

Evaluating control-flow based inductive model  
checking algorithms

RWTH Aachen

Sascha Müller

September 30, 2015



## **Abstract**

IC3 is an incremental algorithm constructing sets of over-approximations of the reachable state space by generating Boolean clauses that are inductive relative to reachability information available within one step. It is fully symbolic and does not require the unwinding of the transition relation unlike many other model checking algorithms. In return it cannot exploit the control-flow information induced by a program. TreeIC3 is an algorithm that remedies this, but reintroduces unwinding.

We present an implementation of the TreeIC3 algorithm and propose further improvements to it by introducing various strategies. Both IC3 and TreeIC3 use a process called generalization to drop irrelevant information in learned clauses. As the major computational effort lies in this procedure, we especially focus on optimizing this computation.

I hereby certify that this master thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged. It has not been accepted in any previous application for a degree.

Aachen, September 30, 2015

---

Sascha Müller

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Preliminaries</b>	<b>6</b>
2.1	Logic . . . . .	6
2.2	Guarded Command Language . . . . .	8
2.3	Control-Flow . . . . .	10
2.4	Model Checking Problem . . . . .	11
<b>3</b>	<b>Framework</b>	<b>12</b>
<b>4</b>	<b>IC3</b>	<b>14</b>
4.1	The Algorithm . . . . .	16
4.2	Generalization . . . . .	19
4.3	SMT . . . . .	22
<b>5</b>	<b>Original TreeIC3</b>	<b>23</b>
5.1	Blocking Phase . . . . .	25
5.2	Propagation . . . . .	27
<b>6</b>	<b>Improvements and Implementation</b>	<b>30</b>
6.1	Predecessor Computation . . . . .	31
6.2	Generalization . . . . .	32
6.2.1	Generalization Contexts . . . . .	33
6.2.2	Syntactical Analysis . . . . .	35
6.2.3	Generalization Pushing . . . . .	36
6.2.4	Using Interpolants . . . . .	39
6.2.5	Evaluation . . . . .	41
6.3	Search Strategies . . . . .	42

6.4	Blocking Strategies . . . . .	45
6.5	Implementational Details . . . . .	48
6.6	Comparison . . . . .	49
<b>7</b>	<b>Conclusion</b>	<b>51</b>

# Chapter 1

## Introduction

Model Checking is the algorithmic analysis of systems to prove properties about their executions, providing a conceptual framework to formalize the fundamental questions and algorithmic procedures to solve them. Model Checking is employed in a wide array of fields, especially for verifying correctness of hardware systems (hardware model checking) and software systems (software model checking).

IC3 [2] is a novel model checking technique for verifying invariant properties of finite state transition systems, introduced originally in the context of hardware model checking and has proven itself to provide impressive performance. It is an incremental algorithm constructing sets of over-approximations of the reachable state space by generating Boolean clauses that are inductive relative to reachability information available within one step. Differing from model checking techniques such as bounded model checking or interpolation based model checking, IC3 abandons the paradigm of unwinding the transition relation and works in a purely symbolic setting. Furthermore, IC3 relies on aggressive use of SAT-solving, giving the algorithm full access to the power of modern state-of-the-art SAT solvers.

Cimatti *et al.* [4] propose a lifting of IC3 onto the context of software model checking. Since there we typically have to deal with infinite transition systems, a richer modelling technique for symbolic representation is required. Cimatti proposed a lifting of the purely Boolean IC3 to IC3 using satisfiability modulo theories, allowing to model programs in a symbolic transition system using first order formulae. Additionally, they extend the IC3 algorithm to the TreeIC3 algorithm which returns to the unwinding paradigm but in return enriches IC3 by the capability of exploiting information induced by

the control-flow of a program. TreeIC3 works by unwinding a program into an abstract reachability tree where each node is associated with a program location, an abstract state formula and an identifier, effectively providing a disjunctive partitioning of the program. Refutations of individual counterexamples then have access to the concrete path information and the solver is exposed to smaller formulae due to the partitioning.

Both IC3 and TreeIC3 rely on techniques for dropping irrelevant symbolic information from the learned clauses. This process is referred to as *generalization* and takes up a majority of the computational effort in exchange for greatly increasing the “usefulness” of the generated clauses.

In this thesis, we present our implementation of the TreeIC3 algorithm and propose further improvements of it by evaluating various different strategies some of the working phases. In particular, we also propose optimization techniques to improve the efficiency of the generalization process, in particular by means of priorly computed generalizations.

The remainder of this thesis is organised as follows. We introduce and fix background notation as well as basic aspects required for the reading of this thesis in Chapter 2. In Chapter 3 we introduce the technical framework into which our implementation has been embedded. Chapter 4 and Chapter 5 formally introduce the IC3 and TreeIC3 algorithm respectively. Finally, in Chapter 6 we present our own implementation and improvements to the existing TreeIC3 algorithm. There, we also evaluate our results and compare them to state-of-the art software model checking. We conclude the thesis in Chapter 7 by summarizing our results.

# Chapter 2

## Preliminaries

In this section we fix preliminary notation and introduce basic concepts as well as further background information required for reading this thesis.

### 2.1 Logic

We introduce some background information regarding propositional logic, first-order logic (FO) and satisfiability (SAT) modulo theories (SMT).

We denote logical formulae with  $\varphi, \psi, \dots$ , sets of logical formulae with  $\Psi, \Phi, \dots, I, T, P$ , variables with  $x, y, \dots$  and sets of variables by  $X, Y, \dots$ .

Throughout this section let  $\tau$  be the FO vocabulary. The *atoms* of FO-formulae are statements of the form  $x = y$ ,  $R(x_1, x_2, \dots, x_n)$  for some  $n$ -ary predicate  $R \in \tau$  and  $f(x_1, \dots, x_n) = y$  for some  $n$ -ary function  $f \in R$ . We also refer to 0-arity predicates as Boolean variables and to 0-arity functions as *theory variables*. Arbitrary FO-formulae using vocabulary  $\tau$  are formed by applying Boolean connectives  $\neg$  (negation),  $\wedge$  (conjunction),  $\vee$  (disjunction), and quantifiers  $\exists x$  (existential),  $\forall x$  (universal). A formula using only Boolean variables and Boolean connectives is a propositional (or Boolean) formula. We use the standard definitions for the shorthand notations of  $\rightarrow$  (implication) and  $\leftrightarrow$  (equivalence). Furthermore, we also apply the standard notions of logical consequence, theories, satisfaction, and validity using the model relation  $\models$ . Typically our formulae are quantifier-free, unless explicitly specified otherwise. A *literal* is an atom  $l$  or its negation  $\bar{l}$ . A *clause* is then a disjunction of literals  $l_1 \vee \dots \vee l_n$  and a *cube* a conjunction of literals  $l_1 \wedge \dots \wedge l_n$  for some  $n \in \mathbb{N}$ . We denote clauses and cubes by  $s, c, d, \dots$ . Moreover, for a

cube  $s = l_1 \wedge \dots \wedge l_n$  we denote the clause consisting of the negated literals with  $\neg s := \bar{l}_1 \vee \dots \vee \bar{l}_n$ . Analogously, we define  $\neg s$  for some clause  $s$ .

A formula is in conjunctive normal form (*CNF*) iff it is a conjunction of clauses and it is in disjunctive normal form (*DNF*) iff it is a disjunction of cubes. For a formula in CNF  $c_1 \wedge \dots \wedge c_n$  we also write  $\{c_1, \dots, c_n\}$  and  $c \in \varphi$  to say that  $c$  is a clause occurring in  $\varphi$ .

We write  $\varphi(X)$  to denote that  $X$  is the set of free variables occurring in the logical formula  $\varphi$ . Sometimes we also write  $\varphi(X_1, \dots, X_n)$  to denote the free variables in  $\varphi$ . A formula with no free variables is also called *sentence*. We assume that for each variable  $x$  there exists a *primed version*  $x'$ . With  $X'$  we denote the set of variables where each variable  $x \in X$  has been replaced by its primed version  $x'$  that is  $X' := \{x' | x \in X\}$ . Similarly, we write  $X^n$  to denote that we apply the priming operator  $n$  times to  $X$ . Let  $\varphi$  be some formula, then we write  $\varphi'$  to denote the formula obtained from  $\varphi$  by adding a prime to each variable occurring in  $\varphi$ . Likewise,  $\varphi^n$  denotes the  $n$  time application of this prime operator.

Initially, IC3 has been proposed in the context of propositional logic. However, as program variables take on values that are not purely Boolean but rather from a more complex domain we also require some background knowledge regarding SMT. The idea of SMT is to enrich a Boolean structure using a fragment of FO or in other words some FO theory  $T$ . We say that  $\varphi$  is *satisfiable modulo*  $T$  iff  $\{\varphi\} \cup T$  is satisfiable.

Program verification relevant SMT theories supported by most modern SMT solvers such as Z3 and Mathsat are for example:

- Linear Real Arithmetic (LRA) is the theory where we have the usual arithmetical functions  $+$ ,  $-$ ,  $\cdot$  which can be applied to variables and numerical constants. Furthermore, the theory supports the relations  $=, \leq, <$  defined in the usual manner.
- Bit-vector (BV) is the theory where we have bit-vectors of a specified length as objects and we can perform the usual arithmetic operations, as well as Boolean operations (such as bitwise and, or, extraction of single bits, etc) on them. Like in LRA we also have the usual relations  $=, \leq, <$ .

BV is also the theory we focus on as it allows modelling software on a bit level while also being decidable, but in general we aim for a framework where the underlying theory can be swapped out as one pleases.

## 2.2 Guarded Command Language

To analyze and model programs we use a simple programming language that allows not only for the usual program constructs but also allows for modelling non-deterministic behavior.

**Definition 1** (Syntax Guarded Command Language). The syntax of the Guarded Command Language (GCL) is defined inductively as:

- **Assume**  $b$  for any Boolean expression  $b$ .  
(Guaranteeing a condition  $b$ )
- $\mathbf{x} := a$  for any variable  $\mathbf{x}$  and expression  $a$ .  
(Variable assignment)
- $c_1; c_2$  if  $c_1, c_2 \in \text{GCL}$ .  
(Sequential execution of  $c_1$  and  $c_2$ )
- $c_1 \square c_2$  if  $c_1, c_2 \in \text{GCL}$ .  
(Non-deterministic choice of either  $c_1$  or  $c_2$ )

We call a *concrete program state* an assignment  $\sigma: \text{Var} \rightarrow \text{Dom}$  where  $\text{Var}$  denotes the set of variables and  $\text{Dom}$  the domain of said variables. We write  $a(\sigma)$  for some expression  $a$  to denote the evaluation result of  $a$  in state  $\sigma$ . With this we can define the semantics of GCL in an operational way as follows:

**Definition 2** (Execution relation of GCL). A GCL command  $c$  in the concrete state  $\sigma$  evaluates to the new state  $\sigma'$  iff  $\langle c, \sigma \rangle \rightarrow \sigma'$  whereas  $\rightarrow$  is the execution relation defined inductively as:

$$\begin{aligned}
 & \text{(assume)} \frac{b(\sigma) = \text{true}}{\langle \mathbf{assume} \ b, \sigma \rangle \rightarrow \sigma} \\
 & \text{(assign)} \frac{}{\langle \mathbf{x} := a, \sigma \rangle \rightarrow \sigma[x \mapsto a(\sigma)]} \\
 & \text{(seq)} \frac{\langle c_1, \sigma \rangle \rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \rightarrow \sigma''}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma''}
 \end{aligned}$$

$$\begin{aligned}
(\text{choice1}) \quad & \frac{\langle c_1, \sigma \rangle \rightarrow \sigma'}{\langle c_1 \sqcap c_2, \sigma \rangle \rightarrow \sigma'} \\
(\text{choice2}) \quad & \frac{\langle c_2, \sigma \rangle \rightarrow \sigma'}{\langle c_1 \sqcap c_2, \sigma \rangle \rightarrow \sigma'}
\end{aligned}$$

Observe that we can model the traditional `if a then b else c` construct using the GCL command

$$(\text{Assume } a; b) \sqcap (\text{Assume } \neg a; c)$$

**Example 1.** The following example models the loop body of a function computing  $n!$  given  $i \geq 0$

$$\text{Assume } i \geq 0; \text{ res} := \text{res} \cdot i; i := i + 1$$

The version of GCL introduced here does not support any loop constructs, but of course we also require the capability of modelling loops in an easy to analyze manner. We do this in the next section using Control-Flow Automata.

