Master's Thesis

Generalisation Methods for Control-Flow Oriented IC3 Algorithms

Frederick Prinz

September 20, 2016

Chair for Software Modeling and Verification
RWTH Aachen University

Referees:
Prof. Dr. Thomas Noll
Prof. Dr. Ir. Joost-Pieter Katoen

Supervisors:
M.Sc. Tim Lange (RWTH Aachen University)
Dr. Martin R. Neuhäußer (Siemens AG)

Abstract

IC3 is a bit-level formal verification method for hardware systems. Given the program in terms of a finite-state transition system, the IC3 algorithm tries to infer an inductive invariant strong enough to prove the correctness of the program. Therefore, it incrementally derives a sequence of overapproximations of the reachable program states. The IC3 generalisation function is a key component of the entire algorithm, because it abstracts from the concrete counterexamples. This abstraction is computed by applying Boolean SAT-solving techniques, such that the resulting generalisation blocks multiple states simultaneously. IC3CFA is based on the idea to lift the principles of Boolean IC3 to software model checking, i.e. SMT. Thereby, the program is given in terms of a control flow automaton (CFA). This thesis presents several approaches to improve the IC3CFA generalisation algorithm, where these approaches try to avoid unnecessary or redundant computations. More precise, since the SMT calls are usually considered to be time consuming and badly scalable, the SMT queries are as far as possible replaced by equivalent syntactical checks. The experimental results show the effectiveness of the presented improvements regarding runtime and memory requirements.

Eidesstattliche Versicherung

____________________________ ____________________________

Name, Vorname Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

____________________________ ____________________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

____________________________ ____________________________

Ort, Datum Unterschrift

Contents

1. Introduction **1**

2. Preliminaries **5**
2.1. Formulas 5
2.2. Guarded Command Language 7
2.3. Control Flow Automata 9

3. IC3 **13**
3.1. Main Algorithm 13
3.2. Generalisation 18
3.3. SMT 20

4. IC3CFA **21**
4.1. Frames 21
4.2. Main Algorithm 23
4.3. Generalisation 27
4.4. Framework 30

5. Generalisation **33**
5.1. Improved Initialisation 33
5.2. Predecessor Computation 36
5.3. Relative Inductiveness 42
5.4. Assumes 44
5.5. Predecessor Cubes 47
5.6. Generalisation Context 50
5.7. Minimal Generalisation 53
5.8. Ordering 55
5.9. Multiple Predecessors 55
5.10. Interpolation 56
5.11. CTGs 59
5.12. New Generalisation Algorithm 61

6. Evaluation **63**
6.1. Performance 64
6.2. Comparison 68

7. Conclusion **73**
7.1. Summary . . . 73
7.2. Future Work . . . 74

Bibliography **75**

A. Benchmark Programs **79**

B. Detailed Experimental Results **83**

Index **i**

List of Figures

1.0.1. Model checking process. 1
1.0.2. Standard CEGAR procedure. 3

2.3.1. The CFA A_{true} of the example program P_{true}. 11

3.1.1. Illustration of the IC3 invariants. 15
3.1.2. Illustration of a CTI c. 15
3.2.1. Intuitive notion of the IC3 generalisation. 18

4.1.1. Illustration of a single frame. 22
4.1.2. Illustration of a frame sequence. 22
4.2.1. Graphical representation of a relative inductive cube c. 23
4.3.1. Intuitive notion of the IC3CFA generalisation. 27
4.4.1. Model checking framework. 31

5.1.1. The CFA A_{false} of the example program P_{false}. 35
5.2.1. Illustration of the WEP of c with respect to the edge $e = (l, cmd, l')$. 37
5.5.1. Graphical representation of a predecessor cube $\widehat{c} \supseteq wep(cmd, c)$. . . 47
5.5.2. Graphical representation of a generalisation $\widehat{c}$ based on the predecessor cube. 49
5.6.1. Intuitive notion of the generalisation context. 51
5.7.1. Intuitive notion of the minimal generalisation. 53
5.10.1. The CFA of the program $Jain1$. 57
5.10.2. Graphical representation of the interpolant I. 58

6.1.1. Graphical comparison with IC3CFAOld. 64
6.1.2. Number of iterations compared to IC3CFAOld. 65
6.1.3. Number of SMT calls in generalisation compared to IC3CFAOld. . 66
6.1.4. Graphical comparison with IC3CFANoGen. 67
6.2.1. Graphical comparison with TreeIC3. 68
6.2.2. Graphical comparison with BMC. 69
6.2.3. Graphical comparison with CEGAR. 70
6.2.4. Graphical comparison with CPAchecker. 71

List of Algorithms

2.3.1. Pseudocode of the example program P_{true}. 11

3.1.1. The main function of the original IC3 algorithm. 16
3.1.2. The strengthen function of the original IC3 algorithm. 17
3.1.3. The propagate function of the original IC3 algorithm. 17
3.2.1. The generalisation function of the original IC3 algorithm. 19

4.2.1. The outer loop of the IC3CFA algorithm. 24
4.2.2. The inner loop of the IC3CFA algorithm. 25
4.3.1. The generalisation of the IC3CFA algorithm. 28
4.3.2. The default generalisation function of the IC3CFA algorithm. . . . 29
4.3.3. The additional generalisation function of the IC3CFA algorithm. . 29

5.1.1. Pseudocode of the example program P_{false}. 35
5.10.1. Pseudocode of the program $Jain1$. 57
5.11.1. The extended IC3CFA generalisation function with CTGs. 60
5.12.1. New generalisation algorithm of the IC3CFA algorithm. 61

1. Introduction

Nowadays, computerised systems are spread over a huge range of applications areas, but analysing these more and more complex systems becomes very difficult. One approach to prove that a given program behaves as intended is formal verification. In particular, we are looking for an automated version of the verification process, e.g. model checking. Model checking is an algorithmic approach to verify the correctness of a given program with respect to the corresponding specification [BK08, CGP99]. However, model checking is an extensive task, because all possible program executions need to be analysed. In contrast to ordinary testing (with test cases), model checking can prove the correctness of the whole program, while testing may only prove the existence of an error.

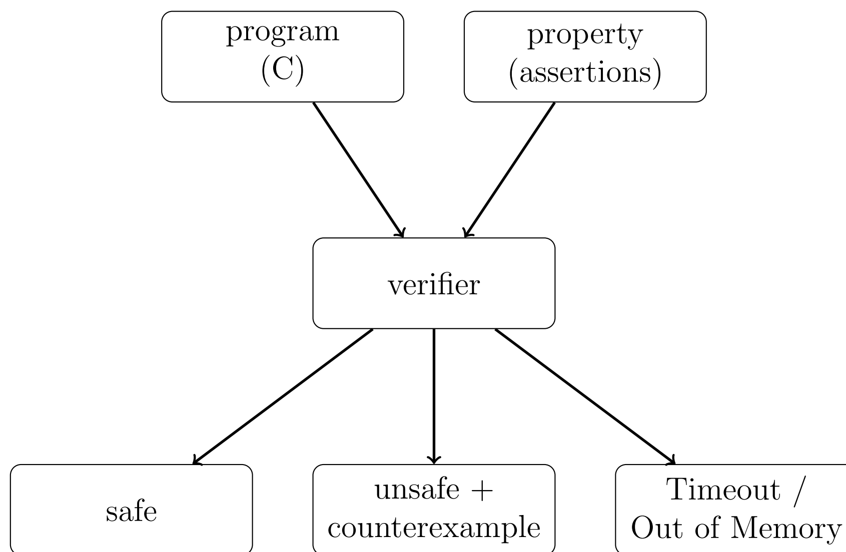

Figure 1.0.1.: Model checking process.

The general model checking process is illustrated in Fig. 1.0.1. More precise, we consider C programs as input files to the verifier in this thesis. The specification is given in terms of assertions in the corresponding C program. Basically, the verifier tries to determine, if the given program is safe, i.e. every assertion evaluates to *true* for all possible executions. Otherwise, if an assertion is violated, the verifier

returns the result unsafe and provides a counterexample, respectively. However, in practice we are also limited by the available resources, i.e. memory and time. Thus, we might get neither safe nor unsafe as possible result because of insufficient resources.

