words, denoted by the type `(_ BitVec 32)`. The `:next` annotations indicate how logic variables define state variables. For instance, `.def0` says "`i` is a state variable and its next state variable is `i+`."

Programs steps have labels, like labels in C. The top of the while loop is labeled *loop*. Initially, the program is at *loop* and remains there until the loop guard is not satisfied, at which point it moves to *done*.

The assert is encoded as the property. The property holds if, whenever the program is at *done* the asserted formula is true.

The following chapter formalizes the logical notation underlying all the math I use. This sets the stage for presenting my contributions in subsequent chapters.

Program:

```
        $i = 0, j = 0$
$loop$:
        while $\star \wedge j < 10$
            if $i < j$ then
                $i = i + 1$
            $j = j + 1$
$done$: assert $j = i \vee j = i + 1$
```

VMT encoding:

```
; state variable decarations
(declare-fun i () (_ BitVec 32)) (declare-fun i+ () (_ BitVec 32))
(declare-fun j () (_ BitVec 32)) (declare-fun j+ () (_ BitVec 32))
(declare-fun loop () Bool)       (declare-fun loop+ () Bool)
(declare-fun done () Bool)       (declare-fun done+ () Bool)
; input declarations
(declare-fun star () Bool)
; :next defs for state vars
(define-fun .def0 () (_ BitVec 32) (! i :next i+))
(define-fun .def1 () (_ BitVec 32) (! j :next j+))
(define-fun .def2 () Bool (! loop :next loop+))
(define-fun .def3 () Bool (! done :next done+))
; initial state
(define-fun .init () Bool
  (! (and loop (not done) (= i #x00000000) (= j #x00000000))
     :init true))
; transition relation
(define-fun .trans () Bool
  (! (or
    (and loop (and star (bvslt j (_ bv10 32)))
              (not (bvslt i j))
         loop+ (not done+)
         (= i+ i) (= j+ (bvadd j (_ bv1 32))))
    (and loop (and star (bvslt j (_ bv10 32)))
              (bvslt i j)
         loop+ (not done+)
         (= i+ (bvadd i (_ bv1 32))) (= j+ (bvadd j (_ bv1 32))))
    (and loop (not (and star (bvslt j (_ bv10 32))))
         loop+ (not done+)
         (= i+ i) (= j+ j))
    (and done (or (= j i) (= j (bvadd i (_ bv1 32)))))
    :trans true)))
(define-fun .prop () Bool
  (! (=> done (or (= j i) (= j (bvadd i (_ bv1 32)))))
     :invar-property 0))
```

Figure 1.2: Program and its concrete encoding as a VMT file. The encoding precisely captures the semantics of each operation, including modular arithmetic. The while loop has the label `loop` and the assert has the label `done`.

Chapter 2.

Basic Techniques: EUF Abstraction & IC3-style Model Checking

This dissertation introduces a complete software model checking algorithm designed using EUF abstraction. Its ideas are indebted to a large amount of prior work in logic-based software verification, algorithms, encodings, model checking, and abstraction. This chapter covers all the necessary ground to contextualize the contributions.

Several ideas were introduced in the 1960's which are fundamental for understanding our techniques. *Verification conditions* were intended to capture exactly the propositions that hold at every program location. Floyd introduced VC generation [18] which was further developed by Manna [19]; see Godefroid and Lahiri [5] for an overview. Hoare introduced *Hoare triples* [20] as a general method for proving software correct by reasoning from precondition to postconditions. Applying these methods to real programs requires a significant amount of creativity, however.

Transition systems were introduced by Keller as an abstract, conceptual model for (parallel) programs [21]. Breaking with tradition, it did not distinguish between control and data, which was "customary" at the time. He also defined an induction principle for proving invariants on transition systems.

Gradually, methods developed to automate the software checking task. Dijkstra contributed Weakest Preconditions [22] as a way of automating some the computation steps of a proof, specifically reasoning by giving rules to construct preconditions for given postconditions. Abstract interpretation [23] provides a formalism for defining and analyzing mechanical abstractions and abstraction-based checkers. Model checking is one of the strong developments along this line.

Model checking, the term and technique, was introduced with an explicit-state algorithm by Clarke and Emerson [24], [25]. Concurrently, Queille and Sifakis [26] introduced a similar technique for Petri nets. Both techniques checked branching time temporal logic properties. Since then there has been significant research into methods for checking safety properties, which simply ask: "can a program reach a given bad state?"

The historical development of these techniques spent a significant amount of effort characterizing the systems as well as the techniques used in those systems. This dissertation will heavily use first-order logic with equality as an encoding language, including the theories of bit-vectors and arrays (Section 2.1). We also use equality with uninterpreted functions (EUF) as our primary abstraction (Section 2.2). The bit-vector and array theories can be abstracted using EUF (Section 2.3). We primarily use transition systems as a logical description of program behavior (Section 2.4). Using these we can formalize model checking (Section 2.5). Next we explain the IC3 algorithm, on which our contributions are based (Section 2.6). We then discuss the CEGAR paradigm for automatically increasing the fidelity of an abstraction (Section 2.7). Finally, we also cover a second program representation, Horn encodings (Section 2.8), since we use them as a source language in Chapter 4.

2.1. First-order Logic with Equality

First-Order Logic with equality (FOL) is a language for describing logical assertions and proofs. Our presentation is modeled after that in the Handbook of Model Checking [27] (chapter 10), a recent book detailing many model checking topics. This presentation anticipates the discussion of interpreted theories in Satisfiability Modulo Theories, below.

Fix a universe of sorts **S** and of variables **X**, each variable of which is associated with a sort from **S**.

Definition 2.1. A *signature* Σ consists of three sets $(\Sigma^{\mathcal{S}}, \Sigma^{\mathcal{P}}, \Sigma^{\mathcal{F}})$: a set of *sort symbols* $\Sigma^{\mathcal{S}} \subseteq S$, a set of *predicate symbols* $\Sigma^{\mathcal{P}}$, and a set of *function symbols* $\Sigma^{\mathcal{F}}$; a total mapping from $\Sigma^{\mathcal{P}}$ to the set of strings over $\Sigma^{\mathcal{S}}$; and a total mapping from $\Sigma^{\mathcal{F}}$ to the set of non-empty strings over $\Sigma^{\mathcal{S}}$. Each predicate symbol $p \in \Sigma^{\mathcal{P}}$ has unique arity n if it is mapped to $\gamma_1\gamma_2 \dots \gamma_n$. Each function symbol $f \in \Sigma^{\mathcal{F}}$ has unique arity n if it is mapped to $\gamma_1\gamma_2 \dots \gamma_n\gamma$.

A signature defines a symbol universe and specifies the types of those symbols.

Definition 2.2 (First-order Logic Syntax)**.** *Terms* and *formulas* in first-order logic are constructed using the following grammar:

type		production	explanation
term (t)	::=	$x \mid y \mid z \mid \cdots$	0-arity (constant) term
	$\mid$	$x_1 \mid x_2 \mid x_3 \mid \cdots$	variable
	$\mid$	$F(t_1, t_2, \dots, t_n)$	uninterp. function (UF)
	$\mid$	ite(f, t_1, t_2)	if-then-else
atom (a)	::=	*true* $\mid$ *false*	Boolean constants
	$\mid$	$t_1 \simeq t_2$	equality atom
	$\mid$	$P(t_1, t_2, \dots, t_n)$	uninterp. predicate (UP)
formula (f)	::=	a	
	$\mid$	$\neg f$	negation
	$\mid$	$f_1 \wedge f_2$	conjunction
	$\mid$	$f_1 \vee f_2$	disjunction
	$\mid$	$\forall x_i .\ f$	for all
	$\mid$	$\exists x_i .\ f$	there exists

The syntax of first-order logic is composed of two kinds of objects: terms and formulas. Terms represent *things* and formulas represent *statements*. Terms are composed of variables, Uninterpreted Function (UF) symbols, and if-then-else (ite). Formulas are made up of equality tests between terms and Uninterpreted Predicate (UP) symbols, combined with arbitrary Boolean operators. $a \implies b$ is syntactic sugar for $\neg a \vee b$. $a \iff b$ is syntactic sugar for $a \implies b \wedge b \implies a$. Formulas can be quantified using the usual quantification operations, although unless we specify otherwise, the use of the word "formula" implies quantifier-free. We usually assume predicate symbols have arity at least 1 and speak of *Boolean (variables)* instead of 0-arity predicates. We also usually assume function symbols have arity at least 1 and speak of *constant* symbols instead of 0-arity functions. It is sometimes convenient, on the other hand, to handle all function and predicate symbols uniformly; the text will make it clear when this is the case.

Sorts restrict the way in which terms and equality atoms may be combined. Functions may only be applied to arguments of the correct sorts; similarly for predicates.

The following definition makes this notion precise.

Definition 2.3 (Well-sorted)**.** Formulas and terms that obey the following conditions are well-sorted.

- Every constant is *well-sorted*.
- For every function symbol F of sort $\gamma_1\gamma_2 \dots \gamma_n\gamma$, $F(t_1, t_2, \dots, t_n)$ is *well-sorted* iff t_i is of sort γ_i and t_i is well-sorted $(i \in \{1, \dots, n\})$.
- For every formula f and well-sorted t_1, t_2, $\text{ite}(f, t_1, t_2)$ is *well-sorted* iff t_1 and t_2 have equal sorts.
- For every well-sorted t_1, t_2, $t_1 \simeq t_2$ is *well-sorted* iff t_1 and t_2 have equal sorts.
- For every predicate symbol P of sort $\gamma_1\gamma_2 \dots \gamma_n$, $P(t_1, t_2, \dots, t_n)$ is *well-sorted* iff t_i is of sort γ_i and t_i is well-sorted $(i \in \{1, \dots, n\})$.
- If f_1 and f_2 are well-sorted formulas, then the following are *well-sorted*:
  - $\neg f_1$
  - $f_1 \wedge f_2$
  - $f_1 \vee f_2$
  - $\forall x_i.\ f_1$
  - $\exists x_i.\ f_1$

Definition 2.4 (Free Variables)**.** For every formula f, $\text{Vars}(f)$ denotes the set of variables free in f.

Definition 2.5 (Formula with Free Variable Annotation)**.** A formula f written

$$f(X_1, X_2, \dots, X_n)$$

means the free variables in f are drawn solely from the set $X_1 \cup X_2 \cup \dots \cup X_n$. Equivalently,

$$\text{Vars}(f(X_1, X_2, \dots, X_n)) \subseteq X_1 \cup X_2 \cup \dots \cup X_n\ .$$

We frequently refer to formulas which are composed only of top-level conjunctions (or disjunctions) of indivisible Boolean statements. The following definitions make this precise.

Definition 2.6. *Atomic formulas* or *atoms* are made up of Boolean identifiers, UP symbols, and equalities between terms.

Definition 2.7 (Atoms and Terms)**.** For an arbitrary formula f:

- Atoms(f) is the set of atoms occurring in f containing no occurrences of ite.
- Terms(f) is the set of terms occurring in f containing no occurrences of ite.

Definition 2.8. A *literal* is a (possibly-negated) atom containing no occurrences of ite.

Definition 2.9. A *clause* is a disjunction of literals.

Definition 2.10. A *cube* is a conjunction of literals.

When convenient, a formula f may be treated as a set of its top-level conjuncts, e.g., $x \simeq 1 \in f$ if $f = (x > 17 \wedge x \simeq 1)$. We typically use this notation for cubes, not arbitrary formulas.

Definition 2.11 (List Notation)**.** The notation $\overline{x}$ denotes a possibly-empty list:

$$\overline{x} = (x_1, x_2, \dots, x_k)\ .$$

The length of the list is $|\overline{x}| = k$. If f is any function, then

$$\overline{f(x)} = (f(x_1), f(x_2), \dots, f(x_k))\ .$$

Definition 2.12 (Substitution)**.** For a formula f, let $f[x \mapsto y]$ denote f with all occurrences of x substituted with y. Let $f[\overline{x} \mapsto \overline{y}]$ denote the simultaneous substitution of y_i for x_i, $i \in \{1, \dots, |\overline{x}|\}$, where $|\overline{x}| = |\overline{y}|$.

One has to take care when substituting into a formula that already binds the variable you're substituting into the formula; this problem is called variable capture. One should rename bound variables to avoid variable capture. We won't run into this problem because we substitute exclusively on quantifier-free formulas.

Simultaneous substitution is *not* simply iterated single substitution. Consider the substitution

$$(x' \simeq y \wedge y' \simeq x)[y, x \mapsto x, y] \ .$$

Iterated substitution produces

$$\begin{aligned} & ((x' \simeq y \wedge y' \simeq x)[y \mapsto x])[x \mapsto y] \\ = \quad & (x' \simeq x \wedge y' \simeq x)[x \mapsto y] \\ = \quad & (x' \simeq y \wedge y' \simeq y) \end{aligned}$$

but simultaneous substitution produces $(x' \simeq x \wedge y' \simeq y)$, exchanging x and y.

It is useful to use functions as maps and be able to update them incrementally. For example, the function can represent an environment mapping variables to their current values. Environments are updated after an assignment statement.

Definition 2.13 (Function Update)**.** Given a function $m : A \to B$, we define single and parallel updates:

- $m_1 = \delta[a \hookrightarrow b]$ is a function for which $m_1(a) = b$ and $m_1(x) = m(x)$ for every $x \neq a$.
- $\delta[\overline{a} \hookrightarrow \overline{b}]$ performs n parallel updates where

$$\overline{a} = (a_1, a_2, \dots, a_n) \quad \text{and} \quad \overline{b} = (b_1, b_2, \dots, b_n) \ .$$

.

We have so far discussed the syntax of FOL. We now move onto its semantics, or meaning.

Definition 2.14 (Σ-Interpretation)**.** For a signature Σ and set of variables $X \subseteq \mathbf{X}$, a *Σ-interpretation $\mathcal{I}$ over X* maps

- every sort $\gamma \in \Sigma^{\mathcal{S}}$ to a non-empty set I_γ, its *domain*;
- every variable $x \in X$ of sort γ to an element $x^{\mathcal{I}} \in I_\gamma$;
- every function symbol $F \in \Sigma^{\mathcal{F}}$ of sort $\gamma_0 \cdots \gamma_n \gamma$ to a total function $F^{\mathcal{I}} : I_{\gamma_0} \times \cdots \times I_{\gamma_n} \to I_\gamma$; and

- every predicate symbol $P \in \Sigma^{\mathcal{P}}$ of sort $\gamma_1 \cdots \gamma_n$ to a relation $P^{\mathcal{I}} \subseteq I_{\gamma_1} \times \cdots \times I_{\gamma_n}$.

Interpretations satisfy a formula or they don't.

Definition 2.15 (Satisfaction of an Interpretation)**.** For every interpretation $\mathcal{I}$, $\mathcal{I} \vDash f$ means that the interpretation $\mathcal{I}$ *satisfies* or *models* the formula f. Satisfaction is defined inductively as follows:

$$\begin{aligned}
&\vDash \mathit{true} && && (2.1)\\
&\mathcal{I} \nvDash \mathit{false} && && (2.2)\\
&\mathcal{I} \vDash t_1 \simeq t_2 && \text{if } \mathcal{I}(t_1) = \mathcal{I}(t_2) && (2.3)\\
&\mathcal{I} \vDash P(t_1, t_2, \dots, t_n) && \text{if } (t_1, t_2, \dots, t_n) \in P^{\mathcal{I}} && (2.4)\\
&\mathcal{I} \vDash \neg f && \text{if } \mathcal{I} \nvDash f && (2.5)\\
&\mathcal{I} \vDash f_1 \wedge f_2 && \text{if } \mathcal{I} \vDash f_1 \text{ and } \mathcal{I} \vDash f_2 && (2.6)\\
&\mathcal{I} \vDash f_1 \vee f_2 && \text{if } \mathcal{I} \vDash f_1 \text{ or } \mathcal{I} \vDash f_2 && (2.7)\\
&\mathcal{I} \vDash \forall x_i.\, f && \text{if for every } x^{\mathcal{I}} \in \text{dom}(x_i),\ \mathcal{I} \vDash f[x_i \mapsto x^{\mathcal{I}}] && (2.8)\\
&\mathcal{I} \vDash \exists x_i.\, f && \text{if for some } x^{\mathcal{I}} \in \text{dom}(x_i),\ \mathcal{I} \vDash f[x_i \mapsto x^{\mathcal{I}}] && (2.9)
\end{aligned}$$

where

$$\mathcal{I}(x) = x^{\mathcal{I}} \tag{2.10}$$

$$\mathcal{I}(F(t_1, \dots, t_n)) = F^{\mathcal{I}}(\mathcal{I}(t_1), \dots, \mathcal{I}(t_n)) \tag{2.11}$$

$$\mathcal{I}(\mathit{ite}(f, t_1, t_2)) = \begin{cases} \mathcal{I}(t_1) & \text{if } \mathcal{I} \vDash f \\ \mathcal{I}(t_2) & \text{if } \mathcal{I} \vDash \neg f \end{cases} \tag{2.12}$$

The algorithms in this dissertation make frequent use of satisfiability checks.

Definition 2.16 (Satisfiability Check)**.** For every interpretation $\mathcal{I}$ and formula f, $\mathcal{I} = \text{SAT}(f)$ implies $\mathcal{I} \vDash f$.

Definition 2.17 (Parameterized ModelAssertion)**.** Let T be a set of terms and Q a set of atoms over some signature Σ. For every model M,

$$\text{ModelAssertion}(M, T, Q)$$

is a conjunction comprised of:

$$\begin{aligned} a \quad & \text{if } M(a) = \textit{true}, a \in Q && (2.13)\\ \neg a \quad & \text{if } M(a) = \textit{false}, a \in Q && (2.14)\\ t_1 \simeq t_2 \quad & \text{if } M(t_1) = M(t_2), t_1, t_2 \in T && (2.15)\\ t_1 \not\simeq t_2 \quad & \text{if } M(t_1) \neq M(t_2), t_1, t_2 \in T && (2.16) \end{aligned}$$

Theorem 2.1 (ModelAssertion Satisfies Model)**.** *For all models M, terms T, and atoms S over signature Σ, then*

$$M \vDash \text{ModelAssertion}(M, T, S)\ .$$

Follows directly from the definition of the ModelAssertion.

Definition 2.18 (ModelAssertion)**.** Let $A_f = \text{Atoms}(f)$ and $T_f = \text{Terms}(f)$ be the set of atoms and terms, respectively, occurring in f. For every model $M \vDash f$,

$$\text{ModelAssertion}(M) = \text{ModelAssertion}(M, A_f, T_f)$$

Theorem 2.2 (ModelAssertion Satisfies Formula)**.** *For all M, f s.t. $M \vDash f$, if a model $M_0 \vDash \text{ModelAssertion}(M)$ then $M_0 \vDash f$.*

This theorem can be shown inductively on the structure of f.

Interpreted Theories

In Satisfiability Modulo Theories (SMT) we often want some symbols to behave according to predetermined axioms. For instance, we want bit-vector operations to behave like bit-vector operations. Following the Handbook [27] we define such theories as classes of models with the same signature.

Definition 2.19. A Σ-*theory* T is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class of models.

We need to relate theories that share sorts, function, and predicate symbols. Subsignatures formalize what it means for those signatures to relate in a consistent way.

Definition 2.20. A signature Σ is a *subsignature* of Ω, written $\Sigma \subseteq \Omega$, if $\Sigma^{\mathcal{S}} \subseteq \Omega^{\mathcal{S}}$, $\Sigma^{\mathcal{P}} \subseteq \Omega^{\mathcal{P}}$, and $\Sigma^{\mathcal{F}} \subseteq \Omega^{\mathcal{F}}$; and every function or predicate symbol of Σ has the same sort as in Ω. In this case, Ω is also a *supersignature* of Σ.

A reduct restricts the symbols and variables to which an interpretation applies but does not otherwise change the interpretation mapping.

Definition 2.21 (Reduct)**.** Let $\mathcal{I}$ be an Ω-interpretation over a set Y of variables. $\mathcal{I}^{\Sigma,X}$ denotes the *reduct of $\mathcal{I}$ to* (Σ, X) if for every $\Sigma \subseteq \Omega$ and $X \subseteq Y$, it is the Σ-interpretation over X obtained by restricting $\mathcal{I}$ to interpret only the symbols in Σ and variables in X.

A formula is satisfiable in a theory if symbols in the formula that come from that theory are consistent with the theory's class of models.

Definition 2.22 (Theory Satisfiability)**.** Let $T = (\Sigma, \mathbf{A})$ be a Σ-theory. A *T-interpretation* is any Ω-interpretation $\mathcal{I}$ for some $\Omega \supseteq \Sigma$ such that $\mathcal{I}^{\Sigma,\emptyset} \in \mathbf{A}$. A formula f is *T-satisfiable* or *satisfiable in T* iff it is satisfied by some T-interpretation $\mathcal{I}$. A set Φ of Ω-formulas *T-entails* an Ω-formula f, written $\Phi \vDash_T f$, iff every T-interpretation that satisfies every formula in Φ also satisfies f. Φ is *T-satisfiable* iff $\Phi \nvDash_T$ *false*. f is *T-valid* iff $\emptyset \vDash_T f$, usually written $\vDash_T f$.

This dissertation relies on two theories: quantifier-free bit-vectors and arrays. In SMT-LIB, these theories are called `QF_BV` and `QF_ABV` [28]. We model program scalars (values and variables) with bit-vectors. We model the program stack and heap with arrays.

Definition 2.23 (Bit-Vector Theory)**.** The *bit-vector signature* Σ_{BV} consists of sorts BV_i for each finite bit width ($i \geq 1$). For each sort there is a set of constants $\{0, 1\}^i$ representing a binary number of size i. We write $x^{[n]}$ to indicate that x is a concrete bit vector of bit width n.

We treat $n = 1$ as Boolean and omit the superscript. We also omit the superscript if it is unimportant or obvious from context.

The bit-vector theory includes a wide variety of interpreted symbols:

- Arithmetic: addition ($+^i$), subtraction ($-^i$), division (signed $\div_s^i$ and unsigned $\div_u^i$), remainder, etc.

- Logical: less-than ($<^i$), less-than or equal, etc., both signed and unsigned;
- Bitwise: concatenation, extraction, and, or, not, etc.

We write using the typical mathematical operators with a bit-width superscript: for example, $x^{[32]} +^{32} y^{[32]}$ for 32-bit addition of x and y. This example is quite horrendous to look at, so we can simplify this to $x +^{32} y$; the operator $+^{32}$ *requires* that its arguments each be 32-bit bit-vectors, so this syntax is no less precise and a bit more pleasant to read. Usually, we write this as $x + y$ when the bit-widths of x and y are clear from context.

All operators are parameterized by a bit width and it is an error to apply them to bit-vectors of a different width. All of the rules about which sorts may be passed to which functions are specified in the `QF_BV` logic description in SMT-LIB.

Definition 2.24 (Array Theory)**.** The *array signature* Σ_{ARR} has sorts $\textsc{Array}_{[\mathrm{i}\mapsto\mathrm{v}]}$ for arrays whose indices are bit-vectors of width i and whose values are bit-vectors of width v. It has the particular function symbols select, store, and const-array. The theory is defined by McCarthy's axioms [17], extended with axioms for extensionality and constant initialization:

$$\forall aije.\ i \simeq j \implies \text{select}(\text{store}(a, i, e), j) \simeq e \tag{2.17}$$

$$\forall aije.\ i \neq j \implies \text{select}(\text{store}(a, i, e), j) \simeq \text{select}(a, j) \tag{2.18}$$

$$\forall ab.\ (\forall i.\ \text{select}(a, i) \simeq \text{select}(b, i)) \implies a \simeq b \tag{2.19}$$

$$\forall ik.\ \text{select}(\text{const-array}(k), i) \simeq k \tag{2.20}$$

We consider a theory of arrays with extensionality and constant-initialized arrays. This theory is based on the `QF_ABV` theory from SMT-LIB. The first two axioms specify array accesses. The third axiom specifies that arrays with identical elements are equal: this feature is known as extensionality. The fourth axiom specifies that every index of a constant-initialized array has the initializer value. We consider this array theory — specifically including extensionality and constant initialization — because of its utility for software verification. Programs commonly bulk-initialize arrays and array equality allows encodings to be composed easily. Also, Z3, the SMT solver we use, supports this theory.

A formula's vocabulary is the uninterpreted symbols it uses.

Definition 2.25 (Vocabulary)**.** Let $T = (\Sigma, \mathbf{A})$ be a Σ-theory and Σ_1 the set of interpreted symbols used in the theory. For every T-formula ϕ, the *vocabulary* of ϕ, denoted $\mathcal{V}(\phi)$, is the subset of symbols from $(\Sigma^{\mathcal{F}} \cup \Sigma^{\mathcal{P}}) \setminus \Sigma_1$ occurring in ϕ.

2.2. EUF

Equality with Uninterpreted Functions (EUF) is a decidable, expressive fragment of first-order logic without quantification. The EUF logic grammar is a restriction of the FOL grammar in Definition 2.2: quantification is disallowed.

EUF has some advantageous properties. Fast congruence closure algorithms run in almost linear time [29]. The algorithm is quite efficient in practice and is amenable to efficient queries under assumptions.

EUF obeys a congruence rule which states that equal function arguments give equal results. In other words, function symbols behave like mathematical functions.

Definition 2.26 (EUF Congruence)**.** The *congruence rule for EUF* states that for all terms $t_1, t_2, \dots, t_n$ and $u_1, u_2, \dots, u_n$ $(n > 0)$, and every model M, the following two conditions hold:

1. For every function symbol F of arity n, if $M \vDash t_1 \simeq u_1, M \vDash t_2 \simeq u_2, \dots, M \vDash t_n \simeq u_n$ then $M \vDash F(t_1, t_2, \dots, t_n) \simeq F(u_1, u_2, \dots, u_n)$.
2. For every function symbol P of arity n, if $M \vDash t_1 \simeq u_1, M \vDash t_2 \simeq u_2, \dots, M \vDash t_n \simeq u_n$ then $M \vDash P(t_1, t_2, \dots, t_n) \iff P(u_1, u_2, \dots, u_n)$.

The congruence closure of a set of EUF equalities partitions the terms into equivalence classes. Congruence closure is the main building block for checking satisfiability of an EUF formula.

To simplify matters, we formalize EUF satisfiability of a conjunction of literals without predicate symbols. This loses no generality, since every predicate symbol can be replaced by equality with a new function symbol, as follows. First, introduce a fresh sort B and fresh constant T of sort B. Next, introduce a fresh function symbol F_P of sort $\gamma_1 \cdots \gamma_n B$ for each predicate symbol P of sort $\gamma_0 \cdots \gamma_n$ and replace every occurrence of $P(t_1, \dots, t_n)$ with $F_P(t_1, \dots, t_n) = \mathsf{T}$ and every occurrence of $\neg P(t_1, \dots, t_n)$ with $\mathsf{F}_P(t_1, \dots, t_n) \neq \mathsf{T}$.

Under this transformation, any set of literals Φ can be partitioned into a set of equalities $E_{\simeq}$ and disequalities $E_{\not\simeq}$. The congruence closure $E^*_{\simeq}$ partitions $E_{\simeq}$ into equivalence classes. Φ is then satisfiable exactly when no equivalent terms are disequal according to $E_{\not\simeq}$. The following definition makes this precise.

Definition 2.27 (Congruence Closure)**.** Let $\Phi = E_{\simeq} \cup E_{\not\simeq}$ be a set of literals over signature Σ and $E_{\simeq}$ be the set of equalities and $E_{\not\simeq}$ the set of disequalities from Φ. The *congruence closure* $E^*_{\simeq}$ of $E_{\simeq}$ is the smallest equivalence relation over terms in Φ that includes $E_{\simeq}$ and satisfies congruence:

For every pair of terms $F(t_1, \dots, t_n)$ and $F(u_1, \dots, u_n)$, if $(t_i, u_i) \in E^*_{\simeq}$ $(i \in \{1, \dots, n\})$, then $(F(t_1, \dots, t_n), F(u_1, \dots, u_n)) \in E^*_{\simeq}$.

Φ is *satisfiable* iff for every $t_1 \neq t_2 \in E_{\not\simeq}$, $(t_1, t_2) \notin E^*$.

To be explicit, we extend normal interpretations to EUF by including an explicit partition of terms from a given formula that is consistent with its interpretation.

Definition 2.28 (EUF Model)**.** For a signature Σ and set of variables $X \subseteq \mathbf{X}$, an *EUF Σ-model $\mathcal{M}$* for f over X is a Σ-interpretation over X along with a congruence closure $\mathcal{M}^*_{\simeq}$ where for every term $t_1, t_2 \in \text{Terms}(f)$, $t_1, t_2 \in \mathcal{M}_{\simeq}$ iff $\mathcal{M} \vDash t_1 \simeq t_2$.

Because we frequently deal with models and their term congruence closures, we develop a nicer syntax for writing them down, in terms of set partitions.

Definition 2.29 (Partition of a Set)**.** A *partition* of a set S is any set P such that:

1. $\left(\bigcup_{U \in P} U\right) = S$
2. for all $U, V \in P$, either $U = V$ or $U \cap V = \emptyset$.

In words, a partition of S groups all the elements of S into non-overlapping subsets. For instance, $\{\{t_1\}, \{t_2, t_3\}\}$ is a partition of $\{t_1, t_2, t_3\}$.

A partition provides an alternative representation for an equivalence relation, such as a congruence closure.

Definition 2.30 (Partition of an Equivalence Relation)**.** Let R be an equivalence relation on some set S. The *R-partition* is the unique set

$$\text{Part}[R] = \bigcup_{x \in S} \{y \mid (x, y) \in R\} .$$

For every $x \in S$, the *equivalence set for x* is $\text{Part}[R](x) = X$, where $X \in \text{Part}[R]$ is the unique set where $x \in X$.

The function notation allows us to go from an element to the set of all elements equivalent to it.

An R-partition retains all information from the equivalence relation R. R can be recovered from its R-partition like so:

$$R = \{(x, y) \mid x \in S, y \in S, S \in \text{Part}[R]\}$$

Written partitions can be cluttered, especially for partitions of terms from EUF models, due to many curly braces and parentheses. We introduce a neater notation for partitions that is just as precise.

Definition 2.31 (Partition Notation)**.** Let $\{S_1, S_2, \dots, S_n\}$ be an arbitrary partition of S. We write this as

$$\{S_1^* \mid S_2^* \mid \cdots \mid S_n^*\}$$

where S_i^* denotes the elements of S_i written without the (implied) curly braces.

For example, $\{x \mid y, z\}$ denotes the partition $\{\{x\}, \{y, z\}\}$ of $\{x, y, z\}$.

Example 2.1 (Possible Partitions)**.** On three terms t_1, t_2, t_3 there are four possible partitions:

$$\{t_1, t_2, t_3\} \qquad \text{all terms equivalent} \tag{2.21}$$

$$\{t_1 \mid t_2, t_3\} \qquad t_1 \neq t_2, t_1 \neq t_3, t_2 = t_3 \tag{2.22}$$

$$\{t_1, t_2 \mid t_3\} \qquad t_1 = t_2, t_1 \neq t_3, t_2 \neq t_3 \tag{2.23}$$

$$\{t_1 \mid t_2 \mid t_3\} \qquad \text{no terms equivalent} \tag{2.24}$$

The Bell number, $B_n = \sum_{i=0}^{n} S(n, i)$, is the number of ways to partition n objects into disjoint sets. The Stirling number of the second kind, $S(n, i)$, is the number of ways to partition a set of n objects into i non-empty subsets.

2.3. EUF Abstraction

EUF can be used to abstract bit-vector and array terms. Such abstraction is described informally in [4]. Abstract formulas over-approximate their concrete counterparts. Recovering the concrete formulas is easy: constant terms (which stand for concrete constants) are mapped to their concrete countererparts; UFs and UPs are mapped to their concrete operations by name. Abstract variables are mapped to their concrete counterparts. Consider a concrete formula $\phi(X)$ and its EUF abstraction $\widehat{\phi}(\widehat{X})$. The relation of the concrete and abstract systems is that the concretization ϕ of any valid EUF formula $\widehat{\phi}$ is valid.

Theorem 2.3 (EUF Abstraction Validity Implies Concrete Validity [4]). *Let ϕ be a formula over the interpreted theories of bit-vectors and arrays and let $\widehat{\phi}$ be ϕ's EUF abstraction. Then:*

$$\vDash \widehat{\phi} \implies \phi \, .$$

Although the general idea of EUF abstraction is not ours, our version is particular to EUFORIA so we cover it in Chapter 3.

2.4. Transition Systems

Definition 2.32 (Transition System [30], [31]). A *transition system* $\mathcal{T} = (X, Y, I, T)$ is a tuple 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. The system's *transition relation* $T(X, Y, X')$ is a formula over the current-state, next-state, and input variables. The set of *next-state variables* is $X' = \{x'_1, x'_2, \ldots, x'_n\}$.

For the remainder of this section, fix such a transition system $\mathcal{T}$.

Formulas over state variables are special because we use them to denote sets of states.

Definition 2.33 (State Formula)**.** A *state formula* $\sigma(X)$ is a formula whose free variables are drawn solely from the state variables of $\mathcal{T}$.

For instance, a single state where $x_1 = 1$ and $x_2 = 2$ can be denoted by formula $x_1 \simeq 1 \wedge x_2 \simeq 2$. The formula $x_1 \simeq 1 \wedge x_2 \simeq x_1 + 1$ also denotes the same state, so a set of states doesn't uniquely identify a formula. We frequently refer to state cubes, which are simply state formulas that are also cubes.

Hereafter, state formulas are identified with the sets of states they denote. For example, the formula $(x_1 \simeq x_2)$ denotes all states where x_1 and x_2 are equal, and other variables may have any value. We may also omit X from σ if the transition system is clear from context.

Definition 2.34 (Transition Formula)**.** A *transition formula* $\tau(X, Y, X')$ is a formula whose free variables are drawn solely from the current-state, next-state, and input variables; and which contains at least two out of three of: a current-state, an input, and a next-state variable.

The latter clause of the definition ensures that state formulas aren't a kind of transition formula. A transition formula talks about a transition.

Definition 2.35 ($\mathrm{Vars}_{\mathcal{T}}$)**.** For state every formula $\sigma(X)$, $\mathrm{Vars}_{\mathcal{T}}(\sigma(X))$ denotes the set of state variables of $\mathcal{T}$ free in σ. $\mathrm{Vars}'_{\mathcal{T}}(\sigma(X))$ denotes the set of next-state variables free in σ.

Definition 2.36 (State Space)**.** The *state space* of $\mathcal{T}$ is the set of all valuations to variables in X; it is denoted by $\Omega(X)$.

Priming makes it easy to talk about current- or next-state variables without explicitly referencing the state variable set. A formula without current-state variables is unchanged after priming; same for next-state variables and unpriming.

Definition 2.37 (Priming)**.** $\mathrm{prime}(\sigma)$ stands for the formula $\sigma[X \mapsto X']$, that is, all state variable occurrences are replaced with primed (i.e., next-state) state variables. $\mathrm{unprime}(\sigma)$ is $\sigma[X' \mapsto X]$.

We use transition systems to represent programs. Each "step" of the transition system involves a single update of each state variable; on this step, input variables are set arbitrarily.

Definition 2.38. There is a *transition* from $\sigma_i(X)$ to $\sigma_j(X)$ under T iff $\sigma_i(X) \wedge T \vDash \sigma_j'(X)$. $\sigma_j(X)$ is a *successor* of $\sigma_i(X)$ and $\sigma_i(X)$ is a *predecessor* of $\sigma_j(X)$ under T.