A *symbolic program state* or also called *abstract state formula* is a formula  $\varphi$  such that  $\sigma \models \varphi$ . Since there may also be other states  $\sigma'$  such that  $\sigma' \models \varphi$  we also say that  $\varphi$  is an *over-approximation*. Using the symbolic representation we can model program states in a way such that the SMT solver can work with them. Since the later algorithms all only process symbolic program states we also write program state or just state if it is clear from the context.

Furthermore, we usually consider the symbolic representation to be in DNF if not explicitly specified otherwise. In this manner we also use cubes to describe sets of concrete program states.

In order to analyze program behavior we also encode the semantics of a GCL term using logical formulae. For each GCL command  $c$  we associate a symbolic representation in form of an FO formula which we denote by  $T_c(X, X')$  with  $X$  modelling the variables of the current state and  $X'$  modelling the variables after executing  $c$ .  $T_c(X, X')$  then models the effect of  $c$  within a logical framework. We will leave out the subscript  $c$  if it is either clear from the context or not relevant.

## 2.3 Control-Flow

As introduced before, we model single loop-free execution sequences using the GCL language. In order to extend the framework to allow the modelling of more complex structures we use the notion of *control-flow automata*.

**Definition 3** (Control-Flow Automaton). A Control-Flow Automaton (CFA) is a tuple  $\mathcal{A} = (L, G)$  where

- $L$  contains the program locations.
- $G \subseteq L \times \text{GCL} \times L$  is the set of labelled control-flow edges modelling the program behavior in form of a GCL command when control flows from one location to another.

In this manner we can encode loops and other complex control-flow behavior in the graph structure of a CFA. Since we need a few more properties to describe programs and also include a notion of correctness we consider the following enriched structure.

**Definition 4** (Program). A Program is a tuple  $\Pi = (L, G, l_0, E)$  where  $(L, G)$  is a CFA,  $E$  is the set of all error locations and  $l_0$  is the initial location.

In theory one can also consider programs with multiple initial locations. However, as this can be easily modelled by a program with one unique initial location we only focus on this case with the aim of keeping the notation simple. We say that a program uses the *large-block encoding* if loop free parts are collapsed into a single GCL statement and that a GCL command is in *static single assignment* (SSA) if each variable is assigned to at most once. For our GCL labels we assume a large-block encoding collapsing only so many statements such that SSA is still upheld.

Using these objects we can finally define what a counterexample is and when a program is considered safe. A path  $\pi$  of length  $n$  is a sequence  $(l_0, c_0, l_1), \dots, (l_{n-1}, c_{n-1}, l_n)$  with  $(l_i, c_i, l_{i+1}) \in G$  for all  $i \leq n$  representing a walk through the CFA. With  $\pi$  we also associate the path formula  $\rho(\pi) := \bigwedge_{i \leq n} T_{c_i}^i(X^i, X^{i+1})$ . We call  $\pi$  *feasible* iff  $\rho(\pi)$  is satisfiable and otherwise we call it *spurious*. If  $l_n \in E$  then we call  $\pi$  a counterexample and if  $\pi$  is feasible then we call it a feasible counterexample and otherwise a spurious counterexample. For simplicity we also assume that all error locations are final (have no outgoing transitions). Finally, we call  $\mathcal{A}$  safe iff all counterexamples are spurious.

## 2.4 Model Checking Problem

In the model checking problem we want to verify whether a given *system* upholds a *property* or not.

**Definition 5** (Transition System).  $S := (X, I, t_1, \dots, t_n)$  is a (symbolic) transition system where  $X$  is a set of state variables,  $I(X)$  a symbolic state representing the initial states and  $t_i(X, X')$  are formulae symbolically representing the transition relation. If we only have a single transition, we denote it by  $T(X, X')$ .

By encoding a given program  $\Pi$  symbolically, we obtain the according transition system. A path in a transition system is a sequence  $\pi := \sigma_0, \dots, \sigma_n$  such that  $\sigma_0 \models I$  and for all  $0 \leq i \leq n$  we have  $\sigma_i, \sigma_{i+1} \models t_j$  for some  $j$ . The set of all paths of a system we denote by  $Paths(S)$ . It remains to consider a formal model of properties.

**Definition 6** (Property). A linear time property, or just property,  $P(X)$  is a set of paths describing expected system behavior.

A system  $S$  fulfills a property iff  $Paths(S) \subseteq P(X)$ , which we also denote by  $S \models P(X)$ . In the *model checking problem* we then want to determine whether  $S \models P(X)$ .

A *safety* property is a special kind of property stating that “some bad event never occurs”. Its dual is the notion of a *liveness* property. Here we state that “something good always eventually occurs”. In our case we are interested in properties describing the safety of a program, that is no error location  $l \in E$  is ever reached. In fact, for program safety as defined in the previous section it suffices to consider a special case of safety properties called *invariant* properties.

**Definition 7** (Invariant Property). An invariant property is a property  $P(X)$  such that there exists a formula  $\varphi(X)$  with

$$\forall \pi = \sigma_0 \dots \sigma_n: \pi \in P \Leftrightarrow \sigma_i \models \varphi \text{ for all } 0 \leq i \leq n$$

We also just write  $P(X)$  or  $P$  to denote the formula or a set of formulae representing the invariant.  $P(X)$  then represents the “good” states and  $\neg P(X)$  the “bad” states.

# Chapter 3

## Framework

Our TreeIC3 algorithm is implemented within an existing proprietary model checking framework. While our benchmarks and experiments are focused on C programs, the framework actually takes as input any file that can be compiled into the Low Level Virtual Machine (LLVM) intermediate representation (IR) [9], thus making our results applicable not only for one specific programming language but instead a wide array including for example C, Fortran and Haskell.

We transform the IR into our own intermediate verification code (IVL) suitable for program verification. This IVL code is then optimized using standard Static Analysis techniques such as expression propagation, needed

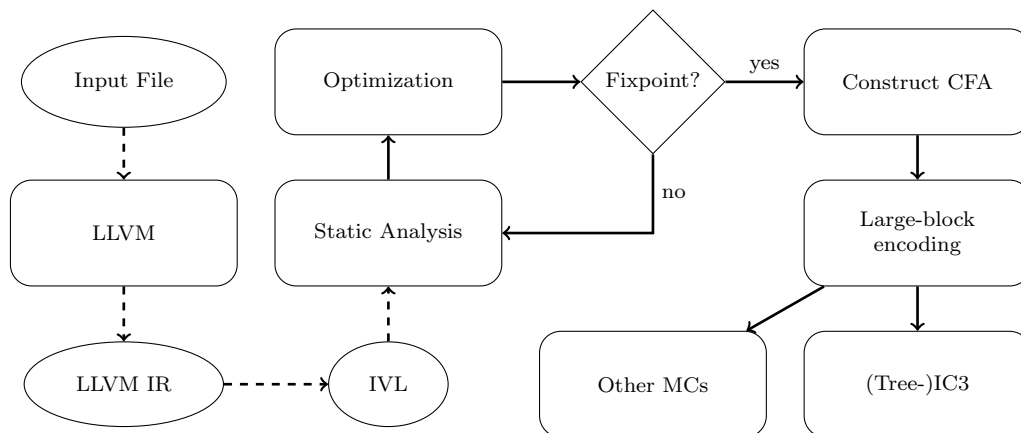


Figure 3.1: Workflow of the framework

variables analysis and program slicing. We use a bit-precise memory model supporting limited pointer operations such as array-element and record-field addressing. We also apply Steensgaard’s pointer analysis [11] to rewrite simple pointer expressions. Upon reaching a fixpoint of these optimizations, we construct a representation of the IVL program according to Def. 4 using CFAs with GCL labels. Additionally, we minimize the CFAs using bisimulation.

Fig. 3.1 summarizes the workflow of the framework. Other software model checking algorithms such as CEGAR and bounded model checking have also been implemented.

While the framework itself is theory unaware, we focus on bit-precise software model checking and therefore use BV theory. For the reasoning engine, the framework supports Z3<sup>1</sup> and MathSat<sup>2</sup>.

---

<sup>1</sup><https://github.com/Z3Prover/z3>

<sup>2</sup><http://mathsat.fbk.eu/>

# Chapter 4

## IC3

Originally proposed by Aaron Bradley [2] in the context of hardware model checking, **I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness, or short IC3, is a novel model checking algorithm for invariant properties. It has many appealing aspects when compared to other model checking techniques such as CEGAR, bounded model checking or interpolation based methods as it completely abandons the paradigm of unwinding the transition relation. Furthermore, it is a SAT based method and therefore capable of leveraging the power of modern state-of-the-art SAT solvers to the model checking problem. The key idea of IC3 is to construct proofs in an inductive manner and strengthening the induction hypothesis should the induction step fail. Consider the following program:

---

```
1:  $x := 1$ 
2: while * do
3:    $x := x + 1$ 
```

---

where \* represents some non-deterministic side-effect free function that we have otherwise no information about. Note that the example is already within the context of software model checking as this is the focal point of this thesis. For simplicity, however, we disregard bit-level issues such as overflows for this example and thus consider  $x$  to be a variable with infinite domain.

Say we want to prove the invariant  $x \geq 0$ . We can easily see that this invariant holds and that we can indeed apply induction to prove this.

- Initially, the invariant is guaranteed as we assign to  $x$  a positive value (base case)

- Assume that  $x \geq 0$  holds in some iteration (induction hypothesis IH)
- In each iteration of the while loop we monotonically increase the value of  $x$  by a positive integer and using the induction hypothesis we can conclude that indeed the invariant  $x \geq 0$  will also hold in the next iteration (induction step or consecution)

However, usually we have to deal with much more complex constructs than in the above example. In particular, we may have dependencies between variables. Consider the following modified program:

---

```

1: x := 1
2: y := 1
3: while * do
4:   x := x + y
5:   y := x + 1

```

---

We now increase  $x$  not by a constant but rather by a variable which is also modified in each iteration of the while loop. Observe that both  $x$  and  $y$  are dependent of each other. Of course, it is still easy to see that the invariant continues to hold but in order to prove it in an inductive manner we also have to extend the previous proof.

In fact, while we cannot prove  $x \geq 0$  using a direct inductive approach we can easily prove the **stronger** invariant  $x \geq 0 \wedge y \geq 0$ . Now we can reason that in each iteration  $x \geq 0$  due to the induction hypothesis. Therefore also  $y \geq 0$  in the current iteration as we are now guaranteed to increase  $y$  by positive values only. We have seen in these two examples that we can prove properties in more complex systems using induction by employing a stronger induction hypothesis. This leads us to the question that given some safety property  $P$  that is fulfilled in some system or program in our context:

Can we programatically strengthen  $P$  such that induction succeeds?

And the answer is, maybe surprisingly, that yes, we can! And not only that, we can even guide the strengthening procedure using counterexamples, meaning that we also only strengthen  $P$  by necessary formulae. To see how this strengthening process works, let us reconsider the while segment of the second program one more time in a more formal manner.

For our property we have  $P = \{x \geq 0\}$  and initially we have due to the assignments that  $I := \{x = 1, y = 1\}$  holds. Clearly, the base case  $I \Rightarrow P$

holds. Next we also encode the assignments in the while loop into this logical framework using  $T := \{\mathbf{x}' = \mathbf{x} + \mathbf{y} \wedge \mathbf{y}' = \mathbf{x}' + 1\}$ . If we can now show that  $P \wedge T \Rightarrow P'$  then induction succeeds.

Since we now have a purely logical framework encoding the induction proof, we can now ask a solver whether  $P \wedge T \Rightarrow P'$  holds. And in this particular case the answer would be **no**. Indeed, requesting a counter example from the solver would yield  $\mathbf{y} < 0$ . Observe that this is directly the negation of the formula  $\mathbf{y} \geq 0$ , which we used to strengthen the induction hypothesis. This kind of counterexample gives rise to a key notion of IC3: Counterexamples to induction or short CTIs. By systematically excluding CTIs we can construct the desired set of formulae. To see how this can be done we now fully introduce the IC3 algorithm.