IC3 is state-of-the-art one of the most prominent model checking algorithm designed for bit-level verification of hardware systems (hardware model checking). It has been originally proposed in [Bra11]. The IC3 algorithm operates on a finite-state transition system representing the given program. Thereby, it tries to prove the correctness of an invariant property by iteratively deriving overapproximations of the reachable state space. In every iteration the algorithm tries to block all potential counterexamples with respect to the desired property. Simultaneously, IC3 relies on an aggressive abstraction, so-called generalisation, of these counterexamples. The whole IC3 algorithm terminates with the result "safe", if a fixpoint in the derived sequence of overapproximations is found. Different experimental results have proven the effectiveness of this approach [Bra11].

IC3CFA is an approach, which tries to lift the ideas of the original IC3 algorithm to software model checking [LNN15]. Thereby, the program is given in terms of a control-flow automaton (CFA). Respectively, the original IC3 algorithm is adapted to incorporate the location information of the CFA. The IC3CFA approach has shown promising results in the domain of software model checking [LNN15].

In this thesis we try to improve the IC3CFA generalisation algorithm, which is part of the entire IC3CFA algorithm. So far, the abstraction of the potential counterexamples in the IC3CFA generalisation relies on a significant amount of requests to the SMT (Satisfiability Modulo Theories) solver. However, the SMT calls in the algorithm are considered to be computational expensive and to scale worse for larger requests. Thus, we try to replace the SMT calls by equivalent syntactical checks, which for example exploit the structure of the CFA. Furthermore, we try to avoid unnecessary computations, where the result is already derived by previous computations. Note that we basically consider an analysis independent of the underlying theory, e.g. bit-vector (BV) logic or linear real arithmetic (LRA).

Another approach, which is also based on the idea to lift the original IC3 algorithm to software model checking, is called TreeIC3 [CG12]. However, this approach is based on abstract reachability trees (ARTs), where the ART is created by unwinding the original control flow automaton. In consequence, each ART node is associated with a program location and a first-order formula. The TreeIC3 al-

gorithm iteratively derives a tree-like structure given in terms of the ART, where the algorithm computes under-approximations of candidate counterexamples.

Bounded model checking (BMC) is a model checking algorithm similar to a breadth-first search [BCC+03, CBRZ01, Lee61]. Starting in the initial program location, the algorithm unwinds a search-tree of all possible program executions, where the maximal length of a path is limited by the specified bound. Since the BMC algorithm is only able to find paths leading to an error location, it can not prove the correctness of the given program.

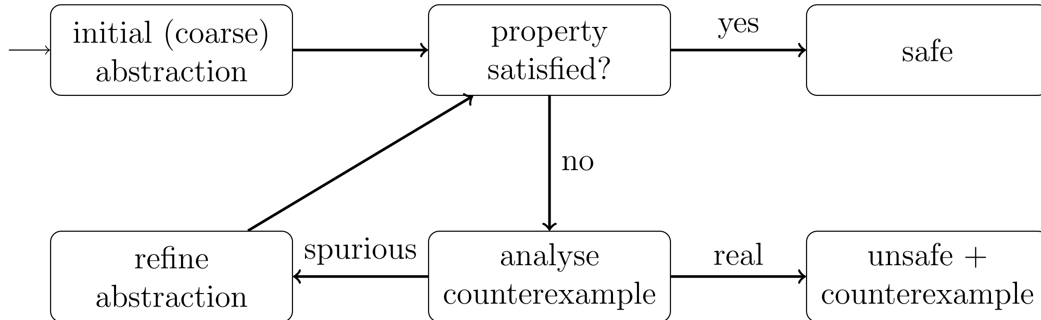

Figure 1.0.2.: Standard CEGAR procedure.

CEGAR ("CounterExample-Guided Abstraction Refinement") is state-of-the-art another popular model checking approach [CGJ+00, CGJ+03]. It is based on the idea to iteratively refine a set of abstractions based on the derived counterexamples. The standard CEGAR procedure is illustrated in Fig. 1.0.2[WGI07]. The algorithm starts with a coarse abstraction and checks, if there is a realisable counterexample. Otherwise, the counterexample is called spurious and the algorithm refines the set of predicates to block this spurious counterexample. The algorithm has proven the correctness of the given program, if there not exists a counterexample any more.

Another approach is based on the idea to combine the static program analyses and the verification process. This combination has been proposed in [BHT07] and is implemented in the model checking tool CPAchecker [BK11]. Different experimental results have shown that the CPAchecker is state-of-the-art one of the fastest model checkers with respect to the given set of benchmark programs (Appendix A) [Bey15].

This thesis is structured as follows. First, we introduce the preliminaries in Chapter 2, which also include the definition of the control-flow automata. Based on the original IC3 algorithm for hardware model checking (Chapter 3), we introduce the IC3CFA algorithm in Chapter 4. In this thesis we mainly consider the generalisation algorithm of IC3CFA, wherefore we present several improvements in Chapter 5. Finally, we evaluate the performance of our new IC3CFA generalisation with regard to the given set of benchmark programs (Appendix A). Furthermore, we compare the new IC3CFA algorithm to other state-of-the-art model checkers (Chapter 6).

2. Preliminaries

In this chapter we introduce the background to formally reason about a given program and its correctness. Therefore, we use first-order formulas to represent the program states (Sec. 2.1) and the Guarded Command Language to model the program behaviour in between (Sec. 2.2). All possible program executions are specified in terms of a control flow automaton, which is introduced in Sec. 2.3. This control flow automaton includes the initial program valuation and the specification of the undesired behaviour.

2.1. Formulas

In this thesis we use quantifier-free first-order logic to reason about program states [Fit90, Smu95].

Definition 2.1.1 (Satisfiability)**.**
Let φ be a quantifier-free first-order formula [Fit90, Smu95]. The formula φ is called satisfiable, iff there exists a model σ, i.e. $\sigma \models \varphi$ [CK90]. We define a function $sat(\varphi)$, such that

$$sat(\varphi) \quad \Leftrightarrow \quad \exists model\ \sigma.\ \sigma \models \varphi.$$

Vice versa, we define a function $unsat(\varphi)$, where

$$unsat(\varphi) \quad \Leftrightarrow \quad \neg(\exists model\ \sigma.\ \sigma \models \varphi).$$

Definition 2.1.1 introduces the model relation $\models$ for first-order formulas. Let φ denote a quantifier-free first-order formula in the following . We define two functions $sat(\varphi)$ and $unsat(\varphi)$ to indicate the existence or the absence of a model with respect to the formula φ (Def. 2.1.1). The set $Var(\varphi)$ contains all variables, which occur in the formula φ. A primed formula φ' indicates that every variable in φ is replaced by their primed version, i.e. $\varphi' = \varphi[x \mapsto x' \mid x \in Var(\varphi)]$.

Let Var denote the set of program variables in the remainder of this section. A model σ over Var represents a concrete program state, where each program variable is assigned a specific value. In consequence, a formula φ is an abstract

representation of a set of program states. In the following, we define several terms specifying the syntactic structure of a formula φ.

Definition 2.1.2 (Literals)**.**
A literal p is a first-order atom over Var or its negation [Smu95].

Definition 2.1.3 (Cubes)**.**
A cube c over Var is a first-order formula, such that

$$c = \bigwedge L$$

where L is a set of literals over Var.

A literal p (Def. 2.1.2) is an atomic formula, i.e. it has no strict sub-formulas. We introduce compound formulas in terms of cubes (Def. 2.1.3) , where each cube is a conjunction of literals. A cube with an empty set of literals $L = \emptyset$ equals the formula *true*. Note that a set of literals L uniquely determines a cube. Thus, we often use the equivalent notation $c = L \equiv \bigwedge L$. The negation of a cube $\neg c$ is called clause.

Definition 2.1.4 (Clausesets)**.**
A clauseset cs is a (quantifier-free) first-order formula, i.e.

$$cs = \bigwedge C$$

where C is a set of clauses over Var.

A clauseset cs is a conjunction of explicitly negated cubes (Def. 2.1.4). Similar to cubes, we also use the equivalent set notation, i.e. $cs = C \equiv \bigwedge C$. We introduce the symbol $\top$ for the clauseset cs with an empty set of clauses $C = \emptyset$, since it is equivalent to the formula *true*. Respectively, the unsatifiable counterpart $cs = \{\neg true\} \equiv false$ is represented by the symbol $\bot$. An example of the different terms presented in this section is shown in Example 2.1.5.