Definition 2.39 (Transition System Execution)**.** A (possibly-infinite) sequence of states $\sigma_0(X), \sigma_1(X), \ldots$ is an *execution* of $\mathcal{T}$ if $\sigma_0(X) \vDash I(X)$ and for every pair $(\sigma_i(X), \sigma_{i+1}(X))$, there is a transition from $\sigma_i(X)$ to $\sigma_{i+1}(X)$, i.e., $\sigma_i(X) \wedge T \vDash \sigma_{i+1}'(X)$.

2.5. Model Checking

A model checker's purpose in life is to check whether a property holds in a given transition system. This dissertation only discusses safety properties, even though there are many other classes of interesting properties. Informally, a safety property specifies that the system doesn't do "something bad." For instance, whether it is possible to reach a state where $x = 0$ is a safety property.

The core idea of model checking is to compute the set of reachable states and see—is my bad state in the reachable set?

To make this idea precise we first define the set of reachable states. Let $\mathcal{T}_{2.5} = (X, Y, I, T)$ be an arbitrary transition system.

Definition 2.40 (Pre- and Post-image States)**.** For every set of states $S \subseteq \Omega(X)$,

$$\text{pre}(S, T) = \{s \in \Omega(X) \mid \exists t \in S.\ s \wedge T \vDash t'\} \tag{2.25}$$

$$\text{post}(S, T) = \{t \in \Omega(X) \mid \exists s \in S.\ s \wedge T \vDash t'\} \tag{2.26}$$

pre and post denote the set of predecessors (respectively, successors) of the states in S. Usually, instead of sets of states, we manipulate formulas denoting them. Let's extend this definition to formulas.

Definition 2.41 (Pre- and Post-images [32])**.** For an arbitrary state formula $R(X)$:

$$\text{post}(R, T) = \text{unprime}(\exists X.\ R \wedge T) \tag{2.27}$$

$$\text{pre}(R, T) = \exists X'.\ (\text{prime}(R) \wedge T) \tag{2.28}$$

post(R, T) is a logical assertion that describes the set of all successor states of R. pre(R, T) is a logical assertion that describes the set of all predecessor states of R. We omit the T argument and just write post(R) if the transition relation is obvious from context. The set of successor states is called the *(post-)image* and the set of predecessor states is called the *pre-image*.

Using post we can define the set of all states reachable in $\mathcal{T}_{2.5}$.

Definition 2.42. For every transition system $\mathcal{T}$, the set of all reachable states in $\mathcal{T}$ is

$$\tilde{R}(\mathcal{T}) = \bigcup_{n=0}^{\infty} \tilde{R}_n \tag{2.29}$$

where $\tilde{R}_i$ is defined inductively as the set of states reachable in i steps: $\tilde{R}_0 = I$ and $\tilde{R}_i = \text{post}(\tilde{R}_{i-1})$ for $i > 0$.

Bad states are those outside the property.

Definition 2.43 (Safety Property)**.** A *safety property* is a state formula, $P(X)$.

Definition 2.44 (Model Checking Problem)**.** Let $P(X)$ be a safety property. The *model checking problem* is to determine whether any state satisfying $\neg P(X)$ is reachable through some execution of T. A *model checking instance*, therefore, is a tuple (X, Y, I, T, P) of a transition system and property.

Definition 2.45 (Counterexample)**.** A *counterexample* to a safety property $P(X)$ is a k-step execution

$$\sigma_0(X), \sigma_1(X), \dots, \sigma_k(X)$$

such that

$$\sigma_k(X) \vDash \neg P(X) .$$

It's not usually feasible to calculate the set of reachable states for a reasonably sized transition system. Often the number of states is astronomical; or those states do not have a concise logical description, or it is difficult to discover; or the set is infinite (though it is finite in this dissertation). Instead, we can show that $\neg P$ is unreachable by finding an inductive invariant.

Definition 2.46 (Inductive Invariant)**.** An *inductive invariant S* has the following properties:

$$I \vDash S \qquad \textit{initiation} \qquad (2.30)$$

$$\text{post}(S) \vDash S \qquad \textit{consecution} \qquad (2.31)$$

If S is an inductive invariant and also $S \vDash P$, then it is an *inductive invariant for P*.

Inductive invariants are a fundamental technique used in the IC3 model checking algorithm, on which EUFORIA is based.

2.6. IC3 Algorithm

Bradley introduced a model checking technique, Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) [16], that builds inductive invariants *incrementally*, meaning that the inductive invariant is built one piece at a time until it holds for every execution of a transition system. This dissertation explores an extension of IC3.

The algorithm in Figure 2.1 forms the basis of the extension in this work. Our presentation is based on a particular implementation of IC3 called Property Directed Reachability (PDR) [33]. The algorithm is defined in terms of three main objects:

1. sets of cubes F_i, where each cube $s(X) \in F_i$ represents a set of unreachable states;
2. $\mathcal{M}$, a queue of proof obligations (pob) $\langle M, i\rangle$ where M is a cube, the level $i \in \mathit{Int}$, and the queue is ordered by lowest level pobs first; and
3. D, the current exploration depth.

IC3 is formalized as CHECK (Figure 2.1). CHECK takes a model checking instance as input. The sequence R_i represents over-approximate reachability frontiers. R_i ($i \in \{1, \ldots, D\}$) is represented implicitly using the set of cubes F_i. $R_0 = I$. A cube $c \in R_i$ means "the states c are definitely unreachable after i transitions, and we don't yet know whether they are unreachable after $j > i$ transitions."

```
          $D$                                                      *current depth*
Globals:  $F_i$                                                    *set of cubes, $i \in \{0, 1, \dots, D\}$*
          $R_i \equiv \bigwedge_{j=i}^{N+1} \bigwedge_{\hat{c} \in F_j} \neg c$      *reachable set (over-approximate)*
1: procedure CHECK($I, T, P$)
2:     INITIALIZE()
3:     while *true* do
4:         if $M = \text{SAT}(R_D \wedge \neg P)$ then
5:             $c \leftarrow$ GETBADCUBE($M$)
6:             if BACKWARDREACH($\langle c, D \rangle$) then
7:                 return *true*                                   ▷ found counterexample
8:         else
9:             NEWFRAME()
10:            if FORWARDPROPAGATE() then
11:                return *false*                                  ▷ found invariant
12: procedure INITIALIZE()
13:     $D \leftarrow 0$
14:     push $F_0 \leftarrow \{I(X)\}$
15:     push $F_\infty \leftarrow \emptyset$
16: procedure NEWFRAME()
17:     push $F_D \leftarrow \emptyset$
18:     $D \leftarrow D + 1$
19: procedure GETBADCUBE($M$)
20:     return ModelAssertion($M$)
21: procedure ADDUNREACHABLECUBE($\langle M, i \rangle$)
22:     for $j \in \{1, \dots, i\}$ do
23:         if $M \subseteq c$ for any $c \in F_j$ then
24:             $F_j \leftarrow F_j \setminus \{c\}$
25:     $F_i \leftarrow F_i \cup \{M\}$
```

Figure 2.1: Word-level IC3.

The SAT query on line 4 asks whether there is a state not satisfying P in R_D. If so, the state is generalized to a cube c and is the first proof obligation for backward reachability. Otherwise, the system is safe up to D steps and so a new from is created. Next, FORWARDPROPAGATE attempts to prove that currently-unreachable states are still unreachable at a greater depth. In the process, the procedure may discover an invariant.

IC3 alternates between two phases: backward reachability (Figure 2.2a) and forward propagation (Figure 2.2b).

Backward reachability (Figure 2.2a) attempts to a construct a counterexample to P with at least D transitions (possibly more). It manages a queue $\mathcal{M}$ of proof obligations that represent potential executions to $\neg P$. At each iteration, it chooses a pob pair $\langle M, i\rangle$. If the pob is at step $i = 0$, then it represents an execution to $\neg P$. Otherwise, IC3 performs a counterexample-to-induction (CTI) query (line 9) to see if cube M is reachable from the current $(i-1)$-step over-approximation (lines 3–9). If so, GENERALIZEFEASIBLE generalizes the pre-state and adds it to the queue (lines 9–12). It also adds the current pob to the queue so that it will be examined again, after possibly proving other pob's unreachable. If M is not reachable, GENERALIZEINFEASIBLE generalizes the unreachable cube M to refine the reachability frontier i and possibly later frontiers also (lines 13–18).

Forward propagation (Figure 2.2b) pushes unreachable cubes forward, attempting to prove states unreachable at a further depth (lines 23–27). Line 28 checks whether two (over-approximate) reachable sets become identical, i.e., $R_i = R_{i+1}$ $(i < N)$. If so, the algorithm terminates having discovered an inductive invariant for P.

Bradley [16] and Een *et al.* [33] give proofs of correctness for IC3 and PDR, respectively.

The presentation of EUFORIA in this dissertation builds on the algorithm CHECK.

2.7. Counterexample-guided Abstraction & Refinement

CEGAR is usually referred to as a framework rather than an algorithm. It is a template for algorithms that must be instantiated for the particular abstraction being used. This dissertation uses transition systems over bit-vectors and arrays as the concrete rep-

```
1: procedure BackwardReach($\langle M_{in}, i_{in} \rangle$)
2:     $\mathcal{M} \leftarrow \langle M_{in}, i_{in} \rangle \epsilon$
3:     while $\mathcal{M} = \langle M, i \rangle \mathcal{M}'$ do
4:         $\mathcal{M} \leftarrow \mathcal{M}'$
5:         if $i = 0$ then
6:             return true
7:         if $\neg$SAT($R_i \wedge M$) then
8:             continue                                          ▷ $M$ known unreachable
9:         if $M_0$ = SAT($\neg M \wedge R_{i-1} \wedge T \wedge M'$) then
10:            $M_s \leftarrow$ GeneralizeFeasible($M_0$)               ▷ Overridden in Chapter 3
11:            $\mathcal{M} \leftarrow \langle M_s, i-1 \rangle \mathcal{M}$
12:            $\mathcal{M} \leftarrow \langle M, i \rangle \mathcal{M}$
13:        else
14:            $\langle M_b, k \rangle \leftarrow$ GeneralizeInfeasible($\langle M, i \rangle$)
15:                                                   ▷ Attempt to generalize more
16:            while $k < D - 1$ and $\neg$SAT($\neg M_b \wedge R_{k-1} \wedge T \wedge M_b'$) do
17:                $\langle M_b, k \rangle \leftarrow$ GeneralizeInfeasible($\langle M_b, k \rangle$)
18:            AddUnreachableCube($\langle M_b, k \rangle$)
19:            if $k < D$ then
20:                $\mathcal{M} \leftarrow \langle M_b, k+1 \rangle \mathcal{M}$
21:    return false
```

(a) Backward reachability.

```
22: procedure ForwardPropagate()
23:     for $i \in \{1, \dots, D-1\}$ do
24:         for $s \in F_i$ do
25:             if $\neg$SAT($R_i \wedge T \wedge s'$) then
26:                 $m \leftarrow$ maximum in $\{i+1, \dots, D+1\}$ at which $s$ blocked
27:                 AddUnreachableCube($\langle s, m \rangle$)
28:         if $F_i$ is empty then
29:             return true                                        ▷ invariant found
30:     return false
```

(b) Forward propagation.

Figure 2.2: Backward reachability and forward propagation.

resentation and EUF abstraction. One could also use Kripke structures over concrete formulas and predicate abstraction.

Abstraction refinement is the process of bringing the over-approximate abstract representation closer to the concrete one. CEGAR doesn't dictate the refinement mechanism; it specifies the trigger, which is the counterexample.

CEGAR is a three-step process. Let $\mathcal{P}$ be the concrete representation of the program, such as a transition system.

1. Compute an initial abstraction $\widehat{\mathcal{P}}$ of $\mathcal{P}$. This typically involves abstracting the property as well. This step also often involves whatever high level text of the program is available.

2. Model check the current abstraction $\widehat{\mathcal{P}}$ which results in either: (a) $\widehat{\mathcal{P}}$ has a counterexample $\widehat{C}$ in $\widehat{\mathcal{P}}$ that violates the abstract property; (b) $\widehat{\mathcal{P}}$ satisfies the abstract property, in which case CEGAR is done. In the former case, CEGAR examines the concretization C of $\widehat{C}$ to determine whether $\mathcal{P}$ has that counterexample and, if so, terminates. If there is no counterexample in $\mathcal{P}$ corresponding to C, $\widehat{C}$ is said to be *spurious* and CEGAR proceeds to step 3.

3. Keying off of $\widehat{C}$, refine the program abstraction and update the abstraction $\widehat{\mathcal{P}}$. Refinement must increase the fidelity of $\widehat{\mathcal{P}}$ enough so that $\widehat{C}$ is no longer a counterexample in $\widehat{\mathcal{P}}$. A subsequent model check of $\widehat{\mathcal{P}}$ may give a different result, so CEGAR returns to step 2.

This high-level presentation of CEGAR is sufficient to discuss EUFORIA. See the Handbook [27], Chapter 13, for a thorough discussion of abstraction refinement and CEGAR.

2.8. Horn Encodings

This section gives a brief explanation of Horn encodings, a different formalism for programs. We will discuss how programs are encoded as Horn in Chapter 4.

Definition 2.47 (Horn Clause)**.** A *Horn clause* or *rule* is a universally quantified formula with a *body* formula and a *head* formula

$$\forall x_1, \ldots, x_m . \underbrace{\bigwedge_{k=1}^{j} P_k(\overline{x}_k) \land \phi(x_1, \ldots, x_m)}_{body} \Rightarrow head \tag{2.32}$$

where for every k, $P_k \in \mathcal{R}$ is an uninterpreted predicate symbol, $\overline{x}_k \subseteq \{x_1, \ldots, x_m\}$, and $|\overline{x}_k| = |P_k|$ [34].

The constraint ϕ is a formula that uses no uninterpreted atoms. *head* must either be an application of an uninterpreted predicate or an interpreted formula. In this dissertation, $j = 1$, which corresponds to linear Horn clauses.

2.9. Model Checking Tools

This section gives a brief description two model checkers, SPACER and IC3IA, which we use when evaluating EUFORIA.

SPACER

SPACER [34]–[36] is a state-of-the-art model checker. It is an over- and under-approximation driven incremental model checker that is tightly integrated with Z3. It is capable of inferring quantified array invariants and uses model-based projection array procedures to lazily instantiate property-directed array axioms. It represents procedure summaries as logical formulas to support checking programs with recursive functions.

IC3IA

IC3IA [37] is an IC3-style CEGAR model checker that implements implicit predicate abstraction. IC3IA's architecture is quite similar EUFORIA's, more similar than SPACER's.

As discussed in Cimatti [37], IC3IA is superior to state-of-the-art bit-level IC3 implementations and can support hundreds of predicates, around an order of magnitude more than what explicit predicate abstraction tools practically support.

Chapter 3.

EUFORIA: A Model Checker for C Programs

This chapter presents the core ideas behind EUFORIA, culminating in a checker that works for C programs with scalar variables. In other words, all C operations and loops are supported; arrays and procedures are not supported.

First we discuss a method for encoding C programs into a logical representation, using LLVM as intermediate representation (Section 3.2). Next, we introduce EUFORIA, a novel model checker built around IC3 (Section 3.3). In the process, we introduce two of our novelties: an efficient, sound EUF abstraction for programs (Section 3.3.1) and a fast projection procedure for generalizing pre-images (Section 3.3.3). This procedure makes it possible to adapt IC3 to EUF transition systems. We then discuss refinement and give proofs of correctness and termination (Sections 3.4 and 3.5). Finally, we present an evaluation of our checker (Section 3.6).

3.1. Introduction

The first chapter introduced the concept of control properties. A variety of important properties are control properties. We gave examples such as dropping elevated privileges, locking disciplines, and typestate properties. These kinds of properties motivate the checker presented in this chapter.

The typical approach for verifying control properties is predicate abstraction [38], [39], 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.

3.2. Encoding C Programs

The way one encodes programs is crucial to the scalability of a model checker, and encodings are often developed alongside checking algorithms. We construct our encoding carefully to maximize the effectiveness of EUF abstraction. For instance, flipped occurrences of commutative operations (e.g., $x + y$ and $y + x$ occur in the program) are normalized to a fixed order. EUFORIA thus avoids learning some refinements about commutativity. We want to apply EUFORIA to C programs, so this section describes an encoding of LLVM programs into a logical representation amenable to analysis.

First we present a pared-down LLVM-like language, MiniLLVM, based on the Vellvm work [40], [41]. Vellvm specifies a formal semantics for LLVM programs. We only require a small subset of the language in order to give a formal description of our encoding. We encode programs in this language as transition systems over SMT formulas.

Figure 3.1 shows the syntax and operational semantics of MiniLLVM. The language has statements, local variables, global variables, the usual suspect operators, (conditional) branches, and a single `@main` function. Statements are grouped into labeled blocks. If a block labeled l_1 ends with a branch to a block labeled l_2, we say l_2 is a successor of l_1 and l_1 is a predecessor of l_2. The language is in Static Single Assignment (SSA) form, so it includes the usual phi nodes. Phi nodes support two predecessors;

MiniLLVM syntax:

Category	Meta-variable		Productions
Types	typ	::=	$\mathbf{i}sz \mid \mathbf{void} \mid typ*$
Constants	$cnst$	::=	$\mathbf{i}sz\ Int \mid (typ*)\mathbf{null}$
Values	val	::=	$id \mid cnst$
Binops	bop	::=	$\mathbf{add} \mid \mathbf{mul} \mid \mathbf{sdiv} \mid \mathbf{load} \mid \mathbf{store} \mid \cdots$
Right-hand-sides	rhs	::=	$val_1\ bop\ val_2$
Commands	c	::=	$id := rhs$
Terminators	tmn	::=	$\mathbf{br}\ val\ l_1\ l_2 \mid \mathbf{return}\ typ\ val$
Phi Nodes	ϕ	::=	$id = \mathbf{phi}\ typ\ [val_1, l_1], [val_2, l_2]$
Instructions	$insn$	::=	$\phi \mid c \mid tmn$
Non-ϕs	ψ	::=	$c \mid tmn$
Blocks	b	::=	$l\ \overline{\phi}\ \overline{c}\ tmn$
Functions	f	::=	$\mathbf{define\ void}$ `@main`$()\{b_s, \overline{b}, b_e\}$
Products	$prod$	::=	$id = \mathbf{global}\ typ\ cnst \mid f$
Module	mod	::=	$\overline{prod}$

Semantic domains:

$$\begin{array}{llll llll} \text{Values} & v & ::= & Int & \text{Environment} & \delta & ::= & id \mapsto v \\ \text{Frames} & \sigma & ::= & (pc, \delta) & \text{Prog Counters} & pc & ::= & l.i \mid l.\mathbf{t} \end{array}$$

MiniLLVM operational semantics:

$$\textsc{Binary-Op}\quad \frac{c = r := val_1\ bop\ val_2 \qquad \langle\!\langle val_1 \rangle\!\rangle_\delta = v_1 \qquad \langle\!\langle val_2 \rangle\!\rangle_\delta = v_2 \qquad \mathbf{eval}(bop, v_1, v_2) = v_3}{f \vdash (l, (c, \overline{c}), tmn, \delta) \longrightarrow (l, \overline{c}, tmn, \delta[r \hookrightarrow v_3])}$$

$$\textsc{Branch}\quad \frac{f[l_3] = (\overline{\phi_3}\ \overline{c_3}\ tmn_3) \qquad \langle\!\langle val \rangle\!\rangle_\delta = v \qquad l_3 = (v\ ?\ l_1 : l_2) \qquad \langle\!\langle \overline{\phi_3} \rangle\!\rangle^l_\delta = \delta'}{f \vdash (l, [], \mathbf{br}\ val\ l_1\ l_2, \delta) \longrightarrow (l_3, \overline{c_3}, tmn_3, \delta')}$$

Figure 3.1: Syntax and operational semantics of MiniLLVM.

extending the encoding to more predecessors is straightforward. The metavariable *id* ranges over MiniLLVM identifiers such as `%x`, `%y`, `%tmp`, and so on. The metavariable l ranges over MiniLLVM labels. *sz* ranges over positive integers.

A MiniLLVM module consists of a number of global variables and a `@main` function. Functions are composed of a list of blocks $b_s, \overline{b}, b_e$ with a distinguished start block b_s and exit block b_e. A basic block has a labeled entry point l, a list of phi nodes $\overline{\phi}$, a list of commands $\overline{c}$, and a terminator instruction *tmn*. Labels are globally unique.

Commands c include the usual set of arithmetic, relational, and bitwise operators, in addition to memory operations **load** and **store**. The **br** and **return** commands branch to another block in the function, or return a value from the function, respectively. In MiniLLVM, there is only one occurrence of **return**, in the exit block of `@main`, which contains no other instructions.

The operational semantics of MiniLLVM is defined as a judgment that relates the current frame to the next frame in function f:

$$f \vdash (pc, \delta) \longrightarrow (pc', \delta') \ .$$

A frame $\sigma = (pc, \delta)$ specifies a control configuration of the program using the program counter *pc* and a map of variables to values δ. The current program counter is $\sigma.pc$ and the current mapping is $\sigma.\delta$. In a procedure, $l.i$ refers to the i'th instruction of block l and $l.\mathbf{t}$ to its terminator. We write $f[l] = b$ if there is a block b with label l in function f. A frame σ can be written in an expanded form as $(l, \overline{c}, tmn, \delta)$ where:

1. l is the label of the block at $\sigma.pc$ and

2. $\overline{c}$ and *tmn* are as-yet-unexecuted commands.

$\langle\!\langle val \rangle\!\rangle_\delta$ denotes the evaluation of a value to an integer, possibly by consulting the local state δ. **eval** computes values using concrete operations on given integers. $\langle\!\langle rhs \rangle\!\rangle_\delta$ denotes evaluating the right-hand side *rhs* using mapping δ.

Phi nodes perform assignments dependent on predecessor blocks. Each phi node has a two-element list where each element is of the form $[val, l]$ and where l must label a predecessor block. In MiniLLVM, phi nodes must occur at the beginning of a block and their assignments are evaluated simultaneously, using the current values of the right-hand sides. Consider the following example:

$$\begin{aligned}
l_0 :&\ \ \cdots \\
l_1 :&\ \ x = \textbf{phi int}\ [y, l_1], [0, l_0] \\
&\ \ y = \textbf{phi int}\ [x, l_1], [1, l_0] \\
&\ \ z := x = y \\
&\ \ \textbf{br}\ z\ l_1\ l_2 \\
l_2 :&\ \ \cdots
\end{aligned}$$

Block l_1 assigns to variables x and y. The respective values assigned to x and y depend on which predecessor was executed immediately before l_1, either l_0 or l_1. The assignments to x and y must be evaluated in parallel. Assume that the current state is l_1, $x = 0$, and $y = 1$. Evaluating the statements sequentially will produce the state $x = 1, y = 1$. Evaluating the statements in parallel will produce the state $x = 1, y = 0$. These results flip the direction of the branch; the parallel assignment is the correct one.

This parallel assignment is handled by $\langle\!\langle \overline{\phi_3} \rangle\!\rangle^l_\delta$ in the Branch rule. $\langle\!\langle \overline{\phi_3} \rangle\!\rangle^l_\delta$ returns a δ' in which variables from δ are updated according to the phi instructions $\overline{\phi_3}$ when the predecessor executed is l. The following exhibits the general case for a block l_3 when predecessor l was executed immediately before l_3:

$$f[l_3] = \begin{bmatrix} \overline{\phi_3} \\ \overline{\psi} \end{bmatrix}$$

where

$$\overline{\phi_3} = \begin{cases} v_1 & = \ \ \textbf{phi}\ \mathit{typ}_1\ [\mathit{val}_1, l], \ldots \\ v_2 & = \ \ \textbf{phi}\ \mathit{typ}_2\ [\mathit{val}_2, l], \ldots \\ \ldots & \\ v_k & = \ \ \textbf{phi}\ \mathit{typ}_k\ [\mathit{val}_k, l], \ldots \end{cases} .$$

Since the predecessor location l must occur once in each phi list, we can without loss of generality write it as occurring first. And since the other predecessor location is by assumption not relevant, we leave it unspecified. Then, the semantics of phi instructions are given as follows:

$$\langle\!\langle \overline{\phi_3} \rangle\!\rangle_\delta^l = \delta[\overline{v} \hookrightarrow \overline{val}] \quad \text{where} \quad \begin{array}{ll} \overline{v} & = (v_1, v_2, \dots, v_k) \\ \overline{val} & = (val_1, val_2, \dots, val_k) \end{array}$$

The Binary-Op rule evaluates a single non-control statement. Arguments to the binary operation are evaluated and then the result of the operation is computed. We don't belabor the exact semantics of each binary operation. Each operation corresponds in a relatively straightforward way to a bit-vector operation in SMT-LIB.

The initial frame is

$$(l.1, \delta_0) \tag{3.1}$$

and execution begins at `@main`. The initial δ_0 is populated with symbolic pointer values for each global variable. Although there is a store that models heap interactions, we leave it unspecified because we describe below how to soundly remove the heap from consideration for our encoding.

3.2.1. Encoding sans Memory

We now describe an encoding of a subset of MiniLLVM without memory operations, global variables, and using only scalar local variables. Subsequently, in Section 3.2.2, we describe how we translate a MiniLLVM program into one without memory operations or globals.

We modify the exit block l_e to read:

$$l_e : \mathbf{br}\ 1\ l_e\ l_e\ .$$

In other words, we change the return to a branch that introduces a self-loop. This indicates a safe program exit.

A program's state variables can be classified into two categories: program (local) variables V and location variables L. Let $\mathcal{L}$ denote the set of all labels in the program, including three distinguished labels, l_s and l_e and l_{err}, denoting the entry point, exit block, and error location of the program, respectively. We use a possibly-subscripted l to denote elements of $\mathcal{L}$. We define the encoder $\mu[\![\cdot]\!]$ to map program variables, operations, and locations to encoding objects. For instance, a variable x of type $\mathbf{i}32$ is encoded as a bit-vector $\mu[\![x]\!] = x^{[32]}$. The encoder maps sorts as follows:

$$\mu[\![\mathbf{i}1]\!] = \textsc{Bool} \tag{3.2}$$

$$\mu[\![\mathbf{i}sz]\!] = \mathrm{BV}_{sz} \tag{3.3}$$

Type $\mathbf{i}1$ is treated specially: it is encoded using Boolean values and operations.

The encoder maps program objects as follows:

$$\mu[\![\mathbf{i}sz\ cnst]\!] = cnst^{[sz]} \tag{3.4}$$

$$\mu[\![\mathbf{i}sz\ x]\!] = x^{[sz]} \tag{3.5}$$

$$\mu[\![v_1\ bop\ v_2]\!] = \mu[\![v_1]\!]\ \mu[\![bop]\!]\ \mu[\![v_2]\!] \tag{3.6}$$

$$\mu[\![l]\!] = \ell \qquad \text{for all } l \in \mathcal{L} \tag{3.7}$$

Variables and constants are encoded as bit-vectors of the appropriate type. Location variables are used to model 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., [42]–[44]). We encode each block label with a single Boolean location variable that becomes true when that block is reached. Location variables allow us to partition a program's state space into a set of disjoint control states. We usually just write ℓ instead of the more verbose $\mu[\![l]\!]$, and may subscript it as we do for labels themselves.

Let M be an arbitrary MiniLLVM program. The transition system encoding of M defined as (X, Y, I, T, P). First, we define the state space. Let $V_\phi = \{\mu[\![x]\!] \mid f[l] = (x = \mathbf{phi}\ typ\ [val_1, l_1], [val_2, l_2])\}$ be the set of variables occurring on the left-hand sides of phi instructions. Let $V_t = \{\mu[\![x]\!] \mid x := rhs\}$ be the rest of the variables. Now, $V = V_\phi \cup V_t$ is the set of bit-vector-encoded variables that occur in M. The set of locations is $L = L_\phi \cup L_b$, where $L_\phi = \{\ell_i \mid f[l_i] = (\phi\ \overline{\phi}\ \overline{c}\ tmn)\}$ is the set of basic blocks containing phi instructions and $L_b = \{\ell_i \mid f[l_i] = \overline{c}\ tmn\}$ is the rest of the blocks. The location variables for the start and exit blocks are ℓ_s and ℓ_e, respectively. There is a designated error location, ℓ_{err}. We can now define the state space as

$$X = V_\phi \cup L_\phi \cup \{\ell_s, \ell_e, \ell_{\mathrm{err}}\} \tag{3.8}$$
$$Y = V_t \cup L_b \ . \tag{3.9}$$

Next we define the transition relation, T. To define T we need to encode basic blocks, which are composed of phi nodes, commands, and a terminator. We discuss each one in turn.

Phi nodes A generic maximal contiguous sequence of phi nodes, for a block labeled l, looks like this:

$$f[l] = \begin{bmatrix} v_1 & = & \textbf{phi}\ \mathit{typ}_1\ [\mathit{val}_{11}, l_1], [\mathit{val}_{12}, l_2] \\ v_2 & = & \textbf{phi}\ \mathit{typ}_2\ [\mathit{val}_{21}, l_1], [\mathit{val}_{22}, l_2] \\ \dots & & \\ v_k & = & \textbf{phi}\ \mathit{typ}_k\ [\mathit{val}_{k1}, l_1], [\mathit{val}_{k2}, l_2] \\ \dots & & \end{bmatrix}$$

We assume predecessors are ordered the same way in each assignment, i.e., l_1 before l_2.

The state update for the state variables is encoded as the conjunction of the ite-trees below:

$$\begin{aligned} \mu[\![v_1]\!]' &\simeq \mathrm{ite}(\ell' \wedge \ell_1, \mu[\![\mathit{val}_{11}]\!], \mathrm{ite}(\ell' \wedge \ell_2, \mu[\![\mathit{val}_{12}]\!], \mu[\![v_1]\!])) \\ \mu[\![v_2]\!]' &\simeq \mathrm{ite}(\ell' \wedge \ell_1, \mu[\![\mathit{val}_{21}]\!], \mathrm{ite}(\ell' \wedge \ell_2, \mu[\![\mathit{val}_{22}]\!], \mu[\![v_2]\!])) \\ \dots & \\ \mu[\![v_k]\!]' &\simeq \mathrm{ite}(\ell' \wedge \ell_1, \mu[\![\mathit{val}_{k1}]\!], \mathrm{ite}(\ell' \wedge \ell_2, \mu[\![\mathit{val}_{k2}]\!], \mu[\![v_k]\!])) \end{aligned} \tag{3.10}$$

The variables are updated when control reaches l (indicated by ℓ') and their values value depend on the predecessor location. Note that this encoding ensures "parallel assignment" as required by the MiniLLVM semantics.

Commands Non-phi instructions are easily encoded. Let $f[l] = (id := rhs)$. These are encoded as global constraints of the form

$$\mu[\![id]\!] \simeq \mu[\![rhs]\!] \; . \tag{3.11}$$

Encoding the right-hand side consults a table mapping MiniLLVM operations into SMT-LIB operations in `QF_BV`. For instance, **add** is mapped to $+_{sz}$ where val_1 and val_2 have type **i***sz*.

Note that *id* is not encoded as a state variable in this case because it is not a phi node assignment. Note further that this assignment does not depend on l. The reason for this is that the program is in SSA form, so *id* is assigned in exactly one program location.

Branches Let l be the label of a basic block whose predecessor labels are $l_1, l_2, \ldots, l_n$. Also assume that l_i.**t** is either **br** val_i l _ or **br** val_i _ l; that is, each predecessor contains a branch to l using val_i and the opposite side of the branch is unspecified. The transfer of control to block l is encoded as follows

$$\text{prime}(\ell) \simeq \bigvee_{i \in \{1,\ldots,n\}} \ell_i \wedge \gamma \text{ where } \gamma = \begin{cases} \mu[\![val_i]\!] \simeq 0 & \text{if } l_i.\mathbf{t} \text{ is } \mathbf{br}\ val_i\ \_\ l \\ \mu[\![val_i]\!] \not\simeq 0 & \text{if } l_i.\mathbf{t} \text{ is } \mathbf{br}\ val_i\ l\ \_ \end{cases} \tag{3.12}$$

This equation captures the fact that reaching ℓ happens exactly when one of the predecessor locations has been reached.

Transition relation Finally, the transition relation T is defined as the conjunction of constraints (3.10), (3.11), and (3.12).

Initial state Initially, the program is at the entry block and nowhere else. This condition is encoded in the initial state:

$$I = \ell_s \bigwedge_{\ell \in L_\phi, \ell \neq \ell_s} \neg \ell \; . \tag{3.13}$$

This reflects the initial frame (3.1) for `@main`.

```
define i32 @main() {
s: ; start
  br label l
l: ; loop
  %i = phi [0, s], [%incr, b]
  %cond = icmp slt %i, 5
  br i1 %cond, label b, label d
b: ; body
  %incr = add %i, 3
  br label l
d: ; done
  %acond = icmp slt %i, 7
  br i1 %acond, label x, label e
err: ; error
  call __VERIFIER_error()
e: ; exit
  ret 0
}
```

```
$i = 0$
while $(i < 5)$ {
    $i = i + 3$
}
assert$(i < 7)$
```

(a) Example LLVM (top) for the pseudocode program (below).

Transition relation T:

$$i' \simeq \text{ite}(\ell_l' \wedge \ell_s, 0, \text{ite}(\ell_l' \wedge \ell_b, incr, i)) \tag{3.15}$$

$$\ell_s' \simeq \mathit{false} \tag{3.16}$$
$$\ell_l' \simeq \ell_s \vee \ell_b \tag{3.17}$$
$$\ell_{\text{err}}' \simeq \ell_d \wedge \neg acond \tag{3.18}$$
$$\ell_e' \simeq (\ell_d \wedge acond) \vee \ell_e \tag{3.19}$$
$$\ell_b \simeq \ell_l \wedge cond \tag{3.20}$$
$$\ell_d \simeq \ell_l \wedge \neg cond \tag{3.21}$$
$$cond \simeq (i < 5) \tag{3.22}$$
$$incr \simeq (i + 3) \tag{3.23}$$
$$acond \simeq (i < 7) \tag{3.24}$$

Initial state I:

$$\ell_s \wedge \neg\ell_l \wedge \neg\ell_{\text{err}} \wedge \neg\ell_e \tag{3.25}$$

Property P:

$$\neg\ell_{\text{err}} \tag{3.26}$$

(b) Encoding of LLVM into (X, Y, I, T, P). $X = \{\ell_s, \ell_l, \ell_{\text{err}}, \ell_e, i\}$. $Y = \{\ell_b, \ell_d, cond, incr, acond\}$.

Figure 3.2: LLVM program and transition system encoding example.

Property The property is

$$P = \neg\ell_{\text{err}} \,. \tag{3.14}$$

Figure 3.2 shows a complete example of an LLVM program and its transition system encoding. Pseudocode for the simple program is also shown. The program has a very simple safety property: that $i < 7$ after the loop executes. The program will execute the body of the loop twice, resulting in $i = 6$; thus the property holds.

The program is not in the MiniLLVM language. In particular, the program has a call to `__VERIFIER_error` to denote the property. MiniLLVM is an idealization to explain the encoding.

There are a couple of interesting features of this encoding. For one, the transition system's state variables are proportional to the number of phi statements in the program, not the number of SSA temporaries. For typical LLVM programs, this *drastically* reduces the state space size compared to treating each temporary as a state variable.