## 4.1 The Algorithm

Without loss of generality we assume that all sets of formulae contain only clauses. Let  $X$  denote a set of variables and let  $S$  be some kind of system (in our context a program) symbolically represented by  $I(X)$  and  $T(X, X')$  where  $I(X)$  describes the initial state of the system and  $T(X, X')$  the transitions. Note that in the special case of programs additional information such as the program counter has to be encoded here as well. Additionally, let  $P(X)$  be an invariant property symbolically describing the set of “good” states. Likewise,  $\neg P(X)$  then represents the set of states that are “bad”. The aim is to show that we cannot reach  $\neg P(X)$  from  $I(X)$ .

As discussed previously this is done in an inductive manner by constructing a strengthening of  $P(X)$ , that we will call  $F(X)$ , capable of performing the induction step. More formally, we want to construct  $F(X)$  such that the following properties hold:

**Definition 8.**  $F(X)$  is relative inductive to  $P(X)$  iff

- $I(X) \models F(X)$  (base case) and
- $F(X) \wedge P(X) \wedge T(X, X') \models P(X')$  (induction step).

This construction constitutes the very core of IC3 and is non-trivial. The idea is to maintain a sequence of sets of formulae  $F_0(X), \dots, F_k(X)$  called frames where  $F_i(X)$  approximates the set of states reachable in at most  $i$  steps for  $i < k$  while still implying  $P(X)$ . Moreover,  $F_k(X)$  is the current set

of states that we consider reachable and that still has to be analyzed. We also refer to this sequence as **trace**. In other words, the frames have to fulfill the following properties:

- $F_0(X) = I(X)$  (we start with all initial states).
- $F_{i+1}(X) \subseteq F_i(X)$  (hence also  $F_i(X) \models F_{i+1}(X)$  and thus we monotonically increase the approximation).
- $F_i(X) \wedge T(X, X') \models F_{i+1}(X')$  ( $F_{i+1}(X')$  reachable from  $F_i(X)$ ).
- $F_i(X) \models P$  for all  $i < k$  ( $F_i(X)$  is a strengthening of  $P(X)$ ).

The algorithm constructs the trace by using two phases. The blocking phase and the propagation phase. Assume we have currently constructed the trace  $F_0(X), \dots, F_k(X)$ . The blocking phase now attempts to verify that we are still safe. In other words we cannot reach a bad state, i.e. we cannot find a CTI. Formally we can now exactly define what CTI is.

**Definition 9** (CTI). A cube  $c$  from which  $\neg P(X)$  is reachable is a counterexample to induction or short CTI on level  $i$  iff  $F_i(X) \wedge c(X) \wedge T(X, X') \wedge \neg P(X')$  is satisfiable.

If we detect a CTI then there are two possibilities:

- (I) The property is not fulfilled or
- (II) the over-approximations are too coarse.

The task of the blocking phase is to decide whether (I) or (II) is the case by backtracking the CTI and checking on each visited level  $i < k$  if the CTI is still reachable. If we can prove that a CTI  $c$  is in fact not reachable on a level  $i < k$  we add  $\neg c$  to  $F_i(X)$ . Otherwise we remember that we failed to block the CTI  $c$  and then try to block the predecessor states of  $c$  on level  $i - 1$ . In the following we leave out annotating the variable sets since they are clear from the context.

Figure 4.1 illustrates the situation and how  $P$ , the bad cube  $c$  and the frames  $F_0, \dots, F_k$  relate to each other. Note that in the illustration  $\neg P$  and  $F_k$  are disjoint but this is not necessarily the case.

To be efficient, the backtracking process is handled via a queue containing pairs  $(c, i)$  where  $c$  is the CTI and  $i$  the level on which we attempt to block it.

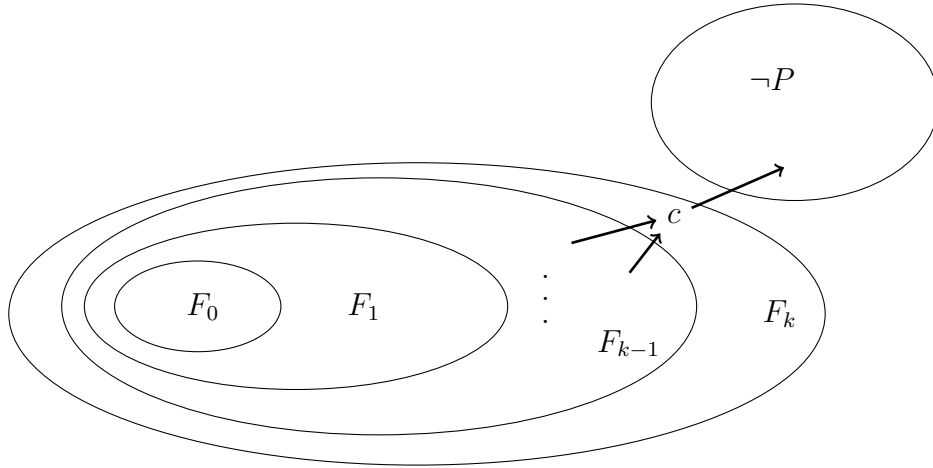


Figure 4.1: IC3

Blocking a CTI on level  $i$  means that we can show that  $F_{i-1} \wedge \neg c \wedge T \models \neg c'$  or in other words  $\neg c$  is inductive relative to  $F_{i-1}$ . We also write this condition as  $\text{relInd}(c, i, T) := F_{i-1} \wedge \neg c \wedge T \wedge c'$ . Taking everything together yields the following algorithm for the blocking phase.

---

**Algorithm 1** Blocking Phase

---

```

1: procedure BLOCK( $k, F_0, \dots, F_k, I, T, P$ )
2:    $\text{ctis} := \{(c, k-1) \in \neg P \mid F_k \wedge T \wedge c \text{ sat}\}$ 
3:    $\text{queue} := \text{Queue.create}(\text{ctis})$ 
4:   while ! $\text{queue.empty}()$  do
5:      $(c, i) := \text{queue.head}()$ 
6:     if  $i = 0$  and  $c \models I$  then return false
7:     if  $\text{relInd}(c, i, T)$  is sat then
8:        $\text{queue.add}(c, i-1)$ 
9:        $\text{queue.addAll}(\text{predecessors}(c, i, T), i-1)$ 
10:    else
11:       $F_i := F_i \cup \{\neg c\}$ 

```

---

Should we ever see a pair  $(c, 0)$  with  $c \models I$  then this means that we failed to block the CTI on all levels and we can return that the property is not fulfilled. Note that by analysing the backtracking process we can also obtain a counterexample path.

Using this approach we can systematically eliminate CTIs such that we either find a counterexample or  $F_k$  becomes relative inductive (i.e. we can no longer find a CTI).

In the second case we then proceed to the propagation phase. In this second phase we then “broaden the frontier” by extending the trace by a new frame  $F_{k+1}$ . The remaining task of the propagation phase is straightforward. The aim is to propagate the learned clauses, or in other words the negated CTIs, as far forward as possible to avoid future recomputations. A simple implementation can be obtained by checking on each level for each clause if we can push it forward to the next higher level frame. Finally, this phase is also used to check for termination. Should we discover two subsequent frames  $F_i, F_{i+1}$  for some  $i < k$  such that  $F_i = F_{i+1}$  then that means that we have already learned all the clauses required to exclude future CTIs and we can therefore safely stop the algorithm and return that the property is indeed satisfied.

---

**Algorithm 2** Propagation Phase

---

```

1: procedure PROPAGATE( $k, F_0, \dots, F_k, T$ )
2:   for  $i = 0; i < k; i++$  do
3:     for  $\text{clause} \in F_i$  do
4:       if  $\text{relInd}(\text{clause}, i, T)$  is unsat then
5:          $F_{i+1} := F_{i+1} \cup \{\text{clause}\}$ 
6:       if  $F_i = F_{i+1}$  then return true

```

---

Executing these two algorithms in alternation then yields the functionally complete IC3 algorithm.

## 4.2 Generalization

While the introduced IC3 algorithm is functional, there are many optimizations that can be done. One key optimization is the so called generalization that Bradely also introduces in his original paper [2]. In fact, when later considering the main TreeIC3 algorithm we will see that not using generalization will yield an algorithm that is noncompetitive with other model checking algorithms.

In order to discuss generalization we first need to take a closer look at a procedure we only briefly considered in the previous part, namely the compu-

tation of the predecessor cubes  $\text{predecessor}(c, i, T)$ . In the purely Boolean case this is easy. Since  $\text{relInd}(c, i)$  is satisfiable, we can request the solver to provide a model  $\mu$  for  $F_{i-1} \wedge \neg c \wedge T \wedge c'$  and then obtain a symbolic formula for the predecessors by dropping the primed variables in  $\mu$  [4]. Constructing the DNF then yields the desired set of cubes.

**Example 2.** Let  $c := x \wedge \bar{z}$  be a bad state,  $T := \{x' \leftrightarrow x \vee y, z' \leftrightarrow z \wedge y\}$  and  $F_{i-1} := \emptyset$ . This yields

$$\text{relInd}(c, i) = (\bar{x} \vee z) \wedge (x' \leftrightarrow x \vee y) \wedge (z' \leftrightarrow z \wedge y) \wedge x' \wedge \bar{z}'$$

From this we obtain the model  $\mu$  with  $\mu(x) = \text{false}$ ,  $\mu(y) = \text{true}$ ,  $\mu(z) = \text{false}$ ,  $\mu(z') = \text{false}$  and  $\mu(x') = \text{true}$  and by dropping  $x', z'$  this gives us the symbolic predecessor  $\text{predecessor}(c, i, T) = \{\bar{x} \wedge y \wedge \bar{z}\}$ .

In the above example we can see that the number of literals is increased through the predecessor computation. While the literals here are all necessary, it can occur that the procedure generates clauses containing literals that are in fact not relevant for the inductivity check and are therefore not strictly required. To see this, consider some strict subcube  $d \subsetneq c$  of a bad cube  $c$  and assume that we have  $\text{relInd}(d, i)$  is unsat. Then already adding  $\neg d$  to  $F_i$  will be sufficient to block the bad state  $c$  since also  $\neg d \subsetneq \neg c$  and hence  $\neg d \models \neg c$ . Literals only containing variables that do not occur either in the transition nor in  $F_i$  are simple such examples. But we can also have more complex literals depending on each other while still not being relevant for the inductivity check. Finding these in a systematical manner and eliminating such literals is the task of generalization. Writing this formally yields:

**Definition 10** (Generalization). Let  $c$  be a cube such that  $\text{relInd}(c, i)$  is unsat. Then a minimal subcube  $g \subseteq c$  is a generalization of  $c$  iff  $\text{relInd}(g, i)$  is also unsat.

Observe that such minimal subcubes are not necessarily unique [3]. Using this we can now also define when a literal is relevant.

**Definition 11** (Relevant Literals). Let  $c$  be a cube. Then a literal  $l \in c$  is relevant iff  $l \in g$  with  $g$  a minimal generalization of  $c$ .

A simple and straightforward algorithm presented in [7] proposes to simply iterate over all literals  $p \in c$  attempting to drop  $p$ . Should the resulting

cube  $c \setminus \{p\}$  still be relative inductive then we keep the new cube, otherwise we revert to  $c$ .

However, it is possible to perform potentially exponentially better by applying the algorithm proposed by Bradely in [3]. The key idea is that typically in “big” systems the number of relevant literals will be “small”.

Consider for example the case that we have  $n$  literals for some  $n \in \mathbb{N}$  with only one relevant literal  $l$ . Then we partition  $c$  into two disjoint cubes  $c_1 \cup c_2 = c$  such that  $||c_1| - |c_2|| \leq 1$ . Attempting to drop the entirety of  $c_1$  will then either reveal  $l \in c_1$  or that there is no relevant literal in  $c_1$ . The same applies if we attempt to drop  $c_2$ . By recursing this approach we obtain a method to find  $l$  in a logarithmic amount of steps not unlike a binary search.