Example 2.1.5.
Let $Var = \{x, y\}$ be the set of program variables. Furthermore, let $p_1 = (y \geq x)$ and $p_2 = (x = 1)$ be two literals over Var. The example cube c_1 contains both literals p_1 and p_2, such that $c_1 = \{y \geq x, x = 1\} = y \geq x \wedge x = 1$. Let $p_3 = (x = 2)$ be another literal, which is part of the cube $c_2 = \{y \geq x, x = 2\}$. We obtain the example clauseset $cs = \{\neg c_1, \neg c_2\}$, where the corresponding formula is given by $\neg(y \geq x \wedge x = 1) \wedge \neg(y \geq x \wedge x = 2)$.

2.2. Guarded Command Language

In this section we introduce the Guarded Command Language, short GCL, to reason about the sequential program behaviour between two program states. GCL has originally been introduced by Dijkstra for predicate transformer semantics [Dij75]. It represents the given sequential program in a standardised way, which is independent of the actual programming language.

Definition 2.2.1 (Expressions)**.**
Let Var denote the program variables. An arithmetic expression a is a first-order formula defined by

$$a ::= z \ | \ v \ | \ a_1 \circ a_2$$

where $z \in \mathbb{Z}, v \in Var$ and $\circ = \{+, -, *, /, \%\}$. Furthermore, a_1 and a_2 are also arithmetic expressions.
A restricted Boolean expression b is a first-order formula defined by

$$b ::= true \ | \ false \ | \ x \circ a \ | \ b_1 \wedge b_2$$

where $x \in Var$ is a variable, $\circ = \{=, \neq, \leq, \geq, <, >\}$ and a is an arithmetic expression. In addition, b_1 and b_2 are also restricted Boolean expressions.

We define first-order expressions (Def. 2.2.1), where we distinguish between arithmetic and restricted Boolean ones. The arithmetic expressions include the basic arithmetic operations on integers, where the operator / denotes the integer division. A restricted Boolean expression either evaluates to $true$ or $false$, where compound expressions are constructed via the conjunction of sub-expression. Note that we use a restricted form of Boolean expressions to guarantee the correctness of our approach presented in Sec. 5.2 [GS93]. However, although we syntactically restrict to the conjunction of multiple expressions, we are also able to represent the negation and disjunction of expressions with the help of GCL commands introduced in the following.

Definition 2.2.2 (Guarded Command Language (GCL))**.**
The syntax of a command in Guarded Command Language (GCL) is defined as follows:

$$cmd ::= assume \ b \ | \ x := a \ | \ cmd_1; cmd_2 \ | \ cmd_1 \Box cmd_2$$

where b is a restricted Boolean expression, x is a variable, a is an arithmetic expression and cmd_1, cmd_2 are GCL commands.

The definition of a GCL command is shown in Def. 2.2.2 , where we distinguish between four command types [Dij75]. The first basic command is the assume

assume b, which is derived from a condition or an assertion in the given program. Note that b is a restricted Boolean expression, i.e. we only allow the conjunction of two sub-expressions. However, we can represent the disjunction $b_1 \vee b_2$ with the help of the choice command $(assume\ b_1)\Box(assume\ b_2)$. Respectively, the negation $\neg b_1$ is handled by applying the De Morgan's laws [CG15] and, if necessary, inverting the arithmetic operators. The second basic command $x := a$ is the common variable assignment. The sequential execution of GCL commands is represented by the sequence rule $cmd_1; cmd_2$, which contains two sub-commands cmd_1, cmd_2. The choice command $cmd_1 \Box cmd_2$ allows us to also model non-deterministic program behaviour, e.g. programs with non-deterministic variables *nondetBool* and *nondetInt*.

Definition 2.2.3 (GCL Semantics)**.**
Let σ, σ' be two concrete (program) states. The semantics of a GCL command *cmd* are specified by the execution relation $\rightarrow$, where $< cmd, \sigma > \rightarrow \sigma'$ is a notation for the statement σ *evaluates to* σ' *under cmd.* The execution relation $\rightarrow$ is inductively defined by:

$$(assume) \quad \frac{b(\sigma) = true}{< assume\ b, \sigma > \rightarrow \sigma}$$

$$(assign) \quad \frac{}{< x := a, \sigma > \rightarrow \sigma[x \mapsto a(\sigma)]}$$

$$(seq) \quad \frac{< cmd_1, \sigma > \rightarrow \sigma' \quad < cmd_2, \sigma' > \rightarrow \sigma''}{< cmd_1; cmd_2, \sigma > \rightarrow \sigma''}$$

$$(choice1) \quad \frac{< cmd_1, \sigma > \rightarrow \sigma'}{< cmd_1 \Box cmd_2, \sigma > \rightarrow \sigma'}$$

$$(choice2) \quad \frac{< cmd_2, \sigma > \rightarrow \sigma'}{< cmd_1 \Box cmd_2, \sigma > \rightarrow \sigma'}$$

The semantics of a GCL command is shown in Def. 2.2.3 [Dij75], where a concrete state σ is a mapping of the program variables Var to an element of their respective domain $Dom(Var)$, i.e. $\sigma : Var \rightarrow Dom(Var)$. Given an initial state σ, the execution relation specifies the resulting states σ' after executing the GCL command *cmd*. We say that σ evaluates to σ' under *cmd*. Regarding the assume command *assume* b, the notation $b(\sigma)$ represents the evaluation of the Boolean expression b under the variable valuation σ. Note that the execution relation is

only specified, if the condition b of the assume command *assume* b evaluates to *true*. Otherwise, there exists no successor state, such that we model the blocking behaviour of an assume in the given program.

Example 2.2.4.
Let x be the single program variable and $\sigma = [x \mapsto 0]$ be a concrete state. We consider an example GCL command

$$cmd_{ex} = assume\ x \geq 0;\ x := x + 1;\ (x := 4 * x) \Box (assume\ true);\ x := x - 1$$

and apply the execution relation $\rightarrow$ starting in σ. Note that $\Rightarrow$ indicates the application of the sequence rule (*seq*).

$$\begin{array}{llr} & < cmd_{ex}, [x \mapsto 0] > & \\ & < assume\ x \geq 0, [x \mapsto 0] > \rightarrow [x \mapsto 0] & \text{(assume)} \\ \Rightarrow & < x := x + 1;\ (x := 4 * x) \Box (assume\ true);\ x := x - 1, [x \mapsto 0] > & \\ & < x := x + 1, [x \mapsto 0] > \rightarrow [x \mapsto 1] & \text{(assign)} \\ \Rightarrow & < (x := 4 * x) \Box (assume\ true);\ x := x - 1, [x \mapsto 1] > & \end{array}$$

We distinguish between the first and the second choice rule, where the first one yields:

$$\begin{array}{llr} & < x := 4 * x, [x \mapsto 1] > \rightarrow [x \mapsto 4] & \text{(choice2)} \\ \Rightarrow & < x := x - 1, [x \mapsto 4] > & \\ & < x := x - 1, [x \mapsto 4] > \rightarrow [x \mapsto 3] & \text{(assign)} \\ \Rightarrow & [x \mapsto 3] & \end{array}$$

Alternatively, if we apply the second choice rule, we get:

$$\begin{array}{llr} & < assume\ true, [x \mapsto 1] > \rightarrow [x \mapsto 1] & \text{(choice1)} \\ \Rightarrow & < x := x - 1, [x \mapsto 1] > & \\ & < x := x - 1, [x \mapsto 1] > \rightarrow [x \mapsto 0] & \text{(assign)} \\ \Rightarrow & [x \mapsto 0] & \end{array}$$

In summary, $\sigma = [x \mapsto 0]$ evaluates to $[x \mapsto 0]$ or $[x \mapsto 3]$ under cmd_{ex}.

2.3. Control Flow Automata

So far, we have introduced GCL commands to model sequential program executions (Sec. 2.2). In this section we introduce a graph-based representation, called control flow automaton (CFA), to represent more complex programs with loop structures.

Definition 2.3.1 (Control Flow Automaton (CFA))**.**
A control flow automaton (CFA) is a tuple

$$A = (L, G, l_{init}, l_E)$$

where

- L is a finite set of program locations,
- $G \subseteq L \times GCL \times L$ is a finite set of transitions labelled with a GCL command,
- $l_{init} \in L$ is an initial program location,
- $l_E \in L$ is an error location.