A second feature is that blocks in the language can be translated independently of another, simplifying the encoder implementation. For instance, in the example, constraints (3.17) and (3.20) are both encoded using (3.12), but (3.20) results in a constraint over primary inputs, since block `b` contains no state variables. This feature is also true of a Single Block Encoding [45], but ours uses larger blocks which encode multiple paths, and which may be more efficient to check in practice.

Correctness L is a cutset of the CFG because it contains the destinations of all the back edges discovered by a DFS and contains ℓ_s, ℓ_e, and ℓ_{err}. Therefore, it defines a cutset program summary as defined by Gurfinkel *et al.* [46].

3.2.2. Encoding Programs with Global Variables

The C programs we consider in our evaluation contain local and global variables of integer types. Encoding global variables is conceptually no different from encoding locals. However, a problem arises after LLVM's front-end processes C programs with globals: all global variables are of pointer type and are accessed through **load** and **store** operations. Globals in LLVM "define pointer values" [47], meaning that they cannot in general be encoded as scalar registers.[2] While stack-allocated local variables can be promoted to scalars via the `mem2reg` pass, LLVM rarely promotes a global variable to a scalar.[3] A compiler that did so would be unsound in general, since other modules could refer to the global.

[2]The reason globals are modeled as memory regions is ultimately that the C language allows one to take the address of any object.

[3]If a global is never changed, sometimes LLVM will promote it to a scalar, but that doesn't help our cause.

Since our encoding applies only to a single module and assumes that all variables are accessed as scalars, I opted to *thread the globals* through the program, which allows promoting the globals to locals. At a high level, the global variable is modeled using pointer variables local to each procedure. Then the locals can be promoted to scalars using `mem2reg`.

Not all global variables are *threadable*. Threadable globals:

- are pointers to bit-vector type (e.g., **i**32 *);
- have local linkage;
- are used only as the memory operand for **load** and **store** operations of the exact same type as the bit-vector type.

Threading the global is implemented as a whole-program LLVM pass. The `Thread-Globals` pass consists of the following steps. We detail the steps for single global variable, g. The generalization to multiple variables is straightforward.[4]

1. Allocate a local variable g_{main} inside `@main` and initialize in the same way that g is initialized (e.g., to 0). Replace every use of g in f with a use of g_{main}.
2. For each function f other than `@main`, add in-parameter in_f and return value out_f. The return value is added to the function signature by using a structure to store multiple return values, if necessary. in_f represents the current value of g on entry, out_f represents the value of g on exit, and g_f stores the value of g within f. Initially, bind g_f to in_f; on exit return out_f. Finally, replace all uses of g with g_f.
3. For each call from h to f, bind g_h to in_f before the call and bind out_f into g_h after the call.

Figure 3.3 shows an example program and its transformation with `ThreadGlobals`. The program in Figure 3.3a has a single global variable `%g` and two function calls to `f`.

[4]A peek behind the curtain: when I implemented `ThreadGlobals`, I had already prototyped the function encoding (Chapter 5). Ultimately, it was too much for one paper to assess EUFORIA with and without functions, so I omitted the encoding from our paper [48]. But in this section I describe `ThreadGlobals` in its general form, with functions.

The program after the `ThreadGlobals` pass is shown in Figure 3.3b. Step 1 introduces the variable `%g_main` (which will replace `@g`) and initializes it to 0 (line 3). For the function `@f`, step 2 introduces variables `%g_f`, `%in_g`, and `%out_g`. `%g_f` (line 21) is initialized to `%in_g` (line 22) and `%out_g` is set to the current value of `%g_f` before returning (line 26). All references to `@g` are replaced with the corresponding variables just introduced. During step 3, the caller, `@main`, binds the actual `%in_g` to the current value of `%g_main` (lines 6 and 10). It also stores the return value from `@f` into `%g_main` (lines 7 and 11).

After `ThreadGlobals`, LLVM's `mem2reg` pass promotes the introduced variables to scalars, so they are encoded as bit-vectors.

3.3. EUFORIA

EUFORIA builds on the model checker IC3 [16] 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.

3.3.1. EUF Abstraction of Transition Systems

A key advantage of our abstraction is that it is cheap to compute. It is defined by an abstraction mapping that performs a linear-time, syntax-directed, structure-preserving transformation of the concrete transition system. This section formalizes the particulars of our abstraction.

We define an *EUF abstraction mapping* $\mathcal{A}[\![\cdot]\!]$ which returns the abstraction of a given concrete sort or transition formula, i.e., a formula over current- and next-state variables, inputs, operations, and constants in `QF_BV`. The mappings below show how sorts are handled. We write uninterpreted objects that are abstractions of concrete objects—sorts, terms, functions, and predicates—in $\mathsf{sans\ serif}$ face, to distinguish them from interpreted objects. For instance, the UF for addition is ADD. The translation is defined in terms of a fresh uninterpreted sort UBV_n corresponding to the interpreted bit-vector sort BV_n.

Definition 3.1 (UBV Sort)**.** For every positive integer n, UBV_n is an uninterpreted sort.

```
@g = global i32 0, align 4
define i32 @main() {
  call void f(3)
  call void f(4)
  %t = load @g
  %t2 = add %t, 1
  store @g, %t2
  call void print(@g)
  ret void
}
define void @f(i32 %x) {
  %y = load @g
  %z = add %y, %x
  store @g, %z
  ret void
}
```

(a) Original LLVM code using a global variable.

```
define i32 @main() {
  %g_main = alloca i32, align 4
  store %g_main, 0
  ; first call to f
  %g1 = load %g_main
  %r1 = call i32 f(3, %g1)
  store %g_main, %r1
  ; start second call to f
  %g2 = load %g_main
  %r2 = call i32 f(4, %g2)
  store %g_main, %r2
  %t = load %g_main
  %t2 = add %t, 1
  store %g_main, %t2
  ; start call to print
  %g3 = load %g_main
  call void print(%g3)
  ret void
}
define i32 @f(i32 %x, i32 %in_g) {
  %g_f = alloca i32, align 4
  store %g_f, %in_g
  %y = load %g_f
  %z = add %y, %x
  store %g_f, %z
  %out_g = load %g_f
  ret %out_g
}
```

(b) LLVM code transformed with `ThreadGlobals`. The global variable `g` becomes local to `main` and modifications to it are threaded through each function invocation.

Figure 3.3: `ThreadGlobals` example. The purpose is to remove global variables from the program by promoting them to local allocations, enabling `mem2reg` to optimize the local accesses into register operations.

Just as bit-vectors have a notation indicating their sort, we define a notation for uninterpreted terms, indicating their sort.

Definition 3.2 (Sized EUF Term)**.** For every term e and positive integer n, we write $e^{\langle n\rangle}$ to indicate that the abstract term e is a term of uninterpreted sort UBV_n.

Definition 3.3 (EUF Abstraction Mapping)**.**

$$
\begin{aligned}
\mathcal{A}[\![\textsc{Bool}]\!] &= \textsc{Bool} \\
\mathcal{A}[\![\mathrm{BV}_n]\!] &= \mathsf{UBV}_n \\
\mathcal{A}[\![b]\!] &= b && \text{for Boolean } b \\
\mathcal{A}[\![x^{[n]}]\!] &= \mathsf{x}^{\langle n\rangle} && \text{for state variable } x \\
\mathcal{A}[\![c^{[n]}]\!] &= \mathsf{c}^{\langle n\rangle} \\
\mathcal{A}[\![\mathrm{ite}(\mathit{cond}, a^{[n]}, b^{[n]})]\!] &= \mathrm{ite}(\mathcal{A}[\![\mathit{cond}]\!], \mathcal{A}[\![a^{[n]}]\!], \mathcal{A}[\![b^{[n]}]\!]) \\
\mathcal{A}[\![a^{[n]} \simeq b^{[n]}]\!] &= \mathcal{A}[\![a^{[n]}]\!] \simeq \mathcal{A}[\![b^{[n]}]\!] \\
\mathcal{A}[\![\neg a]\!] &= \neg\mathcal{A}[\![a]\!] \\
\mathcal{A}[\![a \wedge b]\!] &= \mathcal{A}[\![a]\!] \wedge \mathcal{A}[\![b]\!] \\
\mathcal{A}[\![\mathit{unop}\; a^{[n]}]\!] &= \mathcal{A}[\![\mathit{unop}]\!](\mathcal{A}[\![a^{[n]}]\!]) \\
\mathcal{A}[\![a^{[n]}\; \mathit{bop}\; b^{[n]}]\!] &= \mathcal{A}[\![\mathit{bop}]\!](\mathcal{A}[\![a^{[n]}]\!], \mathcal{A}[\![b^{[n]}]\!])
\end{aligned}
$$

Boolean constants and variables b are mapped to themselves, since EUF logic supports them directly. State variables $x^{[n]}$ and bit-vector constants $c^{[n]}$ are translated to uninterpreted terms of sort UBV_n. Equalities are mapped into abstract equalities and Boolean structure is preserved.

The names *unop* and *binop* are metavariables ranging over arithmetic, relational, and bitwise operations; each is mapped to an uninterpreted variant on mapped arguments. The name of the uninterpreted function or predicate is logically irrelevant in EUF, as long as distinct operations have distinct names. We use distinct UFs for distinct argument sorts, so we subscript our names. For example, for the signed greater-than and bitwise-or operators on n bits:

$$\begin{aligned}
\mathcal{A}[\![x^{[n]} >_s y^{[n]}]\!] &= \mathsf{SGT}_n(\mathcal{A}[\![x^{[n]}]\!], \mathcal{A}[\![y^{[n]}]\!]) \\
\mathcal{A}[\![x^{[n]} \mid y^{[n]}]\!] &= \mathsf{BOR}_n(\mathcal{A}[\![x^{[n]}]\!], \mathcal{A}[\![y^{[n]}]\!])
\end{aligned}$$

The rest of the operators follow this pattern.

This abstraction preserves type safety. By ensuring that each concrete type is translated into a 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.

Definition 3.4 (EUF Concretization Mapping)**.**

$$\begin{aligned}
\mathcal{D}[\![\textsc{Bool}]\!] &= \textsc{Bool} \\
\mathcal{D}[\![\mathsf{UBV}_n]\!] &= \mathrm{BV}_n \\
\mathcal{D}[\![b]\!] &= b && \text{for Boolean } b \\
\mathcal{D}[\![\mathsf{x}^{\langle n\rangle}]\!] &= x^{[n]} \\
\mathcal{D}[\![\mathsf{c}^{\langle n\rangle}]\!] &= c^{[n]} \\
\mathcal{D}[\![\mathrm{ite}(cond, a^{\langle n\rangle}, b^{\langle n\rangle})]\!] &= \mathrm{ite}(\mathcal{D}[\![cond]\!], \mathcal{D}[\![a^{\langle n\rangle}]\!], \mathcal{D}[\![b^{\langle n\rangle}]\!]) \\
\mathcal{D}[\![x^{\langle n\rangle} \simeq y^{\langle n\rangle}]\!] &= \mathcal{D}[\![x^{\langle n\rangle}]\!] \simeq \mathcal{D}[\![y^{\langle n\rangle}]\!] \\
\mathcal{D}[\![x \wedge y]\!] &= \mathcal{D}[\![x]\!] \wedge \mathcal{D}[\![y]\!] \\
\mathcal{D}[\![\neg x]\!] &= \neg\mathcal{D}[\![x]\!] \\
\mathcal{D}[\![uunop\, a^{[n]}]\!] &= \mathcal{D}[\![uunop]\!](\mathcal{A}[\![a^{[n]}]\!]) \\
\mathcal{D}[\![a^{[n]}\ ubop\ b^{[n]}]\!] &= \mathcal{D}[\![ubop]\!](\mathcal{A}[\![a^{[n]}]\!], \mathcal{A}[\![b^{[n]}]\!])
\end{aligned}$$

Definition 3.5 (CTS)**.** A *Concrete Transition System (CTS)* is any transition system over `QF_BV` or `QF_ABV`.

Algorithm 1 (EUFORIA Entry Point).

```
1: procedure EUFORIA($I, T, P$)
2:     $\hat{I}, \hat{T}, \hat{P} \leftarrow$ AbsTrans($I, T, P$)        ▷ construct abstract transition system
3:     while true do
4:         if Check($\hat{I}$, $\hat{T}$, $\hat{P}$) then        ▷ if there is an abstract counterexample
5:             if $cx$ = BuildCx() then
6:                 return $cx$        ▷ found true counterexample
7: procedure AbsTrans($I, T, P$)
8:     $\hat{I}, \hat{T}, \hat{P} \leftarrow \mathcal{A}[\![I]\!], \mathcal{A}[\![T]\!], \mathcal{A}[\![P]\!]$
9:     $\{c_1, c_2, \ldots, c_k\} \leftarrow$ all concrete constants in $I, T, P$
10:    $\hat{T} \leftarrow \hat{T} \wedge \text{distinct}(\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_k)$
11:    return $(\hat{I}, \hat{T}, \hat{P})$
```

Figure 3.4: Entry point to EUFORIA. I, T, and P define a model checking problem. CHECK either converges or discovers an abstract counterexample, which may trigger a refinement. BUILDCX either constructs a concrete program trace from a feasible abstract counterexample or it refines the abstraction.

A CTS may be abstracted with $\mathcal{A}[\![\cdot]\!]$, resulting in an abstract transition system.

Definition 3.6 (ATS). Let $\mathcal{T} = (X, Y, I, T)$ be a concrete transition system. An *Abstract Transition System (ATS)* $(\hat{X}, \hat{Y}, \hat{I}, \hat{T})$ consists of state variables $\hat{X} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\}$, input variables $\hat{Y} = \{\mathsf{y}_1, \mathsf{y}_2, \ldots, \mathsf{y}_m\}$, initial state $\hat{I} = \mathcal{A}[\![I]\!]$, and transition relation $\hat{T}$:

$$\hat{T}(\hat{X}, \hat{Y}, \hat{X}') = \mathcal{A}[\![T(X, Y, X')]\!] \tag{3.27}$$

Because EUF abstraction is sound, if the abstract system cannot reach an unsafe state, then the concrete system will also never reach it.

3.3.2. Algorithm

EUFORIA's entry point is given in Figure 3.4. EUFORIA first abstracts the CTS into an ATS and then performs a CHECK to search for an inductive invariant for the EUF ATS. The procedure ABSTRANS computes the abstraction and adds additional constraints to force all abstract constants to be distinct from one another. We use the term

$$\text{distinct}(\mathsf{c}_1, \ldots, \mathsf{c}_k)$$

to represent this constraint. Some SMT solvers, like z3, support this construct natively.[5] Of course, one can instead use $n(n-1)$ disequalities:

$$\bigwedge_{i \in \{1,\dots,k\}} \bigwedge_{j \in \{i+1,\dots,k\}} \mathsf{c}_i \not\approx \mathsf{c}_j$$

After abstraction and checking, euforia may find an abstract counterexample.

Definition 3.7 (ACX)**.** Let $\hat{\mathcal{T}} = (\hat{X}, \hat{Y}, \hat{I}, \hat{T})$ be an ATS and $\hat{P}$ be a safety property. An n-step *Abstract Counterexample (ACX)* is an execution $\hat{A}_0, \hat{A}_1, \dots, \hat{A}_n$ in $\hat{\mathcal{T}}$ where $\hat{A}_n \vDash \neg\hat{P}$.

If an ACX is found, it is used to construct a Concretized Abstract Counterexample (CACX), which, if feasible, is returned on line 6. If the CACX is spurious, then refinement is attempted on line 5.

3.3.3. Term Projection

Most of the procedure Check (Figure 2.1) maps to the EUF case directly. The operation that needs to be tweaked is GeneralizeFeasible, the pre-image construction on line 10, Figure 2.2a.

When the Counterexample-to-Induction (CTI) query, line 9 of Figure 2.2a, is satisfiable, euforia generalizes the single pre-image 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. We call this process *CTI expansion*.

euforia's expansion procedure is given in Algorithm 2. At a high level, the expansion procedure uses the CTI model M to compute (a) a set of atomic formulas S and (b) a set of terms Q; then it uses S and Q to construct a formula (implied by M) describing many states that reach the target states. TermProj is a projection operator which is the focus of this section. TermProj is a mechanism for reducing the size of the ModelAssertion constructed on line 6.

First we'll illustrate TermProj.

[5] z3 exposes a distinct constraint to the user, but internally the constraint is lowered into disequalities.

Algorithm 2 (CTI Expansion). Precondition: $M \vDash \hat{T} \wedge \hat{s}'$.

```
1: procedure ExpandPreimage($\hat{s}', M$)
2:     $S \leftarrow \emptyset$; $Q \leftarrow \emptyset$
3:     $S, Q \leftarrow \text{TermProj}(M, \hat{T}(\hat{X}, \hat{Y}, \hat{X}') \wedge \hat{s}(X'))$
4:     $S \leftarrow \{p \in S \mid \text{Vars}(p) \cap (\hat{Y} \cup \hat{X}') = \emptyset\}$
5:     $Q \leftarrow \{t \in Q \mid \text{Vars}(t) \cap (\hat{Y} \cup \hat{X}') = \emptyset\}$
6:     $\hat{g} \leftarrow \text{ModelAssertion}(M, Q, S)$
7:     return $\hat{g}$
```

Pre-image generalization procedure used in euforia in lieu of GeneralizeFeasible from Figure 2.2a, line 10. M is the model for the CTI query. ModelAssertion **is defined in Figure 2.17.**

Example 3.1. Consider the following abstract transition relation $\hat{T}$ on variables $\hat{X} = \{\mathsf{x}_1, \mathsf{x}_2\}$:

$$\mathsf{x_1}' \simeq \hat{f}_1 \text{ where } \hat{f}_1 = \text{ite}(\mathsf{x}_1 \simeq \mathsf{x}_2, \mathsf{ADD}(\mathsf{x}_1, 1), \mathsf{SUB}(\mathsf{x}_1, 3)) \tag{3.28}$$

$$\mathsf{x_2}' \simeq \hat{f}_2 \text{ where } \hat{f}_2 = \mathsf{x}_1 \tag{3.29}$$

$$M_{3.1} = \begin{cases} \{\mathsf{x}_1, \mathsf{x}_2, \mathsf{x_2}' \mid 1, \mathsf{ADD}(\mathsf{x}_1, 1), \mathsf{x_1}' \mid 3, \mathsf{SUB}(\mathsf{x}_1, 3)\} \\ \mathsf{GT}^{M_{3.1}} = \{(\mathsf{x}_1, \mathsf{x}_2), (\mathsf{x_1}', \mathsf{x_2}')\} \end{cases}$$

Turning this model into an assertion, we get:

$$\begin{aligned} \text{ModelAssertion}(M_{3.1}) = \mathsf{GT}(\mathsf{x}_1, \mathsf{x}_2) \wedge \mathsf{GT}(\mathsf{x_1}', \mathsf{x_2}') \wedge \\ \mathsf{x}_1 \simeq \mathsf{x}_2 \wedge \mathsf{x}_1 \simeq \mathsf{x_2}' \wedge \mathsf{x}_2 \simeq \mathsf{x_2}' \wedge \\ (\mathsf{ADD}(\mathsf{x}_1, 1) \simeq 1) \wedge (1 \simeq \mathsf{x_1}') \wedge (\mathsf{ADD}(\mathsf{x}_1, 1) \simeq \mathsf{x_1}') \wedge (3 \simeq \mathsf{SUB}(\mathsf{x}_1, 3)) \wedge \\ \mathsf{x}_1 \not\simeq 1 \wedge \mathsf{x}_1 \not\simeq 3 \wedge 1 \not\simeq 3 \end{aligned} \tag{3.30}$$

Instead, euforia performs a traversal TermProj on $\hat{f}_1$ and $\hat{f}_2$ to find relevant constraints, terms, and variables; in this example, it returns:

$$\begin{aligned} \text{TermProj}(M_{3.1}, \hat{T}) = (S, Q) \quad \text{where} \\ S = \{\}, \\ Q = \{1, \mathsf{ADD}(\mathsf{x}_1, 1), \mathsf{x}_1, \mathsf{x}_2\}\ . \end{aligned} \tag{3.31}$$

Algorithm 3. M is a model. S and Q are sets of formulas and terms, respectively. Precondition: $M \vDash f_0$.

```
1: procedure TermProj($M$, $f_0$)
2:     $S \leftarrow \emptyset$; $Q \leftarrow \emptyset$
3:     $r \leftarrow$ TPRec($f_0$)
4:     return $(S \cup \{\text{Lit}(r)\}, Q)$
5: procedure TPRec($f$)
6:     switch $f$ do
7:         case $x$                                              ▷ $x$ a 0-arity term
8:             $Q \leftarrow Q \cup \{x\}$
9:             return $x$
10:        case $\mathsf{F}(t_1, t_2, \dots, t_n)$
11:            $t \leftarrow \mathsf{F}(\text{TPRec}(t_1), \text{TPRec}(t_2), \dots, \text{TPRec}(t_n))$
12:            $Q \leftarrow Q \cup \{t\}$
13:            return $t$
14:        case ite$(c, t_1, t_2)$                          ▷ only traverse satisfied branch
15:            $S \leftarrow S \cup \{\text{Lit}(\text{TPRec}(c))\}$
16:            if $M \vDash c$ then
17:                return TPRec($t_1$)
18:            else
19:                return TPRec($t_2$)
20:        case $b$                                              ▷ $b$ a Boolean variable
21:            return $b$
22:        case $t_1 \simeq t_2$
23:            return TPRec($t_1$) $\simeq$ TPRec($t_2$)
24:        case $\mathsf{P}(t_1, t_2, \dots, t_n)$
25:            return $\mathsf{P}(\text{TPRec}(t_1), \text{TPRec}(t_2), \dots, \text{TPRec}(t_n))$
26:        case $\neg f_1$
27:            return $\neg$ TPRec($f_1$)
28:        case $f_1 \wedge f_2$
29:            if $M \vDash f$ then $S \leftarrow S \cup \{\text{Lit}(\text{TPRec}(f_2))\}$; return TPRec($f_1$)
30:            else if $M \vDash \neg f_1$ then return TPRec($f_1$)
31:            else                                                  ▷ $M \vDash \neg f_2$
32:                return TPRec($f_2$)
33:        case $f_1 \vee f_2$
34:            if $M \vDash f_1$ then return TPRec($f_1$)
35:            else if $M \vDash f_2$ then return TPRec($f_2$)
36:            else                                                  ▷ $M \vDash \neg f$
37:                $S \leftarrow S \cup \{\text{Lit}(\text{TPRec}(f_2))\}$; return TPRec ($f_1$)
```

Figure 3.5: Calculates a model-based set S of constraints and a set Q of terms and variables for a formula (or term) f using model M. Let Lit(b) $= b$ **if** $M \vDash b$ **and** Lit(b) $= \neg b$ **if** $M \vDash \neg b$**. In the body of** TPRec, M **refers to the model from** TermProj.

We then compute ModelAssertion(M, S, Q) which yields our generalized pre-image cube, for which M is an implicant:

$$(\mathsf{x}_1 \simeq \mathsf{x}_2) \wedge (\mathsf{ADD}(\mathsf{x}_1, 1) \simeq 1) \wedge (\mathsf{x}_1 \not\simeq 1) \tag{3.32}$$

The cube does not include the term $\mathsf{SUB}(\mathsf{x}_1, 3)$ because only the true branch of the ite in f_1 needs to be traversed. The predicate $\mathsf{GT}(\mathsf{x}_1, \mathsf{x}_2)$ was excluded since its assignment did not matter.

The implementation of our projection operator TermProj is given in Figure 3.5. Intuitively, TermProj uses the model to prune large portions of the input formula by: (1) only traversing the true branch of ite's; (2) picking a single witness for false conjunctions and, dually, true disjunctions.

M is an implicant of the formula ExpandPreimage returns.

Theorem 3.1. *For every model M such that $M \vDash T \wedge s'$,*

$$M \vDash \textit{ExpandPreimage}(s', M) \ .$$

Proof. See Appendix Theorem 3.1. □

CTI expansion is common to many IC3-style checkers. One option for computing the exact pre-image is existential elimination. Unfortunately, EUF does not admit existential elimination. For instance, there is no equivalent x-less formula for $\exists x.\, P(x)$. Instead, an attractive candidate is the weakest precondition predicate transformer [22] wlp which computes pre-images by substitution and has been used in the context of IC3 [7], [44]. If we assume T is defined in terms next-state functions, i.e., $T = (\overline{x}' \simeq \overline{\tau}(X))$, then $\text{wlp}(R, T) = R[\overline{x} \mapsto \overline{\tau}(X)]$. For example:

$$T = [x' \simeq F(x) \wedge y' \simeq H(k, G(y, m))] \tag{3.33}$$

$$\text{wlp}(Q(x) \wedge P(y), T) = [Q(F(x)) \wedge P(H(k, G(y, m)))] \tag{3.34}$$

Unfortunately, wlp is particularly problematic for EUF, as iterated applications of it can cause EUF terms to grow arbitrarily large, leading to potential non-termination of EUF abstract reachability. CTIGAR [49] 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. EUFORIA can't reliably 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 generalization methods in the context of interpreted theories, such as for linear arithmetic [43], [50] and for polyhedra [51].

Generalizing Unsatisfiable CTI Queries If the CTI query (line 9 of Figure 9) is unsatisfiable, then state $\hat{s}$ is unreachable in i transitions. We want to generalize $\hat{s}$ by finding a set of states (a cube) $\hat{m} \supseteq \hat{s}$ that is unreachable and covers more states than $\hat{s}$, if possible. We use a simple greedy scheme for finding a minimal unsatisfiable set that is given in Algorithm 4. This algorithm is not novel; we include it to flesh out EUFORIA's description.

Algorithm 4 (Generalizing Unsatisfiable CTI Queries)**.**

```
1: procedure GeneralizeInfeasible($\langle \hat{s}, i \rangle$)
2:     $\hat{t} \leftarrow \hat{s}$
3:     $j \leftarrow i$
4:     for each literal $\hat{l}$ in $\hat{s}$ do
5:         $\hat{m} \leftarrow \hat{t} \setminus \hat{l}$                    ▷ test if $\hat{m}$ unreachable if literal $\hat{l}$ removed
6:         if $\neg$SAT($\hat{m} \wedge I$) and $\neg$SAT($\neg\hat{m} \wedge R_{j-1} \wedge \hat{T} \wedge \hat{m}'$) then
7:             $j \leftarrow$ frame $\geq j$ at which $\hat{m}$ is still unreachable
8:             $\hat{t} \leftarrow \hat{m}$                              ▷ literal $\hat{l}$ was not necessary
9:     return $\langle \hat{t}, j \rangle$
```

Generalization of definitely-unreachable cubes. EUFORIA, like PDR, examines the unsat core of the query on line 6 in order to implement line 7.

Algorithm 4 attempts to drop each literal, one at a time, from the input cube. If the resulting cube does not intersect I and is still unreachable, then it is kept for further generalization. Line 7 implicitly examines the unsat core of the CTI check to determine which frontiers contributed to unsatisfiability, potentially generalizing m.

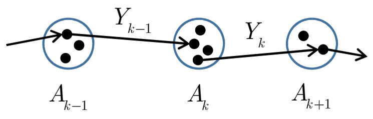
3.4. Refinement

When EUFORIA finds an ACX, it must be checked for feasibility, potentially refining the abstract state space.

Definition 3.8 (Feasibility)**.** An abstract formula $\hat{\sigma}$ is *feasible* if its concretization σ is satisfiable over `QF_BV` or `QF_ABV`.

An abstract counterexample is feasible, therefore, if its concretization is a counterexample in the CTS. Let $\hat{A}_0, \hat{A}_1, \ldots, \hat{A}_n$ be an ACX in $\hat{T}$. The ACX is spurious for at least one 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 $\hat{A}_i$; or
2. $A_{i-1} \wedge T \wedge A_i'$ 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:



The goal of EUFORIA's refinement procedure, BUILDCX (Figure 3.6a), is to produce one or more abstraction refinement lemmas. The following definition reflects all the lemmas that EUFORIA currently learns; it does not account for other possible learning mechanisms.

Definition 3.9. An *abstraction refinement lemma* is an abstract formula $\neg\hat{f}$ such that $\mathrm{SAT}(\hat{T} \wedge \hat{f})$ and $\vDash \neg f$.

In BUILDCX, f is an unsatisfiable subset of constraints whose abstraction is consistent with the current abstract transition relation; the lemma is formed by abstracting f and conjoining $\neg\hat{f}$ to $\hat{T}$.

Algorithm 5. Returns true if counterexample is feasible, false if abstraction is refined
input: counterexample $(\hat{A}_0, \hat{A}_1, \ldots, \hat{A}_n)$ in $\hat{T}$

```
1: procedure BuildCx()
2:     for $i \in \{0, 1, \ldots, n\}$ do                          ▷ check states
3:         if $\neg$SAT($A_i$) then
4:             LearnLemma(UnsatCore())
5:             return false
6:     for $i \in \{0, 1, \ldots, n\}$ do                          ▷ check transitions
7:         if $\neg$SAT($A_{i-1} \wedge T \wedge A_i'$) then
8:             LearnLemma(UnsatCore())
9:             return false
10:    return RefineForward()
```

(a) The first two stages of refinement: examining concretized states and transitions.

Algorithm 6.

```
1: procedure RefineForward()
2:     if $\neg$SAT($I \wedge A_0$) then                            ▷ check initial state
3:         LearnLemma(UnsatCore())
4:         return false
5:     $s_1 \leftarrow \langle$concrete assignment for each state variable, $\{\}\rangle$
6:     for $i \in \{2, 3, \ldots, n\}$ do
7:         if $M$ = SAT($v_{i-1} \wedge pc_{i-1} \wedge T \wedge A_i'$) then
8:             $s_i \leftarrow$ Simulate($M, s_{i-1}, T, A_i$)          ▷ $M$ is the model for the query
9:         else
10:            LearnLemma(UnsatCore())
11:            return false
12:    return true                                          ▷ feasible counterexample
```

(b) Symbolically simulate counterexample

Algorithm 7.

```
1: procedure Simulate($M, \langle v_{i-1}, pc_{i-1}\rangle, T, A_i$))
2:     $v_i \leftarrow$ empty map
3:     for $x_j \in X$ do
4:         $v_i[x_j] \leftarrow$ TPRec$_M$($f_j[X \mapsto v_{i-1}]$)         ▷ update entry for $x_j$ in $v_i$
5:     $s, \_ \leftarrow$ TermProj($M, T$)
6:     $pc_i \leftarrow pc_{i-1} \cup \{l[X \mapsto v_{i-1}] \mid l \in s\}$
7:     return $\langle v_i[Y \mapsto y_i], pc_i[Y \mapsto y_i]\rangle$
```

(c) Steps a symbolic state $s_{i-1} = \langle v_{i-1}, pc_{i-1}\rangle$ forward one step by updating value map (v_i) and path constraint (pc_i) using T.

Figure 3.6: euforia's refinement procedure, BuildCx.

Algorithm 8. Input: unsatisfiable set of constraints c.

```
1: procedure LearnLemma($c$)
2:     $\hat{c} \leftarrow$ AbstractAndNormalize($c$)          ▷ abstract and eliminate input variables
3:     if $c$ contains no inputs then
4:         if Vars($c$) $\subseteq X$ then                      ▷ only present-state vars
5:             Simplify and add lemma $\neg\hat{c}(\hat{X}')$
6:         if Vars($c$) $\subseteq X'$ then                     ▷ only next-state vars
7:             Simplify and add lemma $\neg\hat{c}(\hat{X})$
8:     Simplify and add lemma $\neg\hat{c}$
```

Figure 3.7: Learns a lemma by abstracting the concrete core c and conjoining $\hat{c}$ to $\hat{T}$

BuildCx has three stages. The first stage (lines 3–5) checks whether each $\hat{A}_i$ is feasible ($0 \leq i \leq n$). The second stage (lines 7–9) checks whether each $\hat{A}_{i-1} \wedge \hat{T} \wedge \hat{A}_i'$ is feasible ($0 < i \leq n$). These stages address reasons 1 and 2.

States and transitions are prioritized over the third stage because it is advantageous to learn constraint lemmas, since they make the abstract state space smaller. Nevertheless, euforia must learn across multiple counterexample steps in general. The third stage performs symbolic simulation on the counterexample path to address reason 3 (Figure 3.6b).

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 $\hat{T}$.

The details of symbolic simulation 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 concretized counterexample. If a contradiction is reached, then an unsatisfiable subset is found and used to learn a lemma.

Specifically, RefineForward (Figure 3.6b) 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 3.6c) computes the next state symbolically, in two steps: (1) updating variable assignments by symbolically evaluating each next-state function in T, (2) updating the path constraint with any new input constraints, and (3) uniquely renaming all input variables.

As we have said, the symbolic formula created by this process represents a single execution path through the program being analyzed, with inputs remaining 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 3.7) 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: constraint lemmas and expansion lemmas.

Definition 3.10 (Constraint and Expansion Lemmas)**.** Let $\hat{\mathcal{T}} = (\hat{X}, \hat{Y}, \hat{I}, \hat{T})$ be an abstract transition system. Two types of lemmas are learned during refinement of $\hat{\mathcal{T}}$.

1. A *constraint lemma* restricts the behavior of uninterpreted functions to make them conform more closely to the behavior of their concrete counterparts, using terms from $\hat{\mathcal{T}}$. It is an instance of the axioms for one or more bit-vector operations.

2. An *expansion lemma* introduces new terms not in $\hat{\mathcal{T}}$.

Constraint lemmas only are learned during one-step checks, lines 2–9 in Figure 3.6a. Constraint and expansion lemmas are learned during the symbolic simulation of the concrete counterexample, Figure 3.6b.

A key fact is that constraint lemmas *reduce the size* of the abstract state space. Constraint lemmas constrain the behavior of uninterpreted objects to be consistent with their concrete semantics, i.e., partially interpreting the uninterpreted operations. Expansion lemmas, on the other hand, increase the size of the abstract state space, similar to predicates added during predicate abstraction refinement. As we discuss in Section 3.5, both types of lemmas are necessary for correctness.

There are many options for performing feasibility checks and deriving suitable refinements from them if one or more of them fail (e.g., [52]–[54]). We chose this refinement procedure because our focus is on assessing the suitability of EUF abstraction for control properties, and because it's simple.

3.5. Termination & 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. Throughout, we make the assumption that the projection operator does not fail.

Theorem 3.2. *BackwardReachability terminates with an answer of* true *or* false*.*

Proof. See Appendix Section A.2. □

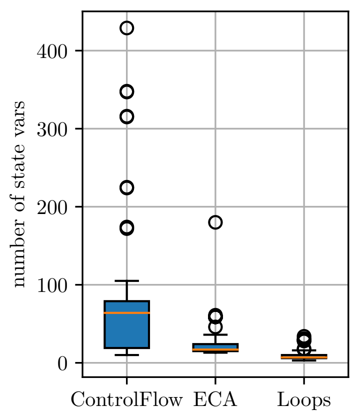
Theorem 3.3. *EUFORIA's refinement procedure increases the fidelity of the ATS, up to expressing all concrete* `QF_BV` *behavior.*

Proof. See Appendix Section A.2. □

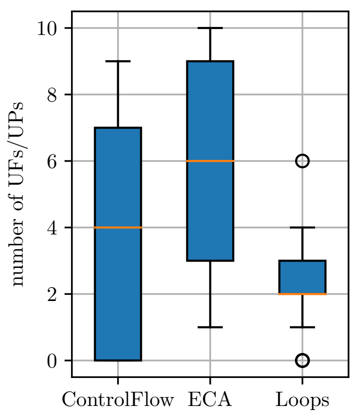
3.6. Evaluation

EUFORIA 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 [56] for EUF solving during backward reachability and Boolector 2.0 [57] for `QF_BV` solving during refinement. EUFORIA as evaluated does not support 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-seeming results if the input program is ill-defined.