In order to adapt the algorithm to deal with an arbitrary amount of relevant literals, we also need to deal with the case that there are indeed relevant literals in both  $c_1$  and in  $c_2$ . As we use this very algorithm also in the TreeIC3 implementation to deal with generalization we give an outline of it in pseudo-code.

---

**Algorithm 3** Generalization

---

```

1: procedure GENERALIZE( $c, i, T, \text{support}$ )
2:   if  $|c| = 1$  then return  $c$ 
3:   else
4:      $(l_0, r_0) := \text{split}(c)$ 
5:     if  $\text{relInd}(l_0 \cup \text{support}, i, T)$  is unsat then
6:        $\text{generalize}(l_0, i, T, \text{support})$ 
7:     else if  $\text{relInd}(r_0 \cup \text{support}, i, T)$  is unsat then
8:        $\text{generalize}(r_0, i, T, \text{support})$ 
9:     else
10:       $l := \text{generalize}(l_0, i, T, \text{support} \cup r_0)$ 
11:       $r := \text{generalize}(l_0, i, T, \text{support} \cup l)$ 
12:      return  $l \cup r$ 

```

---

Calling GENERALIZE with an initially empty support set then computes a minimal generalization of  $c$ .

## 4.3 SMT

Originally Bradley proposed IC3 using only Boolean state variables, a finite state system and propositional logic. However, a purely Boolean approach is often unsuited for modelling programs and as such for software model checking. It is more convenient to apply reasoning at a higher level of abstraction using SAT modulo theories (SMT). In this manner we can model the effects of assignments, expressions and so on in a straightforward manner using a fitting theory such as Linear Real Arithmetic (LRA) or Bit-vector logic (BV).

In [4] Cimatti *et al.* propose such a lifting of IC3 to support SMT. In fact, lifting IC3 to SMT is in most aspects straightforward. As IC3 itself reasons mostly on the level of literals, cubes and clauses we can simply replace the solver engine with an SMT solver and have the solver cope with the underlying theory.

There is however one operation where the transition to SMT is non-trivial. Namely the computation of the predecessor states  $\text{predecessor}(c)$ . The algorithm we considered earlier can now no longer be applied, as the state variables are encoded within the theory atoms [4].

Cimatti proposes a theory-dependent approach for dealing with this issue. For example, for theories admitting quantifier elimination we can quantify over the primed variables and then eliminate them using quantifier elimination. However, in the context of our framework we desire a solution that is theory independent. We will discuss this approach in the context of the main TreeIC3 algorithm.

# Chapter 5

## Original TreeIC3

One primary strength of IC3 is its independence of the unwinding paradigm and its capability of performing model checking by using a fully symbolic representation of the system.

However, this raises the question whether a complete abandonment of the unwinding paradigm might not be too radical as it would leave us without means of exploiting some properties of the program. Or to be more precise, since we are working with CFAs, exploiting the control-flow of the CFA.

In fact, when blocking a bad state on level  $k$  in IC3 we have no information about the path used to reach the bad state and therefore have to implicitly consider all paths of length of  $k$ . By combining IC3 with an unwinding based approach we do gain the knowledge of the exact path used to reach the bad state and can therefore block it more efficiently. This train of thought leads to the TreeIC3 algorithm originally proposed in [4].

In the following we present the individual building blocks making up the TreeIC3 algorithm.

The key idea is to unwind the CFA into an abstract reachability tree (ART) using a depth-first search (DFS) approach. For each node in the ART we memorize the corresponding CFA location, a unique identifier, and also a set of clauses describing properties the program state must have at this point. We also refer to this formula as the abstract state formula. Formalizing this notion of ARTs yields the following definition when given some input CFA.

**Definition 12** (Abstract reachability tree). An abstract reachability tree (ART) of a CFA  $\mathcal{A} = (L, G, E)$  is a labelled graph  $\mathcal{T} = (V, T)$  with

- $V$  containing vertices of the form  $(i, \varphi, l)$  where  $i$  is a unique identifier,

$l \in L$  a program location and  $\varphi$  a state formula describing logical properties that must definitely hold if the program has reached this point.

- $T \subseteq V \times \text{GCL} \times V$  being a set of labelled edges such that  $T$  defines a tree. Furthermore, if  $(k, \psi, m) \in V$  is a successor of  $(i, \varphi, l) \in V$  with edge label  $c$  we have that  $k > i$ ,  $(l, c, m) \in G$  and  $\varphi \wedge c \models \psi'$

Typically, IC3 would encode information  $c$  about a specific location  $l$  using a clause of the form  $\neg(x_{pc} = pc_l) \vee c$ . Since we now directly use the control flow information, unlike IC3, we do not have to encode it into the transition relation and therefore only use the clause  $c$ .

Also, instead of using one all-encompassing transition relation  $T$  we now use several transition relations  $t_e$ , each symbolically encoding the transition when using edge  $e$  in the CFA.

Therefore, in this setting we obtain a different criterion for checking for relative inductiveness. If we want to check at step  $i$  whether we can block a bad cube  $c$  we now have to verify that  $F_{i-1} \wedge t_{i-1} \models \neg c'$  holds, whereas  $t_{i-1}$  is the transition used at step  $i - 1$  and  $F_{i-1}$  is the clause set associated with the ART node visited at step  $i - 1$ . Analogously to `relInd(c, i, T)` we now obtain the TreeIC3 version.

$$\text{treeRelInd}(c, i) := F_{i-1} \wedge t_{i-1} \wedge c'$$

Here, we can see a major advantage of TreeIC3. While in IC3 each inductiveness check required the entire transition relation we are capable of focusing the solver only on the relevant part, namely the actually used transition. In this manner the SMT solver receives smaller and easier to handle formulae.

The algorithm roughly repeats the following steps:

- Unroll the CFA until an error location has been found and propagate learned clauses to new nodes. (Not explicitly considered in the original paper [4]. We refer to this as the Search Phase)
- Block the error location using an adapted TreeIC3 blocking algorithm. (Blocking Phase)
- Perform propagation. (Propagation Phase)

Additionally to the Blocking and the Propagation Phase, we now first find the paths to the error locations using a Search Phase. Instead of having one global trace as with IC3 we now have several traces, each for one path in the ART. Likewise, we now have a different termination criterion. Instead of closing one single trace, we now need to close each path on the ART individually. For this we need to define what it means to “close a path in the ART”.

**Definition 13.** Let  $\mathcal{T} = (V, T)$  be an ART and  $n = (l, \varphi, k) \in V$ . We say that  $n$  is covered if either

- (i) There exists another node  $n' = (l, \psi, h) \in V$  such that  $h < k$ ,  $\psi \models \varphi$  and  $n'$  is not covered.
- (ii)  $n$  has an (indirect) ancestor for which (i) holds.

We say that  $\mathcal{T}$  is safe iff no error location is reachable or more formally for all  $(i, \varphi, e) \in V$  with  $e \in E$  we have that  $\varphi \models false$ .

If a node  $n$  is covered by a node  $n'$  then  $n'$  can simulate  $n$  or in other words, whenever we could reach an error state from  $n$  we must also be capable of reaching it from  $n'$ . Therefore we can safely call a path that has a covered node closed. Observe that we do not allow a node to be covered by an already covered node. In this manner we avoid creating cycles in the covering relation.

Additionally, if a path ends in a node with the abstract state formula  $\varphi = false$  then we can also safely close this path as it is impossible to reach an error location from this node. Using these two criteria we obtain the termination condition that every leaf of the ART must either be covered or have a state formula equivalent to *false*.

## 5.1 Blocking Phase

The most interesting phase of this algorithm is certainly the Blocking Phase. Here we can use the additional information we possess by knowing the exact path to its utmost by blocking not just a bad state but more specifically blocking it along a specific path. In accordance to the original paper by Cimatti [4] we refer to this new blocking algorithm also as TreeIC3-Block-Path. The algorithm itself is very similar to the IC3 blocking algorithm, we mainly have to adjust relative inductivity checks and we have to construct the frames according to the error path. For the error path let  $\pi := (l_1, true, 1), \dots, (l_e, \varphi_n, m)$

be a path of length  $n$  for some  $n \in \mathbb{N}$  in the ART where  $l_1$  is the initial location and  $l_e$  the error location.

---

**Algorithm 4** TreeIC3 Blocking Phase

---

```

1: procedure TREEIC3-BLOCK-PATH( $\pi$ )
2:    $F := [\text{true}, \dots, \varphi_i, \dots, \varphi_n]$ 
3:    $\text{ctis} := \text{predecessors}(t_{n-1}, \text{true})$ 
4:    $\text{queue} := \text{Queue.create}(\text{ctis})$ 
5:   while ! $\text{queue.empty}()$  do
6:      $(c, i) := \text{queue.head}()$ 
7:     if  $i = 0$  then return false
8:     if  $\text{treeRelInd}(c, i)$  is sat then
9:        $\text{queue.add}(c, i-1)$ 
10:       $\text{queue.addAll}(\text{predecessors}(t_{i-1}, c), i-1)$ 
11:     else
12:        $g := \text{GENERALIZE}(c, i, t_{i-1}, [])$ 
13:        $F_i := F_i \cup \{\neg g\}$ 
14:   Check-Covered-Nodes( $\pi$ )

```

---

The procedure `Check-Covered-Nodes( $\pi$ )` checks for all nodes  $n$  on the path  $\pi$  whether all nodes considered covered by  $n$  are still covered by  $n$ . Observe that we have to perform this uncovering, as adding clauses to  $n$  reduces the number of concrete states possible at node  $n$ . Thus, the set of states of some node that was considered covered by  $n$  might no longer be covered.

Also, `predecessor( $t, c$ )` is the SMT adapted function for the predecessor computation of a cube  $c$  yielding all predecessor cubes  $c_i$  with respect to the transition  $t$ . This is also where we can exploit our now available knowledge of the concrete path. In IC3 we did not possess the additional information of the concrete transition  $t$  but had to use the complete transition table  $T$  during the predecessor computation. Using  $t$  we can focus only on paths that are relevant for blocking the bad cube. Note that even if we have a concrete path  $\pi$ , within our original program we might still have multiple paths as we are employing the large-block encoding. We will discuss the implementation of the `predecessor( $t, c$ )` function at a later point and more in-depth.

Since we also need to consider the step down from the global transition relation in the generalization procedure and call it using the according transition  $t$ , we also included the call to `GENERALIZE` in the listing of

**TreeIC3-Block-Path.** It should be noted that in **GENERALIZE** we now have to use the new relative inductiveness condition `treeRelInd(c, i)` as well.

Observe that outside of the changes required to adapt the Blocking Algorithm to the new framework of ARTs, we can indeed directly adapt the old IC3 Blocking Algorithm.

## 5.2 Propagation

We have seen that we can easily adapt the IC3 Blocking Algorithm into the new setting while also exploiting the now available information of the concrete paths. But what about Propagation? In IC3 propagating meant pushing a clause from frame  $F_i$  to frame  $F_{i+1}$ . Of course, this type of propagation is still valid but in order to increase the chance of coverage and thus the chance of successfully closing a path in the ART, it is not sufficient. Consider for example the following extract of some ART.

In the ART depicted in Fig. 5.1 we have found the error location  $l_e$  and successfully blocked the path, generating the clause  $x = 1$  at node 1 in the process. Expanding the node 2 then yields a node to which we cannot propagate  $x = 1$ . However, at node 3 we have again  $x = 1$ . But as the straightforward IC3 propagation algorithm fails to propagate  $x = 1$  to node 3 we cannot consider node 3 to be covered by node 1. In consequence, TreeIC3 is then forced into a non-terminating loop as we will find the error location

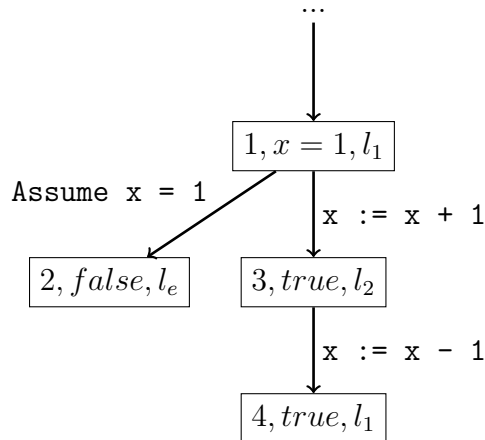


Figure 5.1: IC3 propagation in TreeIC3

$l_e$  again and again.