The formal definition of a CFA is shown in Def. 2.3.1 [All70]. Intuitively, each node $l \in L$ represents a specific program location, i.e. a specific value of the program counter. The initial location $l_{init} \in L$ represents the entry of the program. An edge between two nodes represents a sequential program execution, thus it is labelled with a GCL command. An edge leading to the error location $l_E \in L$ indicates that the desired property of the program is violated. The property is violated, if an assertion in the program evaluates to $false$. In the following, we assume that there is at most one edge between two locations and that the error location l_E has no outgoing edges. Furthermore, we apply Large-Block Encoding (LBE) to group several loop-free program locations, such that we minimise the overall graph [BCG+09, BKW10].

Definition 2.3.2 (Transition Formulas)**.**
Let Var denote the program variables. Furthermore, let (L, G, l_{init}, l_E) be a CFA and $e = (l, cmd, l') \in G$ be an edge between the locations $l, l' \in L$. The transition formula T_e is a quantifier-free first-order formula over the variables Var and Var'. It is derived from the GCL command cmd by the recursive auxiliary function

$$f(cmd) = \begin{cases} b & \text{if } cmd = (assume\ b) \\ x' = a & \text{if } cmd = (x := a) \\ f(cmd_1)[y' \setminus y_i] \wedge f(cmd_2)[y \setminus y_i] & \text{if } cmd = (cmd_1; cmd_2) \\ f(cmd_1) \vee f(cmd_2) & \text{if } cmd = (cmd_1 \Box cmd_2) \end{cases}$$

where $y \in Var(cmd)$ and $y_i \notin Var(cmd)$ is a fresh variable.

Since we check, if an edge in the CFA is realisable, we have to compute a (quantifier-free) first-order formula of this edge. Intuitively, a transition formula (Def. 2.3.2) specifies a mapping from a source to a target valuation with respect to

the variables Var. The common unprimed variables are used to encode the source program state, whereas the new primed ones encode the target state.

Example 2.3.3.
We introduce an example program P_{true} (Alg. 2.3.1).

Algorithm 2.3.1 Pseudocode of the example program P_{true}.

```
1: procedure MAIN(x, y)
2:     assume(y < x)
3:     while true do
4:         assert(y < x ∨ (x ≠ 1 ∧ x ≠ 2))
5:         y ← y − 1
6:     end while
7: end procedure
```

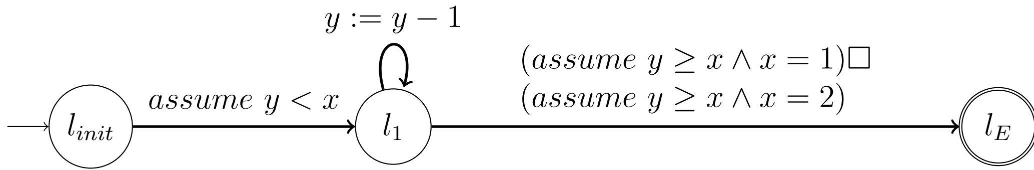

Figure 2.3.1.: The CFA A_{true} of the example program P_{true}.

The corresponding CFA A_{true} of the program P_{true} is shown in Fig. 2.3.1. More formally, it is defined by

$$A_{true} = (L, G, l_{init}, l_E)$$

where

$$\begin{aligned} L =& \{l_{init}, l_1, l_E\}, \\ G =& \{(l_{init}, assume\ y < x, l_1), (l_1, y := y - 1, l_1), \\ & (l_1, assume\ y \geq x \wedge x = 1; assume\ y \geq x \wedge x = 2, l_E)\} \end{aligned}$$

The transition formulas are given by

$$\begin{aligned} T_{(l_{init}, assume\ y<x, l_1)} &= y < x, \\ T_{(l_1, y:=y-1, l_1)} &= (y' = y - 1), \\ T_{(l_1, assume\ y \geq x \wedge x=1; assume\ y \geq x \wedge x=2, l_E)} &= (y \geq x \wedge x = 1 \wedge y \geq x \wedge x = 2). \end{aligned}$$

3. IC3

IC3 ("Incremental Construction of Inductive Clauses for Indubitable Correctness") is state-of-the-art one of the most important bit-level formal verification methods for hardware systems. The algorithm has been originally developed by Bradley [Bra11] and tries to incrementally infer an inductive invariant strong enough to prove the correctness of the given program.

We present the main algorithm of IC3 in Sec. 3.1, where the generalisation function is introduced in Sec. 3.2. Although IC3 originally focuses on Boolean variables, we introduce a first extension to Satisfiability Modulo Theories (SMT) in Sec. 3.3.

3.1. Main Algorithm

The IC3 algorithm operates on a transition system over Boolean variables, where the transition systems models the given program. The transitions system is formally defined as follows (Def. 3.1.1) [Kel76].

Definition 3.1.1 (Transition Systems (TS)).
A (finite-state) transition system is a tuple

$$TS = (\bar{x}, T, I)$$

where

- $\bar{x}$ is a finite set of Boolean variables,
- $T(\bar{x}, \bar{x}')$ is propositional formula describing the transition relation,
- $I(\bar{x})$ is a predicate specifying the initial states.

Given a set of Boolean variables $\bar{x}$, the propositional formula $I(\bar{x})$ evaluates to $true$, iff the variable valuation $\bar{x}$ represents an initial state. Accordingly, the transition formula $T(\bar{x}, \bar{x}')$ yields $true$, iff we can reach the program state with valuation $\bar{x}'$ from $\bar{x}$ within one step. Similar to priming the first order formulas (Sec. 2.1), the primed variables $\bar{x}'$ characterise the target state, while the unprimed ones $\bar{x}$ specify the source state.

Definition 3.1.2 (Inductiveness).
Let $TS = (\bar{x}, I, T)$ be a transition system and P be a propositional property over the variables $\bar{x}$. Then P is called inductive, if

- $I \Rightarrow P$ (initiation),
- $P \wedge T \Rightarrow P'$ (consecution).

The desired behaviour of the given program is specified by a propositional property $P(\bar{x})$. Note that if we consider the propositional property $E(\bar{x})$ specifying the error states, we set $P(\bar{x}) = \neg E(\bar{x})$. To prove the correctness of the given program, we show that the property P is an invariant, i.e. it is inductive with respect to the transition system TS (Def. 3.1.2). However, the inductiveness of P is not essential for the correctness of the program. Thus, the IC3 algorithm instead derives an strengthening F, such that $F \wedge P$ is inductive. The inductive strengthening $F(\bar{x})$ of P is formally given in terms of a clauseset over the variables $\bar{x}$, i.e. it is a (quantifier-free) first-order formula. We say that the strengthening F is relative inductive with respect to a property P, iff $I \Rightarrow F$ and $P \wedge F \wedge T \Rightarrow F'$ holds. The computation of a single inductive strengthening F is often complex and computational expensive, such that the IC3 algorithm iteratively derives a sequence of frames $F_0 ... F_k$. The index $k \in \mathbb{N}$ is called the maximal frame index and every frame F_i $(0 \leq i \leq k)$ overapproximates the i-step reachable states R_i from an initial location.

Definition 3.1.3 (IC3 Invariants).
The IC3 algorithm incrementally derives a sequence of frames $F_0, ..., F_k$, which has to satisfy the following invariants

- $I \Rightarrow F_0$,
- $\forall 0 \leq i < k.\ F_i \Rightarrow F_{i+1}$,
- $\forall 0 \leq i \leq k.\ F_i \Rightarrow P$,
- $\forall 0 \leq i < k.\ F_i \wedge T \Rightarrow F'_{i+1}$.

The IC3 algorithm preserves the invariants shown in Def. 3.1.3 for the frame sequence $F_0 ... F_k$. First, the frame F_0 is equal to initial states I, such that it is the most precise overapproximation of reachable states. Second, a frame F_{i+1} contains all states reachable with at most $i + 1$ steps, which by definition also includes the i-step reachable states F_i. Furthermore, the frame F_{i+1} contains at least all one-step successor of the states in F_i. Finally, every frame has to satisfy the desired property P. Otherwise, the IC3 algorithm terminates with the result $false$. The IC3 invariants are graphically depicted in Fig. 3.1.1.