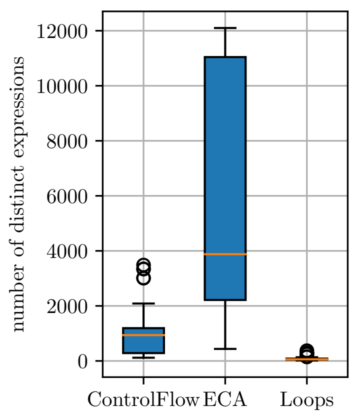
We evaluated EUFORIA on 752 benchmarks containing safety property assertions from the SV-COMP'17 competition [58]. 516 are safe and 236 are unsafe. We ran all the benchmarks on 2.6 GHz Intel Sandy Bridge (Xeon E5-2670) machines with 2



(a) State variables



(b) Uninterpreted elements



(c) Benchmark size

Figure 3.8: 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.

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.[6] Some static characteristics of these benchmarks are presented in Figure 3.8.

We evaluated EUFORIA against IC3IA [37], a CEGAR-style IC3-based checker that implements implicit predicate abstraction. It is covered in Section 2.9. We chose IC3IA largely because it is similar to EUFORIA, with one essential difference: it uses predicate abstraction instead of EUF abstraction. 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 file [59] which is readable by IC3IA.[7]

[6]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.

[7]When these experiments were conducted, EUFORIA's front-end processed LLVM bitcode directly, in-

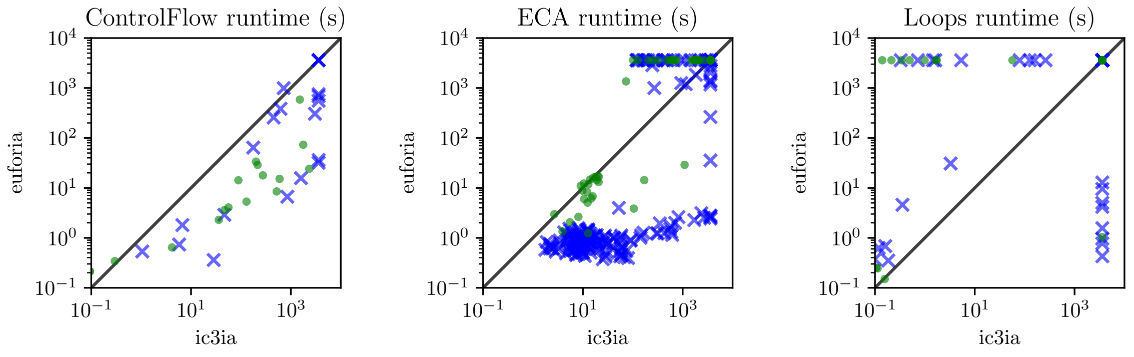


Figure 3.9: Scatter plot of runtimes broken down by benchmark set. Timeout was set to one hour. Safe benchmarks show with green dots, unsafe with blue x's.

Our evaluation sought answers to the following questions:

1. When EUFORIA performs relatively well, why?

2. When EUFORIA performs relatively poorly, why?

3. Does EUFORIA require more clauses than IC3IA to accomplish verification?

4. How does convergence depth compare?

Figure 3.9 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.

stead of VMT. As a result, EUFORIA's runtime numbers include the time it takes to encode the transition system and property. IC3ia does not need to do this. Thus EUFORIA's numbers are slightly higher than they could be (up to 30 seconds).

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 3.10 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 3.11. 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.

Discussion

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:

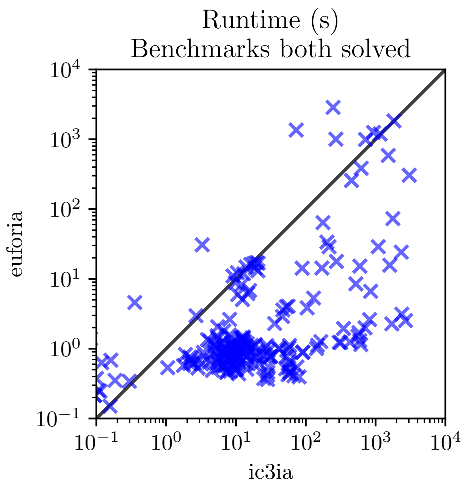


Figure 3.10: Runtime of EUFORIA and IC3IA on 249 benchmarks for which both checkers terminated within an hour. EUFORIA solves most instances more quickly than IC3IA.

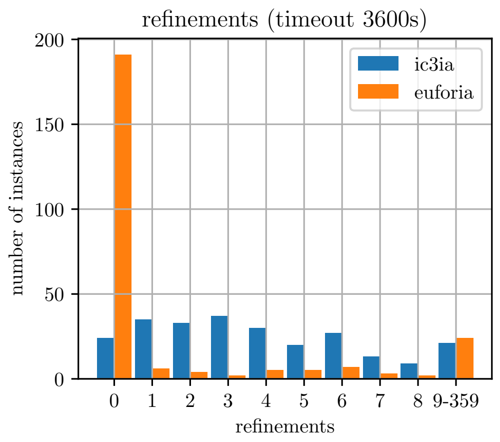


Figure 3.11: Number of instances grouped by how many refinements were required to solve them, on benchmarks both checkers finished. The key take away is that EUFORIA is able to solve many instances with very few refinements.

Example 3.2.

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

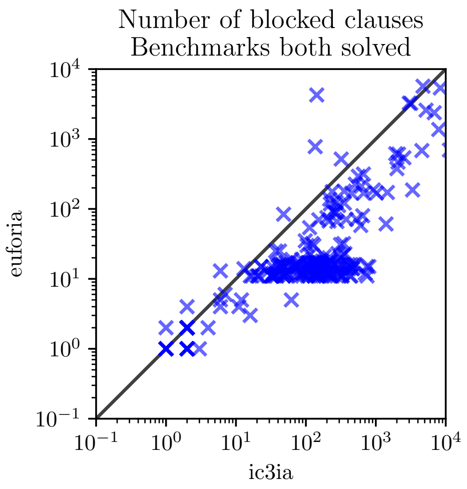


Figure 3.12: Number of blocked clauses during solving for all benchmarks solved by EUFORIA and IC3ia. Overall, EUFORIA seems to add fewer cubes.

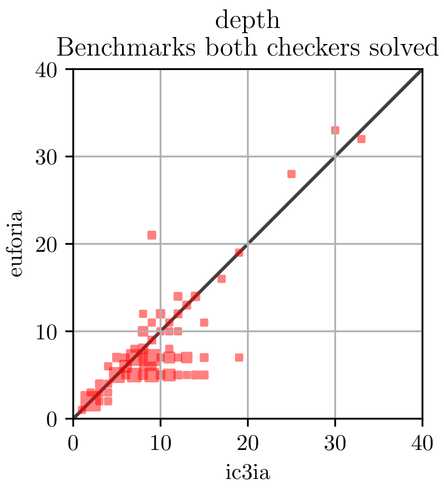


Figure 3.13: Frame depth after convergence for both EUFORIA and IC3ia. The area of the squares is proportional to the number of different benchmarks terminating at the given depths.

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 3.12 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 3.13 shows the termination depths of EUFORIA and IC3ia. Generally, the termination depths of both checkers are comparable.

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.

3.7. Related Work

Abstraction in general has been employed extensively to address verification complexity [15], [60]–[62]. Counterexample-Guided Abstraction Refinement (CEGAR) was introduced by Kurshan [63] and refined and generalized by Clarke *et al.* [15], [64] [65].

3.7.1. EUF Abstraction

The first application of EUF to verification was for equivalence checking pipelined microprocessors [2], [66], [67]. EUF is a major technique for software equivalence checking [68]–[71], compiler translation validation [3], [72], and global value numbering [73]. Recently, EUF programs as such have been investigated for decidability [74]. As a modeling tool, uninterpreted functions are often used in place of operations that are costly to reason about, like nonlinear functions [75], or where no automated reasoning is available (e.g., [76], [77]).

Applications such as these prompted an avalanche of research integrating EUF into reasoning tools. EUF plays an important role in solvers such as CLU [78], UCLID [79], and Simplify [80] (see [4] for an extensive bibliography). EUF provides a "base theory" for arrays [81] because it provides a natural over-approximation for array theory terms. EUF is also used as a "layered theory" for bit-vectors [82] for the same reason. Gulwani *et al.* investigate generic EUF support for abstract interpreters [83], [84].

Reveal [6] applied EUF to abstracting wide datapaths with uninterpreted functions. It also included a CEGAR flow for analyzing counterexamples and refining the abstraction, but it was limited to sequential circuit descriptions. Averroes [7] is the procedure that inspired EUFORIA. Building on Reveal, Averroes performs an IC3/PDR style check on an uninterpreted transition system. EUFORIA is a software model checker while Averroes is a hardware model checker. As a hardware model checker, Averroes is tightly integrated with its hardware-based frontend. EUFORIA supports a flexible input format independent of the source program. As we'll see in Chapter 4, EUFORIA supports array reasoning. Moreover, EUFORIA's abstract IC3-based search is guaranteed to terminate.

3.7.2. Predicate Abstraction

Predicate abstraction (PA) is the dominant technique in control property verification (see the Handbook, Chapter 15 for a detailed introduction [85]). Predicate abstraction allows abstracting a program with respect to a set of predicates. Given a set of predicates $p_1, \dots, p_n$, PA projects the program into a state space over valuations of these predicates. This technique was originally applied to abstract state graphs using the PVS theorem prover [38].

PA is arguably the most widespread type of automatic abstraction and has been applied to virtually every program analysis task. Well-known projects include SLAM [61], [86], BLAST [87], and IC3IA [37]. SLAM's approach is to abstract the program into a program on Boolean variables alone [88], which preserves control and abstracts data with respect to a set of predicates. SLAM checks properties like, "this program only locks unlocked locks, and unlocks locked locks." SLAM checks its Boolean program with pushdown techniques using binary decision diagrams (BDDs) [89], supports procedure summaries [90], and refines using a CEGAR scheme [91]. BLAST introduces lazy abstraction: 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 [92]). IMPACT implicitly computes the predicate abstraction, to avoid this cost [93]. YASM [94] extends predicate abstraction to liveness properties. UFO [95] combines predicate abstraction's over-approximations (e.g., SLAM) with interpolation-driven under-approximation (e.g., IMPACT) in a single framework.

EUF abstraction is nearly "free" in that it does not require any calls to a theorem prover (cf. [13], a syntactic abstraction in which a theorem prover is called numerous times). Moreover, our approach directly abstracts operations as well as predicates, in order to more effectively target control properties. One can look at our term projection as a form of predicate abstraction over EUF formulas, in which the universe of predicates is implicitly explored.

3.7.3. IC3 Extensions

Applications and extensions of the IC3 algorithm abound. To adapt IC3 to SMT-based model checking, two issues must be resolved: how to generalize SAT to SMT and how to represent the program counter.

Hoder and Bjørner [50] and Cimatti and Griggio [43] present the first SMT-based software model checkers built in IC3 style. Both generalize IC3 to the theory of linear rational arithmetic. Komuravelli *et al.* generalizes IC3 to linear integer arithmetic [36] and to arrays [34]. Bjørner and Gurfinkel [96] integrate polyhedral abstract interpretation with IC3 to compute safe convex polyhedral invariants. Karbyshev *et al.* generalize IC3 for inferring the presence or absence of quantified universal invariants [97].

Hoder *et al.* [50] generalize IC3 to apply to Horn-based pushdown analysis; their algorithm, GPDR, is capable of verifying programs with recursive functions. Software Proof-based Abstraction with CounterExample-based Refinement [35] — SPACER — connects CEGAR with proof-based abstraction. SPACER is like a chain dance between a bounded (under-approximate) safety check and an unbounded (over-approximate) inductivity check. Subsequently, RecMC (implemented in SPACER) over- and under-approximates procedure summaries in co-operation with the previous scheme. SPACER 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). EUFORIA is more vanilla: it adapts IC3 to the EUF theory in an efficient way, and does nothing in particular for recursive functions.

IC3 has been adapted to use predicate abstraction, with a couple of different refinement schemes. CTIGAR's [49] refinement is triggered by individual queries during backward reachability. IC3IA's [37] refinement is triggered whenever an abstract counterexample is found and uses interpolation to derive new predicates. Our work abstracts using EUF, which is a different mechanism from each of these, and is bit-precise in its concrete representation.

SMT-based model checkers handle the program counter either symbolically or explicitly. Explicit handling can use an abstract reachability tree, similar to lazy abstraction [43]. Alternatively, Lange *et al.* adapt IC3 to control flow automata [44] by associating with each program location its own copy of IC3's over-approximate frames. Welp and Kuehlmann use IC3 to refine loop invariants [98] as well as a hybrid approach of cooperating IC3 solvers that are each responsible for disjoint parts of the program to verify [51]. In contrast, our work only uses a symbolic representation for the program counter, only one copy of IC3's frames, and applies to entire programs.

3.7.4. Term Projection and Interpolation

Craig interpolation is frequently used in model checkers (BLAST [87], IMPACT [93], Whale [99], SPACER [35], etc.) for computing sets of states. These states are used for computing proofs of safety (among other things). (See the Handbook, Chapter 14 [100] for an overview of interpolation and model checking.) Term projection is, at first blush, another way to solve the same problem.

Definition 3.11 (Interpolant)**.** Given an inconsistent formula $A \wedge B$, an *interpolant* is a formula I such that:

1. I uses the common vocabulary of A and B (i.e., $\mathcal{V}(I) \subseteq (\mathcal{V}(A) \cap \mathcal{V}(B))$, and
2. $A \implies I$ and $I \implies \neg B$.

Example scenario: let $A = s_0 \wedge T$ and $B = s_1'$. A represents a transition from states s_0 and B represents a set of next states. Assume that $A \wedge B$ is inconsistent, indicating the transition is infeasible. The common vocabulary of A and B is the next-state variables. Thus, an interpolant for A and B represents a state formula. The predicates used in this formula can be used to refine a predicate abstraction.

Interpolants are computed for inconsistent formulas. Pre-image generalization, on the other hand, begins with a consistent formula and attempts to generalize it. Interpolation doesn't directly apply, therefore, to the problem of generalizing pre-images.

Nevertheless, interpolation can be used as an alternative to term projection in the following way ([49] is closely related). When the CTI query $R_{i-1} \wedge T \wedge t'$ (line 9 of Figure 2.2a) is satisfiable, there is a state s in R_{i-1}. The following query may be unsatisfiable:

$$s \wedge T \wedge \neg t' \tag{3.35}$$

If it is, compute an interpolant I for $A = s$ and $B = (T \wedge \neg t')$. By the properties of interpolation, I's vocabulary makes it a state formula, $A \implies I$, and

$$\begin{array}{rcl} & I & \implies \neg(T \wedge \neg t') \\ = & I & \implies \neg T \vee t' \end{array} .$$

Therefore, $I \wedge T \implies t'$ — that is, every state in I reaches t' in one transition. I is a generalized pre-image. Over EUF transition systems, the query (3.35) may be SAT (due to the non-deterministic nature of UFs), so this method may not be reliable.

3.7.5. Control Properties

Wolper provides a precise characterization of what he terms data-independent programs [11]. These reactive programs consume and produce data from a set D; the programs may make decisions based on the data, but they do not modify it. Put another way, the only operation you can perform on data is equality testing [13]. Such programs are data-independent if they do not change their behavior under transformations of the input values, except for the corresponding output values. The central result is that proofs of properties over infinite domains D be reduced to proofs of properties over finite domains. Namjoshi *et al.* define a predicate abstraction that is precise over such programs [13]. Lazić and Nowak give a semantic characterization of data-independence [12] which Benalycherif *et al.* extend to hardware property verification [101].

Our techniques generalize beyond data-independent programs. The programs we check may liberally depend on data inputs when the property does not. Our cheap abstraction allows us to ignore such dependencies.

Typestate properties [14] are a broad subset of control properties, though EUFORIA is not limited to checking typestate properties. A typestate property is expressed as a finite-state automaton which encodes legal state changes that the program may make. Typestate is frequently used to encode API protocols, which are usage rules for APIs [102].

Chapter 4.

EUFORIA with Arrays

This chapter extends the EUFORIA algorithm to support reasoning about programs with memory. Encoding real programs that use arbitrary pointer manipulations is complicated. Instead of implementing our own front-end, we translated SEAHORN's Horn-encoded output into transition systems (Section 4.2). We introduce a simple abstraction for arrays (Section 4.3). As a result, the core model checking algorithm doesn't change. The main changes are introduced during refinement. We introduce a method for abstraction refinement (Section 4.4) that significantly improves the scalability of EUFORIA, compared to the refinement from the previous chapter (Section 3.4). We prove that EUFORIA still terminates (Section 4.5). We experimentally evaluate EUFORIA and find that it solves more than 100 more SV-COMP benchmarks than SPACER, a leading model checker (Section 4.6). We conclude with related work (Section 4.7).

4.1. Introduction

Arrays and array-like structures are pervasively used for software development. From C/C++ arrays and vectors to Python lists, it is difficult to find software that doesn't use and manipulate arrays. Despite this, research of software model checkers has largely focused on finding numerical invariants and proving numerical properties of programs. As results of the software verification competition (SV-COMP) show, even when model checkers support arrays, there are a significant number of programs that cannot be automatically verified — some for a lack of expressivity and some for a lack of performance. Our focus is on the latter.

The key challenge that we face is adequately controlling theory reasoning in the

SMT solver underlying the model checker. While SMT solvers typically have an array theory and can therefore directly solve array problems, the interface that SMT solvers provide does not provide for adequate incrementality and hinting to enable maximal performance. For instance, we find that, in our SV-COMP benchmarks, as many as 90% of the array lemmas that the SMT solver is learning are either redundant or ultimately irrelevant. Most lemmas either do not advance the cause of the model checker or were thrown away by the SMT solver due to imperfect caching. Thus time spent learning those lemmas was wasted effort.

To eliminate this waste, we do incremental inductive model checking on top of an equality with uninterpreted functions (EUF) theory [48]. This removes the need for SMT array theories in the core incremental model checking process, relegating the array theory solely to abstraction refinement operations, and yielding a thousand-fold reduction in the number of operations that do redundant or irrelevant work. Additionally this means that array lemmas are only learned where they are pertinent to proving or disproving the property.

Moreover our strategy addresses a fundamental tension. On the one hand, incremental model checkers [16], which construct a safety proof bit by bit, are particularly scalable because their many individual queries are simple to solve and generalize. On the other hand, these queries lack error path information that could simplify overall checking.

For example, consider model checking the following program, assuming that `a`, `b`, and `f` are distinct constant values:

```
    int[] A; int i, a, b, f;
ℓ1: A[3] = f;
ℓ2: A[1] = a;
    A[2] = b;
    assume(1 <= i <= 3);
    if (A[i] == f);
ℓ3:   error();
    else
ℓ4:   exit();
```

The model checker attempts to find values for `i` which lead to the error at location ℓ_3. Of course it can reach ℓ_3 if $\texttt{i} = 3$, which the checker takes two SMT queries to discover. The first query corresponds to reaching ℓ_3, where $\texttt{A[i]} = \texttt{f}$, from ℓ_2. The solver deduces $i \notin \{1, 2\}$, meaning the property may yet be violated, so the checker moves on to the next query, which corresponds to reaching the failure from ℓ_1. The first query involves two array stores and one read; the SMT array theory will generate theory lemmas to deduce that `A[i]` is not set to `f` by any assignment from ℓ_2. Several of these lemmas ultimately do not matter, however, since the property is discovered to be violated by the antecedent assignment at ℓ_1.

EUFORIA's array abstraction avoids much of this redundant work. To set the stage for the abstraction, we begin by discussing how to go from programs to transition systems via Horn encodings.

4.2. Encoding Memory-Manipulating Programs

It's no trivial matter to encode memory-manipulating programs using array constraints. A standard approach is to (1) compute a set of disjoint *regions* of memory and (2) encode each such region as an array in `QF_ABV`. SeaHorn [103] does this, as do SMACK [104], CBMC [105], ESBMC [106], and CASCADE [107]. Care must be taken when computing regions to prevent them from collapsing into a single region, which would inhibit further analysis.

So, we implemented a program encoding different from the previous Section 3.2. SeaHorn outputs a Horn encoding as an SMT-LIB formula which we translate to a transition system encoding. We call this encoding Horn2VMT [108].

4.2.1. Translating Horn to Transition Systems

We begin with a motivating example: a program encoded as Horn which we will translate into a transition system.

Example 4.1 (Program with Assert).

```
function main()
 $l_E$: $i \leftarrow 0$
 $l_L$: while ($i < 5$)
           $i \leftarrow i + 3$
 $l_M$: assert $i < 7$
```

This program is safe. The while loop executes (deterministically) twice before reaching the assert with the value $i = 6$.

The program's Horn encoding is defined over interpreted symbols $\{<, +\}$ and relation symbols $\mathcal{R} = \{E, L, M, U\}$; relations model the program's control locations and state [109]. Specifically, the nullary relation E models the entry point of *main*. L is unary and models the while loop. M is also unary and models the location on exit from the loop, at the assert. U is nullary and models assertion failure.

Example 4.2 (Horn Encoding of Example 4.1)**.** The complete Horn encoding is defined as the conjunction of the following constraints:

$$\textit{true} \Rightarrow E \qquad (4.1) \qquad\qquad E \Rightarrow L(0) \qquad (4.2)$$

$$\forall x.\ L(x) \wedge (x < 5) \Rightarrow L(x+3) \qquad (4.3) \qquad \forall x.\ L(x) \wedge \neg(x < 5) \Rightarrow M(x) \qquad (4.4)$$

$$\forall x.\ M(x) \wedge \neg(x < 7) \Rightarrow U \qquad (4.5)$$

A Horn clause solver asks the question, "is the relation U non-empty?" All relations are considered empty unless proven otherwise, using the rules above. Below, when I say a relation is reachable, I mean that the relation is derivable from the rules above.

(4.1) This rule is read as, "The entry point E is always reachable."

(4.2) "If E is reachable, then L is reachable and $0 \in L$." The fact that $0 \in L$ corresponds to the assignment $i \leftarrow 0$.

(4.3) "If L is reachable and $x \in L$ for any $x < 5$, then $x + 3 \in L$." This rule models one trip through the while loop by adding elements to the relation L.

(4.4) "If L is reachable and $x \in L$ for any $x \geq 5$, M is reachable and $x \in M$."

(4.5) "If M is reachable and $x \geq 7$, U is reachable."

To use a VMT-capable model checker to answer the question, "is the relation U non-empty?" we show below how to encode Horn clauses (4.1)–(4.5) as a transition system. First we define a simple notation to preserve the value of a set of variables across a transition.

Definition 4.1 (Value Preservation)**.** For any set of variables S,

$$\pi[S] \equiv (\bigwedge_{x \in S} x' = x) \ .$$

Example 4.3 (Transition System Encoding of Example 4.2)**.** The corresponding transition system $\mathbb{A} = (X, \emptyset, I, T)$ and property P are:

$$X = \{\ell_E, \ell_L, \ell_M, \ell_U, P_{L,1}, P_{M,1}\} \tag{4.6}$$

$$I = (\neg \ell_E \wedge \neg \ell_L \wedge \neg \ell_M \wedge \neg \ell_U) \tag{4.7}$$

$$P = (\neg \ell_U) \tag{4.8}$$

and T is defined as the disjunction of the following constraints:

$$(\ell'_E \wedge \pi[X \setminus \{\ell_E\}]) \tag{4.1*}$$

$$(\ell_E \wedge \ell'_L \wedge (P'_{L,1} = 0) \wedge \pi[X \setminus \{\ell_L, P_{L,1}\}]) \tag{4.2*}$$

$$(\ell_L \wedge (P_{L,1} < 5) \wedge \ell'_L \wedge (P'_{L,1} = P_{L,1} + 3) \wedge \pi[X \setminus \{\ell_L, P_{L,1}\}]) \tag{4.3*}$$

$$(\ell_L \wedge \neg(P_{L,1} < 5) \wedge \ell'_M \wedge (P'_{M,1} = P_{L,1}) \wedge \pi[X \setminus \{\ell_M, P_{M,1}\}]) \tag{4.4*}$$

$$(\ell_M \wedge \neg(P_{M,1} < 7) \wedge \ell'_U \wedge \pi[X \setminus \{\ell_U\}]) \tag{4.5*}$$

The variables $\ell_E, \ell_L, \ell_M, \ell_U$ are Boolean *relation variables* that correspond to the relation symbols in the Horn clauses. $P_{L,1}, P_{M,1}$ are integer *place variables* that correspond to elements inhabiting Horn clause relations.

Each disjunct of T corresponds to a single Horn rule; (4.1*) corresponds to (4.1), (4.2*) to (4.2), and so on. It is possible in $\mathbb{A}$ to reach states $(\ell_L \wedge P_{L,1} = \mathbf{0})$, $(\ell_L \wedge P_{L,1} = \mathbf{3})$, and $(\ell_L \wedge P_{L,1} = \mathbf{6})$[8], meaning $\{0, 3, 6\} \subseteq L$. Moreover, every reachable state satisfies P, implying that clauses (4.1)–(4.5) are not satisfiable.

4.2.2. General Translation

We now present our general translation from set of n linear Horn clauses over $\mathcal{R}$ into a transition system $\mathbb{G} = (X, Y, I, T)$ such that a reachability query (i.e., whether a

[8]Boldface indicates the only difference among the three formulas.

relation is derivable) holds if and only if a safety property on T fails. Recall from Section 2.8 that a Horn clause has the following form:

$$\forall x_1, \dots, x_m. \underbrace{\bigwedge_{k=1}^{j} P_k(\overline{x}_k) \land \phi(x_1, \dots, x_m)}_{\text{body}} \Rightarrow \mathit{head} \tag{4.9}$$

Without loss of generality, we assume each Horn clause head is a relation occurrence and that we wish to solve a single query, a 0-ary relation U.[9]

The states of the resulting transition system are defined over a finite set of state variables $X = \{\ell_R \mid R \in \mathcal{R}\} \cup \{P_{R,i} \mid R \in \mathcal{R}, 1 \leq i \leq k \text{ where } R \text{ has arity } k\}$: Boolean relation variables ℓ_R and place variables $P_{R,i}$; and fresh primary inputs of the form $\{Y_{j,x} \mid 1 \leq j \leq n\}$, as explained below. Horn clauses are translated with the help of the syntactic mapping $\mathcal{H}[\![\cdot]\!]$ defined over quantifier-free formulas.

Definition 4.2 (Horn Translation Mapping $\mathcal{H}[\![\cdot]\!]$)**.** Let (possibly-subscripted) e, f, g, s, t be expressions:

$$\mathcal{H}[\![x]\!] = Y_{i,x} \qquad \text{quantified variable } x \text{ occurs in rule } i \tag{4.10}$$

$$\mathcal{H}[\![R(x_1, \dots, x_k)]\!] = \ell_R \land P_{R,1} \simeq \mathcal{H}[\![x_1]\!] \land \cdots \land P_{R,k} \simeq \mathcal{H}[\![x_k]\!] \tag{4.11}$$

$$\mathcal{H}[\![F(e_1, \dots, e_k)]\!] = F(\mathcal{H}[\![e_1]\!], \dots, \mathcal{H}[\![e_k]\!]) \qquad \text{for interpreted } F \tag{4.12}$$

$$\mathcal{H}[\![s \simeq t]\!] = \mathcal{H}[\![s]\!] \simeq \mathcal{H}[\![t]\!] \tag{4.13}$$

$$\mathcal{H}[\![f \land g]\!] = \mathcal{H}[\![f]\!] \land \mathcal{H}[\![g]\!] \tag{4.14}$$

$$\mathcal{H}[\![\neg f]\!] = \neg\mathcal{H}[\![f]\!] \tag{4.15}$$

During translation, T is treated as a disjunction. For every Horn clause with atom A in its body, $(\forall x_1, \dots, x_k. A \land \phi \Rightarrow \mathit{head})$, add the following disjunct to T: $\mathcal{H}[\![A \land \phi]\!] \land \mathrm{prime}(\mathcal{H}[\![\mathit{head}]\!]) \land \pi[X \setminus \mathrm{Vars}_{\mathbb{G}}(\mathit{head})]$. The initial state $I = (\bigwedge_{\ell_R} \neg\ell_R)$ and the property $P = \neg\ell_U$. By cases it can be tediously but straightforwardly shown that if a single Horn clause is satisfiable, the resulting transition system has a corresponding satisfying assignment.

[9]If we wish to solve a more complex query, for example $P(1, 4, x)$ (for $P \in \mathcal{R}$), simply modify the Horn clauses as follows: add a fresh relation symbol U to $\mathcal{R}$ and a rule $(\forall x. P(1, 4, x) \Rightarrow U)$.

Let β be a map from variables to terms. It is indexed by relation- and place-variables. Initially, $\beta(\ell) = \mathit{false}$ for every relation-variable ℓ and $\beta(P) = P$ for every place-variable P.

Algorithm 9 ("Functional" VMT Re-encoding of Horn)**.**

```
 1: procedure HORN2FVMT
 2:     $T \leftarrow \mathit{true}$
 3:     for the $i$'th Horn clause $(\forall x_1, \dots, x_m . A(\overline{x}_m) \wedge \phi(\overline{x}_m) \implies H(\overline{x}_m))$ do
 4:         $\beta(\ell_H) \leftarrow \mathrm{ite}(\mathit{act}_i, \mathit{true}, \beta(\ell_H))$
 5:         for $j \in \{1, \dots, m\}$ do
 6:             $\beta(P_{H,j}) \leftarrow \mathrm{ite}(\mathit{act}_i, \mathcal{H}[\![x_j]\!], \beta(P_{H,j}))$
 7:         $g_i \leftarrow \mathcal{H}[\![A(\overline{x}_m) \wedge \phi(\overline{x}_m)]\!]$
 8:         Conjoin to $T$: $(\mathit{act}_i \implies g_i)$
 9:     for each $x, t$ such that $\beta(x) = t$ do
10:         Conjoin to $T$: $x' \simeq t$
11:     Conjoin to $T$ one-hot constraints on $\{\mathit{act}_i\}$
```

Figure 4.1: Algorithm to compute functional VMT encoding from Horn encoding.

4.2.3. Proof of Correctness

Theorem 4.1. *The transition system has the property that the state ℓ_R is reachable in T if and only if relation R is derivable under the Horn clauses.*

Proof. See Appendix Section A.3. □

4.2.4. VMT Re-encoding Algorithm

Our translation takes linear time and uses space linear in the number of Horn clauses and the relation symbols. The number of state variables is proportional to the sum of the relation symbol arities. In addition, an n-step Horn derivation corresponds to an $O(n)$-length execution.

This encoding is simple but it does not have the *functional* characteristics we need. EUFORIA excels when the next state is a function of the current state. So, our actual implementation incorporates a few tweaks. At a high level, T is defined as a conjunction rather than a disjunction. State variables updates are defined in a single, top-level equation, similar to the encoding defined in Section 3.2.

The algorithm for producing T is given in Figure 4.1. It maintains a map β from variables to terms. Once the algorithm reaches line 9, $\beta(x) = \tau_x$ means that the state-update function for variable x is represented by $x' = \tau_x(X, Y)$.

The algorithm processes each Horn clause once. The order of processing may produce a different, equivalent result.[10] For the i'th Horn clause, it allocates an *activation variable* act_i. When act_i is true the translation g_i of the body of the corresponding Horn clause must hold in the current state, enabling the place and relation variable assignments specified on lines 4 and 6. The last line of the procedure forces the system to choose exactly one activation variable per transition. Omitting this might introduce spurious transitions in T.

Finally, the resulting transition system (X, Y, I, T) is defined over state variables X and inputs Y as defined in Section 4.2.2, and T as produced by the algorithm in Figure 4.1. The final T has the form:

$$\bigwedge_{x \in X} (x' = \tau_x(X, Y)) \wedge \bigwedge_{1 \leq i \leq n} (\mathit{act}_i \implies g_i(X, Y)) \wedge \mathrm{OneHot}(\{\mathit{act}_i\}) \tag{4.16}$$

where $\mathrm{OneHot}(S)$ constrains exactly one act_i variable to be true in every satisfying assignment.

Below we show the Horn2FVMT output corresponding to the Horn encoding above.

[10]The order of processing affects the order of ite's constructed by lines 4 and 6; the different orders produce equivalent state updates.

Example 4.4 (Horn2FVMT Output for Example 4.2)**.**

$$\begin{aligned}
\mathit{act}_1 &\implies \mathit{true} && (4.17)\\
\mathit{act}_2 &\implies \ell_E && (4.18)\\
\mathit{act}_3 &\implies \ell_L \wedge (P_{L,1} < 5) && (4.19)\\
\mathit{act}_4 &\implies \ell_L \wedge \neg(P_{L,1} < 5) && (4.20)\\
\mathit{act}_5 &\implies \ell_M \wedge \neg(P_{M,1} < 7) && (4.21)\\
\ell'_E &= \mathit{act}_1 && (4.22)\\
\ell'_L &= \mathit{act}_2 \vee \mathit{act}_3 && (4.23)\\
\ell'_M &= \mathit{act}_4 && (4.24)\\
\ell'_U &= \mathit{act}_5 && (4.25)\\
P'_{L,1} &= \mathsf{ite}(\mathit{act}_3, P_{L,1} + 3, \mathsf{ite}(\mathit{act}_2, 0, P_{L,1})) && (4.26)\\
P'_{M,1} &= \mathsf{ite}(\mathit{act}_4, P_{L,1}, P_{M,1}) && (4.27)
\end{aligned}$$

The state variables, initial state, and property are defined as in Example 4.3. The two transition systems have equivalent behavior.

4.3. Array Abstraction

To avoid the overhead of instantiating array axioms, array operations and terms may be abstracted. The operations select, store, and const-array are mapped into corresponding uninterpreted functions, select, store, and $\mathsf{const\text{-}array}$ by extending the EUF abstraction mapping $\mathcal{A}[\![\cdot]\!]$ to array terms and operations as follows:

Definition 4.3 (EUF Abstraction Mapping with Arrays—cf. Section 3.3.1)**.**

$$\begin{aligned}
\mathcal{A}[\![a : \text{Array}]\!] &= \mathsf{a} && (4.28)\\
\mathcal{A}[\![\text{select}(a, i)]\!] &= \mathsf{select}(\mathcal{A}[\![a]\!], \mathcal{A}[\![i]\!]) && (4.29)\\
\mathcal{A}[\![\text{store}(a, i, x)]\!] &= \mathsf{store}(\mathcal{A}[\![a]\!], \mathcal{A}[\![i]\!], \mathcal{A}[\![x]\!]) && (4.30)\\
\mathcal{A}[\![\text{const-array}(k)]\!] &= \mathsf{const\text{-}array}(\mathcal{A}[\![k]\!]) && (4.31)
\end{aligned}$$

The array abstraction fits neatly into EUFORIA's data abstraction approach. In fact, this abstraction approach keeps EUFORIA reasoning at the pure (quantifier-free) uninterpreted function level, for which there are efficient decision procedures.

4.4. Array Refinement

EUF reachability may find an Abstract Counterexample (ACX). Due to EUF abstraction, the Concretized Abstract Counterexample (CACX) may not be a counterexample in the CTS. Consider the following model checking problem.