In order to combat this issue, Cimatti [4] proposed to complement the IC3 style propagation by a second type of propagation which we will refer to here as *path propagation* and which was originally considered in the context of lazy abstraction using interpolants in [10].

The idea is simple: Let  $n := (i, \varphi, l)$  be some leaf,  $n' := (k, \psi, l')$  some ancestor of  $n$  and  $\pi$  the path from  $n$  to  $n'$ . We then consider the set of clauses  $C_{n,n'}$  containing all clauses  $c \in \varphi$  but  $c \notin \psi$  and such that  $\varphi \wedge \rho(\pi) \models c$ . By adding  $C_{n,n'}$  to  $\psi$  we then successfully propagate all clauses that were valid in  $n$  and are still valid in  $n'$  to  $n'$ .

However, applying this idea directly is not sound. Consider for example the ART depicted in Fig. 5.2 where we leave out the edge labels for convenience. In the presented example we have initially discovered the error location  $l_e$  at node 3 but managed to block it at node 1, learning some clause  $c$  in the process. Further expansion reveals the node 4 for which we use path propagation to propagate  $c$  from 1 to 4, effectively covering 4. We then unwind until we find node 6. As we have  $\models \text{true} \rightarrow \text{true}$  we have that node 6 is covered by node 2. However, closing the path at this point is not sound as we have no guarantee that we can block  $l_e$  again.

The example also shows us the reason as to why the naive approach

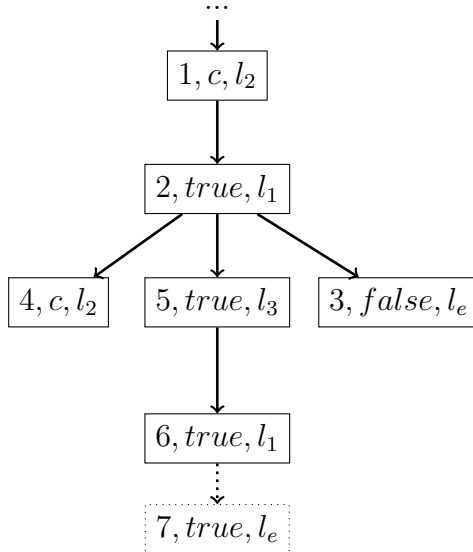


Figure 5.2: Naive path propagation

towards path propagation is flawed. Informally, node 2 has not learned the “reason” as to why  $c$  should hold at node 4. Should, however, node 2 learn this reason (in form of some clause  $d$ ) and if we can apply path propagation to propagate  $d$  from node 2 to node 6, then and only then, can we safely close the path at node 6.

This clause generation can be achieved by smartly applying the blocking algorithm **TreeIC3-Block-Path**. Instead of blocking an entire path, that is blocking all CTIs computed by  $\text{predecessors}(\varphi, \mathfrak{t}, \text{true})$ , we only block those states  $s$  such that  $s \models \neg C_{n,n'}$ . Or in other words, we initialize the queue of the blocking algorithm with cubes  $\text{predecessors}(\varphi, \mathfrak{t}, \neg C_{n,n'})$ . Putting everything together yields the final propagation algorithm.

---

**Algorithm 5** TreeIC3 Path Propagation

---

```

1: procedure STRENGTHEN( $\mathbf{n} = (1, \varphi, \mathbf{h})$ ,  $\mathbf{n}' = (1, \psi, \mathbf{h}')$ )
2:    $\pi := [\mathbf{n}, \dots, \mathbf{n}']$ 
3:    $\mathbf{m} := |\pi|$ 
4:    $\mathbf{C}_{\mathbf{n}, \mathbf{n}'} := \{c \mid c \in \varphi, \varphi \wedge \rho(\pi) \models c^m, \psi \not\models c\}$ 
5:   if  $\mathbf{C}_{\mathbf{n}, \mathbf{n}'} \neq \emptyset$  then
6:     TreeIC3-Block-Path( $\pi, \neg \mathbf{C}_{\mathbf{n}, \mathbf{n}'}$ )
7:      $\psi := \psi \cup \mathbf{C}_{\mathbf{n}, \mathbf{n}'}$ 
8:     Uncover-Covered-Nodes( $\mathbf{n}'$ )

```

---

We can observe that if  $\mathbf{n}'$  is a direct successor of  $\mathbf{n}$ , then the algorithm performs in the same manner as standard IC3 propagation. By applying **STRENGTHEN** systematically to each new node in the ART and whenever a node learns a new clause we obtain the propagation phase.

# Chapter 6

## Improvements and Implementation

While we have seen the general framework of the TreeIC3 algorithm, many important details including the predecessor computation have not been discussed yet. We discuss these aspects in this chapter in the context of our implementation of the TreeIC3 algorithm within our framework and also present various improvements and strategies for some of the phases of the algorithm.

For our reasoning engine we primarily use Z3, except for computing interpolants as this is not yet supported by Z3 for the theory of bit-vectors. Therefore, as interpolation engine we use MathSat.

We apply large-block encoding to collapse several program blocks into one GCL command in order to reduce the size of the CFAs.

We also evaluate the results compared to a state-of-the-art software model checker. As benchmarks we use the benchmark set used in the original TreeIC3 paper [4] enriched by the instances of the Bit-vector categories of the Software Model Checking competition<sup>1</sup> 2015 and additional benchmark instances for testing the correctness of our implementation. For our benchmark settings we consistently apply a timeout of 1800s and a memory limit of 4GB. The benchmarks are executed on a cluster of nodes with 2.1GHz cores.

---

<sup>1</sup><http://sv-comp.sosy-lab.org/2015/>

## 6.1 Predecessor Computation

While Cimatti [4] has considered TreeIC3 using LRA theory, our aim is to obtain an algorithm that is not dependent on the underlying theory and can thus in particular support the theory of Bit-vectors.

As discussed previously, employing SMT requires adjusting the algorithm for computing predecessors of abstract states. Cimatti proposes computing under-approximations of the predecessor states instead, blocking a path by repeated application of the path blocking algorithm. However, there is also another way to simplify the predecessor computation using approximations. Indeed, by using *weakest preconditions* following [12] we can obtain a safe over-approximation of the predecessors.

**Definition 14.** Let  $c$  be a GCL command and  $\varphi$  an abstract state formula. The weakest precondition (wp) of  $\varphi$  with respect to  $c$  is an abstract state formula  $\psi := wp\llbracket c, \varphi \rrbracket$  such that for any concrete states  $\sigma, \sigma'$  with  $\sigma' \models \varphi$  we have for all executions  $\langle c, \sigma \rangle \rightarrow \sigma'$  iff  $\sigma \models \psi$ .

In other words,  $wp\llbracket c, \varphi \rrbracket$  yields a formula symbolically encoding all the concrete states such that we can execute  $c$  and then obtain a concrete state covered by  $\varphi$ . This definition, however, yields a notion of weakest preconditions that is in a sense too strict for our GCL framework.

To see this, recall that the GCL language allows for nondeterministic constructs in the form of choice  $c_1 \sqcap c_2$ . According to Def. 14, the weakest precondition requires that for any execution starting in some state  $\sigma$  with  $\sigma \models wp\llbracket c_1 \sqcap c_2, \varphi \rrbracket$  we must also have for some  $\sigma'$  with  $\sigma' \models \varphi$ ,

$$\begin{aligned} &\langle c_1 \sqcap c_2, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle \rightarrow \sigma' \\ \text{and } &\langle c_1 \sqcap c_2, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle \rightarrow \sigma' \end{aligned}$$

However, this would imply for states  $\sigma$  such that  $\langle c_1, \sigma \rangle \rightarrow \sigma'$  but  $\langle c_2, \sigma \rangle \not\rightarrow \sigma'$  that  $\sigma \not\models wp\llbracket c_1 \sqcap c_2, \varphi \rrbracket$ . But clearly, in order to prove that we cannot arrive in the bad state  $\varphi$  we also must block such  $\sigma$ . This train of thought leads to a slightly modified version of the notion of a weakest precondition, the *weakest existential precondition*.

**Definition 15.** The weakest existential precondition (wep) of  $\varphi$  with respect to  $c$  is  $\psi := wep\llbracket c, \varphi \rrbracket$  such that for any concrete states  $\sigma, \sigma'$  with  $\sigma' \models \varphi$  we have that there exists some execution such that  $\langle c, \sigma \rangle \rightarrow \sigma'$  iff  $\sigma \models \psi$ .

Using this adjusted definition we can approximate the bad predecessor states that we require to be blocked. Furthermore, by applying the approach presented in [8] we can compute these preconditions efficiently while remaining ignorant of the underlying theory and even obtain quantifier-free formulae by employing the following construction:

$$\begin{aligned}
wep[[x := a, \varphi]] &:= \varphi[x \mapsto a] \\
wep[[\text{Assume } b, \varphi]] &:= \varphi \wedge b \\
wep[[c_1 \square c_2, \varphi]] &:= wep[[c_1, \varphi]] \vee wep[[c_2, \varphi]] \\
wep[[c_1; c_2, \varphi]] &:= wep[[c_1, wep[[c_2, \varphi]]]]
\end{aligned}$$

**Example 3.** Let  $c := \text{Assume } i \geq 0; \text{res} := \text{res} \cdot i; i := i + 1$ . The weakest existential precondition of  $c$  with respect to  $\varphi := \text{res} = n! \wedge i = n$  is

$$\begin{aligned}
wep[[c, \varphi]] &= wep[[\text{Assume } i \geq 0, \varphi[\text{res} \mapsto \text{res} \cdot i, i \mapsto i + 1]]] \\
&= wep[[\text{Assume } i \geq 0, \text{res} \cdot i = n! \wedge i + 1 = n]] \\
&= i \geq 0 \wedge \text{res} \cdot i = n! \wedge i + 1 = n
\end{aligned}$$

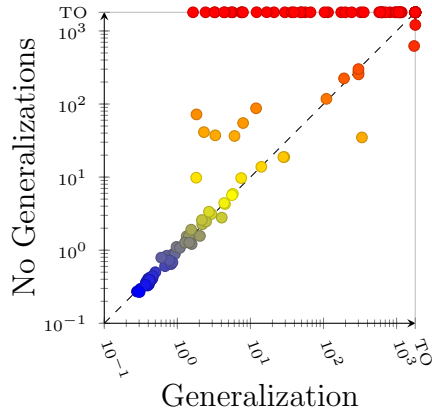
As with traditional IC3 we then obtain the individual bad states (bad cubes) by transforming the wep into DNF.

By setting  $\text{predecessor}(t_c, \varphi) := \text{DNF}(wep[[c, \varphi]])$  we then finally obtain the desired procedure.

## 6.2 Generalization

We have found in our experiments that in many benchmarks over 80% of the computation time is spent in the generalization procedure. However, despite this tremendous overhead, the performance boost gained by employing generalization makes this technique indispensable also in the TreeIC3 algorithm.

The benchmark results shown in Fig. 6.1 illustrate how standard generalization as presented in the previous chapter 5 fares within our implementation. The scatter plot shows the solving times of the generalizationless variant in relation to TreeIC3 using generalization; cutting off at our timeout (TO). Instances that went out of memory are not displayed in the plot. In the table we list for both approaches the number of solver instances (# solved) and the solving time used to solve them (solve  $t$ ). Note that timeouts are not included in the solving time.



Algorithm	# solved	solve $t$
No Generalization	71/178	470s
Generalization	110/178	17044s

Figure 6.1: Benchmarks of the generalization optimizations

We can observe, that not only are more instances solveable using generalization, we are also capable of solving most of the previously solveable instances in less time. Only in very few instances can we observe generalization slowing the algorithm down. As these results indicate, optimizing the generalization procedure is of utmost importance and one could even argue key to the performance of the TreeIC3 algorithm in general. In the following we discuss some techniques to optimize the process.