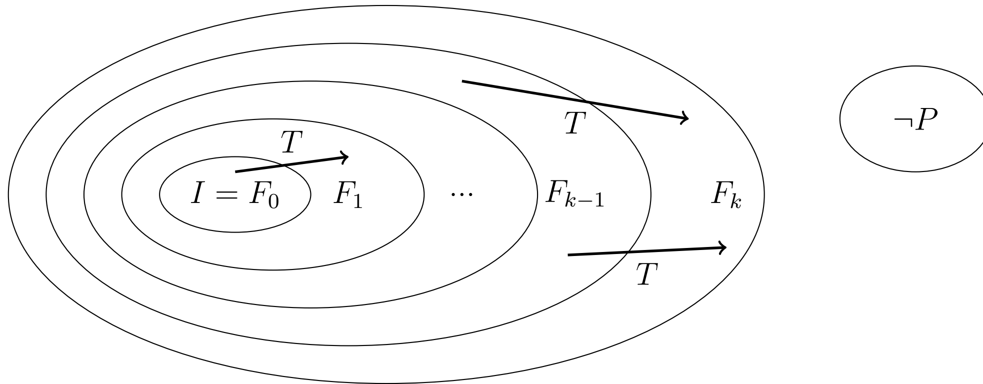

Figure 3.1.1.: Illustration of the IC3 invariants.

Definition 3.1.4 (Counterexample to Induction (CTI)).
Let $TS = (\bar{x}, I, T)$ be a transition system. A cube c over the set of variables $\bar{x}$ is a counterexample to induction at index $i \in \mathbb{N}$ with respect to the propositional property P, iff

$$sat(F_i \wedge c \wedge T \wedge \neg P')$$

Program states in the frame F_k, from which we can reach a one step successor state in $\neg P$, are called counterexamples to induction (Def. 3.1.4). The CTIs are formally given in terms of a set of cubes, i.e. a (quantifier-free) first-order formulas. An example CTI c is illustrated in Fig. 3.1.2.

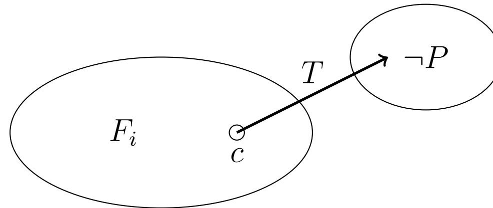

Figure 3.1.2.: Illustration of a CTI c.

Each CTI c has to be blocked in the frame F_k to prove the correctness of the given program. In terms of IC3, the algorithm tries to strengthen the frame sequence $F_0 ... F_i$ to exclude c and all its recursive predecessors. If it fails to block c, a state violating the property is reachable from an initial state of the transition system. Respectively, the IC3 algorithm provides a counterexample path in the transition system (Def. 3.1.5) [Bra11].

Definition 3.1.5 (Counterexample)**.**
Let $n \in \mathbb{N}$ and $F_0 ... F_n$ be a sequence of frames derived by the IC3 algorithm. A counterexample is formally defined as a sequence of concrete (program) states $\sigma_0 ... \sigma_n$ in the transition system, such that

- $\forall 0 \leq i \leq n.\ \sigma_i \models F_i$
- $\forall 0 \leq i < n.\ (\sigma_i, \sigma_{i+1}) \models T$
- $\sigma_n \models \neg P$

The pseudocode of the IC3 main function is shown in Alg. 3.1.1 [Bra11]. The function returns *true*, iff the given transition system TS satisfies the desired property P. Otherwise, it returns $false$ and provides a counterexample.

Algorithm 3.1.1 The main function of the original IC3 algorithm.

```
 1: function BOOL MAIN(TS, P)
 2:     if sat(I ∧ ¬P) ∨ sat(I ∧ T ∧ ¬P') then          ▷ check 0-/1-step cex.
 3:         return false
 4:     F_0 ← {I}, F_1 ← {P}                              ▷ initialise frames
 5:     for k = 1...∞ do
 6:         if ¬strengthen(TS, P, k) then                 ▷ block CTIs
 7:             return false
 8:         F_{k+1} ← {P}
 9:         propagate(TS, P, k)
10:         if ∃1 ≤ i ≤ k. F_i = F_{i+1} then             ▷ check termination
11:             return true
12:     end for
13: end function
```

At the beginning, the IC3 algorithm checks, if there exists a 0-/1-step counterexamples. A 0-step counterexample is a concrete state σ, which satisfies the initial condition I and violates P. A 1-step counterexample is a direct successor of an initial state, which violates the property P. The main loop of the algorithm iterates over the maximum frame index $k \in \mathbb{N}$. Basically, it is separated into a blocking and a propagation phase. In the first phase, the algorithm tries to strengthen the frame sequence $F_0 ... F_k$ to block all CTIs. If the strengthening has been successful, we increase the index k by one and continue with the second phase. In the propagation phase, we push as many cubes as possible to the next frame. The whole algorithm terminates with the result *true*, if we have found an invariant, i.e. a fixpoint in the frame sequence $F_0 ... F_k$.

Algorithm 3.1.2 The strengthen function of the original IC3 algorithm.

```
1: function BOOL STRENGTHEN(TS, P, k)
2:     Queue ← {(c, k) | c is CTI}                    ▷ initialise queue with CTIs
3:     while Queue ≠ ∅ do
4:         (c, i) ← Queue.pop()                        ▷ get element from queue
5:         if i = 0 then
6:             return false
7:         if sat(F_{i−1} ∧ ¬c ∧ T ∧ c') then
8:             Queue ← Queue ∪ {(ĉ, i − 1) | sat(F_{i−1} ∧ ¬c ∧ ĉ ∧ T ∧ c')}
9:             Queue ← Queue ∪ {(c, i)}
10:        else
11:            g ← genMicIC3(TS, c, i)                 ▷ generalise c
12:            F_i ← F_i ∪ {¬g}
13:    end while
14:    return true
15: end function
```

Algorithm 3.1.2 shows the pseudocode of the function *strengthen*. The queue is initialised with all CTIs, which have to be blocked at the frame F_k. An element (c, i) in the queue is called proof obligation of the cube c and index $i \in \mathbb{N}$. For each proof obligation (c, i), we block the cube c in the frame F_i, if c is relative inductive with respect to the predecessor frame F_{i-1}. Otherwise, we compute the predecessor cubes violating the relative inductiveness and add new proof obligations to the queue, respectively. If we get a proof obligation with index $i = 0$, we have reached an initial state of the program. Thus, the property P is violated and the function returns $false$ including a counterexample. If all CTIs are blocked, i.e. the queue is empty, the function returns $true$ to indicate the successful strengthening of the frame sequence $F_0 ... F_k$. Note that the generalisation function $genMicIC3$ is considered in detail in Sec. 3.2. For the moment, we assume that $g = c$.

Algorithm 3.1.3 The propagate function of the original IC3 algorithm.

```
1: procedure PROPAGATE(TS, P, k)
2:     for i = 1...k do
3:         for ¬c ∈ F_i do
4:             if unsat(F_i ∧ T ∧ ¬c') then
5:                 F_{i+1} ← F_{i+1} ∪ {¬c}
6:         end for
7:     end for
8: end procedure
```

Algorithm 3.1.3 shows the pseudocode of the function *propagate* [Bra11], which tries to propagate known information to a frame with an higher index. We move a cube c to the next frame F_{i+1}, iff it is relative inductive with respect to the frame F_i.

3.2. Generalisation

The generalisation is one of the key components of IC3, because it allows us to block multiple program states simultaneously. It relies on the aggressive application of SAT-solving, such that we abstract from the concrete program state.

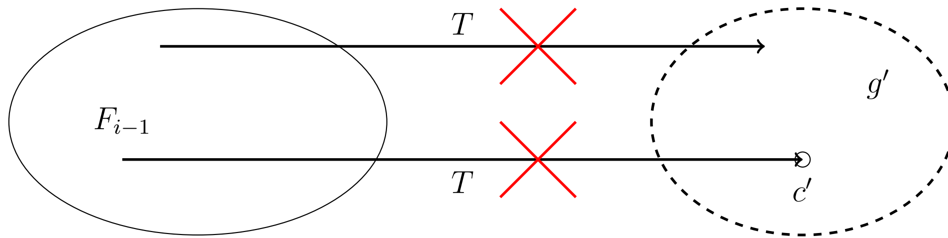

Figure 3.2.1.: Intuitive notion of the IC3 generalisation.