Example 4.5 (Model Checking Problem Requiring Refinement). $\mathcal{E}_{4.4} = (X, Y, I, T, P)$ where

$$\mathcal{E}_{4.4} = (\{a, i\}, \emptyset, [\text{select}(a,i) \simeq 3], [a' \simeq \text{store}(a,i,3)]) \quad \text{and} \quad P = [\text{select}(a,i) \simeq 3],$$

$$\hat{\mathcal{E}}_{4.4} = (\{\mathsf{a}, \mathsf{i}\}, \emptyset, [\mathsf{select}(\mathsf{a},\mathsf{i}) \simeq 3], [\mathsf{a}' \simeq \mathsf{store}(\mathsf{a},\mathsf{i},3)]) \quad \text{and} \quad \hat{P} = [\mathsf{select}(\mathsf{a},\mathsf{i}) \simeq 3]\ .$$

The property P is its own safety invariant. Nevertheless, $\hat{P}$ does not hold in $\hat{\mathcal{E}}_{4.4}$, since EUF abstraction does not preserve the relationship between store and select. This example has the two-step CACX $(I, \text{select}(a,i) \not\simeq 3)$ which is infeasible in the `QF_ABV` theory. EUFORIA uses this contradictory CACX to *refine*, or increase the fidelity of, the array abstraction. Refinement is accomplished by conjoining formulas, called lemmas, to the abstract transition relation.

For Example 4.5, EUFORIA learns an instance of McCarthy's axiom (2.17), to eliminate the spurious behavior caused by the abstraction:

$$\mathsf{a}' \simeq \mathsf{store}(\mathsf{a},\mathsf{i},3) \implies \mathsf{select}(\mathsf{a}',\mathsf{i}) \simeq 3$$

This lemma constrains the abstract state space of $\hat{\mathcal{E}}_{4.4}$ and is therefore a constraint lemma. We will further discuss lemmas after we present our implementation of abstraction refinement.

4.4.1. Implementation of Abstraction Refinement

The previous chapter presented an implementation of abstraction refinement (Section 3.4). In the presence of array constraints, this implementation proved to be in-

Algorithm 10.

```
1:  procedure BuildBmcCx()
2:      $\mathcal{B} \leftarrow$ BmcFormula()                              ▷ phase one
3:      if $\neg$SAT($\mathcal{B}$) then
4:          RefineWithInterpolants(UnsatCore())
5:          return false
6:      return true                                      ▷ feasible counterexample
7:  procedure RefineWithInterpolants(core)                        ▷ phase two
8:      $\mathcal{B}_{HC} \leftarrow$ BuildHorn(core)
9:      $\mathcal{M} \leftarrow$ HornSolve($\mathcal{B}_{HC}$)
10:     for $i \in \{1, \dots, n\}$ do
11:         $p_i \leftarrow$ GetInterpolant($\mathcal{M}, i$)
12:         $p_{i+1} \leftarrow$ GetInterpolant($\mathcal{M}, i+1$)
13:         $l \leftarrow p_{i-1}(X) \wedge \text{body}_i(X, Y, X') \wedge \neg p_i(X')$
14:         LearnLemma($l$)
```

euforia's refinement procedure, BuildBmcCx.

efficient, solving hundreds fewer benchmarks. This section describes a different approach. Instead of symbolic simulation-based image computation, euforia constructs a single formula representing the bounded-model check query for the CACX. Further, it augments this formula with extra constraints that may make it easier to solve. If that formula is inconsistent, euforia calculates interpolants for the query and learns from those.

Instead of RefineForward (Figure 3.6b) euforia calls BuildBmcCx, presented in Algorithm 10. It performs a bounded model check (BMC) of the entire counterexample instead of symbolic simulation. If that check is inconsistent, then euforia calculates interpolants from which it derives expansion lemmas. We use a Horn clause solver (spacer) for convenience to calculate the interpolants; but the interpolants could be obtained using any interpolating theorem prover for `QF_ABV`.

BuildBmcCx has two phases.

BuildBmcCx phase one, BMC solving In phase one (lines 2–3), BmcFormula constructs the instance as below by explicitly renaming variables and using multiple copies of T:

$$\begin{aligned}\mathcal{B} = A(X_0) \wedge I(X_0) \wedge T(X_0, Y_1, X_1) \wedge \\ A(X_1) \wedge T(X_1, Y_2, X_2) \wedge \ldots \wedge \\ A(X_{n-1}) \wedge T(X_{n-1}, Y_n, X_n) \wedge A(X_n)\end{aligned}$$

$\mathcal{B}$ is then checked for feasibility. Solving BMC queries is challenging for several reasons. First, there are multiple copies of T which makes for a large formula. Second, T is monolithic — it encodes the entire program, though we expect that only part of the program is relevant for a given counterexample step. Third, even if we could reduce T at each step by removing irrelevant parts, using a large-step encoding [45] for T means that the reduced T would likely still contain a whole pile of nested Boolean logic, not all of which is necessarily relevant.

At a high level, we address these difficulties by conjoining extra constraints $\mathcal{Q}$ onto $\mathcal{B}$ that significantly prune its search space. These constraints are derived from abstract models gathered during EUFORIA's EUF reachability. We use our projection procedure, TermProj, given in Figure 3.5, to derive these extra constraints from the abstract transition relation. We now detail how we construct and solve $\mathcal{B} \wedge \mathcal{Q}$.

Let $\hat{M}_i^{i+1}$ denote the abstract model for the transition $(\hat{A}_i, \hat{A}_{i+1})$ in the abstract counterexample $(0 \leq i < n)$. Q is the concretization of an abstract formula

$$\hat{\mathcal{Q}} = \bigwedge_{0 \leq i < n} \text{TermProj}(M_i^{i+1}, \hat{T}(\hat{X}_i, \hat{Y}_{i+i}, \hat{X}_{i+1})) \; .$$

The intuition is that the constraints in $\hat{\mathcal{Q}}$ help prune irrelevant states of the counterexample, focusing the BMC query. We then preprocess $\mathcal{B} \wedge \mathcal{Q}$ by an equation solving pass that performs Gaussian elimination and variable elimination.[11] Variables assigned to constants at the top-level will be removed, possibly opening up other elimination opportunities. Linear constraints are solved, leading to further variable elimination.

We've outlined two strategies that attempt to simplify the BMC query: (1) adding extra constraints and (2) preprocessing with elimination passes. These strategies address difficulties two (T is monolithic) and three (T contains much nested logic). In

[11]The `solve-eqs` tactic in z3.

practice, their combination achieves efficiency far beyond what either does in isolation. From this point on in the discussion, $\mathcal{B}$ refers to the BMC query augmented with $\mathcal{Q}$.

Ultimately, if $\mathcal{B}$ is feasible (BuildBmcCx line 6), it is a counterexample to the property. If $\mathcal{B}$ is infeasible, BuildBmcCx enters phase two.

BuildBmcCx phase two, interpolants Phase two is implemented in RefineWithInterpolants. BuildHorn creates an inductive interpolant sequence problem from $\mathcal{B}$.

Definition 4.4 (Inductive Interpolant Sequence [110])**.** Let $F_1 \wedge F_2 \wedge \cdots \wedge F_n$ be an unsatisfiable formula. An *inductive interpolant sequence* is a sequence $I_0, I_2, \dots, I_n$ such that

1. $I_0 = \mathit{true}$ and $I_n = \mathit{false}$,
2. for all $i \in \{1, \dots, n\}$, $I_{i-1} \wedge F_i \vDash I_i$, and
3. for all $i \in \{0, \dots, n\}$, $\mathrm{Vars}(I_i) \subseteq \mathrm{Vars}(F_1, \dots, F_i) \cap \mathrm{Vars}(F_{i+1}, \dots, F_n)$.

In the context of refinement learning, $\mathcal{B}$ is the unsatisfiable formula and the variables in common for condition 3 are the state variables.

BuildHorn leverages $\mathcal{B}$'s UNSAT core to create $\mathcal{B}_{HC}$, using only the constraints from $\mathcal{B}$ that occur in the core. $\mathcal{B}_{HC}$ is a set of recursion-free Horn clauses in which fresh uninterpreted predicates P_i stand for step-wise interpolants:

$$\mathcal{B}_{HC} = \begin{cases} P_0(X_0) & \Longleftarrow \ \mathit{true} \\ P_1(X_1) & \Longleftarrow \ P_0(X_0) \wedge A^\star(X_0) \wedge I(X_0) \wedge T^\star(X_0, Y_1, X_1) \\ P_2(X_2) & \Longleftarrow \ P_1(X_1) \wedge A^\star(X_1) \wedge T^\star(X_1, Y_2, X_2) \\ \vdots & \\ P_n(X_n) & \Longleftarrow \ P_{n-1}(X_{n-1}) \wedge A^\star(X_{n-1}) \wedge T^\star(X_{n-1}, Y_n, X_n) \\ \mathit{false} & \Longleftarrow \ P_n(X_n) \end{cases} \tag{4.32}$$

where $F^\star = \bigwedge\{f \in F \mid f \in \mathrm{UnsatCore}(\mathcal{B})\}$ for $F \in \{A, T\}$.[12] These Horn clauses are

[12] $\mathcal{B}_{HC}$ could be computed without $\mathcal{B}$'s UNSAT core, but using it promotes learning concise lemmas, because it substantially reduces the complexity of the Horn clause bodies. See equation (4.33).

satisfiable by construction since $\mathcal{B}$ is infeasible. The solution to (4.32) is an inductive interpolant sequence [110].

The Horn solver produces, for each predicate P_i, a formula $p_i(X_i)$ consistent with (4.32). For each solution p_i, EUFORIA constructs a lemma from the corresponding Horn clause as follows:

$$\neg[p_{i-1}(X) \wedge body_i(X, Y, X') \wedge \neg p_i(X')] \qquad i \in \{1, \ldots, n\} \tag{4.33}$$

where $body_i$ stands for the interpreted body portion from the rule whose head is P_i.

We now return to the topic of expansion lemmas. Consider the following three-line program:

```
$x \leftarrow 3$
$x \leftarrow x + 3$
**assert** $x < 7$
```

Consider an (infeasible) 2-step counterexample $(x = 3, x \geq 7)$ and its corresponding set of Horn clauses:

$$P_0(3) \tag{4.34}$$

$$P_1(x_1) \Longleftarrow P_0(x_0) \wedge x_1 \simeq x_0 + 3 \tag{4.35}$$

$$\mathit{false} \Longleftarrow P_1(x_1) \wedge x_1 \geq 7 \tag{4.36}$$

A solution is $P_0(x) = (x \simeq 3)$ and $P_1(x) = (x \simeq 6)$ which results in the following lemmas (see (4.33)):

$$\neg[\mathsf{x} \simeq 3 \wedge \mathsf{x}' \simeq \mathsf{ADD}(\mathsf{x}, 3) \wedge \mathsf{x}' \not\simeq 6] \tag{4.37}$$

$$\neg[\mathsf{x} \simeq 6 \wedge \neg\mathsf{LT}(\mathsf{x}, 7)] \tag{4.38}$$

where ADD and LT are UF's used to abstract addition and less-than, respectively.

The key take-away here is that these lemmas introduce the new term 6 into the abstraction, which previously only contained terms from the program text, namely 3, i, 7, and the addition and less-than. These lemmas increase the granularity of the abstraction. This kind of learning is similar to learning new predicates in a predicate abstraction (e.g., [37]).

Lemmas are expansion lemmas only when the interpolants contain new terms. Using our method implies that the interpolation system itself decides whether a particular lemma is expansive or not; EUFORIA does not make this decision explicitly. EUFORIA's back-end uses SPACER to solve $\mathcal{B}_{HC}$.

Refinement is not guaranteed to succeed. The interpolation problem is constructed over the quantifier-free theory of bit-vectors and arrays. We require quantifier-free interpolants but interpolants for arrays in general are not quantifier-free [111]. Moreover, the interpolant back-end may give up.

Properties of arrays, such as sortedness, typically require quantification. Other examples include modeling variables in parameterized programs [112] and, more generally, stating properties about every element (or every index) of an array. Our restriction to quantifier-free interpolants means we will not be able to discover invariants for these properties. Nevertheless, EUFORIA can prove statements about particular array indices and values, similar to the way our bit-vector invariants learn facts about particular inputs to arithmetic operators. The programs targeted in our evaluation typically do not depend on complex array properties.

To sum up, constraint lemmas specialize UFs to particular concrete behaviors. Expansion lemmas increase the granularity of the EUF abstraction. EUFORIA learns array lemmas only if they crop up in a CACX's contradiction, ensuring that the lemmas are directly relevant to the property that is being checked. Empirically speaking, contradictions usually feature a small handful of UFs which are ultimately relevant to the property, resulting in targeted lemmas. Our process avoids most of the expense array reasoning, as we will see in the evaluation.

4.4.2. Exceptionally Lazy Learning of Array Lemmas

Fundamentally, the procedure LEARNLEMMA (Figure 3.7) learns its lemmas by negating formulas found to be unsatisfiable in `QF_ABV` and conjoining them to $\widehat{T}$. It also simplifies the formulas in order to generalize the lemmas as much as possible, specifically by eliminating input variables (line 2). We eliminate input variables from formulas by (1) collecting top-level equalities and computing their equality closure, resulting in equivalence classes of terms; and (2) substituting every input with a member of its equivalence class that doesn't contain inputs (if possible). Next, if the lemma formula is a state formula, then two versions are learned: one on current-state variables and one on next-state variables (lines 3–7).

Consequently, EUFORIA generates property-directed instantiations of array theory axioms. For instance, here is a lemma learned in one of our benchmarks:

Example 4.6.

$$A \simeq \text{const-array}(0) \implies \text{select}(A, i) \simeq 0$$

This lemma is an instance of axiom (2.20). We also find instances of McCarthy's axiom (2.17):

Example 4.7.

$$\text{select}(A', i) \simeq 0 \vee i' \not\simeq i \vee A' \not\simeq \text{store}(A, i', 0)$$

Array lemmas may also include bit-vector function symbols to learn targeted lemmas about composite behavior:

Example 4.8.

$$B \simeq \text{store}(A, i, 0) \implies \text{extract}(7, 0, \text{select}(B, i)) \simeq 0$$

Finally, some lemmas combine multiple array axioms:

Example 4.9.

$$\text{store}(B, i, 0) \not\simeq A \vee \text{store}(A, i, 0) \simeq A$$

This lemma relates stores and array extensionality. It is not a direct instance of any axiom (2.17)–(2.20), but rather a consequence of several instantiations.

Note that LEARNLEMMA is not specialized to produce array lemmas. Rather, it generalizes formulas from unsatisfiable refinement queries that themselves pinpoint which array lemma instantiations to learn. This design allows LEARNLEMMA to produce lemmas that are property-directed combinations of array theory axiom instantiations.

4.5. Termination & Correctness

Most of the proof from Section 3.5 carries over. The only new terms are array terms. In our language, array terms are restricted to be from bit-vectors to bit-vectors—

meaning an array can only store a finite number of items. Hence, the concrete state space is still finite and refinement will eventually discover every conceivable fact about the state space of a given program.

4.6. Evaluation

To evaluate EUFORIA, we rely on benchmarks from SV-COMP'17 [58], as they are widely used and relatively well understood. We evaluate on C programs from the Systems_DeviceDriversLinux64_ReachSafety benchmark set, hereafter abbreviated *DeviceDrivers*. This set contains 64-bit C programs and contains "problems that require the analysis of pointer aliases and function pointers." EUFORIA was originally designed for control properties, so our benchmark set includes benchmarks with control properties and arrays.

We consider two other model checkers, SPACER and IC3IA, which are introduced in Section 2.9. We also evaluated ELDARICA [113], a predicate-abstraction based CEGAR model checker that supports integers, algebraic data types, arrays, and bit vectors. Unfortunately, ELDARICA either threw errors, ran out of time, or ran out of memory on all of our benchmarks, so we do not consider it further.

We use SeaHorn as a front-end to encode programs into Horn clauses. SeaHorn [103] is a verification condition (VC) generator for C and C++ programs that uses LLVM in order to optimize and generate large-step, Horn clause benchmarks in SMT-LIB declare-rel format [114]. Note that we use the term benchmark to refer both to the C programs and their encoded counterparts. Since SeaHorn is not able to produce bit-vector encoded benchmarks, we modified it to produce bit-vector VCs.[13] Moreover, since EUFORIA does not yet support procedure calls, we instruct SeaHorn to inline all procedures, resulting in linear Horn clauses. We ran SeaHorn on each benchmark, limiting it to one hour of runtime and 8GB of memory. SeaHorn can fail to produce a usable benchmark due to lack of resources or because the input is trivially solved during optimization. All told, SeaHorn produced 948 *DeviceDrivers* Horn clause benchmarks out of 2703 original C programs. 687 are safe and 261 are unsafe.

SPACER natively supports Horn clauses, but EUFORIA and IC3IA take VMT files as in-

[13]We worked from SeaHorn commit id `8e51ef84360a602804fce58cc5b7019f1f17d2dc`.

put. The VMT format [59] is a syntax-compatible extension of the SMT-LIB format that specifies a syntax for labeling formulas denoting initial state, the transition relation, and property. In order to create comparable benchmarks for euforia and ic3ia, we translate the Horn clause benchmarks into VMT using Horn2VMT (Section 4.2), resulting in 948 VMT files that correspond to the 948 Horn benchmarks. The benchmarks range in size from 2^9 to more than 2^{23}, with a median size of 2^{19}; this size is the number of distinct SMT-LIB expressions used to define (I, T, P). When compressed with gzip, their sizes range from 2K to 153 MB.

All checkers run on 2.6 GHz Intel Sandy Bridge (Xeon E5-2670) machines with 2 sockets, 8 cores with 64GB RAM, running RedHat Enterprise Linux 7. Each checker run was assigned to one socket during execution and was given a 30 minute timeout. For every benchmark solved by any checker, we verified that its result was consistent with other checkers.

euforia Compared with spacer

Figure 4.2 shows a scatter plot of runtime for euforia and spacer on *DeviceDrivers* benchmarks. Overall, euforia solves 491 benchmarks and spacer solves 386. euforia times out on 33 benchmarks that spacer solves. spacer times out on 138 benchmarks that euforia solves.

When spacer solves euforia's timeouts In the 33 cases where spacer was able to solve a benchmark that euforia could not, we identified several causes:

1. spacer's preprocessor is able to solve 19 benchmarks without even invoking search. By comparison, euforia's front-end takes excessive time to parse and normalize the benchmarks. euforia parses VMT files using MathSAT5, since it the simplest API to do so. In addition to parsing, MathSAT normalizes and simplifies the resulting formula.

2. Another 12 benchmarks are quite large, and the overhead of a monolithic transition relation dominates euforia's abstract reachability. To explain: SeaHorn produces an *explicitly sliced* transition relation which spacer exploits by making

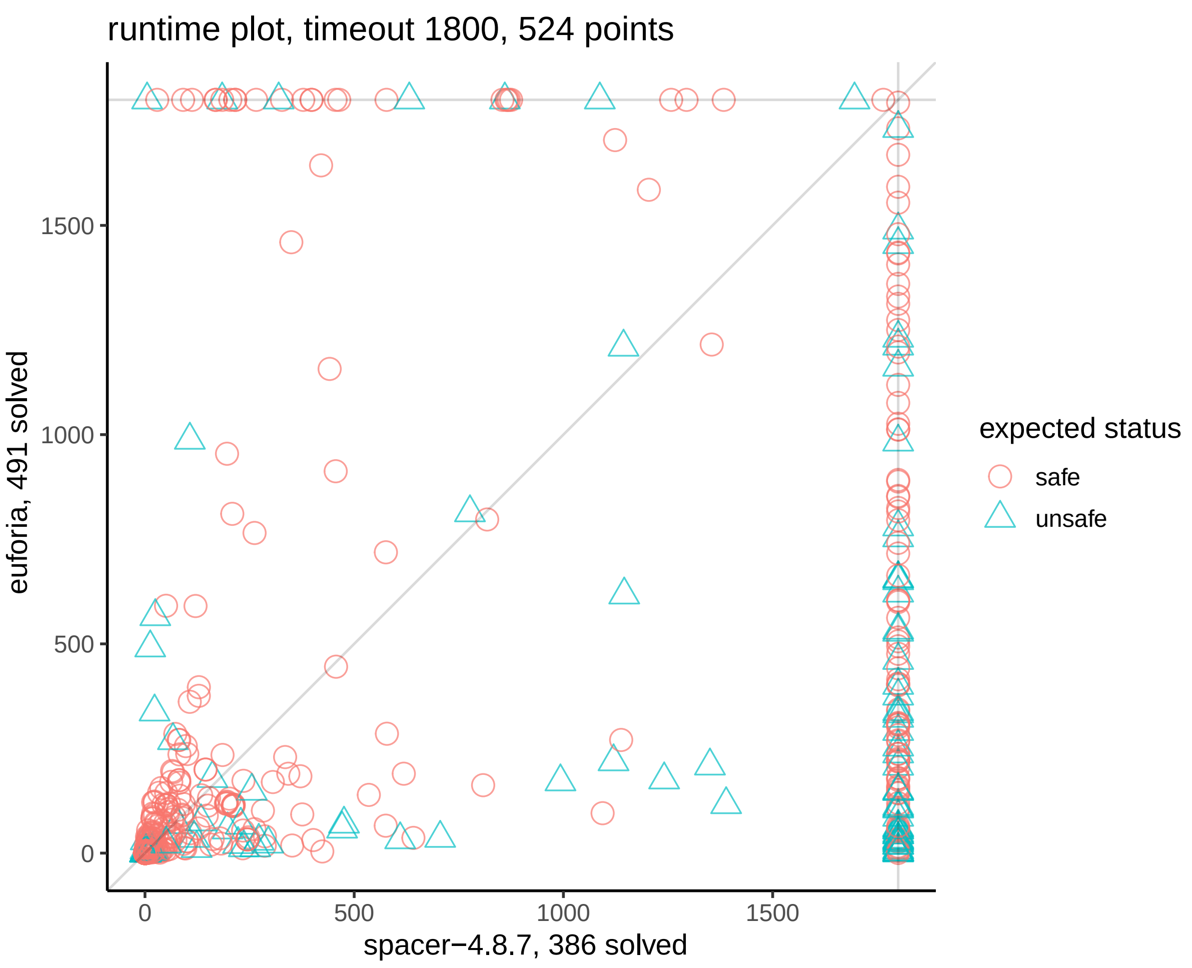


Figure 4.2: Benchmarks solved by either solver (or both). Note the points on the right hand side of this plot. Each point is a benchmark that EUFORIA solved within 30 minutes that SPACER did not solve during that time.

sliced incremental queries. EUFORIA consumes and queries a monolithic transition relation as produced by Horn2VMT.

3. In one benchmark EUFORIA gets stuck in a single interpolation query. We suspect this is because some interpolation queries generated by EUFORIA are unexpectedly difficult for SPACER.

In the last un-accounted for benchmark, there was no obvious cause. We believe that front-end improvements would address the issues identified in item 2. For instance, SPACER's preprocessor could be made independent of z3 so that it could be applied before Horn2VMT.[14] Alternatively, EUFORIA could be integrated into z3 so that it could exploit the same preprocessing as SPACER, but exploring this remains future work.

When EUFORIA solves SPACER's timeouts In the 138 cases where EUFORIA was able to solve a benchmark that SPACER did not, we examined causes. In over half of the cases, SPACER gets stuck solving concrete incremental queries. In the other 52 cases, SPACER gives up before the timeout (it returns unknown). In other words, in every case individual queries were unable to be tackled given the resources constraints. Therefore we emphasize that, in contrast, EUFORIA has the strong benefit of making individual queries predictably fast.

We wondered: is EUFORIA only winning because it hardly needs to do refinement? The answer is no. Figure 4.3 shows the same scatter plot as Figure 4.2 but restricted to EUFORIA-solved benchmarks that required at least one abstraction refinement. It shows that EUFORIA requires refinement for many of the benchmarks for which SPACER times out.

EUFORIA Compared with IC3IA

Figure 4.4 shows a scatter plot of our results compared with IC3IA. IC3IA solves 128 benchmarks total. Excepting three of these, EUFORIA solves *all* the benchmarks that IC3IA solves, usually in orders of magnitude less time. Our results are significant because IC3IA and EUFORIA are quite similar: both implement a PDR-style [33] algorithm,

[14]We tried dumping the benchmark after SPACER's preprocessing step, but the benchmark was no longer guaranteed to be Horn, so it was not a valid input for encoding to VMT with Horn2VMT.

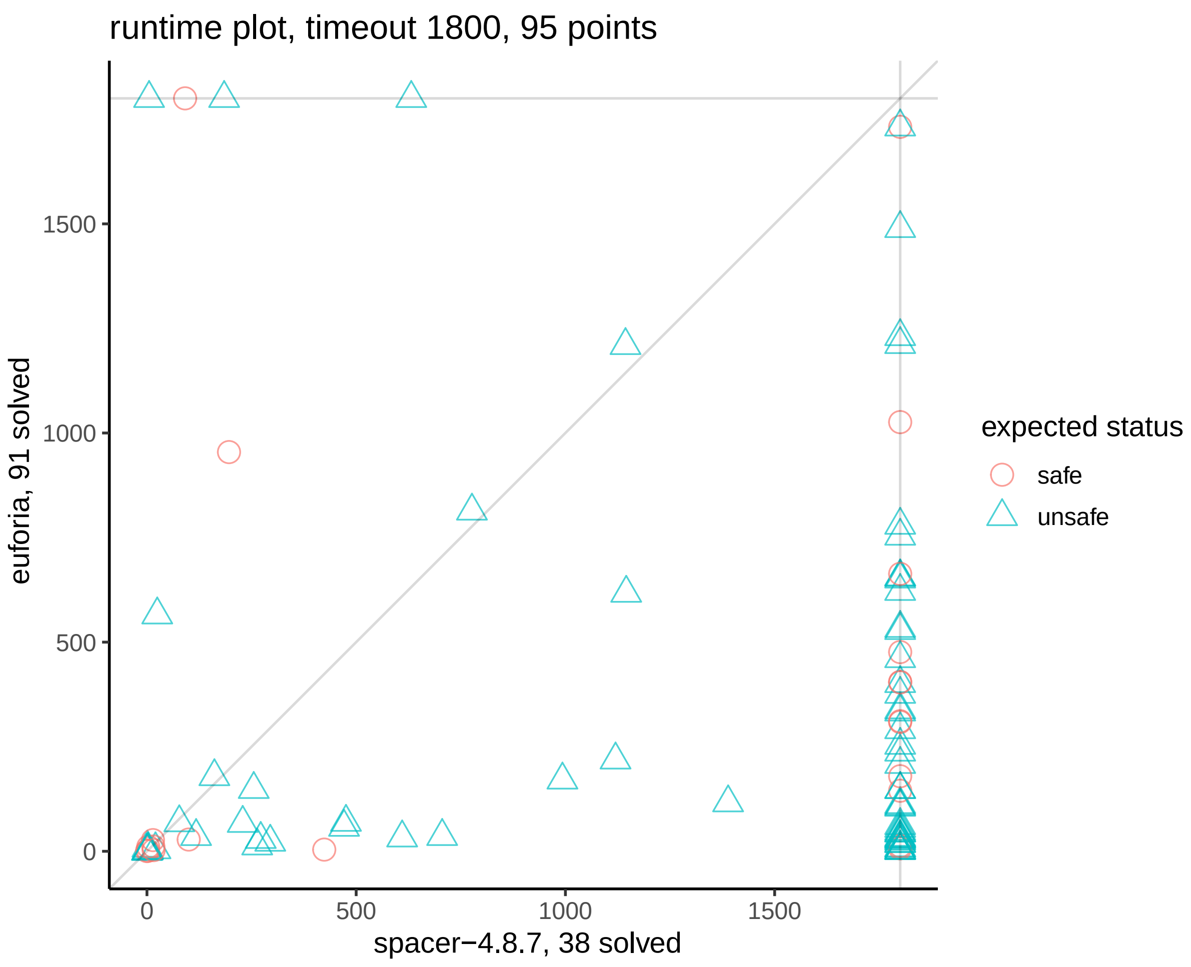


Figure 4.3: EUFORIA VS SPACER restricted to those benchmarks that require *at least one* abstraction refinement.

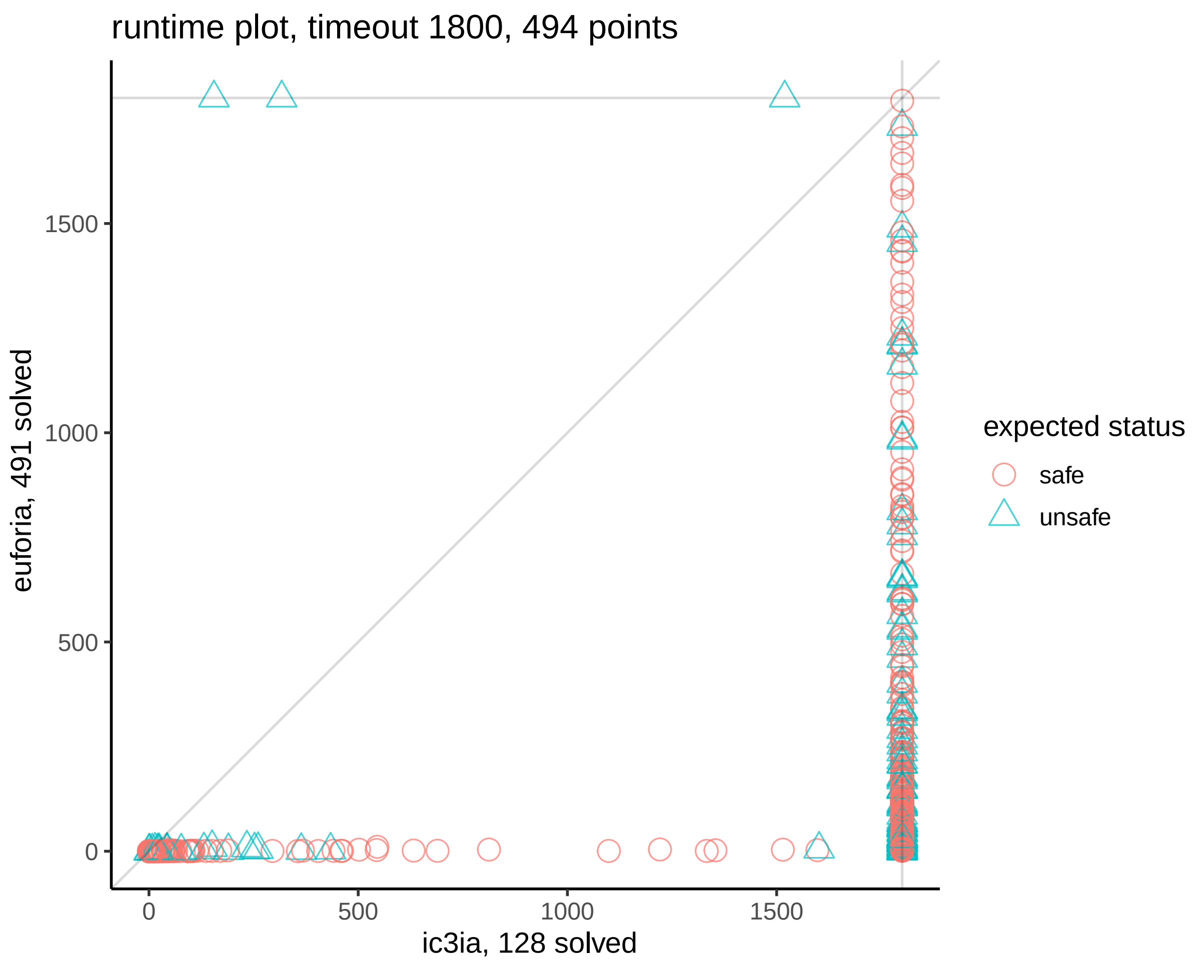


Figure 4.4: Scatter plot of euforia vs ic3ia.

both operate on *exactly* the same VMT instance encoding, and both are written it C++. They differ in two respects: (1) ic3ia uses (implicit) predicate abstraction and euforia uses EUF abstraction; (2) ic3ia's SMT solver backend is MathSAT5 and euforia's is z3.

On the benchmarks where euforia times out, two benchmarks get stuck after several seconds in an interpolant query; the other learns a pile of lemmas but doesn't converge in time.

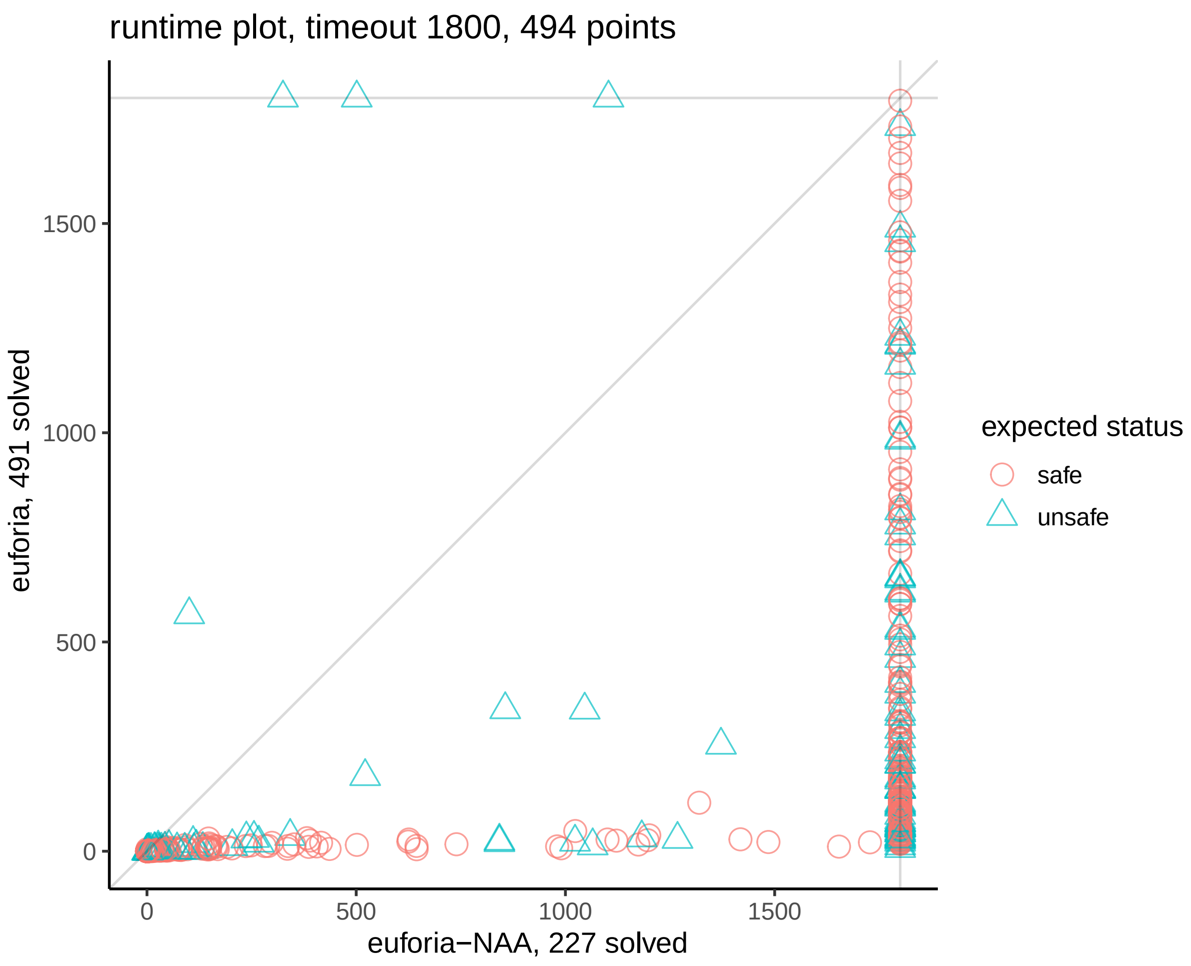


Figure 4.5: EUFORIA$_{\text{NAA}}$ **(no array abstraction) (x axis) compared with** EUFORIA **(y axis).**

EUFORIA and Array Abstraction

For solvers that use lazy theory lemma learning or a trigger-based saturation method [115], array lemmas will be learned in response to property-directed queries. Does EUFORIA's array abstraction really provide a benefit over such an approach?