### 6.2.1 Generalization Contexts

Firstly, we consider the issue that often we have to compute generalizations of the same cube but in “different contexts”. Where a context is consists of the abstract state formula and the transition relative to which we generalize the cube, or more formally:

**Definition 16.** A generalization context (GC) is a tuple  $gc := (C, t)$  where  $C := \{c_1, \dots, c_n\}$  is an abstract state formula in CNF and  $t$  a symbolic transition.

For the generalization of some cube  $c$  with respect to some context  $gc$ , we modify the previous notation to adapt to this by changing the syntax of the

generalization call to  $g := \text{generalize}(c, gc)$ . We can then exploit previous computations to speedup the generalization of some cube we had already seen and that now has to be generalized in a context that is “stronger”. Consider the following example:

**Example 4.** Let  $c := l_1 \wedge l_2 \wedge l_3$  be some cube and  $gc := (\{c_1\}, t)$  a generalization context in which we encounter  $c$  such that for the generalization we obtain  $\text{generalize}(c, gc) = l_1$ .

Or in other words  $\text{treeRelInd}(g, c_1, t)$  is unsat. Assume  $c$  is encountered at a later time again within the context  $gc' := (\{c_1, c_2\}, t)$ . Then clearly

$$\begin{aligned} & \text{treeRelInd}(g, c_1, t) = c_1 \wedge t \wedge g' \text{ is unsat} \\ \Rightarrow & \text{treeRelInd}(g, c_1 \wedge c_2, t) = c_1 \wedge c_2 \wedge t \wedge g' \text{ is unsat} \\ \Rightarrow & \text{generalize}(c, gc') = l_1 \end{aligned}$$

We have thus computed  $\text{generalize}(c, gc') = l_1$  without having performed an additional SMT query. Indeed, we will see that if we have a stronger context, then we can drop at least the literals that were also dropable in the weaker context. To define what it means for a context to be stronger we use simple set inclusion.

**Definition 17.** Define the relation  $\sqsubseteq$  such that  $gc := (C, t) \sqsubseteq (C', t') =: gc'$  iff  $C \subseteq C'$  and  $t = t'$  for any  $gc, gc'$ .

Note that we are using a purely syntactical criterion, in contrast to semantically checking the implication using the query  $\models C \rightarrow C'$ , since we are attempting to avoid SMT calls.

**Lemma 1.** Let  $c$  be a cube and  $gc = (C, t), gc' = (C', t')$  generalization contexts. Then

$$gc \sqsubseteq gc' \Rightarrow \text{generalize}(c, gc') = \text{generalize}(\text{generalize}(c, gc), gc')$$

*Proof.* Assume  $gc \sqsubseteq gc'$ . Let further  $g_{gc} := \text{generalize}(c, gc)$  and also  $g_{gc'} := \text{generalize}(c, gc')$ . We then have

$$\begin{aligned} & \text{treeRelInd}(g_{gc}, C, t) = C \wedge t \wedge g'_{gc'} \text{ is unsat} \\ \Rightarrow & \text{treeRelInd}(g_{gc}, C', t) = C' \wedge t \wedge g'_{gc'} \text{ is unsat since } C \subseteq C' \\ \Rightarrow & \text{generalize}(c, gc') \subseteq \text{generalize}(c, gc) \text{ since } t = t' \\ \Rightarrow & \text{generalize}(c, gc') = \text{generalize}(\text{generalize}(c, gc), gc') \end{aligned}$$

by minimality and since  $\text{generalize}(c, gc) \subseteq c$ . □

Let now  $GC(c)$  denote the set of all contexts we have witnessed the cube  $c$  in until now. Define  $min(c, gc) := min\{gc' \mid gc' \sqsubseteq gc, gc' \in GC(c)\}$  to be the weakest context in which we have encountered  $c$ . We have then already computed  $g_{gc} := generalize(c, min(c, gc))$ .

Now we can substitute our previous generalization call by then instead calling  $generalize(c, gc') = generalize(g_{gc}, gc')$  and hence eliminating  $|c \setminus g_{gc}|$  literals without having to perform an SMT Query.

## 6.2.2 Syntactical Analysis

There are various small syntactical checks that we can perform to reduce the number of SMT calls. In particular, the following two have proven useful. For the following let  $c$  be a bad cube and  $gc = (C, t)$  the current generalization context.

Firstly, say we know some literal  $l$  to hold in  $C$  and  $t$  does not modify a variable occurring in  $l$ . To model this, we define the function  $prime(l, t)$  that replaces variables only mentioned in  $t$  by their primed version.

**Lemma 2.** Consider  $l \in c$  and  $l \in C$ .

$$l = prime(l, t) \Rightarrow l \notin generalize(c, gc)$$

*Proof.* Assume  $l = l'$ . Assume further  $l \in generalize(c, gc) =: g$  and define  $g_l := g \setminus \{l\}$ . From this we obtain the properties:

- (I)  $treeRelInd(g, C, t) = C \wedge t \wedge g'$  is unsat since  $g$  generalization and
- (II)  $treeRelInd(g_l, C, t) = C \wedge t \wedge g'_l$  is sat by minimality of  $g$

However, we also have

$$\begin{aligned} C \wedge t \wedge g'_l &= C \wedge l \wedge t \wedge g'_l && \text{since } l \in C \\ &= C \wedge prime(l, t) \wedge t \wedge g'_l && \text{is sat by (II)} \\ \Rightarrow C \wedge l' \wedge t \wedge g'_l &= C \wedge t \wedge g' && \text{is sat} \end{aligned}$$

but this is unsat according to (I). □

We can thus drop all literals fulfilling the premises of Lemma 2. A second simple syntactical check we can make, is to eliminate all literals that are guaranteed by some **Assume**  $l$  statement in the concrete GCL command of  $t$  which we will denote by  $gcl$ . Note that the assume statement may not occur within a choice or  $l$  is not guaranteed to hold after executing  $gcl$ . We define the set of guaranteed literals

$$\text{guarantee}(c, gcl) := \{l \in c \mid \text{Assume } l \text{ occurs in } gcl \text{ outside of a choice}\}$$

**Lemma 3.** Consider  $l \in c$ . Then  $l \in \text{guarantee}(c, gcl) \Rightarrow l \notin \text{generalize}(c, gc)$

*Proof.* Assume  $l \in \text{guarantee}(c, gcl)$  and  $l \in \text{generalize}(c, gc) =: g$ . Also define again  $g_l := g \setminus \{l\}$ . Following the same line of argumentation as in the previous proof we have

$$\begin{aligned} \text{treeRelInd}(g_l, C, t) &= C \wedge t \wedge g'_l \\ &= C \wedge t \wedge l \wedge g'_l && \text{since } l \in \text{guarantee}(c, gcl) \\ &= C \wedge t \wedge g' && \text{by Def of } g_l \\ &= \text{treeRelInd}(g, C, t) \end{aligned}$$

which is again a contradiction as  $\text{treeRelInd}(g_l, C, t)$  must be sat but  $\text{treeRelInd}(g, C, t)$  is unsat.  $\square$

Note that states containing these kind of literals are indeed generated due to the use of weakest preconditions for compute the predecessors.

As we can see we can reduce the number of queries by over  $x\%$ .

### 6.2.3 Generalization Pushing

We have already seen a way to optimize the generalization process using previously computed generalizations. We can further exploit previous computations by “pushing” generalizations onto the next blocking level.

**Example 5.** To elaborate, consider the ART pictured in Fig. 6.2. Nodes are only labelled by their location and the generated cubes denoted alongside each node and we successfully block the path leading to the error state E.

Let  $gc_2, gc_3$  denote the generalization contexts of nodes 2, 3 respectively. Clearly  $x = 55$  is an irrelevant literal the generalization can safely drop. Hence,  $\text{generalize}(x = 55 \wedge i + 1 > 10, gc_2) = i + 1 > 10$  and we also

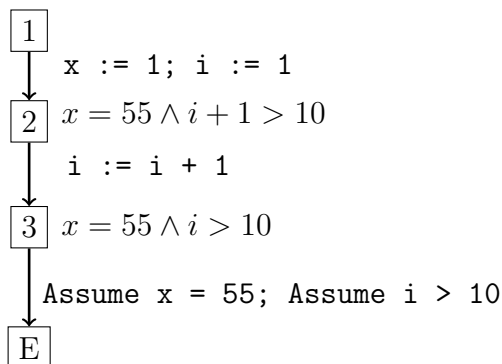


Figure 6.2: Generated states from blocking the path leading from 1 to E.

have  $\text{generalize}(x = 55 \wedge i > 10, gc_3) = i > 10$ . Observe that we have the following connection:

$$\begin{aligned}
 & \text{wep}[\![i := i + 1, \text{generalize}(x = 55 \wedge i > 10, gc_3)]\!] \\
 &= \text{wep}[\![i := i + 1, i > 10]\!] \\
 &= i + 1 > 10 \\
 &= \text{generalize}(x = 55 \wedge i + 1 > 10, gc_2)
 \end{aligned}$$

In other words, if we had some kind of function  $\text{wep}^{-1}$  capable of inverting the weakest existential precondition construction we would obtain the identity:

$$\begin{aligned}
 & \text{generalize}(x = 55 \wedge i > 10, gc_3) \\
 &= \text{wep}^{-1}[\![i := i + 1, \text{generalize}(x = 55 \wedge i + 1 > 10, gc_2)]\!]
 \end{aligned}$$

allowing us to compute  $\text{generalize}(x = 55 \wedge i > 10, gc_3)$  from the prior generalization  $\text{generalize}(x = 55 \wedge i + 1 > 10, gc_2)$  in a purely syntactical manner without having to perform additional SMT Queries.

In this kind of manner, we aim to greatly reduce the number of queries given that we have computed the generalization of a previous step. In the following we restrict ourselves to the case where we have a 1:1 link between the predecessor generalization and the current generalization, that is  $|DNF(\text{wep}[\![c, \varphi]\!])| = 1$ .

Following our approach in example 5 we define the inverse of the weakest existential precondition, which we could also consider a post-condition

**Definition 18.** Let  $wep^{-1}$  be a function such that  $wep^{-1}[[c, wep[[c, \varphi]]] = \varphi$ .

In our implementation we directly obtain  $wep^{-1}$  by inverting the original  $wep$  construction. In order to prove the main property, that is an identity that allows us to compute new generalizations from generalizations of direct predecessors, we first require one more technical result.

**Lemma 4.** Let  $gc$  be some generalization context then

$$\mathbf{generalize}(wep[[c, s]], gc) = wep[[c, \mathbf{generalize}(s, gc)]]$$

*Proof.* We prove the claim by structural induction over  $c$ . Let in the following  $g := \mathbf{generalize}(wep[[c, s]], gc)$ .

- $c = x := a$ . Then  $wep[[c, s]] = s[x \mapsto a]$ . We also have for the context  $gc = (C, t_c) = (C, x' = a)$  for some  $C$  and  $g$  is a generalization such that  $g \subseteq s[x \mapsto a]$  is a minimal cube with

$$\mathbf{treeRelInd}(g, C, x = a) = (C \wedge x = a \wedge g') \text{ is unsat}$$

But then also  $(C \wedge x = a \wedge h')[x \mapsto a]$  with  $h \subseteq s$  such that  $h[x \mapsto a] = g$  must be unsat and  $h$  is also a minimal such subcube (otherwise  $g$  would not be minimal as well). Hence,

$$\begin{aligned} \mathbf{generalize}(wep[[c, s]], gc) &= \mathbf{generalize}(s[x \mapsto a], gc) = g \\ &= h[x \mapsto a] = \mathbf{generalize}(s, gc)[x \mapsto a] = wep[[c, \mathbf{generalize}(s, gc)]] \end{aligned}$$

- $c = \mathbf{Assume } b$ . Then

$$\begin{aligned} \mathbf{generalize}(wep[[c, s]], gc) &= \mathbf{generalize}(s \wedge b, gc) \\ &= \mathbf{generalize}(s, gc) \wedge b = wep[[c, \mathbf{generalize}(s, gc)]] \end{aligned}$$

since  $t_c = b$ .