In this section we introduce the standard generalisation function of the IC3 algorithm (Alg. 3.1.1), where we block a generalised cube g with respect to the original cube c. More precise, we call the generalisation algorithm in the function *strengthen* (Alg. 3.1.2) previous to adding the negated cube to the frame F_i. Thereby, the cube c, which has to be blocked at index i, is relative inductive with respect to the transition relation T and the predecessor frame F_{i-1}, i.e. $sat(F_{i-1} \wedge \neg c \wedge T \wedge c') = false$. The computed generalisation g has to satisfy the following conditions. First, it has to be a syntactical subset of the original cube, such that we guarantee to block at least c. Second, the generalised cube g has to be still relative inductive with respect to the predecessor frame F_{i-1}, i.e. $sat(F_{i-1} \wedge \neg g \wedge T \wedge g')$ yields $false$. Intuitively, the relative inductiveness asserts to block only unreachable states, such that the frame F_i remains a safe overapproximation after adding the clause $\neg g$. The intuitive notion of the generalisation is illustrated in Fig. 3.2.1.

Algorithm 3.2.1 The generalisation function of the original IC3 algorithm.

```
1: function CUBE GENMICIC3(TS, c, i)
2:     for p ∈ c do
3:         ĉ ← c \ {p}                                   ▷ drop literal p
4:         c ← down(TS, c, ĉ, i)
5:     end for
6:     return c
7: end function
8: function CUBE DOWN(TS, c, ĉ, i)
9:     while true do
10:        if sat(I ∧ ĉ) then
11:            return c
12:        else if unsat(F_{i−1} ∧ ¬ĉ ∧ T ∧ ĉ′) then
13:            return ĉ
14:        else
15:            S ← {s | sat(F_{i−1} ∧ ¬ĉ ∧ s ∧ T ∧ ĉ′)}    ▷ get predecessors
16:            for s ∈ S do
17:                ĉ ← ĉ ⊔ s                          ▷ compute least upper bound
18:            end for
19:    end while
20: end function
```

The pseudocode of the standard generalisation function $genMicIC3$ is shown in Alg. 3.2.1 [HBS13]. Given a cube c to be blocked, we compute the generalisation g by dropping one literal $p \in c$ after the other. We call the function *down* (Alg. 3.2.1) and check, if the temporary cube $\widehat{c}$ is still relative inductive with respect to the predecessor frame. If $\widehat{c}$ is relative inductive, then we continue with the cube $\widehat{c}$. Otherwise, we compute the predecessor cubes S of c, which violate the relative inductiveness. The least upper bound $\sqcup$ is the syntactical intersection of two cubes, i.e. $c_1 \sqcup c_2 = c_1 \cap c_2$. Within the context of the generalisation, it yields an overapproximation of the temporary cube $\widehat{c}$ and all predecessors S. However, if the temporary cube $\widehat{c}$ implies an initial state, we fall back to the previous cube c. Note that the resulting generalisation g is a syntactical subset $g \subseteq c$ of the original cube c, such that g always implies c. Beside the introduced standard generalisation algorithm, there is also an optimised function, which potentially performs exponentially better [BM07].

3.3. SMT

The original IC3 algorithm has been designed for checking bit-level hardware systems, i.e. it restricts to Boolean variables and propositional logic. An approach to lift the IC3 algorithm to software model checking has been presented in [CG12]. Therefore, we make use of an SMT (satisfiable modulo theories) solver, which for example supports Linear Real Arithmetic (LRA) and bit-vector (BV) logic. The remaining lifting of the original IC3 algorithm is straightforward [CG12], except the predecessor computation becomes non-trivial. To get the predecessor cubes of a cube c with respect to T, we have to compute the entire preimage or an under-approximation of the preimage. While there is a trade-off between the computational effort and the precision, a common approach uses quantifier elimination for theories supporting this elimination [CG12, WHM13, Mon08]. However, the quantifier elimination makes the whole algorithm theory-dependent. In this thesis we will present a theory independent approach based on our model checking framework.

4. IC3CFA

In this chapter we present the basic IC3CFA algorithm, which has been originally introduced in [LNN15]. IC3CFA is based on the original IC3 algorithm, where the program is given in terms of a CFA. In the remainder of this chapter, we assume that the CFA is specified by (L, G, l_{init}, l_E) (Def. 4.1.1).

First, we adjust the definition of frames to incorporate the CFA information (Sec. 4.1). In Sec. 4.2 we show the pseudocode of the standard IC3CFA functions and give an explanation. Analogously, we introduce the IC3CFA generalisation function (Sec. 4.3). The whole IC3CFA algorithm is embedded into an existing framework, which is outlined in Sec. 4.4.

4.1. Frames

In this section we extend the original IC3 frames to incorporate the CFA.

Definition 4.1.1 (Frames).
Let $i \in \mathbb{N}$ be an index and (L, G, l_{init}, l_E) be a CFA. Furthermore, let $R_{(i,l)}$ denote the set of reachable state at location $l \in L$ within i steps from the initial one l_{init}. A frame $F_{(i,l)}$ at index i and location l is a clauseset, which overapproximates the reachable states at the corresponding location l, i.e.

$$R_{(i,l)} \Rightarrow F_{(i,l)}$$

An IC3CFA frame is characterised by an index $i \in \mathbb{N}$ and a location $l \in L$ in the given CFA. Similar to an original IC3 frame, it contains the negation of blocked cubes, i.e. it is a clauseset representing a (quantifier-free) first-order formula. All frames are represented by a two dimensional matrix of index and location. The symbol $\top$ denotes the empty clauseset $\top = \emptyset \equiv true$. In contrast, $\bot$ is the empty frame $\bot = \{\neg true\} \equiv false$. Each frame $F_{(i,l)}$ overapproximates the reachable states at location $l \in L$ within i-steps from the initial location l_{init}. An example IC3CFA frame is graphically depicted in Fig. 4.1.1.

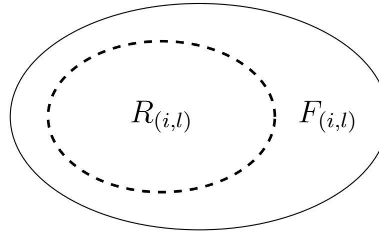

Figure 4.1.1.: Illustration of a single frame.

Given a location $l \in L$ in the CFA, the IC3CFA algorithm derives a sequence of frames $F_{(0,l)}...F_{(k,l)}$ similar to the original IC3 algorithm (Fig. 4.1.2). The index $k \in \mathbb{N}$ again denotes the maximum frame index.

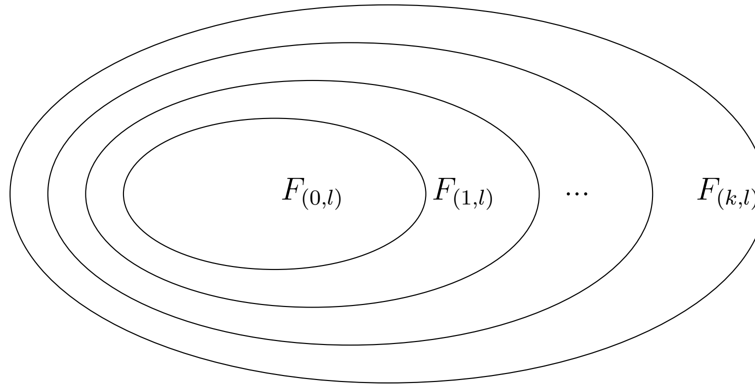

Figure 4.1.2.: Illustration of a frame sequence.

Definition 4.1.2 (Frame Relation)**.**
Let $i, j \in \mathbb{N}$ be two indices and $l, l' \in L$ be two locations in the CFA (L, G, l_{init}, l_E). Given the frames $F_{(i,l)}$ and $F_{(j,l')}$, we define the syntactic frame relation $\sqsubseteq_F$ by

$$F_{(i,l)} \sqsubseteq_F F_{(j,l')} \quad \Leftrightarrow \quad F_{(j,l')} \subseteq F_{(i,l)}$$

To reason about the syntactical relation between two frames $F_{(i,l)}$ and $F_{(j,l')}$, we introduce a relational symbol $\sqsubseteq_F$ (Def. 4.1.2). Basically, it inverts the common subset relation with respect to the sets, such that it reflects the intuitive notation of a greater frame. An example is shown in the following (Ex. 4.1.3).