To address this question, we modified EUFORIA to compute a hybrid abstraction using the theory of EUF and arrays. It abstracts bit-vector operations into UFs (as before), but uses array theory operations for arrays. Call this configuration EUFORIA$_{\text{NAA}}$, for No Array Abstraction.

As demonstrated in Figure 4.5, EUFORIA$_{\text{NAA}}$ is significantly slower almost everywhere and strictly slower in all cases but four. One important difference between

EUFORIA and EUFORIA$_{\text{NAA}}$ is an enormous disparity in array theory lemmas learned by the underlying SMT solver. Between configurations, the difference of the number of array theory lemma instantiations is almost two orders of magnitude (1.9), on 95% of the benchmarks; almost four orders of magnitude (3.8), on 50% of the benchmarks; and more than seven orders of magnitude (7.2), on 5%. To calculate this result, we measure the number of array theory axiom instantiations in the underlying SMT solver (z3). Then, for each benchmark, we took the difference of the logs (base 10) between the two configurations; this quantity is proportional to the order of magnitude difference between the numbers.

We conclude that EUFORIA$_{\text{NAA}}$ spends a lot of time reasoning about arrays despite the fact that EUFORIA required relatively little array reasoning to solve the same benchmarks. Moreover, compared to SPACER's 386 solves, EUFORIA$_{\text{NAA}}$ solves only 227 instances, which (1) shows that array abstraction is critical to performance and (2) gives some additional evidence that SPACER's array projection helps its runtime.

EUFORIA in Itself—the Role of Lemmas

This section discusses EUFORIA's learned lemmas as detailed in Section 3.4. Lemmas in general play a relatively minor role; they're only required in 19% of benchmarks that EUFORIA solved (91). Moreover, only 22 benchmarks required interpolants. Figure 4.6 shows the count of total lemmas learned, broken down by whether EUFORIA learned array lemmas or non-array lemmas. First, we can see that there is a trend that EUFORIA learns fewer array lemmas than data lemmas. Second, all but two benchmarks required fewer than 100 lemmas. These results suggest our benchmarks only depend sparingly on the behavior of memory manipulations, and confirm the suitability of EUFORIA's abstraction. SPACER solves 34 of these benchmarks; out of 34, 14 benchmarks require array lemmas and 20 do not.

4.7. Related Work

The relationship between EUF and the theory of arrays has been long recognized [111], [116] and analyzed [117] and exploited in decision procedures [112] and in the imple-

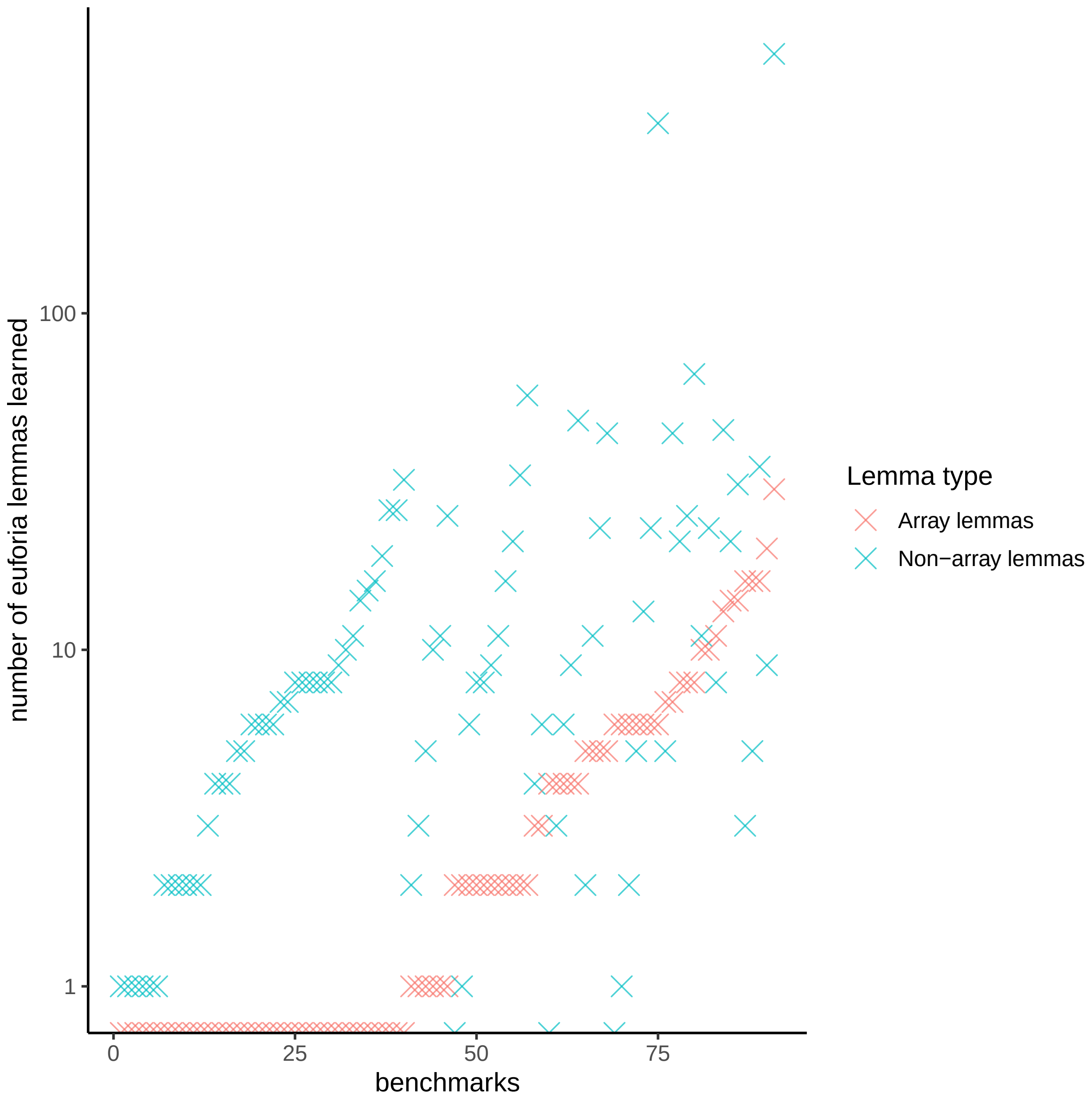


Figure 4.6: Breakdown of lemmas as array-related and non array-related on the subset of benchmarks (91) for which any lemma learning was required (y axis is log scale).

mentation of several SMT solvers, including Yices [118] and z3 [56]. Array terms are compiled into EUF or a ground theory to instantiate the needed array axioms. Our approach lifts EUF outside the SMT solver, to the model checking level, and refines it on demand.

Komuravelli *et al.* introduce a model-based projection for pre-images in order to rewrite array operators into terms in a scalar theory [34]; this algorithm is implemented in SPACER [36] used in our evaluation. Predicate abstraction applies to programs with arrays directly [37], with the limitation that quantifier-free interpolants do not exist in general for the theory of arrays [111]. We inherit that limitation, but contribute a different, inexpensive way to place array constraints in pre-images and refine them lazily.

Broadly, SMT solvers solve constraints over arrays in three ways (sometimes combined): (1) by rewriting selects and stores into a finite number of terms and axiom instantiations in a ground theory, possibly combined with EUF [112], [115]–[117], [119]–[123]; (2) by abstraction-refinement procedures over the array constraints [124], [125]; (3) by rewriting into (non-abstract) representations which are solved with specialized algorithms [81], [126], [127]. The issue addressed by our paper is applicable to each of these: we use an abstraction that inexpensively supports (limited) array reasoning and we only invoke an SMT array solver at the last possible moment.

4.7.1. Extensions To Other Theories

The EUF abstraction described so far has been applied to the quantifier-free theory of bit vectors and arrays. What about other theories?

It is likely that EUF abstraction over-approximates any other SMT-LIB theories.[15] So long as interpreted function symbols behave like functions (i.e., obey functional consistency), UFs over-approximate them. This includes the theories of Integers, Reals, Strings, and Floating Point. What is unclear is the effect of EUF abstraction on refinement in these theories. Synthesizing refinement lemmas may be more or less difficult, depending on the theory.

[15] `http://smtlib.cs.uiowa.edu/theories.shtml`

Chapter 5.

EUFORIA with Functions

This chapter extends EUFORIA to support checking programs with functions. We introduce a novel encoding for programs with non-recursive functions (Section 5.2). This encoding enables EUFORIA as already described to reason about programs with functions. We formalize this encoding (Section 5.3) and detail its implementation (Section 5.4). Finally, we report some experiments comparing this encoding to a fully-inlined encoding (Section 5.5).

5.1. Introduction

Typical interprocedural program analyses must be function-aware. Support for functions is baked into the analysis algorithm. Broadly, analyses either approximate the call stack (via call strings) or approximate the mapping of inputs to outputs (via function summaries) [128]. GPDR introduced a summary-based approach for IC3 [50], which SPACER builds upon [36].

These approaches share an important drawback: one has to modify the model checker. To retrofit an intraprocedural checker to analyze programs with functions, one usually does one of two things:

1. Full function inlining. When a function call is inlined, the body of the function is copied to the call site. The function's local variables are renamed to avoid conflicts with the local variables at the call site. If all call sites of the function are inlined, the function itself can be removed from the analysis.

Inlining may lead to exponential blowup in the size of the program. Recursive functions, moreover, can't be inlined in general.

2. Abstraction. A function call $x = f(y)$ can be replaced by an assignment $x = \star$. If the function is irrelevant to the property, then the analysis will return a correct answer. But if it is relevant, then the analysis may result in a false positive: it may claim the program has a bug due to a value stored in x, when in fact $f(x)$ never returns any such value.

Instead, we propose an encoding that is concise and precise for non-recursive programs. It is concise because it does not clone functions, as required by full inlining, so its size is linear in the input program's size. It is precise because it preserves the exact program semantics. Moreover, the encoding makes it possible to use existing intraprocedural analyses transparently.

We implemented this encoding as SeaHornVMT, a new SeaHorn backend [103]. SeaHorn is an extensible, LLVM-based verification framework. We show experimentally that euforia, an ic3-based model checker, is able to discover reusable facts about functions. In effect, we use ic3 to automatically derive function summaries by exploiting our encoding. Moreover, our encoding does not restrict the abstraction used by the model checker.

Unfortunately, our encoding is limited to non-recursive programs. Nevertheless, we believe such an encoding is useful. The stack size for C programs is determined by the operating system, so C programs usually don't heavily rely on recursive functions. Often, recursive functions are part of the "data" of programs, rather than "control." Typical examples used in motivating recursive function handling involve numerical properties [36], [50], [99], [110]. Other recursive functions involve linked data structures (lists, trees, graphs). We are not aiming to prove the correctness of algorithms manipulating such data structures, such as tree rebalancing, graph reachability, etc. Our focus is on control verification.

5.2. Function Encoding

We introduce a novel encoding in order to support programs with functions. The core insight behind the encoding is that without recursion, at runtime the program will never repeat a call to a function already on the stack. Therefore, the possible stack configurations can be represented as a graph rooted at the program entry function. A call graph is a graph in which each vertex represents a function and there is an edge from vertex f to vertex g iff f calls g.

Since there is no recursion, all possible call sequences are representable as a finite number of finite paths through the call graph. All call sites are finitely enumerable as well[16]. Instead of pushing a stack frame for a call, we can use a normal branch instruction and "activate" the corresponding (site, edge)-pair; instead of popping the stack on return, we again use a normal branch and "deactivate" the corresponding (site, edge)-pair. Assuming that all edges are initially inactive, we will never activate an active edge (or deactivate a inactive one) because there are no recursive functions.

One advantage of our encoding is that indirect function calls are easily supported: we simply use an ite to test which function is being invoked and activate the corresponding edge.

5.2.1. Example

The encoding introduces pseudo-control *wait* and *return* states that model control inside a function (wait) and returning from the function (return). We illustrate with an example.

Consider a function f that takes a single integer argument and returns an integer value. A skeleton for f is given below on the left, along with code that calls f on the right.

```
function int $f$(int $u$)              function main()
  $l_s$:                                  $l_i$: $v \leftarrow f(4)$
      ... use $u$ and set $z$ ...              ...
  $l_e$: return $z$                          $l_j$: $q \leftarrow f(v)$
```

l_s and l_e denote the function's start and exit locations, and l_e denotes the unique location in the function's body corresponding to a return statement. In addition to any declared local variables, the function's state variables include its formal argument u as well as a special variable *rv* that stores its return value. There are two calls to this

[16]Assuming the source file isn't infinite.

function at locations l_i and l_j. The constraints needed to correctly encode these calls consist of:

- Constraints to transition to the callee's entry. Control enters f from either call, so the state updates for the entry location to f are:

$$l'_s = l_i \vee l_j \tag{5.1}$$

- Constraints that set the function's formal parameter u for the call:

$$u' \simeq \text{ite}(l_i, 4, \text{ite}(l_j, v, u)) \tag{5.2}$$

- Constraints that update the function's return value *rv* once function exit is reached:

$$rv' \simeq \text{ite}(l'_e, z, rv) \tag{5.3}$$

- Constraints that copy the return value to caller-variables v and q. To handle this correctly, control must first be transferred to f and the calling function(s) must wait for f to finish before updating v and q. Let w_i, r_i and w_j, r_j denote pseudo control states that correspond to waiting at and returning to locations l_i and l_j, respectively. The updates to these four pseudo states are:

$$w'_i \simeq l_i \vee (w_i \wedge \neg l_e) \qquad w'_j \simeq l_j \vee \left(w_j \wedge \neg l_e\right) \tag{5.4}$$

$$r'_i \simeq w_i \wedge l_e \qquad r'_j \simeq w_j \wedge l_e \tag{5.5}$$

  In these equations, once the formal arguments of f are updated by the actual arguments, the caller enters a call-site-specific waiting state $w = \textit{true}$ and remains in that state as long as f has not reached its end, indicated by l_e being false. When f exits, l_e becomes true and control is transferred to the caller, indicating a return $r_i = \textit{true}$ or $r_j = \textit{true}$. Updates to variables v and q can now be made:

$$v' \simeq \text{ite}(r'_i, rv, v) \qquad q' \simeq \text{ite}(r'_j, rv, q) \tag{5.6}$$

This encoding supports nesting of function calls, but not recursion. Note that wait states may be simultaneously true, even though multiple locations are never simultaneously true. Nested function calls may lead to several w's being true at the same time (each one represents a stack frame).

5.3. Formal Presentation

We extend the MiniLLVM language from Figure 3.1 to include functions and function calls. We call this language FunLLVM and it is shown in Figure 5.1. Instead of a single `@main`, FunLLVM supports defining arbitrary top-level functions with the **define** directive. These are called with the **call** command. Functions may not be passed as parameters.

In order to model multiple-parameter functions, this language supports fixed-size vectors of values with the type **v**sz. Therefore, the values domain is over n-tuples of integers. Frames now include the current function identifier and program states include a stack of function calls. A module is now a list of function definitions. For this presentation we assume functions have no side effects. In practice, side effects can be handled by introducing updated versions of side-effected parameters and threading them through the encoding.

The semantics is defined as a judgment *mod* $\vdash S \longrightarrow S'$ meaning that in the module *mod*, executing program state S results in new state S'. A program state $S = (M, \overline{\sigma})$ is composed of a memory state M and list of stack frames $\overline{\sigma}$. FunLLVM is call-by-value and follows typical call & return semantics. A call evaluates its parameter in the caller's context, transfers control to the callee, and pairs the evaluated parameter with the callee's argument and in the callee's context binds them (rule Call). A return instruction inspects the highest stack frame to find the caller's location and binds the caller's variable to the return value of the callee, resuming execution after call instruction in the caller (rule Return).

We assume that every function returns a value; this simplifies the presentation but generalizing to functions returning **void** is straightforward. Our encoding captures functions that do not terminate; they will correctly never return to the caller. We also assume each function f has unique entry block b_s and exit block b_e. b_e contains only a **return** instruction. No other blocks contain returns. Further, we assume that

FunLLVM syntax:

Category	Meta-variable		Productions
Types	typ	::=	**isz** ∣ **vsz** ∣ **void** ∣ $typ*$ ∣ id
Constants	$cnst$	::=	**isz** Int ∣ $(typ*)$**null**
Values	val	::=	id ∣ $cnst$ ∣ **vsz** $[\overline{val}]$
Binops	bop	::=	**add** ∣ **mul** ∣ **sdiv** ∣ **load** ∣ **store** ∣ $\cdots$
Right-hand-sides	rhs	::=	$val_1\ bop\ val_2$
Arguments	arg	::=	$typ\ id$
Parameters	$param$	::=	$typ\ val$
Commands	c	::=	$id :=$ **call** $typ_0\ id_0(param)$
		∣	$id := rhs$
Terminators	tmn	::=	**br** $val\ l_1\ l_2$ ∣ **return** $typ\ val$
Phi Nodes	ϕ	::=	$id =$ **phi** $typ\ [val_1, l_1], [val_2, l_2]$
Instructions	$insn$	::=	ϕ ∣ c ∣ tmn
Non-ϕs	ψ	::=	c ∣ tmn
Blocks	b	::=	$l\ \overline{\phi}\ \overline{c}\ tmn$
Functions	f	::=	**define** $typ\ id(\overline{arg})\{b_s, \overline{b}, b_e\}$
Module	mod	::=	**module** $\overline{f}$

FunLLVM semantic domains:

Values	v	::=	Int^n	Environment	δ	::=	$id \mapsto v$
Frames	σ	::=	$(fid, l, insn, tmn, \delta)$	Prog Counters	pc	::=	$l.i$ ∣ $l.\mathbf{t}$
Prog States	S	::=	$M, \overline{\sigma}$				

FunLLVM operational semantics:

Call

$$\frac{c = r := \textbf{call}\ typ_1\ id_1(typ\ val) \qquad f_1 = \textbf{define}\ typ_1\ id_1(typ\ p)\{b_s, \overline{b}, b_e\} \\ \langle\!\langle val \rangle\!\rangle_\delta = v \qquad \sigma_0 = (f_0, l_0, (c, \overline{c_0}), tmn_0, \delta) \qquad b_s = l_1\ \overline{c_1}\ tmn_1}{mod \vdash M, (\sigma_0, \overline{\sigma}) \longrightarrow M, ((id_1, l_1, \overline{c_1}, tmn_1, \{\}[p \mapsto v]), \sigma_0, \overline{\sigma})}$$

Return

$$\frac{c = r := \textbf{call}\ typ_1\ id_1(typ\ val) \qquad \langle\!\langle val_1 \rangle\!\rangle_\delta = v_1 \qquad \sigma_0 = (id_0, l_0, (c, \overline{c_0}), tmn_0, \delta_0)}{mod \vdash M, ((id_1, l_1, [], \textbf{return}\ val_1, \delta_1), \sigma_0, \overline{\sigma}) \longrightarrow M, ((id_0, l_0, \overline{c_0}, tmn_0, \delta_0[r \mapsto v_1]), \overline{\sigma})}$$

Figure 5.1: FunLLVM syntax and semantics, defined as an extension of MiniLLVM (Section 3.2, Figure 3.1), with new elements highlighted.

if a function call occurs in a basic block, it is the first instruction and is immediately followed by an unconditional branch instruction. In other words, calls are isolated in blocks. We describe the encoding under these assumptions. Later, we will explain how to satisfy them. To simplify our presentation, we use an auxiliary meta-level encoding function *nite*. It creates a nested if-then-else tree from two (equal-length) lists of conditions and values.

Definition 5.1 (Nested If-then-else)**.**

$$\mathit{nite}(\overline{c}, \overline{v}, v) = \mathrm{ite}(c_1, v_1, \mathrm{ite}(c_2, v_2, \dots \mathrm{ite}(c_n, v_n, v)))) \quad \text{where} \quad \begin{aligned} \overline{c} &= (c_1, c_2, \dots, c_n) \\ \overline{v} &= (v_1, v_2, \dots, v_n) \ . \end{aligned}$$

The remainder of this section defines a transition system encoding (X, Y, I, T) for a program containing a function f, defined generically as:

$$\textbf{define}\ \mathit{typ}\ f(\mathit{arg})\{b_s, \overline{b}, b_e\} \text{ where } b_e = \textbf{return}\ \mathit{val}\ .$$

There are n calls to f in the program; let l_j ($j \in \{1, \dots, n\}$) denote the block location of the call numbered j. Let p_j denote the actual parameter value for the j'th call to f. Let v_j denote the caller-return variable for the j'th call to f. So, the j'th call looks like this:

$$l_j\colon\ v_j = \textbf{call}\ f(p_j)$$

We extend the encoder $\mu[\![\cdot]\!]$ from Section 3.2.1. The encoding state space for a FunLLVM program contains the same state variables as for a MiniLLVM program, along with some extras. Additionally, X contains, for each function f as above:

1. an argument-variable $\mu[\![arg]\!]$ representing the input to f;

2. a return-variable *rv* of sort $\mu[\![\mathit{typ}]\!]$; and

3. wait w_j and return r_j variables for each call site.

Figure 5.2 shows the constraints to encode non-recursive functions. (5.7) shows how arguments are set depending on call site. Upon entering a function, the system enters wait state that indicates it is waiting to return from f. In (5.8), a call binds the

$$\mu[\![arg]\!]' \simeq nite(\bar{\ell}, \overline{\mu[\![p]\!]}, \mu[\![arg]\!]) \quad (5.7)$$

Update function arguments to actuals.

$$\begin{aligned} w_1' &\simeq (\ell_s' \wedge \ell_1) \vee (w_1 \wedge \neg \ell_e) \\ w_2' &\simeq (\ell_s' \wedge \ell_2) \vee (w_2 \wedge \neg \ell_e) \\ &\cdots \\ w_j' &\simeq (\ell_s' \wedge \ell_j) \vee (w_j \wedge \neg \ell_e) \end{aligned} \quad (5.8)$$

Update stack by recording which callee to wait for.

$$rv' \simeq \text{ite}(\ell_e', \mu[\![val]\!], rv) \quad (5.9)$$

Update return value at end of called function.

$$\begin{aligned} \mu[\![v_1]\!]' &\simeq \text{ite}(r_1', rv, \mu[\![v_1]\!]) \\ \mu[\![v_2]\!]' &\simeq \text{ite}(r_2', rv, \mu[\![v_2]\!]) \\ &\cdots \\ \mu[\![v_j]\!]' &\simeq \text{ite}(r_j', rv, \mu[\![v_j]\!]) \end{aligned} \quad (5.10)$$

Update caller variable with callee return value.

(a) Data and pseudo control updates.

$$\ell_s' \simeq (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_j) \quad (5.11)$$

Transfer control to callee entry from some call site.

$$\begin{aligned} r_1' &\simeq \ell_e \wedge w_1 \\ r_2' &\simeq \ell_e \wedge w_2 \\ &\cdots \\ r_j' &\simeq \ell_e \wedge w_j \end{aligned} \quad (5.12)$$

Return from callee to appropriate caller.

(b) Control updates.

Figure 5.2: Encoding of non-recursive function calls.

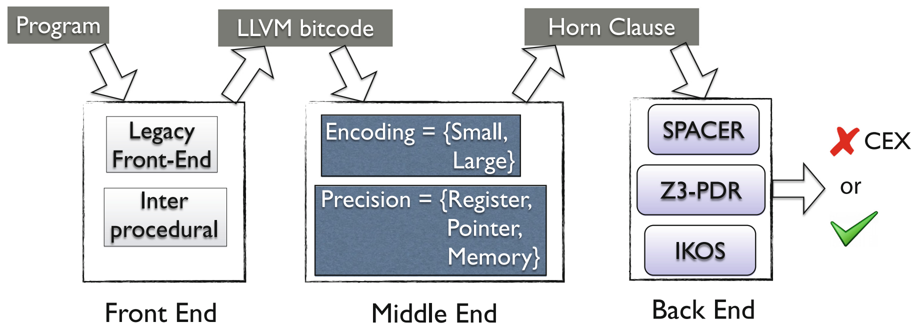


Figure 5.3: Overview of the SeaHorn architecture, taken from [103].

corresponding wait state, which stays true so long as the function has not returned. (5.9) handles storing the return value of f into rv once the function has finished. On return from f, i.e., when ℓ_e holds, the call-site variable v_j must be set to f's return value; (5.10) shows these state updates. r_j denotes the state immediately after returning to the corresponding call site. Note that once f returns, the corresponding wait state w_j will deactivate.

Next we define the control updates. (5.11) defines the function entry; it is entered from one of the call sites. In (5.12), returning to a call site occurs when the system is in the corresponding wait state and at the end of the function.

The transition relation, then, is defined as the conjunction of the constraints from Section 3.2 as well as those in Figure 5.2. The initial state I and inputs Y are unchanged from the MiniLLVM encoding.

5.4. Implementation

We implemented this encoding as a backend for SeaHorn [103], called SeaHornVMT. SeaHorn encodes LLVM programs using a multi-stage process shown in Figure 5.3. SeaHorn's front-end is quite sophisticated, performing inlining, dead code elimination, SSA form, CFG simplifications, loop invariant code motion, scalar replacement of aggregates, etc. It performs a three-phase shape analysis that is context- and field-sensitive [129], [130]. This results in a data structure graph where each node represents a (potentially infinite) set of memory objects, edges represent points-to relationships, and distinct nodes represent disjointness. Nodes are also typed and represent distinct fields distinctly. Node updates are represented in a memory SSA form, allowing heap manipulations to be modeled as conveniently as scalar variables.

SeaHorn provides small-block and large-block Horn encodings. A large-block encoding [45] summarizes cycle-free sections of code by a single transition. A small-block encoding (called a single-block encoding in [45]) models each basic block by a distinct transition. Large-block encodings are typically much more efficient, so we integrated our VMT encoding with SeaHorn's large-block encoding.

SeaHorn large-block encoding [46] is expressed in terms of a cutpoint graph. Our implementation alters this graph by adding new cutpoints for function calls. We now precisely describe the cutpoint graph, how our implementation alters it, and the implementation of the resulting encoding.

Definition 5.2. (Cutset and Cutpoint [46]) Let $G = (V, E)$ be a graph. A set of vertices $S \subseteq V$ is a *cutset* of G iff S contains a vertex from every cycle in G. Equivalently, the graph $(V \setminus S, E \setminus ((S \times V) \cup (V \times S)))$ is acyclic. Every $s \in S$ is a *cutpoint*.

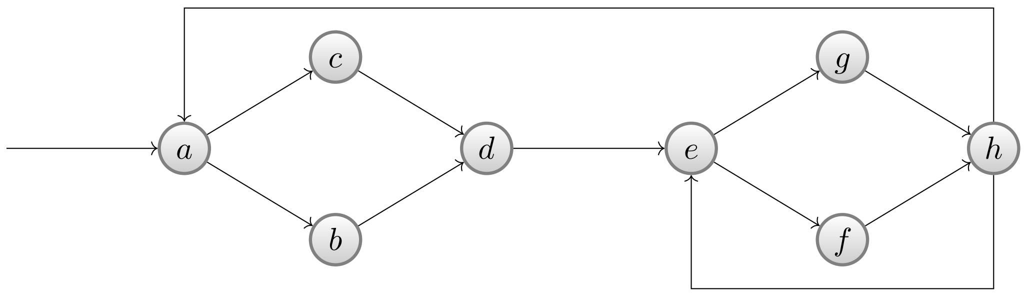
Definition 5.3 (Cutpoint Graph)**.** Let $G = (V, E)$ be a graph and $C \subseteq V$ a cutset. The *C-cutpoint graph* is $G_C = (C, E_C)$ where, for every $n_1, m \in C$, if there exists a path $n_1, n_2, \ldots, n_k, m$ in G such that $n_2, \ldots, n_k \notin S$, then $(n_1, m) \in E_C$.

If the cutpoint is clear from context, we omit the cutset qualifier on the cutpoint graph.

Definition 5.4 (L-Block)**.** Let $G_C = (C, E_C)$ be a cutpoint graph for the CFG $G = (V, E)$. For each edge $(n, m) \in E_C$, let $C_{(n,m)}$ be the set of all vertices (excluding m unless $m = n$) reachable on any path $n, n_2, \ldots, n_k, m$ in G such that $n_2, \ldots, n_k \notin C$. $C_{(n,m)}$ is the *lblock* (for "large block") corresponding to the edge (n, m) with respect to the cutset C for G.

Example 5.1 (CFG and its Cutpoint Graph)**.**

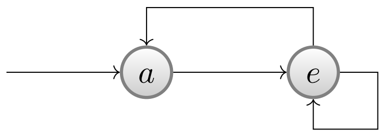
A control flow graph G.



Let $C = \{a, e\}$ be a cutset for G. The cutpoint graph is $G_C = (C, E_C)$ where

$$E_C = \{(a,e),(e,e),(e,a)\}\ .$$

The cutpoint graph looks like this:



and the large blocks associated with each cutpoint graph edge are:

$$\begin{aligned}C_{(a,e)} &= \{a,b,c,d\}\\ C_{(e,e)} &= \{e,f,g,h\}\\ C_{(e,a)} &= \{e,f,g,h\}\ .\end{aligned}$$

The cutpoint graph defines a summary of the program; this is made precise in [46]. Using a cutpoint graph and its lblocks, SeaHornVMT produces a transition system. Each edge in the cutpoint graph is encoded as a single transition, using the edge's lblock. The subgraph induced by an lblock is acyclic so the lblock is encoded using combinational logic only.

The encoding presented in the previous section assumes that each function call is isolated in a block. LLVM allows an arbitrary number of calls to occur in a single basic block because the stack is implicit in LLVM's representation. Our encoding makes the stack explicit and it is simpler to implement if we isolate calls in blocks.

To ensure that function calls fit our assumptions, we implemented an IR pass, `SplitBBAtCalls`. `SplitBBAtCalls` either modifies a given block or it does not. Blocks not containing calls are not modified; blocks with at least one call are modified. Blocks containing calls fit the pattern on the left; assume that $\overline{c_0}$ contains no calls. Such basic blocks are transformed to the set of blocks on the right:

$$
\begin{array}{rl}
l\colon & \overline{c_0} \\
& v = \textbf{call } f(a^1, \ldots, a^k) \\
& \overline{c_1}
\end{array}
\qquad\qquad
\begin{array}{rl}
l\colon & \overline{c_0} \\
& \textbf{br } \textit{true } l_{call} \_ \\
l_{call}\colon & v = \textbf{call } f(a^1, \ldots, a^k) \\
& \textbf{br } \textit{true } l_1 \_ \\
l_1\colon & \overline{c_1}
\end{array}
$$

where l_{call} and l_1 are fresh block names. Afterward, the block l_1 is recursively processed, as it might contain further calls. At the call site for f, `SplitBBAtCalls` splits block l into three blocks:

1. l: This block contains non-call instructions up to (but not including) the call. We insert an unconditional branch from l to l_{call}.

2. l_{call}: This block isolates the call site. We insert an unconditional branch from l_{call} to l_1.

3. l_1: This block contains the commands after the call. Note that l_1 may contain further call instructions.

Since control flows from l to l_{call} to l_1, the transformed program behaves the same as the original program. Only two branch instructions are introduced for each call, so the transformed program is linear in the size of the original one.

One particular problem this transformation addresses is the following. Consider a block like this:

$$
\begin{array}{rl}
l_1\colon & t_1 = x + y \\
& t_2 := \textit{op}(\ldots) \\
l_2\colon & q = \textbf{phi } [t_1, l_1], \ldots \\
& r = \textbf{phi } [t_2, l_1], \ldots
\end{array}
$$

Suppose that *op* is an add instruction. The MiniLLVM encoding for these blocks works out fine. If we assume that x and y are state variables, the set of state variables is $X = \{x, y, q, r\}$. The set of inputs is $Y = \{t_1, t_2\}$.

Instead, suppose that $\textit{op}(\ldots) = \textbf{call } f()$ for some f. Since t_1 is calculated before the call to f and used after the call to f, its value must be "remembered" across the call to f. Therefore, t_1 must be a state variable, not an input.

SeaHornVMT adds cutpoints as follows. First, we run `SplitBBAtCalls` after all of SeaHorn's optimizations but immediately before encoding. Second, SeaHornVMT adds each newly-isolated call block to the cutset. This way we get maximal benefit from the lblock encoding since we introduce extra cutpoints only for calls. SeaHorn-VMT then outputs interprocedural encoding in VMT format.

5.5. Evaluation

RQ1 How does the wait-encoding compare with the same programs, fully inlined?

RQ2 Can we expect that as number of calls increases, we get more benefit from this encoding, under ideal conditions?

RQ3 Does this encoding work with another solver, such as ic3ia?

RQ1: Wait Encoding vs. Full Inlining

The configuration where all functions are inlined is called Inline. The wait encoding is called Wait. LLVM calculates a heuristic cost for a given call site to a given function. This cost accounts for things like the potential code size increase, the stack size increase, the savings of removing a function, etc. If this cost is less than the "inline threshold," then the call site is inlined. We set the inline threshold to 8191 for the Wait-encoded benchmarks. We determined empirically that 8191 was a good balance between no inlining and full inlining.

To generate the large-block benchmarks, SeaHorn was limited to an hour of wall time and 7 GB of memory. SeaHorn may fail to encode because of time-out, mem-out, or because SeaHorn crashes during optimization. We observed all three of these cases. Out of 948 original benchmarks (the same used in Chapter 4), this process produced 709 benchmarks.