- $c = c_1; c_2$  with the claim holding for  $c_1, c_2$ . Then

$$\begin{aligned} \mathbf{generalize}(wep[[c, s]], gc) &= \mathbf{generalize}(wep[[c_1, wep[[c_2, s]]], gc) \\ &= wep[[c_1, \mathbf{generalize}(wep[[c_2, s]], gc)]] \\ &= wep[[c_1, wep[[c_2, \mathbf{generalize}(s, gc)]]]] \\ &= wep[[c, \mathbf{generalize}(s, gc)]] \end{aligned}$$

follows directly using the hypothesis.

Recall that we ignore the choice case as we currently assume our statements to be choice-free.  $\square$

**Theorem 1.** Let  $s$  be the current state under consideration in context  $gc$  and  $g = \text{generalize}(\text{wep}[[c, s]], gc')$  be the generalization from the previous step with context  $gc'$  such that  $\text{generalize}(s, gc') \supseteq \text{generalize}(s, gc)$ . Then the following holds:

$$\text{generalize}(s, gc) \subseteq \text{wep}^{-1}[[c, g]]$$

*Proof.* By combining our partial results we obtain the desired claim.

$$\begin{aligned} \text{wep}^{-1}[[c, g]] &= \text{wep}^{-1}[[c, \text{generalize}(\text{wep}[[c, s]], gc')]] \\ &= \text{wep}^{-1}[[c, \text{wep}[[c, \text{generalize}(s, gc)]]]] && \text{by Lemma 4} \\ &= \text{generalize}(s, gc') && \text{by Def 18} \\ &\supseteq \text{generalize}(s, gc) && \text{by premise} \end{aligned}$$

$\square$

In this manner we can reuse the generalization result of the predecessor if  $\text{generalize}(s, gc') \supseteq \text{generalize}(s, gc)$ . We need this condition in order to ensure that we can switch contexts from  $gc'$  to  $gc$  as seen in the last step of the proof. Note that we can easily check this condition by checking if

$$\text{treeRelInd}(\text{generalize}(s, gc'), C, t) \text{ is unsat.}$$

The desired condition then follows by minimality. Summarizing these findings, we can potentially reuse the predecessor generalization  $g$  by employing only one additional query and then calling the old generalization procedure with  $\text{wep}^{-1}[[c, g]]$  as the new input.

## 6.2.4 Using Interpolants

We currently generate our clauses using the negated, generalized predecessor cubes. In other words, we ultimately generate clauses that are syntactically derived from the *wep* construction. While this works well in many cases, due to the strict syntactical nature the model checker might fail to capture important clauses that are easy to derive in a semantical manner.

Consider for example the program `JAIN` shown in Fig. 6.3 where the expression `nondet()` returns some new, nondeterministic but positive variable

in each call. Thus, we nondeterministically increase  $x$  by a random even value. However, we can easily see that since  $x$  is initialized with an uneven value and since bit-vectors have domains of even size,  $x$  can never assume an even value, in particular not 0.

However, our current implementation is not capable of capturing this abstract semantical information. In fact, our TreeIC3 would attempt to prove for each loop iteration of JAIN that  $l_e$  is still unreachable since the generated clauses each speak about some different nondeterministic variable generated by `nondet()` and are too concrete to be used.

What we require, is some kind of mechanism that allows us to drop the unnecessary dependence of the clauses on some specific nondeterministic variable. Generalization cannot provide this as we only remove information on the level of literals. Here, however, we would like to obtain more general literals. We can achieve this using interpolation.

**Definition 19.** Let  $(A, B)$  denote a pair of formulae such that  $A \wedge B$  is inconsistent. An *interpolant* for  $(A, B)$  is a formula  $\varphi := \text{interpolate}(A, B)$  such that

- $\models A \rightarrow \varphi$
- $\varphi \wedge B$  is unsat
- $\varphi$  only contains symbols shared between  $A$  and  $B$

Using Craig's Interpolation Lemma [6] always exists and is also computable. Interpolants have been often considered in the field of model checking and present an alternative method for clause generation. We would,

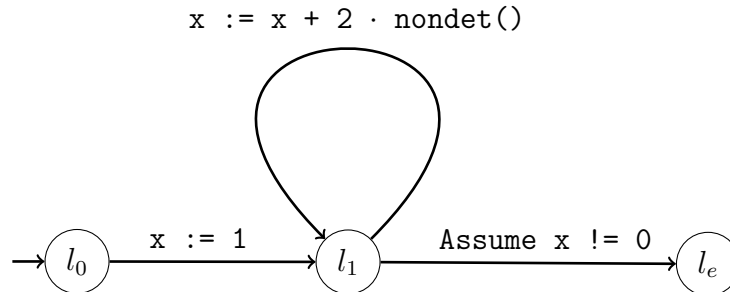


Figure 6.3: Program JAIN

however, like to look at a different way of applying interpolants, namely for simplifying our literals and hence dropping unnecessary semantic dependencies within the literals. Let  $g$  be a generalization in context  $gc = (C, t)$ . We then use interpolation to further simplify  $g$  by constructing the pair  $(A, B)$  as follows

- $A := g'$
- $B := C \wedge t$

Observe that this is unsat due to  $g$  being a generalization. The clause we then generate is not  $\neg g$  but instead

$$\neg \text{unprime}(\text{interpolate}(C \wedge t, g'))$$

whereas  $\text{unprime}(\varphi)$  replaces all occurring primed variables by their unprimed counterparts. Using this approach instances such as JAIN become solvable as we are now capable of abstracting away unnecessary information in our theory atoms.

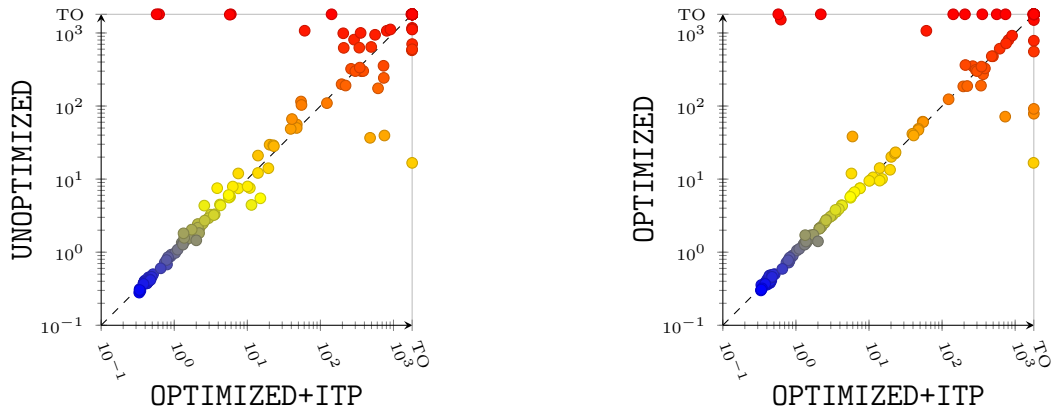
However, our experiments have shown that the computation of interpolants becomes more complicated as the size of the clause sets  $C$  increases. We therefore limit ourselves to only perform interpolation for “small”  $C$ .

## 6.2.5 Evaluation

In this section we evaluate our algorithmic improvements within the context of our implementation.

We consider the following algorithms:

- UNOPTIMIZED uses the unoptimized generalization procedure.
- OPTIMIZED uses generalization enriched by the the syntactical techniques we presented in the prior sections (analysis and reuse of priorly computed generalizations).
- OPTIMIZED+ITP also adds the generalization interpolation. For the maximum clause size we set a limit of 3, as we have found that a bigger limit tends to increase the runtime without yielding more solvable instances.



Algorithm	# solved	solve $t$
UNOPTIMIZED	110/178	17404s
OPTIMIZED	110/178	7721s
OPTIMIZED+ITP	114/178	12268s

Figure 6.4: Benchmarks of generalization optimizations

Fig. 6.4 shows the benchmark results. Firstly, we can observe that our optimizations indeed improve the runtime of the algorithm. Furthermore, OPTIMIZED+ITP is capable of solving more instances, but is generally less efficient due to the additional overhead of computing the interpolants even in situations where we might not require them.

### 6.3 Search Strategies

Until now we have left the order in which we unwound the CFA into an ART up to chance. In fact, the original TreeIC3 algorithm makes no note of using a special approach for unwinding the CFA. However, the positive results on guiding the clause generation process by means of analysing CTIs (i.e. error paths) raise the question whether we can also guide the unwinding itself using error paths.

Consider for example the ART shown in Fig. 6.5 where blocking the path to 6 yielded that 2 is actually an unreachable node. Unwinding by preferring paths leading to error locations instead yields the smaller ART illustrated in Fig. 6.6. since after proving that 2 is actually an unreachable node,

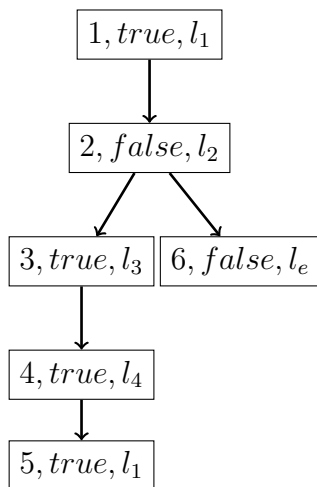


Figure 6.5: Unoptimally unwound ART

we no longer need to consider the successor leading to the location  $l_3$ . In fact, whenever we unwind a path and end up covering the child nodes we do not gain new information in form of generated clauses. On the other hand, unwinding a path leading to an error location (i.e. a counterexample) allows for the generation of new clauses that might in the future be helpful not only for covering other nodes but also for pruning entire sub trees as shown in Fig. 6.6.

But what if we have to decide between two nodes which both are part of some counterexample? We now require some kind of measure telling us how “useful” it would be to consider one path over the other. But how do we measure the “usefulness” of a counterexample?

One possible idea would be using shortest counterexamples, as we can trigger the path blocking algorithm as soon as possible hoping to generate

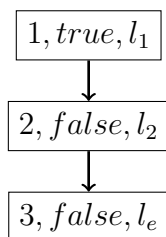


Figure 6.6: Optimally unwound ART

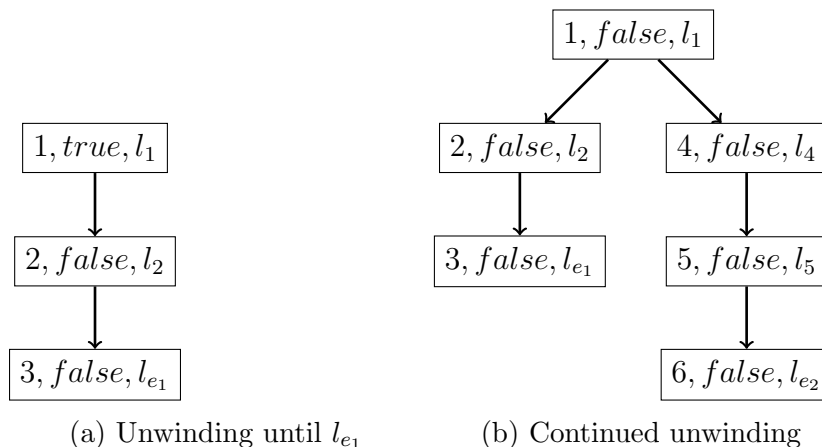


Figure 6.7: Failure of static unwinding

useful clauses. However, this may yield sub optimal results in cases similar to the one displayed in Fig. 6.7.