Example 4.1.3.
Let $i, j \in \mathbb{N}$ be two indices and (L, G, l_{init}, l_E) be a CFA. The frame $F_{(i,l)} = \{\neg c_1\}$ contains only the blocked cube c_1, i.e. the negation of c_1. Respectively, the frame $F_{(j,l')} = \{\neg c_1, \neg c_2\}$ contains two blocked cubes c_1 and c_2. It holds that

$$\{\neg c_1\} \subseteq \{\neg c_1, \neg c_2\} \quad \Rightarrow \quad F_{(i,l)} \subseteq F_{(j,l')} \quad \Rightarrow \quad F_{(j,l')} \sqsubseteq_F F_{(i,l)}.$$

In consequence, the frame $F_{(i,l)}$ is greater than $F_{(j,l')}$, i.e. $F_{(i,l)}$ contains more states.

4.2. Main Algorithm

In this section we introduce the main algorithm of IC3CFA [LNN15].

Definition 4.2.1 (Relative Inductiveness)**.**
Let $i \in \mathbb{N}$ be an index and (L, G, l_{init}, l_E) be a CFA with edge $e = (l, cmd, l') \in G$. Given the frame $F_{(i-1,l)}$ and a cube c, we define a function $relInd(F_{(i-1,l)}, e, c)$, such that

- $l \neq l'$:
$$relInd(F_{(i-1,l)}, e, c) \quad \Leftrightarrow \quad unsat(F_{(i-1,l)} \wedge T_e \wedge c')$$
- $l = l'$:
$$relInd(F_{(i-1,l)}, e, c) \quad \Leftrightarrow \quad unsat(F_{(i-1,l)} \wedge \neg c \wedge T_e \wedge c')$$

The cube c is called relative inductive with respect to the edge e, iff the function $relInd(F_{i-1,l}, e, c)$ evaluates to true.

First, we have to adjust the definition of relative inductiveness to the CFA (Def. 4.2.1). We check the relative inductiveness with respect to a specific edge $e \in G$ in the CFA, where we distinguish between e being a self-loop or not.

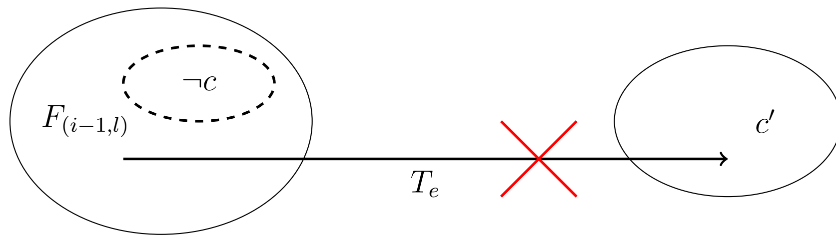

Figure 4.2.1.: Graphical representation of a relative inductive cube c.

Given a relative inductive cube c at index $i \in \mathbb{N}$, there is no satisfiable predecessor state with respect to the edge $e = (l, cmd, l') \in G$ and frame $F_{(i-1,l)}$. Intuitively, there is no transition T_e from a state in $F_{(i-1,l)}$ to a state in c' (Fig. 4.2.1).

Similar to the original IC3 algorithm, IC3CFA preserves the following invariants for all frames (Def. 4.2.2).

Definition 4.2.2 (IC3CFA Invariants)**.**
Let (L, G, l_{init}, l_E) be a CFA. The IC3CFA algorithm incrementally derives a sequence of frames $F_{(0,l)}, ..., F_{(k,l)}$ for all locations $l \in L$, which have to satisfy the following invariants

- $F_{(0,l_{init})} = \top,\ \forall l \neq l_{init}.\ F_{(0,l)} = \bot,$
- $\forall l \in L,\ 0 \leq i < k.\ F_{(i,l)} \Rightarrow F_{(i+1,l)},$
- $\forall 0 \leq i \leq k.\ \neg\exists F_{(i,l_E)},$
- $\forall l' \in L \setminus \{l_E\},\ e = (l, cmd, l') \in G,\ 0 \leq i < k.\ F_{(i,l)} \wedge T_e \Rightarrow F'_{(i+1,l')}.$

At index $i = 0$, all locations in the CFA, except the initial one l_{init}, are not reachable, such that the corresponding frames are set to $\bot \equiv false$. In contrast, the frame of the initial location $F_{(0,l_{init})}$ is set to the $\top \equiv true$. Every frame $F_{(i+1,l)}$ with respect to a specific location $l \in L$ always includes the locations reachable within i-steps, i.e. $F_{(i,l)} \Rightarrow F_{(i+1,l)}$. We omit the frame for the error location l_E, because it implicitly represents the undesired behaviour of the given program. If it is yet reachable, the algorithm provides a counterexample, respectively [LNN15]. Given an edge $e = (l, cmd, l') \in G$ between two locations $l, l' \in L$, the frame $F_{(i+1,l')}$ at least includes all successor states of the states specified by $F_{(i,l)}$.

Algorithm 4.2.1 The outer loop of the IC3CFA algorithm.

1: **function** BOOL OUTERLOOP(A)
2:     **if** $l_{init} = l_E \vee (\exists e = (l_{init}, cmd, l_E) \in G.\ sat(T_e))$ **then**   ▷ check 0-/1-step cex.
3:         **return** $UNSAFE$
4:     $F_{(0,l_{init})} \leftarrow \top, \forall l \neq l_{init}.\ F_{(0,l)} \leftarrow \bot$ ▷ initialise frames
5:     $\forall l.\ F_{(1,l)} \leftarrow \top$
6:     **for** $k = 1...\infty$ **do**
7:         **if** $\neg innerLoop(A, k)$ **then**   ▷ block CTIs
8:             **return** $UNSAFE$
9:         $\forall l.\ F_{(k+1,l)} \leftarrow \top$
10:        **if** $\exists 1 \leq i \leq k. \forall l.\ F_{(i,l)} = F_{(i+1,l)}$ **then**   ▷ check termination
11:            **return** $SAFE$
12:    **end for**
13: **end function**

The main function of the IC3CFA algorithm is shown in Alg. 4.2.1 [LNN15], where $A = (L, G, l_{init}, l_E)$ is the CFA representing the given program. Basically, this function *outerLoop* is similar to the IC3 main function (Def. 3.1.1). We first check, if there are any 0-/1-step counterexamples. A 0-step counterexample exists, if the initial location l_{init} equals the error location l_E. Respectively, a realisable edge $e = (l_{init}, cmd, l_E) \in G$ between both locations l_{init} and l_E indicates a 1-step counterexample. Note that an edge e is called realisable, if the corresponding transition formula T_e is satisfiable. At the beginning of the algorithm, the initial state l_{init} is the only reachable one, such that we initialise the frame $F_{(0,l_{init})}$ with $true$. All other locations $l \neq l_{init}$ are initialised with $false$. In the following, all frames $F_{(0,l)}$ with an index $i > 0$ are initialised with the trivial overapproximation $true$. The function *outerLoop* iterates over the maximal frame index $k \in \mathbb{N}$ and tries to block all CTIs in each iteration. Therefore, the function *innerLoop* is called, which is similar to the IC3 function *strengthen* (Def. 3.1.2). If the strengthening of the frames fails, the algorithm returns $UNSAFE$ including a counterexample. The unreachability of the error location l_E is proven, if the frames remain constant for a certain index $i \in \mathbb{N}$ and all locations in the CFA. If such a fixpoint is found, the algorithm returns $SAFE$ to indicate the correctness.

Algorithm 4.2.2 The inner loop of the IC3CFA algorithm.

```
1:  function BOOL INNERLOOP(A, k)
2:      Queue ← {(c, k, l) | e = (l, cmd, l_E) ∈ G,  c = getPredecessor(e, true)}
3:      while Queue ≠ ∅ do
4:          (c, i, l') ← Queue.pop()                      ▷ get element from queue
5:          if i = 0 then
6:              return false
7:          for e = (l, cmd, l') ∈ G do                   ▷ loop over predecessor edges
8:              if relInd(F_(i−1,l), e, c) then
9:                  g ← gen(i, l', c)                      ▷ generalise c
10:                 F_(i,l') ← F_(i,l') ∪ {¬g}
11:             else
12:                 Queue ← Queue ∪ {(ĉ, i − 1, l) | ĉ = getPredecessor(e, c)}
13:                 Queue ← Queue ∪ {(c, i, l')}
14:         end for
15:     end while
16:     return true
17: end function
```

The pseudocode of the function *innerLoop* is shown in Alg. 4.2.2 [LNN15]. The queue is initialised with a proof obligation for each counterexample to induction

(CTI), where a CTI is a predecessor cube of the error location l_E. A proof obligation (c, i, l) is formally a three tuple, which specifies the cube c to be blocked at index $i \in \mathbb{N}$ and location $l \in L$. The function *innerloop* iterates over the queue, until all proof obligations are processed. Note that the function *Queue.pop*() always returns the proof obligation with the minimal index in the queue. Given a cube c to be blocked, the cube c has to be relative inductive with respect to all predecessor edges. Otherwise, the feasible predecessor cubes of c are added as new proof obligation to the queue. After successfully blocking all predecessor cubes, the clause $\neg c$ is added to the frame $F_{(i,l)}$. Note that the generalisation of c is covered in Sec. 4.3.