Figure 5.4 shows a scatter plot of runtimes using a large-block encoding. Wait solves 388 benchmarks and Inline 387. 348 were solved by both configurations; of these, 233 (67%) are below the diagonal, suggesting that the Wait encoding improves runtime overall. The two configurations are complementary in that each has nearly

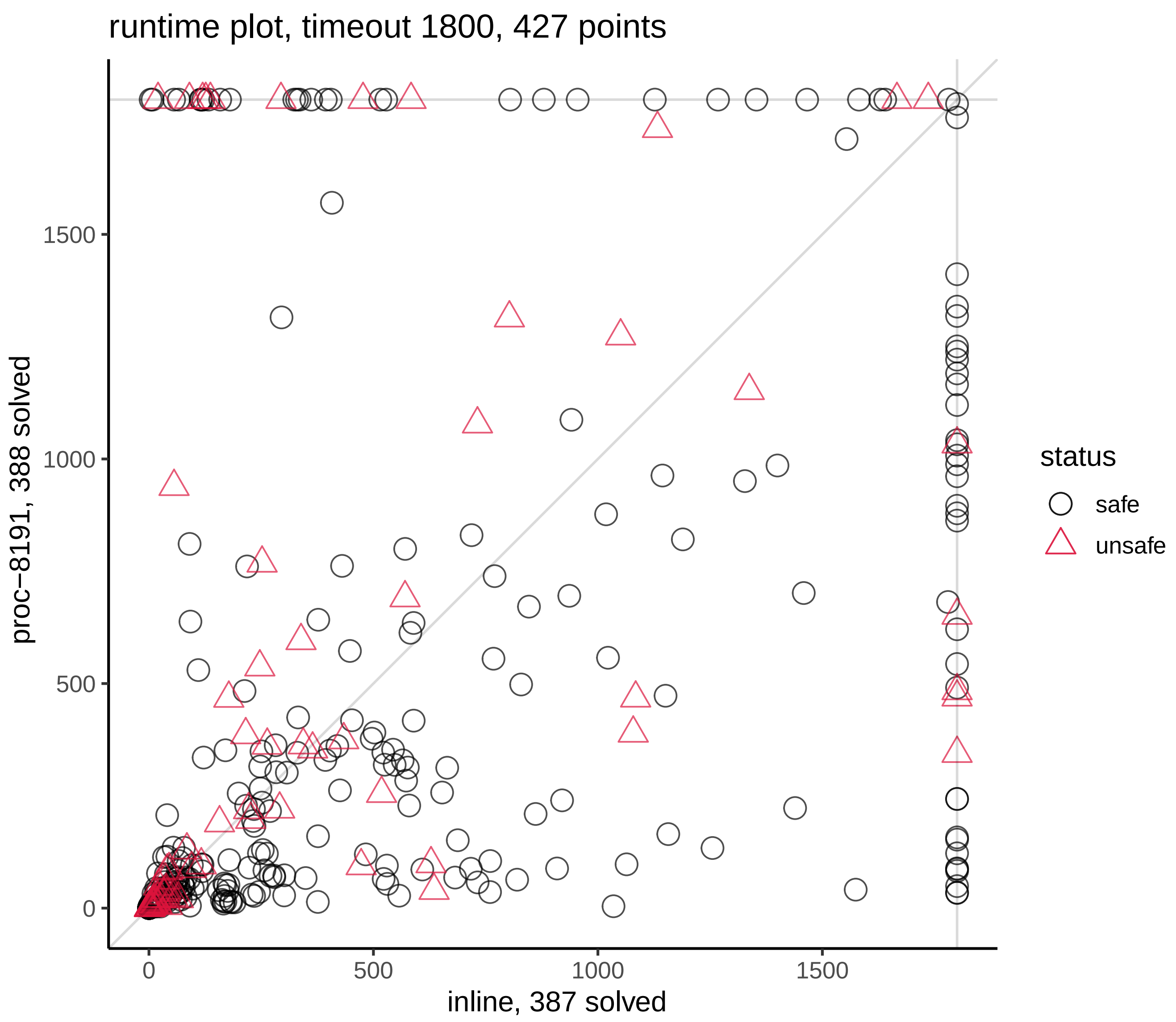


Figure 5.4: Inline configuration vs Wait encoding.

40 uniquely solved benchmarks. A portfolio model checker would do well to try both encodings.

When Wait times out (but Inline doesn't) There are 39 benchmarks for which Wait times out.

1. 20 of these benchmarks slow down due to longer proofs despite the fact that the Wait encoding is smaller in overall size and there are 10's of calls to functions.
2. On 15 benchmarks EUFORIA gets stuck in a refinement step, but does not get stuck on the Inline version.
3. The remaining 4 may fit into the first category, but it's hard to tell.

When Inline times out (but Wait doesn't) There are 40 benchmarks for which Inline times out.

1. 23 benchmarks suffer at least 2x blowup in size under Inline which causes them to be much slower than Wait.
2. 11 benchmarks get stuck during refinement.
3. 3 benchmarks have longer proofs under Inline but this isn't due to a much larger size.

The remaining 2 benchmarks don't fit into the previous categories and we're not yet sure exactly what causes the difference.

As discussed above, SeaHornVMT produces its Wait encoding by inserting a cut-point for each function call. This interacts with the large-block encoding, sometimes quite negatively compared to inlining. As a result, it is difficult to tell if an improvement (or lack thereof) is due to the block size or to the Wait encoding. Take the following program, for instance:

```
function main()
  $l_1$: $f()$
  $l_2$: $f()$
  $l_3$: $f()$
  $l_4$: $f()$
  $l_5$: $f()$
  $l_6$:
```

```
function $f()$
  $x \leftarrow x + y$
```

```
$l_1$: $x \leftarrow x + y$
$l_2$: $x \leftarrow x + y$
$l_3$: $x \leftarrow x + y$
$l_4$: $x \leftarrow x + y$
$l_5$: $x \leftarrow x + y$
$l_6$:
```

If every call to f is inlined, what results is the program on the right. This program can be encoded as a single large block comprised of the five addition statements. The large-block encoding of this program uses two locations: one corresponding to l_1–l_5 and one for l_6. The Wait encoding of this program uses 12 locations: 2 for the entry and exit of f, 5 for each call site, and 5 for each return site. The Wait encoding requires examining many more transitions, compared to the inlined encoding.

The number of state variables may change between encodings, again because of the block size. One might expect that Wait-encoded benchmarks will contain the same state variables as Inline-encoded ones (aside from variables introduced to encode calls themselves). But this is not so, again because of the large block encoding. For instance:

```
$l$: $x \leftarrow a + b$
   $y \leftarrow b * c$
   $t \leftarrow f(x, y)$
   $z \leftarrow t * x * y$
   if $\star$ then
       goto $l$
   ... use $z$ ...
```

When Wait-encoded, x and y are state variables because they are used after the call to f, that is, after the transitions to the entry, and eventually the exit, of f. When Inline-encoded, x and y are simply names for $a + b$ and $b * c$, respectively, and are not encoded as state variables.

To get a clearer picture of the Wait encoding's effect — in other words, to remove the interaction with block size — we might use an encoding in which each instruction is encoded as a single transition. Such an encoding would be totally impractical for verification, but it would allow us to more directly compare Inline-encoded and Wait-encoded benchmarks.

Unfortunately, SeaHorn does not easily support such an encoding. A small-block encoding is as close as we can get. In the small-block encoding, each transition in encodes executing a single basic block.

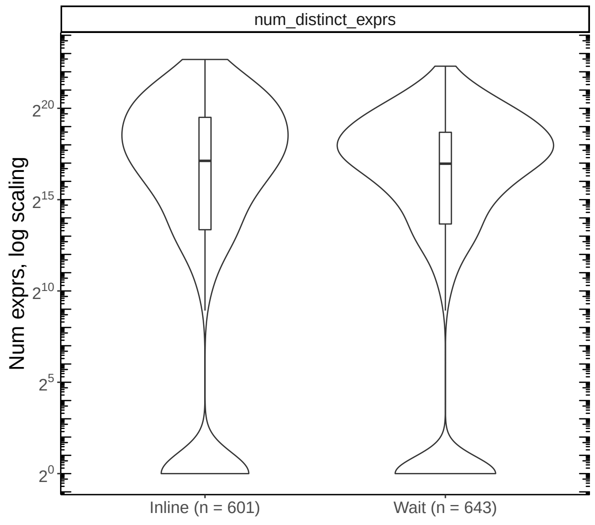
In the small-block configuration, SeaHorn was given 5 minutes of wall time and 1 GB memory for encoding. As a result, some benchmarks failed during inlining that did not fail under the large-block configuration. Furthermore, the Inline configuration fails more often than Wait because it uses more resources. All told, there are 599 Inline benchmarks and 643 Wait benchmarks.

Figure 5.5 shows size information for the small-block benchmarks. I use violin plots to show aggregate data. A violin plot [131] is an augmented box plot. Box plots show four features of a variable: the median, spread, asymmetry, and outliers. The median is denoted by a line in the box. The box extends up and down from the 1st to the 3rd quartiles, that is, the 25% lowest values are below the box and the 25% highest are above it, with the rest inside. The violin plot omits the outliers and adds a density trace, which is a smoothed histogram. The trace is plotted symmetrically about the vertical midline of the plot.

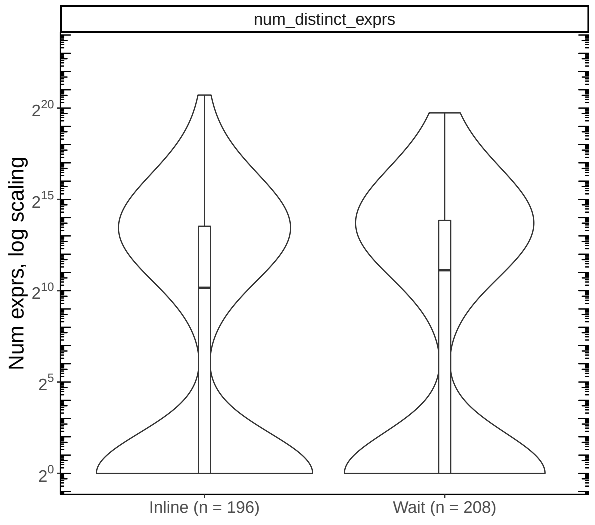
Figure 5.6 shows three measurements from the benchmark data: the number of calls, functions, and the call ratio. These are measured on the intermediate LLVM representation. The call ratio is the ratio of calls to functions. The most salient conclusion from these plots is that the call ratio is low: it never exceeds one. While we don't know an optimal call ratio for the wait encoding, a higher ratio is likely to provide more benefit, as we will see in our discussion of **RQ2**.

The Inline configuration contains functions, which was initially surprising. All function calls are inlined, as can be seen by the lack of function calls in the `num-calls` plots in Figure 5.6. We discuss this in the subsequent Limitations section, as it is an artifact of the current implementation.

In Figure 5.7, Wait solves 17 benchmarks not solved by Inline. On 5 of these benchmarks, Inline crashes or runs out of resources during encoding. On 10 of these benchmarks, Inline produces a much larger benchmarks and Wait's benchmark is able to be solved more quickly and easily. On the remaining 2 benchmarks the previous explanations don't hold and the reason is yet unknown.



(a) All benchmarks.



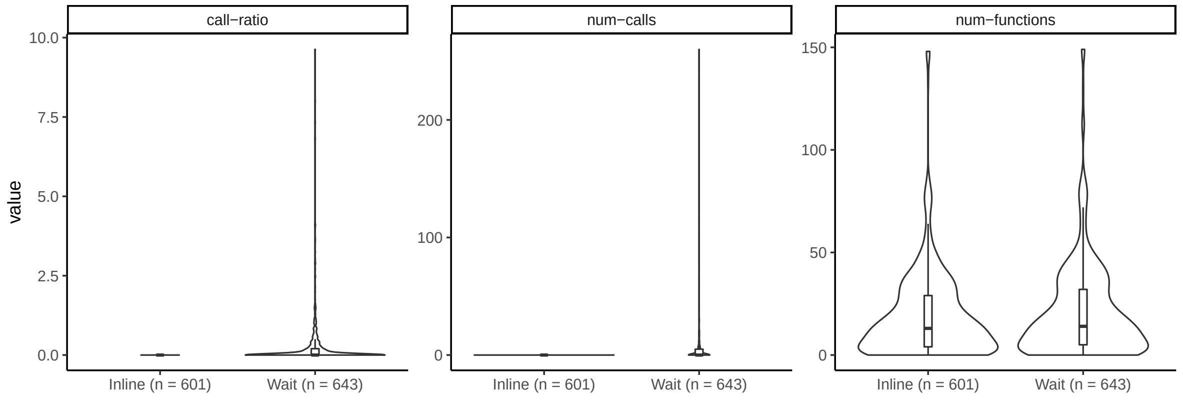
(b) Solved benchmarks.

Figure 5.5: Benchmark size distribution. The size of a benchmark is the number of distinct expressions used to define the initial state, transition system, and property formulas.

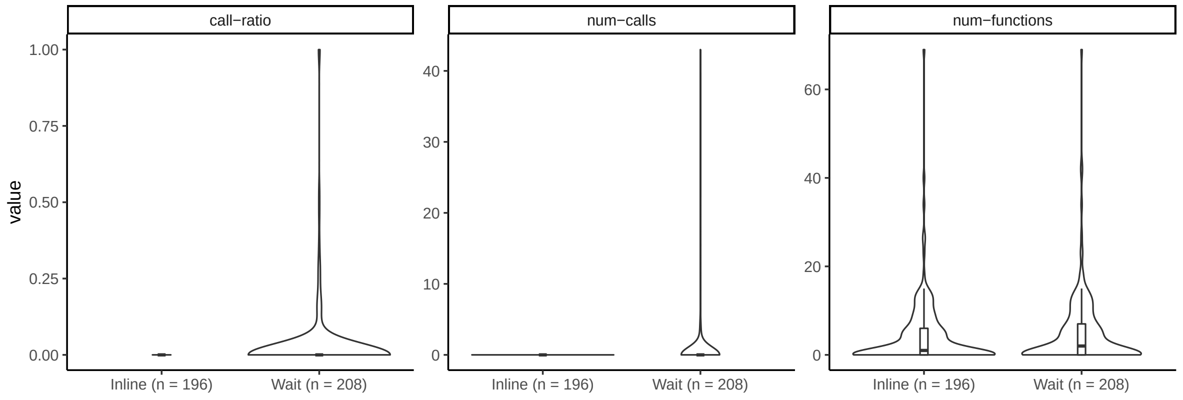
Conclusion In both the large- and small-block encodings, we see frequently that when full inlining blows up (even as little as doubling in size), EUFORIA takes advantage of the concise Wait encoding and solves more quickly. Optimizing the encoding by better inlining or more selectively applying it may show a better overall improvement. For example, one could calculate the improvement that results from doing inlining to see if it reduces the number of large blocks. If so, do inlining. Otherwise, don't.

Questions for future work:

- Why does Wait performance not improve over Inline when there are many function calls?
- What causes the Wait proofs to get longer even if the benchmark is smaller than Inline?



(a) Comparing configurations on aggregate function measurements over all benchmarks.



(b) Comparing configurations on aggregate function measurements over benchmarks solved by each configuration. The call ratio never exceeds one. This suggests our encoding isn't likely to give much benefit.

Figure 5.6: Aggregate benchmark info about functions using small-step encoding. The call ratio is the number of calls divided by the number of functions.

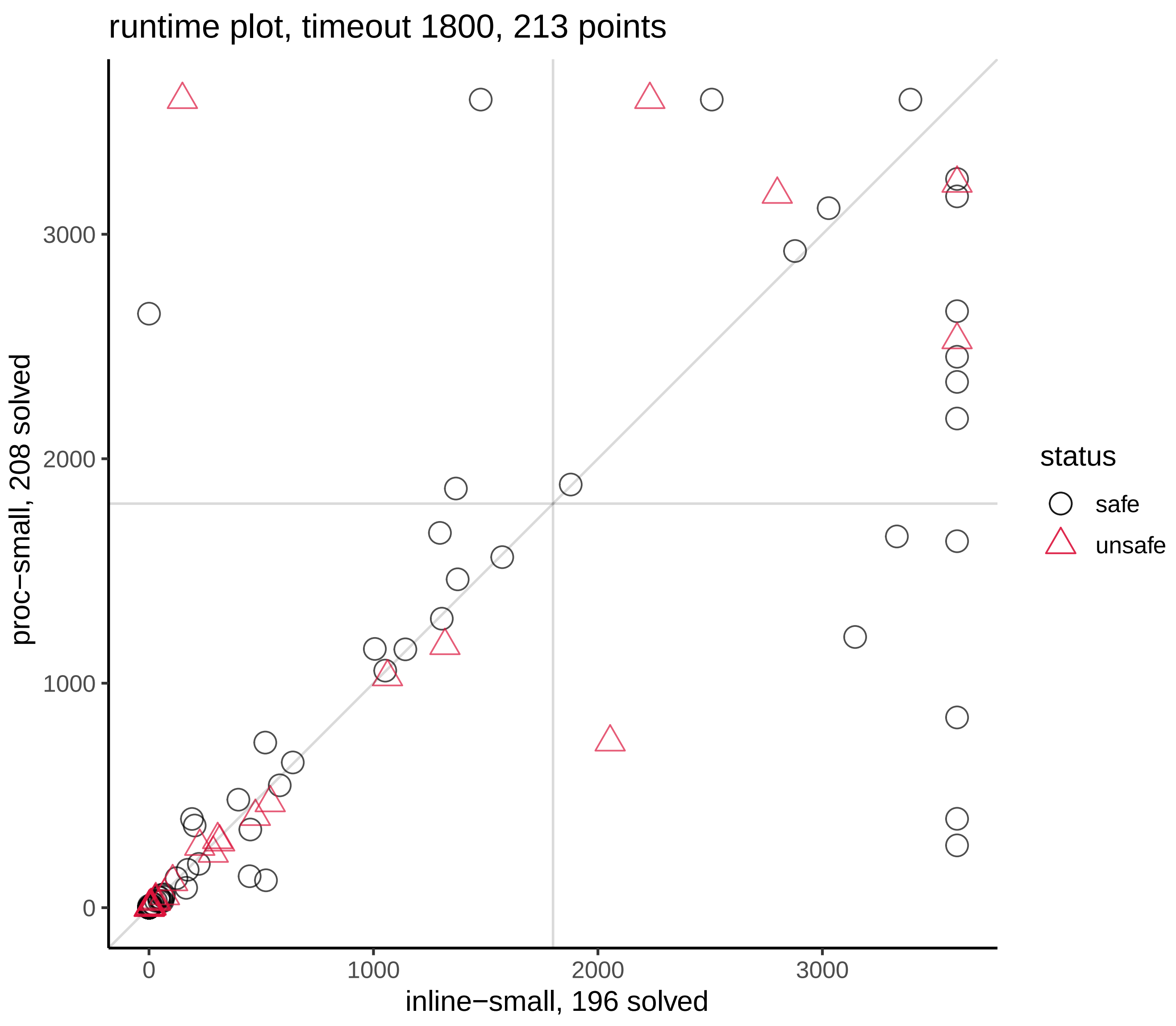
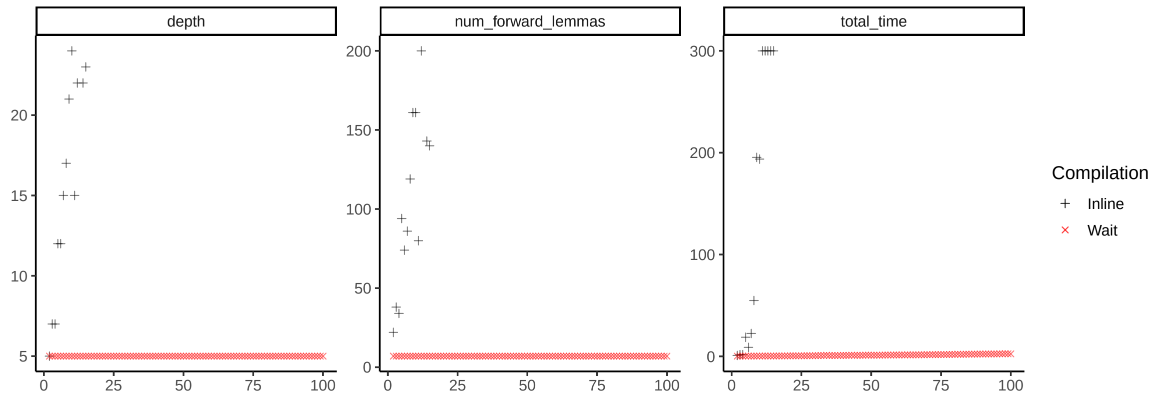


Figure 5.7: Inline configuration vs Waitusing small-block encoding.

```c
int i;
void loop() {
  do {
    i += 3;
  } while (i < 5);
  __VERIFIER_assert(i < 7);
}

int main() {
  i = 0; loop(); /* this line repeats some number of times */
  /* ... more calls to loop() */
  return 0;
}
```

(a) Microbenchmark template. Each instance of this microbenchmark varies the number of calls to `loop`.



(b) Microbenchmark results under Inline and Wait encodings. The x axis is the number of calls to `loop`. The y axis for total_time is in seconds.

RQ2: Reusable Reasoning

This section demonstrates an important feature of our encoding: EUFORIA can discover information that is reusable across different calling contexts. We created a sequence of benchmarks to show the potential of this reuse. The benchmark template in Figure 5.8a is used to derive the sequence.

It consists of a function `loop` which satisfies a simple assertion. When checking `loop` in isolation, EUFORIA generates 7 refinement lemmas to prove that `loop` satisfies the assertion. The nth instance of the benchmark repeats the call to `loop` n times. We tested values of n between 2 and 100 which means the call ratio is between 2 and

100, respectively. We tested the Inline encoding against the Wait-0 encoding, where 0 refers to the inline threshold. Using Wait-0 guarantees that no inlining is performed at all.

Figure 5.8b shows the results of solving the benchmark sequence using the Wait-0 and Inline encodings. In the Inline configuration, the number of lemmas grows number of call sites n increases. Each call site is inlined, resulting in n distinct while loops for n distinct call sites. Each lemma needs to be learned for each distinct occurrence of the while loop. Using the Wait encoding, however, the number of lemmas stays constant at 7! The lemmas are learned once for the body of `loop` and reused at all call sites.

Furthermore, the Inline configuration increases convergence depth and total time as a function n. But the Wait encoding converges at a constant depth (5) and the time looks to increase linearly (and slowly). All Wait runtimes are below three seconds, even for $n = 100$.

We did not test Inline benchmarks for n beyond 15, since the runtime exceeds our timeout of 5 minutes. Using the Wait encoding, every benchmark up to $n = 100$ is solved in at most a couple of seconds.

RQ3: IC3IA

We ran IC3IA on the 709 large-step benchmarks. The runtime scatter plot for these experiments is shown in Figure 5.9. The configurations are again complementary, since Wait solves 16 unique benchmarks and Inline solves 14 uniques. The Wait configuration solves two more benchmarks than Inline. 134 benchmarks are solved by both configurations; of these, 63 (47%) are below the diagonal. Since these results are consistent with EUFORIA's, we did not deeply investigate these benchmarks. These results show that the Wait encoding is usable with other model checkers.

5.5.1. Limitations

The implementation has several limitations. It is not conceptually necessary to add cutpoints for each function call, as SeaHornVMT does. All that a cutset requires is a vertex from every cycle in the control flow graph, as discussed in Section 5.4. A clev-

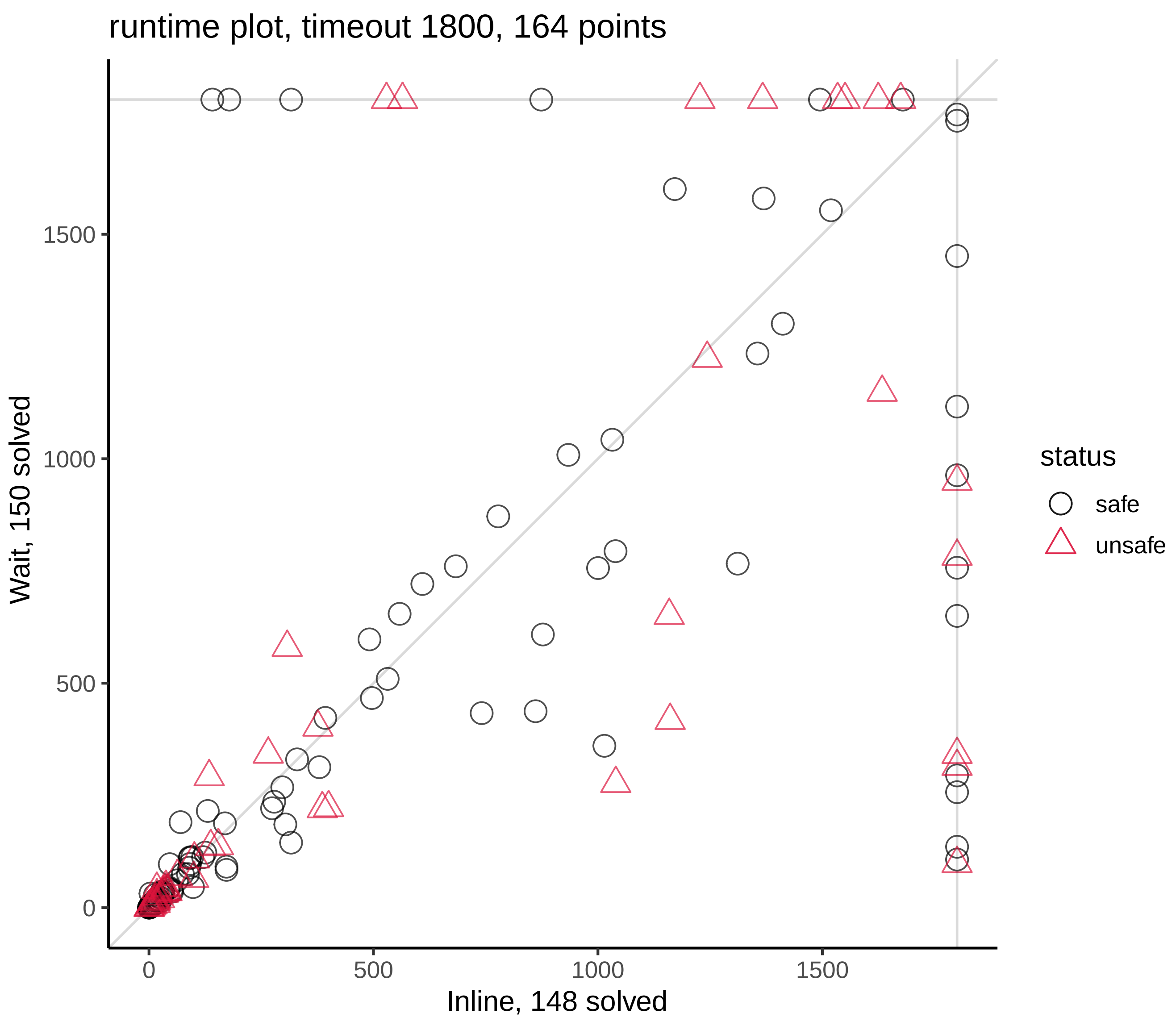


Figure 5.9: Runtime scatter plot for IC3IA.

erer implementation would do the flow analysis necessary to minimize the number of new cutpoints.

We frequently found that encoded benchmarks contained functions that were never called. In these cases the function is *used* because its address is stored into a data structure; but it isn't called. SeaHorn (including LLVM's passes) does not use a pass that detects such cases so that the function can be removed. As a result, the benchmark encoding is cluttered with many infeasible transitions, negatively affecting the underlying SMT solving.

5.6. Related Work

gpdr [50] integrates procedures — encoded as linear Horn clauses — with ic3. The core change is that proof obligations are arranged in a DAG instead of a list. A satisfiable pob generates two predecessors instead of one. If a node is proved unreachable, its sibling is removed from the DAG as well because it no longer matters to the current potential counterexample. A counterexample is found when all the leaves of the DAG are in I. To facilitate finding counterexamples, gpdr maintains a cache that under-approximates up to length-n executions. Nodes are added to the cache when in I and, recursively, when both children are in the cache. Therefore, when the root of the counterexample DAG is in the cache, gpdr terminates with a counterexample.

spacer [36] maintains both over- and under-approximations of procedures. It maintains a symbolic over-approximation (similar to that maintained by ic3 frontiers). It also maintains a symbolic under-approximation which facilitates call site analysis without inspecting the callee's body.

SLAM [61] computes procedure summaries in terms of its predicate abstraction, using BDDs as the underlying data structure. Whale [99] computes summaries from interpolants derived from infeasible executions in which callees are under-approximated. Others that use Craig Interpolation for function handling include HSF [132], Duality [133], Ultimate Automizer [134], and Eldarica [135].

Sharir and Pnueli [128] identify a problem with treating calls and returns as branches: facts about call sites bodies can "leak into" other call sites. Treating calls and returns as branches amounts to, in the terminology of Reps *et al.* [136], "considering all paths rather than considering only the interprocedurally realizable paths." The problem occurs because branches corresponding to a call are no longer associated with their corresponding return branches. As a result, paths which should be infeasible are feasible. For the example of Section 5.2.1, if calls and returns were treated as branches, the following path would be considered feasible:

$$l_i \to^c l_s \to \cdots \to l_e \to^r l_j \, .$$

This path corresponds to a call to f that returns to the wrong call site (it should return to l_i, where the call originated).

This situation cannot happen in our encoding. Calls are treated as branches *along with* stack information, in the form of wait and return states. The stack information pairs calls and returns properly so that only interprocedurally realizable paths are explored.

DAG inlining [137] is a procedure for handling programs with no loops and no recursion. It constructs a DAG unfolding of a program for deciding bounded reachability queries. The DAG unfolding achieves exponential savings compared to a tree unfolding of the call graph. Our approach, on the other hand, does not unfold the call graph at all, but encodes all possible call sequences implicitly by assignments to the wait states.

Stratified inlining [138] is a method for iteratively refining approximations of (possibly-recursive) procedure bodies on demand. The technique does not eliminate the cost of inlining, but delays it. By contrast, we do not require any inlining.

Chapter 6.

Epilogue

6.1. Conclusions

We presented an approach for the automatic verification of safety properties of programs using EUF abstraction on top of incremental, inductive model checking. 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. EUF has particularly efficient decision procedures. 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 evaluations show 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.

6.2. Future Work

- Some control properties require reasoning localized data operations. Specific code fragments in a program may be critical for verifying the property, even if by and large data is irrelevant. 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. In other words, uninterpreted functions are replaced with their concrete counterparts, enabling concrete reasoning mixed with EUF abstraction.

- The term projection procedure presented in Chapter 3 works for some EUF formulas. It works almost all the time for transition systems whose encoding we control. It appears that, at least, it *doesn't* work when the existential formula $\exists V.\ \phi_{qf}(V, W)$ implies an atom $a(W)$ involving a term not in ϕ_{qf}.

  We want to characterize exactly when this projection works and when it does not, either syntactically (e.g., "it works on certain types of formulas") or semantically (e.g., "it works when the quantifier-free model has a certain property"). Moreover, we wish to extend the projection to arbitrary EUF formulas. We believe this can be done via a grounding procedure, that is, by enumerating ground formulas using the quantifier-free formula until sufficient to imply the desired existential. The trouble is that we can generate groundings indefinitely; even if bounded, the number of groundings may explode and many of them are irrelevant, so enumerating them would be a waste. Further work would investigate how to make an EUF MBP procedure that is efficient for model checking.

- EUFORIA supports non-recursive functions. What about recursive functions? EUF provides an opportunity to preserve function bodies and abstract recursive calls using UFs. Such an abstraction over-approximates the recursion. Without refinement, the abstraction may still be sufficient to prove some properties. Further work is required to determine how to refine this efficiently.

- We noticed that the LLVM 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. Wagner *et al.* have also identified situations in which compilers produce code suboptimal for verification [139]. Moreover, recent work [140] 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. Exploring alternative front-ends tailored for verification remains important future work.

Appendix A.

Supplements

A.1. Term Projection

The following two theorems assume that the model M is the one that is globally used by TPRec.

Theorem A.1 (TPRec Preserves Term Equivalence). *For every EUF term t and model M,*

$$\text{Part}[M^*_{\simeq}](\text{TPRec}(t)) = \text{Part}[M^*_{\simeq}](t)$$

Proof. We proceed by structural induction on the term t using the definition of TPRec in Figure 3.5.

- x: Handled on line 7. Trivial because $\text{TPRec}(x) = x$.
- $\mathsf{F}(t_1, t_2, \dots, t_n)$: Line 10 returns $\mathsf{F}(\text{TPRec}(t_1), \text{TPRec}(t_2), \dots, \text{TPRec}(t_n))$. By induction hypothesis, TPRec preserves equivalence of each argument, i.e.,

  $$\text{Part}[M^*_{\simeq}](\text{TPRec}(t_i)) = \text{Part}[M^*_{\simeq}](t_i) \quad \text{for all } i \in \{1, \dots, n\} .$$

  By EUF congruence on F's arguments $\text{Part}[M^*_{\simeq}](\text{TPRec}(t_i))$ and $\text{Part}[M^*_{\simeq}](t_i)$:

  $$\text{Part}[M^*_{\simeq}](\mathsf{F}(\text{TPRec}(t_1), \text{TPRec}(t_2), \dots, \text{TPRec}(t_n))) = \text{Part}[M^*_{\simeq}](\mathsf{F}(t_1, t_2, \dots, t_n))$$

  which is what we want to show.
- $\text{ite}(c, t_1, t_2)$: Line 14 returns either $\text{TPRec}(t_1)$ or $\text{TPRec}(t_2)$.

1. If $M \vDash c$, then $\text{Part}[M^*_{\simeq}](f) = \text{Part}[M^*_{\simeq}](t_1)$ and

$$\text{Part}[M^*_{\simeq}](\text{TPRec}(t_1)) = \text{Part}[M^*_{\simeq}](t_1)$$

by induction hypothesis.

2. If $M \vDash \neg c$, then $\text{Part}[M^*_{\simeq}](f) = \text{Part}[M^*_{\simeq}](t_2)$ and

$$\text{Part}[M^*_{\simeq}](\text{TPRec}(t_2)) = \text{Part}[M^*_{\simeq}](t_2)$$

by induction hypothesis.

□

Theorem A.2. *For every EUF formula f and model M,*

$$M \vDash \text{TPRec}(f) \textit{ iff } M \vDash f \ .$$

Proof. We prove this by structural induction on f.

- $t_1 \simeq t_2$: Line 22. By induction hypothesis,

$$M(\text{TPRec}(t_1)) = M(t_1) \quad \text{and} \quad M(\text{TPRec}(t_2)) = M(t_2) \ .$$

  By congruence on equality, $M(\text{TPRec}(f)) = M(f)$.

- $\mathsf{P}(t_1, t_2, \ldots, t_n)$: Line 24. Same argument as for an uninterpreted function F.

- $\neg f_1$: Line 26. By the induction hypothesis, $M(\text{TPRec}(f_1)) = M(f_1)$. By congruence on negation, $M(\text{TPRec}(\neg f_1)) = M(\neg f_1)$.

- $f_1 \wedge f_2$: Line 28. If $M \vDash f$, then $M(f_1) = M(\text{TPRec}(f_1))$ by induction hypothesis. Otherwise, $M(f) = \textit{false}$, which happens if either (or both) $M(f_1) = \textit{false}$ or $M(f_2) = \textit{false}$. If $M \vDash \neg f_1$, $M(\text{TPRec}(f_1)) = M(f_1)$ by induction hypothesis and $M(f_1) = M(f)$. If $M \vDash \neg f_2$, $M(\text{TPRec}(f_2)) = M(f_2)$ by induction hypothesis and $M(f_2) = M(f)$.

- $f_1 \vee f_2$: Line 33. If $M \vDash f_1$ then $M(\text{TPRec}(f_1)) = M(f_1)$ by induction hypothesis and $M(f_1) = M(f)$. If $M \vDash f_2$ then $M(\text{TPRec}(f_2)) = M(f_2)$ by induction hypothesis and $M(f_2) = M(f)$. Otherwise, $M \vDash \neg f$ then $M(\text{TPRec}(f_1)) = M(f_1)$ by induction hypothesis and $M(\text{TPRec}(f_1)) = M(f)$.