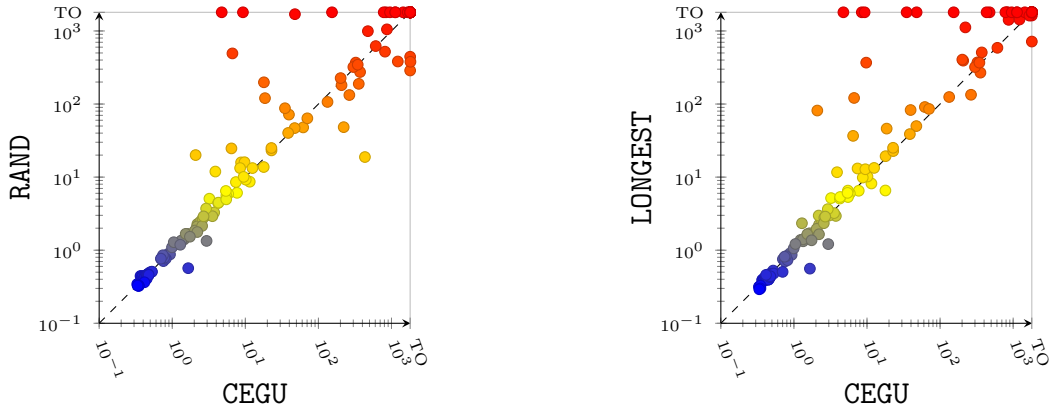
Using the shortest counterexamples first approach we first find  $l_{e_1}$  managing to block the path at node 2, as illustrated on Fig 6.7a. We then discover  $l_{e_2}$  blocking it at node 1 and thus pruning the entire sub tree of 1. Had we considered  $l_{e_2}$  first we would have saved us the effort of having to discover and block  $l_{e_1}$ .

The example shows that the depth at which a counterexample is blocked can largely influence its usefulness. However, it is not clear how we should determine this property in a static manner. We therefore propose considering a more dynamic approach.

Whenever we successfully prune a sub tree we update the unwinding order to first consider the counterexample that has triggered the pruning. In this manner, we obtain a strategy that can deal with all future occurrences of a sub tree in the ART and hence accelerate the pruning process. Since we are using counterexamples to guide us, we refer to the resulting unwinding strategy as **C**ounterexample **g**uided **u**nwindings or in short **CEGU**.

As a pathological case to show the impact unwinding orders can have on the TreeIC3 performance, we also consider the strategy **LONGEST** which prefers the longest counterexamples. If our hypothesis about short counterexamples being generally preferable is correct, this strategy should yield results that are visible worse than using **CEGU**.

We refer to the default strategy as **RAND** since using some arbitrary un-



Algorithm	# solved	solve $t$
RAND	107/178	8240s
CEGU	110/178	7721s
LONGEST	100/178	7852s

Figure 6.8: Benchmarks of the search strategies

winding order is effectively equal to using some random static unwinding order.

The benchmarks illustrated in Fig 6.8 suggest that the CEGU heuristic fails in some individual instances but is tendencially indeed superior to RAND. In particular, we can solve 3 additional instances while still being visibly faster. Also, we can see from the bad performance of LONGEST that this impact of the unwinding order is not a mere fluke. Summarazing these results we can come to the conclusion that while it is not clear how optimal our CEGU strategy is, we have established that different strategies in the unwinding process can impact performance not only in simple examples but also in real life benchmarks.

## 6.4 Blocking Strategies

Throughout this thesis we have employed a fixed strategy for blocking bad states by checking on every level whether or not we can block the path. However, we can, in theory, be exponentially better.

For the following part, let  $\pi$  be some path such that we only have unique

abstract state predecessors, i.e. we always have  $|\text{predecessor}(t_c, \varphi)| = 1$  at all steps when blocking  $\pi$ .

**Definition 20.** Let  $c_i$  be the bad state computed at step  $i$ . We say that a node  $V_i = (l, \varphi, h)$  is

- *blocking* with respect to  $\pi$  if  $\text{treeRelInd}(c_i, i)$  is unsat
- $V_i$  *inductive* with respect to  $\pi$  otherwise.

We also denote blocking nodes by  $B_i$  and inductive nodes by  $I_i$ .

**Lemma 5.** Let  $\pi = V_1, \dots, V_k$  be some path, then it is of the form

$$\begin{aligned} I_1, \dots, I_{n-1}, B_n, \dots, B_{n+m} & \text{ if } \pi \text{ is spurious} \\ I_1, \dots, I_k & \text{ otherwise} \end{aligned}$$

for some  $n, m$  with  $k = n + m$ .

*Proof.* Assume  $\pi$  is not spurious. Then all  $V_i, 1 \leq i \leq k$  must be inductive nodes, otherwise  $\pi$  could be blocked.

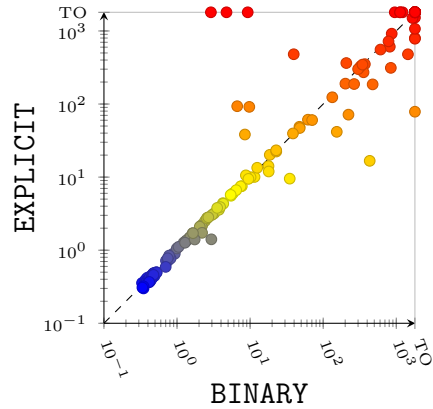
Now assume  $\pi$  is spurious. Then there exists some minimal index  $n$  such that  $V_n$  is blocking. Since  $n$  is minimal we have that all  $V_i$  with  $1 \leq i < n$  are inductive nodes. Set  $m := k - n$  and consider some node  $V_{n+i}$  with  $1 \leq i \leq m$ . Assume  $V_{n+i}$  to be inductive. Then there must exist some concrete state  $\sigma$  such that we can reach  $V_1$  from  $V_{n+i}$ . Hence,  $\pi_{n+i} := V_1, \dots, V_{n+i}$  is feasible. However, then  $\pi_n := V_1, \dots, V_n$  must also be feasible and since  $V_n$  is blocking we have that  $\pi_n$  is not feasible; contradicting the feasibility of  $\pi_{n+i}$ . Thus  $V_{n+i}$  is a blocking node.  $\square$

Clearly, since  $I_1$  to  $I_n$  are inductive nodes the blocking algorithm will fail in blocking  $\pi$  until reaching  $B_n$ . In other words, we are interested in finding the blocking node  $B_n$  and the original blocking algorithm searches for this node in the manner of a linear search. This of course raises the question, whether we can instead use a searching strategy that is closer to that of a binary search.

And indeed, since according to Lemma 5 we have that every node occurring prior to  $B_n$  is inductive and every node occurring after  $B_n$  is blocking we can search for  $B_n$  in a binary style manner. The blocking process thus only requires  $\mathcal{O}(\log k)$  many sat queries.

However, we only obtained this result for the event that we only have unique abstract predecessor states. In order to generalize the theory, we introduce a third node labelling.





Algorithm	# solved	solve $t$
EXPLICIT	110/178	7721s
BINARY	112/178	12287s

Figure 6.10: Benchmarks of the blocking strategies

## 6.5 Implementational Details

In this section we want to discuss some small optimization and implementational details that we have seen to perform well in our experiments.

**Simplifier and syntactical checks** We use a simplifier provided by the SMT Solver Engine, Z3 in our case, to not only simplify clauses but also in hope of reducing the number of clauses that are semantically equivalent, but not syntactically without causing significant overhead. With this, we increase the success rate of using syntactical checks for finding implications as  $C_1 \subseteq C_2$  implies  $\models C_1 \rightarrow C_2$ . In fact, since the number of nodes in an ART may be huge in practical applications we only use the syntactical check between nodes that are not on the same path to efficiently compute the covering relation.

**Hashing** TreeIC3 has a tendency to recompute many objects including weakest preconditions, DNFs of formulae, generalizations, the validity of certain formulae and so on, when performing the path blocking algorithm on the individual paths. We recommend making heavy use of hashing struc-

tures to not only hash the end results but also in between results to avoid recomputations also for objects that are similar to the already computed ones.

**Timeouts** We use different timeouts for SMT queries that are necessary (eg. required to block a path) and queries that are not necessary for the soundness of the algorithm, but for the performance (eg. generalization, path propagation). Our experiments have shown that especially when employing a BV theory the TreeIC3 algorithm might generate formulae that are not big in size, but still very difficult for the SMT Solver. By using a smaller timeout for not necessary calls we can prevent the algorithm from getting “stuck” in such difficult calls.

**Solver instances** Works concerning themselves with IC3 such as [7] recommend using one solver instance per frame. Translating this approach into the TreeIC3 framework would mean to introduce one solver instance per node (or edge) in the ART. However, in our implementation this has quickly lead to memory issues on moderately sized benchmarks. We therefore only employ a global solver instance.

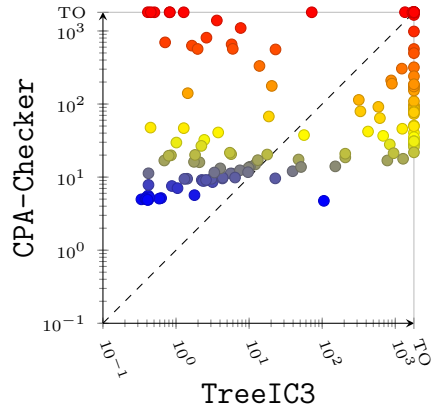
**Pruning** Before executing the path blocking algorithm we check for each node if the abstract state formula is compatible with the transition. If not, we can prune the sub tree beneath such a node and skip the blocking process.

## 6.6 Comparison

In this section we compare our implementation with the state-of-the-art C code model checker CPA-Checker<sup>1</sup>, a tool sporting a bit-precise CEGAR implementation that became the overall winner of the 2015 software model checking competition. We consider a subset of our benchmarks using only the Cimatti and the SVCOMP instances. Furthermore, we set our TreeIC3 implementation to use the blocking strategy `BINARY`, the search strategy `CEGU` and `OPTIMIZED+ITP` as the generalization procedure, since these settings maximized the number of solveable instances. In the following we refer to our implementation using these strategy settings just as TreeIC3.

---

<sup>1</sup><http://cpachecker.sosy-lab.org/achieve.php>



Algorithm	# solved	solve $t$
TreeIC3	100/150	12639s
CPA-Checker	120/150	12680s

Figure 6.11: Comparison with CPA-Checker

Fig 6.11 illustrates the comparison between our implementation and CPA-Checker. Although CPA-Checker is capable of solving 20 more instances while mostly also remaining superior in the direct solve time comparison, we can see that there are also many instances that TreeIC3 can solve more efficiently. Observe that there are surprisingly few instances that both algorithms can solve in a similar timespan, suggesting that the two algorithms are complementary in nature. In fact, algorithmic approaches combining IC3 and CEGAR have been proposed in newer works [1, 5].

# Chapter 7

## Conclusion

In the course of this thesis we have introduced the fully symbolic, original IC3 algorithm by Bradely [2] and its extension to SMT by Cimatti [4]. Furthermore, we have also introduced Cimatti's TreeIC3 algorithm [4] which reintroduces unwinding into IC3 and is in exchange capable of exploiting a programs control-flow. We have also presented key parts of our own implementation of the TreeIC3 algorithm and considered various improvements for it.

While the improvements by means of strategies in the search and the blocking phase have made only a rather small impact, our optimizations for the generalization procedure have yielded decent results. The key approach is to reuse priorly computed generalizations by checking if they are still valid in a current generalization context. Of these optimizations, We have both proven the correctness, and confirmed the improvement experimentally by means of benchmarks.

Although, as the evaluation has shown, we have not yet reached the level of dedicated state-of-the-art software model checkers such as CPA-Checker, our implementation is independent of the underlying theory and can be applied to any programming language with access to an LLVM compiler making our approach far more flexible.

# Bibliography

- [1] Johannes Birgmeier, Aaron R Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (ctigar). In *Computer Aided Verification*, pages 831–848. Springer, 2014.
- [2] Aaron R Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.
- [3] Aaron R Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Formal Methods in Computer Aided Design, 2007. FMCAD’07*, pages 173–180. IEEE, 2007.
- [4] Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In *Computer Aided Verification*, pages 277–293. Springer, 2012.
- [5] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. Springer, 2014.
- [6] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, 1957.
- [7] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125–134. IEEE, 2011.
- [8] Cormac Flanagan and James B Saxe. Avoiding exponential explosion: Generating compact verification conditions. *ACM SIGPLAN Notices*, 36(3):193–205, 2001.

- [9] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.
- [10] Kenneth L McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136. Springer, 2006.
- [11] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.
- [12] Edsger Wybe and Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.