□

Theorem 3.1, pg. 63. *For every model M such that $M \vDash T \wedge s'$,*

$$M \vDash \textsc{ExpandPreimage}(s', M) \ .$$

Proof. The call to ModelAssertion(M, Q, S) ensures that the returned formula is satisfied by M. □

A.2. Termination & Correctness

Theorem 3.2, pg. 69. *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 pre-image generalization procedure, ExpandPreimage (Algorithm 2), 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 GeneralizeInfeasible (Algorithm 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 3.3, pg. 69. *euforia's refinement procedure increases the fidelity of the 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 $\widehat{A}_0, \widehat{A}_1, \ldots, \widehat{A}_n$ in $\widehat{T}$ where each $\widehat{A}_i$ $(0 \leq i \leq n)$ is a state formula. Beginning in any single state $\sigma_0 \in A_1 \wedge I$, for all $1 \leq i \leq n$,

1. Check whether $\sigma_{i-1} \wedge T \wedge A'_i$ is satisfiable.
2. If so, form new state σ_i using the concrete assignments to all variables X'
3. If not, call $\textsc{LearnLemma}(c)$ 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 3.4) is essentially the same as this procedure, except RefineForward attempts to generate stronger lemmas that refute multiple spurious concrete paths at once. □

A.3. Horn2VMT

Theorem A.3. *The transition system has the property that the state ℓ_R is reachable in T if and only if relation R is derivable under the Horn clauses.*

Proof. Direction ($\Leftarrow$): We proceed by induction on the length of the derivation of R. All relations are initially empty; this is correctly modeled by the definition of I. Length-1 derivations use a single Horn clause whose body contains no uninterpreted relations with head R. Such a clause translates to a transition that can be similarly satisfied without relation variables and which also satisfies ℓ'_R.

Consider a relation R derivable in $n+1$ steps. Its last derivation step involves some rule with head R; by the induction hypothesis, a state satisfying the body of this rule is reachable in T. Examining T and the definition of $\mathcal{H}[\![\cdot]\!]$ allows us to conclude that the next state satisfies ℓ'_R.

Direction ($\Rightarrow$): We proceed by induction on the number of transitions. Initially, no relation variable is reached, due to I's definition. Next, suppose that $I \wedge T \wedge \ell'_R$ is satisfied. T guarantees that the body of the Horn rule corresponding to the transition is satisfied, so R is derivable.

Assume that some state $\sigma \wedge \ell'_R$ is reachable after $n+1$ steps. By the induction hypothesis, the current state, which took n steps to reach, has a Horn derivation. The current state corresponds to the body of a rule with head R, since the only transitions to ℓ'_R in T correspond to such rules. Therefore R is derivable. $\square$

If we do not assume that the Horn clauses are linear, Direction ($\Leftarrow$) doesn't work, because one can produce a rule that can never be activated (e.g., body $R(1) \wedge R(2)$ translates to $\ell_R \wedge P_{R,1} = 1 \wedge P_{R,1} = 2$, a contradiction).

Bibliography

[1] K. L. McMillan, "Symbolic model checking: An approach to the state explosion problem," Ph.D. dissertation, Carnegie Mellon University, May 1992.

[2] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification*, D. L. Dill, Ed., ser. Lecture Notes in Computer Science, vol. 818, Springer, 1994, pp. 68–80.

[3] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for Construction and Analysis of Systems*, B. Steffen, Ed., ser. Lecture Notes in Computer Science, vol. 1384, Springer, 1998, pp. 151–166, ISBN: 3-540-64356-7. DOI: `10.1007/BFb0054170`. [Online]. Available: `https://doi.org/10.1007/BFb0054170`.

[4] D. Kroening and O. Strichman, *Decision Procedures – An Algorithmic Point of View, Second Edition*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016, ISBN: 978-3-662-50496-3. DOI: `10.1007/978-3-662-50497-0`. [Online]. Available: `https://doi.org/10.1007/978-3-662-50497-0`.

[5] P. Godefroid and S. K. Lahiri, "From program to logic: An introduction," in *Tools for Practical Software Verification*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds., vol. 7682, Springer Berlin Heidelberg, 2012, pp. 31–44.

[6] 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*, I. Cervesato, H. Veith, and A. Voronkov, Eds., ser. Lecture Notes in Computer Science, vol. 5330, Springer, 2008, pp. 343–352.

[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., vol. 8559, Springer International Publishing, 2014, pp. 849–865.

[8] A. Langley, *Apple's SSL/TLS bug*, `https://www.imperialviolet.org/2014/02/22/applebug.html`, [Online; accessed September 28, 2018], 2014.

[9] 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, ISBN: 1-58113-612-9. DOI: `10.1145/586110.586142`. [Online]. Available: `http://doi.acm.org/10.1145/586110.586142`.

[10] 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.

[11] P. Wolper, "Expressing interesting properties of programs in propositional temporal logic," in *ACM Symposium on Principles of Programming Languages*, ACM Press, 1986, pp. 184–193. DOI: `10.1145/512644.512661`. [Online]. Available: `https://doi.org/10.1145/512644.512661`.

[12] R. Lazić and D. Nowak, "A unifying approach to data-independence," in *Concurrency Theory*, C. Palamidessi, Ed., ser. Lecture Notes in Computer Science, vol. 1877, Springer, 2000, pp. 581–595, ISBN: 3-540-67897-2. DOI: `10.1007/3-540-44618-4\_41`. [Online]. Available: `https://doi.org/10.1007/3-540-44618-4%5C_41`.

[13] K. S. Namjoshi and R. P. Kurshan, "Syntactic program transformations for automatic abstraction," in *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds., ser. Lecture Notes in Computer Science, vol. 1855, Springer, 2000, pp. 435–449. DOI: `10.1007/10722167\_33`. [Online]. Available: `https://doi.org/10.1007/10722167%5C_33`.

[14] 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. DOI: `10.1109/TSE.1986.6312929`. [Online]. Available: `https://doi.org/10.1109/TSE.1986.6312929`.

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

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

[17] J. McCarthy, "Towards a mathematical science of computation," in *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*, North-Holland, 1962, pp. 21–28.

[18] R. W. Floyd, "Assigning meanings to programs," in *Symposia in Applied Mathematics*, American Mathematical Society, vol. 19, 1967.

[19] Z. Manna, "The correctness of programs," *Computer and System Sciences*, vol. 3, no. 2, pp. 119–127, May 1969.

[20] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[21] R. M. Keller, "Formal verification of parallel programs," *Commun. ACM*, vol. 19, no. 7, pp. 371–384, 1976. DOI: 10.1145/360248.360251. [Online]. Available: https://doi.org/10.1145/360248.360251.

[22] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[23] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *ACM Symposium on Principles of Programming Languages*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds., ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. [Online]. Available: http://doi.acm.org/10.1145/512950.512973.

[24] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*, Springer, 1981, pp. 52–71.

[25] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach," in *Symposium on Principles of Programming Languages*, J. R. Wright, L. Landweber, A. J. Demers, and T. Teitelbaum, Eds., ACM Press, 1983, pp. 117–126, ISBN: 0-89791-090-7. DOI: 10.1145/567067.567080. [Online]. Available: https://doi.org/10.1145/567067.567080.

[26] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, M. Dezani-Ciancaglini and U. Montanari, Eds., ser. Lecture Notes in Computer Science, vol. 137, Springer, 1982, pp. 337–351, ISBN: 3-540-11494-7. DOI: 10.1007/3-540-11494-7_22. [Online]. Available: https://doi.org/10.1007/3-540-11494-7%5C_22.

[27] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018, ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8.

[28] 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.

[29] R. Nieuwenhuis and A. Oliveras, "Fast congruence closure and extensions," *Inf. Comput.*, vol. 205, no. 4, pp. 557–580, 2007. DOI: 10.1016/j.ic.2006.08.009. [Online]. Available: https://doi.org/10.1016/j.ic.2006.08.009.

[30] 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.

[31] 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.

[32] N. Piterman and A. Pnueli, "Temporal logic and fair discrete systems," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Springer, 2018, pp. 27–73, ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8_2. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_2.

[33] 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.

[34] A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan, "Compositional verification of procedural programs using Horn clauses over integers and arrays," in *Formal Methods in Computer-Aided Design*, R. Kaivola and T. Wahl, Eds., IEEE, 2015, pp. 89–96, ISBN: 978-0-9835678-5-1.

[35] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, "Automatic abstraction in SMT-based unbounded software model checking," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds., ser. Lecture Notes in Computer Science, vol. 8044, Springer, 2013, pp. 846–862. DOI: 10.1007/978-3-642-39799-8_59. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8%5C_59.

[36] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-based model checking for recursive programs," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, vol. 8559, Berlin, Heidelberg: Springer-Verlag, 2014, pp. 17–34, ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9.

[37] 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*, E. Ábrahám and K. Havelund, Eds., ser. Lecture Notes in Computer Science, vol. 8413, Springer, 2014, pp. 46–61, ISBN: 978-3-642-54861-1. DOI: 10.1007/978-3-642-54862-8_4.

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

[39] 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.

[40] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, “Formalizing the LLVM intermediate representation for verified program transformations,” in *Symposium on Principles of Programming Languages*, J. Field and M. Hicks, Eds., ACM, 2012, pp. 427–440. DOI: 10.1145/2103656.2103709. [Online]. Available: https://doi.org/10.1145/2103656.2103709.

[41] ——, “Formal verification of ssa-based optimizations for LLVM,” in *Programming Language Design and Implementation*, H.-J. Boehm and C. Flanagan, Eds., ACM, 2013, pp. 175–186. DOI: 10.1145/2491956.2462164. [Online]. Available: https://doi.org/10.1145/2491956.2462164.

[42] Z. Manna and A. Pnueli, *Temporal verification of reactive systems – safety*. Springer, 1995, ISBN: 978-0-387-94459-3.

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

[44] 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, ISBN: 978-0-9835678-5-1.

[45] 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, ISBN: 978-1-4244-4966-8. DOI: 10.1109/FMCAD.2009.5351147.

[46] A. Gurfinkel, S. Chaki, and S. Sapra, “Efficient predicate abstraction of program summaries,” in *NASA Formal Methods*, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., ser. Lecture Notes in Computer Science, vol. 6617, Springer, 2011, pp. 131–145, ISBN: 978-3-642-20397-8. DOI: 10.1007/978-3-642-20398-5_11. [Online]. Available: https://doi.org/10.1007/978-3-642-20398-5%5C_11.

[47] LLVM Project, *LLVM language reference manual*, accessed September 10, 2020. [Online]. Available: https://llvm.org/docs/LangRef.html#global-variables.

[48] D. Bueno and K. A. Sakallah, "EUFORIA: Complete software model checking with uninterpreted functions," in *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, C. Enea and R. Piskac, Eds., ser. Lecture Notes in Computer Science, vol. 11388, Springer, 2019, pp. 363–385, ISBN: 978-3-030-11244-8.

[49] J. Birgmeier, A. R. Bradley, and G. Weissenbacher, "Counterexample to induction-guided abstraction-refinement (CTIGAR)," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 831–848, ISBN: 978-3-319-08866-2. DOI: `10.1007/978-3-319-08867-9_55`. [Online]. Available: `https://doi.org/10.1007/978-3-319-08867-9_55`.

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

[51] 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, ISBN: 978-1-4503-2153-2. DOI: `10.7873/DATE.2013.168`.

[52] D. Kroening, A. Groce, and E. M. Clarke, "Counterexample guided abstraction refinement via program execution," in *International Conference on Formal Engineering Methods*, J. Davies, W. Schulte, and M. Barnett, Eds., ser. Lecture Notes in Computer Science, vol. 3308, Springer, 2004, pp. 224–238, ISBN: 3-540-23841-7. DOI: `10.1007/978-3-540-30482-1_23`. [Online]. Available: `https://doi.org/10.1007/978-3-540-30482-1_23`.

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

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

[55] O. Tange, *GNU Parallel 2018*. Ole Tange, Mar. 2018, ISBN: 9781387509881. DOI: `10.5281/zenodo.1146014`. [Online]. Available: `https://doi.org/10.5281/zenodo.1146014`.

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

[57] 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.

[58] D. Beyer, “Software verification with validation of results - (report on SV-COMP 2017),” in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay and T. Margaria, Eds., ser. Lecture Notes in Computer Science, vol. 10206, 2017, pp. 331–349.

[59] F. B. Kessler, *The VMT format*, (accessed May 13, 2020), 2020. [Online]. Available: `https://nuxmv.fbk.eu/index.php?n=Languages.VMT`.

[60] 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.

[61] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, “Automatic predicate abstraction of C programs,” in *Programming Language Design and Implementation*, M. Burke and M. L. Soffa, Eds., ACM, 2001, pp. 203–213, ISBN: 1-58113-414-2. DOI: `10.1145/378795.378846`. [Online]. Available: `https://doi.org/10.1145/378795.378846`.

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

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

[64] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003. DOI: `10.1145/876638.876643`. [Online]. Available: `https://doi.org/10.1145/876638.876643`.

[65] F. Balarin and A. L. Sangiovanni-Vincentelli, “An iterative approach to language containment,” in *Computer Aided Verification*, C. Courcoubetis, Ed., ser. Lecture Notes in Computer Science, vol. 697, Springer, 1993, pp. 29–40, ISBN: 3-540-56922-7. DOI: `10.1007/3-540-56922-7\_4`. [Online]. Available: `https://doi.org/10.1007/3-540-56922-7%5C_4`.

[66] F. Corella, “Automated high-level verification against clocked algorithmic specifications,” in *Computer Hardware Description Languages and their Applications*, D. Agnew, L. J. M. Claesen, and R. Camposano, Eds., ser. IFIP Transactions, vol. A-32, North-Holland, 1993, pp. 147–154, ISBN: 0-444-81641-0.

[67] R. E. Bryant, S. M. German, and M. N. Velev, “Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic,” *ACM Trans. Comput. Log.*, vol. 2, no. 1, pp. 93–134, 2001. DOI: `10.1145/371282.371364`. [Online]. Available: `https://doi.org/10.1145/371282.371364`.

[68] D. W. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. P. Rajan, "Embedded software verification using symbolic execution and uninterpreted functions," *Int. J. Parallel Program.*, vol. 34, no. 1, pp. 61–91, 2006. doi: 10.1007/s10766-005-0004-8. [Online]. Available: https://doi.org/10.1007/s10766-005-0004-8.

[69] S. Gulwani and A. Tiwari, "Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions," in *European Symposium on Programming*, P. Sestoft, Ed., ser. Lecture Notes in Computer Science, vol. 3924, Springer, 2006, pp. 279–293, isbn: 3-540-33095-X. doi: 10.1007/11693024_19. [Online]. Available: https://doi.org/10.1007/11693024%5C_19.

[70] B. Godlin and O. Strichman, "Regression verification: Proving the equivalence of similar programs," *Softw. Test. Verification Reliab.*, vol. 23, no. 3, pp. 241–258, 2013. doi: 10.1002/stvr.1472. [Online]. Available: https://doi.org/10.1002/stvr.1472.

[71] N. P. Lopes and J. Monteiro, "Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 4, pp. 359–374, 2016. doi: 10.1007/s10009-015-0366-1. [Online]. Available: https://doi.org/10.1007/s10009-015-0366-1.

[72] A. Zaks and A. Pnueli, "Program analysis for compiler validation," in *Workshop on Program Analysis for Software Tools and Engineering*, S. Krishnamurthi and M. Young, Eds., ACM, 2008, pp. 1–7, isbn: 978-1-60558-382-2. doi: 10.1145/1512475.1512477. [Online]. Available: https://doi.org/10.1145/1512475.1512477.

[73] S. Gulwani and G. C. Necula, "A polynomial-time algorithm for global value numbering," *Sci. Comput. Program.*, vol. 64, no. 1, pp. 97–114, 2007. doi: 10.1016/j.scico.2006.03.005. [Online]. Available: https://doi.org/10.1016/j.scico.2006.03.005.

[74] U. Mathur, P. Madhusudan, and M. Viswanathan, "Decidable verification of uninterpreted programs," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 46:1–46:29, 2019. doi: 10.1145/3290359. [Online]. Available: https://doi.org/10.1145/3290359.

[75] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "An abstract domain of uninterpreted functions," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds., ser. Lecture Notes in Computer Science, vol. 9583, Springer, 2016, pp. 85–103, isbn: 978-3-662-49121-8. doi: 10.1007/978-3-662-49122-5_4. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5%5C_4.

[76] B.-Y. E. Chang and K. R. M. Leino, "Abstract interpretation with alien expressions and heap structures," in *Verification, Model Checking, and Abstract Interpretation*, R. Cousot, Ed., ser. Lecture Notes in Computer Science, vol. 3385, Springer, 2005, pp. 147–163, ISBN: 3-540-24297-X. DOI: 10.1007/978-3-540-30579-8_11. [Online]. Available: https://doi.org/10.1007/978-3-540-30579-8%5C_11.

[77] Z. Kincaid, J. Cyphert, J. Breck, and T. W. Reps, "Non-linear reasoning for invariant synthesis," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 54:1–54:33, 2018. DOI: 10.1145/3158142. [Online]. Available: https://doi.org/10.1145/3158142.

[78] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds., ser. Lecture Notes in Computer Science, vol. 2404, Springer, 2002, pp. 78–92. DOI: 10.1007/3-540-45657-0_7. [Online]. Available: https://doi.org/10.1007/3-540-45657-0%5C_7.

[79] S. K. Lahiri and S. A. Seshia, "The UCLID decision procedure," in *Computer Aided Verification*, R. Alur and D. Peled, Eds., ser. Lecture Notes in Computer Science, vol. 3114, Springer, 2004, pp. 475–478. DOI: 10.1007/978-3-540-27813-9_40. [Online]. Available: https://doi.org/10.1007/978-3-540-27813-9%5C_40.

[80] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005. DOI: 10.1145/1066100.1066102. [Online]. Available: https://doi.org/10.1145/1066100.1066102.

[81] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, "A decision procedure for an extensional theory of arrays," in *IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, 2001, pp. 29–37.

[82] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani, "A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., ser. Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 547–560. DOI: 10.1007/978-3-540-73368-3_54. [Online]. Available: https://doi.org/10.1007/978-3-540-73368-3%5C_54.

[83] S. Gulwani, A. Tiwari, and G. C. Necula, "Join algorithms for the theory of uninterpreted functions," in *Foundations of Software Technology and Theoretical Computer Science*, K. Lodaya and M. Mahajan, Eds., ser. Lecture Notes in Computer Science, vol. 3328, Springer, 2004, pp. 311–323, ISBN: 3-540-24058-6. DOI: 10.1007/978-3-540-30538-5_26. [Online]. Available: https://doi.org/10.1007/978-3-540-30538-5%5C_26.

[84] S. Gulwani and A. Tiwari, "Combining abstract interpreters," in *Programming Language Design and Implementation*, M. I. Schwartzbach and T. Ball, Eds., ACM, 2006, pp. 376–386, ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1134026. [Online]. Available: https://doi.org/10.1145/1133981.1134026.

[85] R. Jhala, A. Podelski, and A. Rybalchenko, "Predicate abstraction for program verification," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Springer, 2018, pp. 447–491, ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8_15. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_15.

[86] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Communications of the ACM*, vol. 54, no. 7, pp. 68–76, 2011.

[87] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Symposium on Principles of Programming Languages*, J. Launchbury and J. C. Mitchell, Eds., ACM, 2002, pp. 58–70. DOI: 10.1145/503272.503279. [Online]. Available: https://doi.org/10.1145/503272.503279.

[88] T. Ball and S. Rajamani, "Boolean programs: A model and process for software analysis," Tech. Rep. MSR-TR-2000-14, Feb. 2000, p. 29. [Online]. Available: https://www.microsoft.com/en-us/research/publication/boolean-programs-a-model-and-process-for-software-analysis/.

[89] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for boolean programs," in *SPIN Model Checking and Software Verification Workshop*, K. Havelund, J. Penix, and W. Visser, Eds., ser. Lecture Notes in Computer Science, vol. 1885, Springer, 2000, pp. 113–130, ISBN: 3-540-41030-9. DOI: 10.1007/10722468_7. [Online]. Available: https://doi.org/10.1007/10722468%5C_7.

[90] T. Ball, T. D. Millstein, and S. K. Rajamani, "Polymorphic predicate abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 2, pp. 314–343, 2005.

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

[92] 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*, T. Margaria and W. Yi, Eds., ser. Lecture Notes in Computer Science, vol. 2031, Springer, 2001, pp. 268–283, ISBN: 3-540-41865-2. DOI: 10.1007/3-540-45319-9_19. [Online]. Available: https://doi.org/10.1007/3-540-45319-9_19.

[93] K. L. McMillan, "Lazy abstraction with interpolants," in *Computer Aided Verification*, T. Ball and R. B. Jones, Eds., ser. Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 123–136, ISBN: 3-540-37406-X. DOI: 10.1007/11817963_14. [Online]. Available: https://doi.org/10.1007/11817963%5C_14.

[94] A. Gurfinkel and M. Chechik, “Why waste a perfectly good abstraction?” In *Tools and Algorithms for the Construction and Analysis of Systems*, H. Hermanns and J. Palsberg, Eds., ser. Lecture Notes in Computer Science, vol. 3920, Springer, 2006, pp. 212–226, ISBN: 3-540-33056-9. DOI: `10.1007/11691372\_14`. [Online]. Available: `https://doi.org/10.1007/11691372%5C_14`.

[95] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, “Ufo: A framework for abstraction- and interpolation-based software verification,” in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds., ser. Lecture Notes in Computer Science, vol. 7358, Springer, 2012, pp. 672–678. DOI: `10.1007/978-3-642-31424-7\_48`. [Online]. Available: `https://doi.org/10.1007/978-3-642-31424-7%5C_48`.

[96] N. Bjørner and A. Gurfinkel, “Property directed polyhedral abstraction,” in *Verification, Model Checking, and Abstract Interpretation*, D. D’Souza, A. Lal, and K. G. Larsen, Eds., ser. Lecture Notes in Computer Science, vol. 8931, Springer, 2015, pp. 263–281, ISBN: 978-3-662-46080-1. DOI: `10.1007/978-3-662-46081-8_15`. [Online]. Available: `https://doi.org/10.1007/978-3-662-46081-8_15`.

[97] 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, 7:1–7:33, Mar. 2017, ISSN: 0004-5411. DOI: `10.1145/3022187`. [Online]. Available: `http://doi.acm.org/10.1145/3022187`.

[98] 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, ISBN: 978-3-9815370-2-4. DOI: `10.7873/DATE.2014.127`.

[99] A. Albarghouthi, A. Gurfinkel, and M. Chechik, “Whale: An interpolation-based algorithm for inter-procedural verification,” in *Verification, Model Checking, and Abstract Interpretation*, V. Kuncak and A. Rybalchenko, Eds., ser. Lecture Notes in Computer Science, vol. 7148, Springer, 2012, pp. 39–55, ISBN: 978-3-642-27939-3. DOI: `10.1007/978-3-642-27940-9\_4`. [Online]. Available: `https://doi.org/10.1007/978-3-642-27940-9%5C_4`.

[100] K. L. McMillan, “Interpolation and model checking,” in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Springer, 2018, pp. 421–446, ISBN: 978-3-319-10574-1. DOI: `10.1007/978-3-319-10575-8\_14`. [Online]. Available: `https://doi.org/10.1007/978-3-319-10575-8%5C_14`.

[101] L. Benalycherif and A. McIsaac, “A semantic condition for data independence and applications in hardware verification,” *Electron. Notes Theor. Comput. Sci.*, vol. 250, no. 1, pp. 39–54, 2009. DOI: `10.1016/j.entcs.2009.08.004`. [Online]. Available: `https://doi.org/10.1016/j.entcs.2009.08.004`.

[102] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *International Symposium on Software Testing and Analysis*, P. G. Frankl, Ed., ACM, 2002, pp. 218–228, ISBN: 1-58113-562-9. DOI: `10.1145/566172.566212`. [Online]. Available: `https://doi.org/10.1145/566172.566212`.

[103] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn verification framework," in *Computer Aided Verification*, D. Kroening and C. S. Pasareanu, Eds., ser. Lecture Notes in Computer Science, vol. 9206, Springer, 2015, pp. 343–361, ISBN: 978-3-319-21689-8.

[104] Z. Rakamaric and M. Emmi, "SMACK: decoupling source language details from verifier implementations," in *Computer Aided Verification*, 2014, pp. 106–113.

[105] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds., ser. Lecture Notes in Computer Science, vol. 2988, Springer, 2004, pp. 168–176.

[106] L. C. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Trans. Software Eng.*, vol. 38, no. 4, pp. 957–974, 2012. DOI: `10.1109/TSE.2011.59`. [Online]. Available: `https://doi.org/10.1109/TSE.2011.59`.

[107] W. Wang, C. Barrett, and T. Wies, "Cascade 2.0," in *Verification, Model Checking, and Abstract Interpretation*, K. L. McMillan and X. Rival, Eds., ser. Lecture Notes in Computer Science, vol. 8318, Springer, 2014, pp. 142–160, ISBN: 978-3-642-54012-7. DOI: `10.1007/978-3-642-54013-4_9`. [Online]. Available: `https://doi.org/10.1007/978-3-642-54013-4_9`.

[108] D. Bueno and K. Sakallah, "Horn2VMT: Translating horn reachability into transition systems," in *Workshop on Horn Clauses for Verification and Synthesis*, 2020, To appear.

[109] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, Eds., ser. Lecture Notes in Computer Science, vol. 9300, Springer, 2015, pp. 24–51. DOI: `10.1007/978-3-319-23534-9\_2`.

[110] P. Rümmer, H. Hojjat, and V. Kuncak, "Classifying and solving horn clauses for verification," in *Verified Software: Theories, Tools, Experiments*, E. Cohen and A. Rybalchenko, Eds., ser. Lecture Notes in Computer Science, vol. 8164, Springer, 2013, pp. 1–21, ISBN: 978-3-642-54107-0. DOI: `10.1007/978-3-642-`

54108-7_1. [Online]. Available: https://doi.org/10.1007/978-3-642-54108-7%5C_1.

[111] D. Kapur, R. Majumdar, and C. G. Zarba, "Interpolation for data structures," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, M. Young and P. T. Devanbu, Eds., ACM, 2006, pp. 105–116. DOI: 10.1145/1181775.1181789. [Online]. Available: https://doi.org/10.1145/1181775.1181789.

[112] A. R. Bradley, Z. Manna, and H. B. Sipma, "What's decidable about arrays?" In *Verification, Model Checking, and Abstract Interpretation*, E. A. Emerson and K. S. Namjoshi, Eds., ser. Lecture Notes in Computer Science, vol. 3855, Springer, 2006, pp. 427–442, ISBN: 3-540-31139-4. DOI: 10.1007/11609773_28. [Online]. Available: https://doi.org/10.1007/11609773%5C_28.

[113] H. Hojjat and P. Rümmer, "The ELDARICA horn solver," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. Bjørner and A. Gurfinkel, Eds., IEEE, 2018, pp. 1–7, ISBN: 978-0-9835678-8-2. DOI: 10.23919/FMCAD.2018.8603013. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8603013.

[114] N. Bjørner, K. L. McMillan, and A. Rybalchenko, "Program verification as satisfiability modulo theories," in *Workshop on Satisfiability Modulo Theories*, P. Fontaine and A. Goel, Eds., ser. EPiC Series in Computing, vol. 20, EasyChair, 2012, pp. 3–11. [Online]. Available: https://easychair.org/publications/paper/qGkT.

[115] L. M. de Moura and N. Bjørner, "Generalized, efficient array decision procedures," in *Formal Methods in Computer-Aided Design*, IEEE, 2009, pp. 45–52, ISBN: 978-1-4244-4966-8. DOI: 10.1109/FMCAD.2009.5351142. [Online]. Available: https://doi.org/10.1109/FMCAD.2009.5351142.

[116] D. Kapur and C. G. Zarba, "A reduction approach to decision procedures," University of New Mexico, Tech. Rep., 2005.

[117] A. Goel, S. Krstić, and A. Fuchs, "Deciding array formulas with frugal axiom instantiation," in *International Workshop on Satisfiability Modulo Theories*, ser. SMT '08, Princeton, New Jersey, USA: ACM, 2008, pp. 12–17, ISBN: 978-1-60558-440-9. DOI: 10.1145/1512464.1512468. [Online]. Available: http://doi.acm.org/10.1145/1512464.1512468.

[118] B. Dutertre and L. M. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, T. Ball and R. B. Jones, Eds., ser. Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 81–94, ISBN: 3-540-37406-X. DOI: 10.1007/11817963_11. [Online]. Available: https://doi.org/10.1007/11817963%5C_11.

[119] N. Suzuki and D. Jefferson, "Verification decidability of presburger array programs.," CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1977.

[120] J. Jaffar, "Presburger arithmetic with array segments," *Inf. Process. Lett.*, vol. 12, no. 2, pp. 79–82, 1981. DOI: 10.1016/0020-0190(81)90007-7. [Online]. Available: https://doi.org/10.1016/0020-0190(81)90007-7.

[121] C. Lynch and B. Morawska, "Automatic decidability," in *IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, 2002, p. 7, ISBN: 0-7695-1483-9. DOI: 10.1109/LICS.2002.1029813. [Online]. Available: https://doi.org/10.1109/LICS.2002.1029813.

[122] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz, "New results on rewrite-based satisfiability procedures," *ACM Trans. Comput. Log.*, vol. 10, no. 1, 4:1–4:51, 2009. DOI: 10.1145/1459010.1459014. [Online]. Available: https://doi.org/10.1145/1459010.1459014.

[123] J. Christ and J. Hoenicke, "Weakly equivalent arrays," in *International Symposium on Frontiers of Combining Systems*, ser. FroCoS 2015, Wroclaw, Poland: Springer-Verlag, 2015, pp. 119–134, ISBN: 978-3-319-24245-3.

[124] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., ser. Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 519–531.

[125] R. Brummayer and A. Biere, "Lemmas on demand for the extensional theory of arrays," *JSAT*, vol. 6, no. 1-3, pp. 165–201, 2009. [Online]. Available: https://satassociation.org/jsat/index.php/jsat/article/view/74.

[126] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "A write-based solver for SAT modulo the theory of arrays," in *Formal Methods in Computer-Aided Design*, A. Cimatti and R. B. Jones, Eds., IEEE, 2008, pp. 1–8. DOI: 10.1109/FMCAD.2008.ECP.18. [Online]. Available: https://doi.org/10.1109/FMCAD.2008.ECP.18.

[127] P. Habermehl, R. Iosif, and T. Vojnar, "What else is decidable about integer arrays?" In *Foundations of Software Science and Computational Structures*, R. M. Amadio, Ed., ser. Lecture Notes in Computer Science, vol. 4962, Springer, 2008, pp. 474–489.

[128] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds., Prentice-Hall, 1981, ch. 7, pp. 189–233.

[129] C. Lattner, A. Lenharth, and V. S. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proceedings of the Conference on Programming Language Design and Implementation*, J. Ferrante and K. S. McKinley, Eds., ACM, 2007, pp. 278–289. DOI: `10.1145/1250734.1250766`. [Online]. Available: `http://doi.acm.org/10.1145/1250734.1250766`.

[130] A. Gurfinkel and J. A. Navas, "A context-sensitive memory model for verification of C/C++ programs," in *Static Analysis Symposium*, F. Ranzato, Ed., ser. Lecture Notes in Computer Science, vol. 10422, Springer, 2017, pp. 148–168, ISBN: 978-3-319-66705-8. DOI: `10.1007/978-3-319-66706-5_8`. [Online]. Available: `https://doi.org/10.1007/978-3-319-66706-5_8`.

[131] J. L. Hintze and R. D. Nelson, "Violin plots: A box plot-density trace synergism," *The American Statistician*, vol. 52, no. 2, pp. 181–184, 1998. DOI: `10.1080/00031305.1998.10480560`.

[132] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *Programming Language Design and Implementation*, M. W. Hall and D. A. Padua, Eds., ACM, 2012, pp. 405–416. DOI: `10.1145/2254064.2254112`. [Online]. Available: `https://doi.org/10.1145/2254064.2254112`.

[133] K. L. McMillan, "Lazy annotation revisited," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 243–259, ISBN: 978-3-319-08866-2. DOI: `10.1007/978-3-319-08867-9\_16`. [Online]. Available: `https://doi.org/10.1007/978-3-319-08867-9%5C_16`.

[134] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski, "Ultimate automizer with smtinterpol - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Piterman and S. A. Smolka, Eds., ser. Lecture Notes in Computer Science, vol. 7795, Springer, 2013, pp. 641–643, ISBN: 978-3-642-36741-0. DOI: `10.1007/978-3-642-36742-7\_53`. [Online]. Available: `https://doi.org/10.1007/978-3-642-36742-7%5C_53`.

[135] P. Rümmer, H. Hojjat, and V. Kuncak, "Disjunctive interpolants for horn-clause verification," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds., ser. Lecture Notes in Computer Science, vol. 8044, Springer, 2013, pp. 347–363. DOI: `10.1007/978-3-642-39799-8\_24`. [Online]. Available: `https://doi.org/10.1007/978-3-642-39799-8%5C_24`.

[136] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Symposium on Principles of Programming Languages*, R. K. Cytron and P. Lee, Eds., ACM Press, 1995, pp. 49–61.

[137] A. Lal and S. Qadeer, "DAG inlining: A decision procedure for reachability-modulo-theories in hierarchical programs," in *Programming Language Design and Implementation*, D. Grove and S. Blackburn, Eds., ACM, 2015, pp. 280–290, ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737987. [Online]. Available: https://doi.org/10.1145/2737924.2737987.

[138] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds., ser. Lecture Notes in Computer Science, vol. 7358, Springer, 2012, pp. 427–443. DOI: 10.1007/978-3-642-31424-7_32. [Online]. Available: https://doi.org/10.1007/978-3-642-31424-7%5C_32.

[139] J. Wagner, V. Kuznetsov, and G. Candea, "-OVERIFY: Optimizing programs for fast verification," in *Hot Topics in Operating Systems*, P. Maniatis, Ed., USENIX Association, 2013. [Online]. Available: https://www.usenix.org/conference/hotos13/session/wagner.

[140] 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. DOI: 10.1017/S1471068415000204. [Online]. Available: https://doi.org/10.1017/S1471068415000204.

[141] N. Sharygina and H. Veith, Eds., vol. 8044, Lecture Notes in Computer Science, Springer, 2013.

[142] T. Ball and R. B. Jones, Eds., vol. 4144, Lecture Notes in Computer Science, Springer, 2006, ISBN: 3-540-37406-X. DOI: 10.1007/11817963. [Online]. Available: https://doi.org/10.1007/11817963.

[143] P. Madhusudan and S. A. Seshia, Eds., *Computer Aided Verification*, vol. 7358, Lecture Notes in Computer Science, Springer, 2012.

[144] R. Kaivola and T. Wahl, Eds., *Formal Methods in Computer-Aided Design*, IEEE, 2015, ISBN: 978-0-9835678-5-1.

[145] J. Launchbury and J. C. Mitchell, Eds., *Symposium on Principles of Programming Languages*, ACM, 2002.

[146] E. A. Emerson and A. P. Sistla, Eds., *Computer Aided Verification*, vol. 1855, Lecture Notes in Computer Science, Springer, 2000.

[147] *Formal Methods in Computer-Aided Design*, IEEE, 2009, ISBN: 978-1-4244-4966-8.

[148] W. Damm and H. Hermanns, Eds., *Computer Aided Verification*, vol. 4590, Lecture Notes in Computer Science, Springer, 2007.

[149] A. Biere and R. Bloem, Eds., *Computer Aided Verification*, vol. 8559, Lecture Notes in Computer Science, Springer, 2014, ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9.