Example 4.2.3.
We consider the example program P_{true} (Alg. 2.3.1) with the CFA A_{true} (Fig. 2.3.1).

i: \ l:	l_{init}	l_1
0	$\top$	$\bot$
1	$\top$	$\{\neg(y \geq x \wedge x = 1),$ $\neg(y \geq x \wedge x = 2)\}$
2	$\top$	$\{\neg(y \geq x \wedge x = 1),$ $\neg(y \geq x \wedge x = 2)\}$

Table 4.1.: Resulting frames of the IC3CFA algorithm.

Table 4.1 shows the resulting frames of the IC3CFA algorithm. At the beginning, we initialise the frame $F_{(0,l_1)}$ with the empty frame, i.e. $F_{(0,l_1)} = \bot = \{\neg true\}$. In contrast, the frame $F_{(0,l_{init})}$ of the initial location l_{init} is initialised with $\top = \emptyset$. We introduce the abbreviations $e_1 = (l_{init}, assume\ y < x, l_1)$ and $e_2 = (l_1, y := y - 1, l_1)$. At the first iteration $i = 1$, we derive two CTIs $c_1 = y \geq x \wedge x = 1$ and $c_2 = y \geq x \wedge x = 2$, which is explained in detail in Sec. 5.2 [Lei05]. Both CTIs have to be blocked at location l_1. The first cube c_1 yields $relInd(\top, e_1, c_1) = true$ and $relInd(\bot, e_2, c_1) = true$, such that the negation of c_1 is added to the frame $F_{(1,l_1)}$. Respectively, the same result holds for the cube c_2. After blocking all CTIs, we continue with the second iteration $i = 2$. The relative inductiveness checks for the CTI c_1 are $relInd(\top, e_1, c_1)$ and $relInd(\{\neg(y \geq x \wedge x = 1), \neg(y \geq x \wedge x = 2)\}, e_2, c_2)$, where both evaluate to *true*. The same holds for the second CTI c_2.

For $i = 1$ and $i = 2$ we get the same frames, such that the algorithm terminates with the result *true*.

4.3. Generalisation

Based on the IC3CFA main algorithm (Sec. 4.2), we introduce the IC3CFA generalisation function.

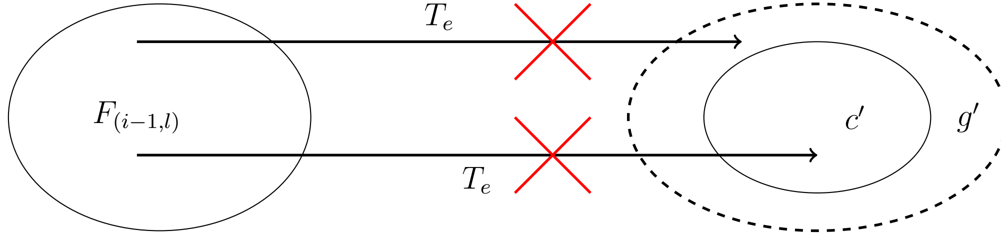

Figure 4.3.1.: Intuitive notion of the IC3CFA generalisation.

Intuitively, given a cube c at index $i \in \mathbb{N}$, the generalisation function tries to find a generalisation g including more states than the original cube c. However, the generalisation g still has to be relative inductive with respect to a specific a specific edge $e = (l, cmd, l') \in G$ and the corresponding frame $F_{(i-1,l)}$. The intuitive notion of the generalisation is illustrated in Fig. 4.3.1.

Definition 4.3.1 (Generalisation)**.**
Let $i \in \mathbb{N}$ be an index and (L, G, l_{init}, l_E) be a CFA. Let $e = (l, cmd, l') \in G$ be an edge and c be a cube to be blocked at location $l' \in L$. An (edge-based) generalisation $g = gen_{(i,l')}(c, e)$ of c with respect to the index i and edge e is a cube, where the function $gen_{(i,l')}(c, e)$ satisfies the following conditions:

- the function $gen_{(i,l')}(c, e)$ is deterministic, i.e.

$$\forall cubes\ g, \widehat{g}.\ g = gen_{(i,l')}(c, e) \wedge \widehat{g} = gen_{(i,l')}(c, e) \quad \Rightarrow \quad g = \widehat{g}$$

- the generalisation $g = gen_{(i,l')}(c, e)$ is a subset of the original cube c, i.e.

$$g = gen_{(i,l')}(c, e) \subseteq c$$

- the generalisation $g = gen_{(i,l')}(c, e)$ is relative inductive with respect to the edge e, i.e.

$$relInd(F_{(i-1,l)}, e, c) \Rightarrow relInd(F_{(i-1,l)}, e, g)$$

The formal definition of a generalisation $g = gen_{(i,F_{(i-1,l)})}(c,e)$ with respect to a specific edge $e \in G$ is shown in Def. 4.3.1. Although not explicitly stated by the original IC3 algorithm, we assume that the function $gen_{(i,F_{(i-1,l)})}(c,e)$ computing the generalisation has to be deterministic. This deterministic behaviour guarantees reproducible results. Furthermore, we restrict the generalised cube g to be a syntactical subset of the original cube c, i.e. $g \subseteq c$. Finally, the generalisation g still has to be inductive relative to the frame $F_{(i-1,l)}$.

Definition 4.3.2 (Location-based Generalisation)**.**
Let $i \in \mathbb{N}$ be an index and (L, G, l_{init}, l_E) be a CFA. The location-based generalisation of the cube c and location $l' \in L$ is defined by

$$gen_{(i,l')}(c) = \bigcup_{e \in G} gen_{(i,l')}(c,e)$$

Since we block a generalised cube $g \subseteq c$ at a specific CFA location $l' \in L$, we have to combine the (edge-based) generalisation of all predecessor edges. The so called location-based generalisation $gen_{(i,l')}(c)$ is introduced in Def. 4.3.2, where the cube c has to be generalised a index $i \in \mathbb{N}$ and location $l' \in L$. Note that $gen_{(i,l')}(c)$ is also a syntactical subset of the original cube c, i.e. $gen_{(i,l')}(c) \subseteq c$.

Algorithm 4.3.1 The generalisation of the IC3CFA algorithm.

```
1: function CUBE GEN(i, l', c)
2:     res ← true
3:     for e = (l, cmd, l') ∈ G do
4:         if |c| > 4 then                          ▷ choose generalisation method
5:             res ← res ∪ genBinary(F_(i−1,l), e, c, true)
6:         else
7:             res ← res ∪ genMic(F_(i−1,l), e, c, true)
8:     end for
9:     return res
10: end function
```

Algorithm 4.3.1 shows the entry point of the IC3CFA generalisation, which is called by the function *innerLoop* (Alg. 4.2.2). For cubes with at most three literals, we choose the default generalisation function *genMic*, which is similar to the IC3 generalisation (Alg. 3.2.1). Otherwise, for larger cubes we call the function *genBinary* (Alg. 4.3.3), which potentially performs exponentially better [BM07].

Algorithm 4.3.2 The default generalisation function of the IC3CFA algorithm.

```
1: function CUBE GENMIC(F, e, c, support)
2:     res ← c ∪ support
3:     for p ∈ c do                                  ▷ loop over literals
4:         tmp ← res \ {p}
5:         if relInd(F, e, tmp) then
6:             res ← tmp
7:     end for
8:     return res
9: end function
```

Given a cube c, the IC3CFA default generalisation function $genMic$ tries to drop each literal p of c. If the reduced cube is still relative inductive with respect to the given edge $e \in G$ and frame $F_{(i-1,l)}$, the algorithm continues with the reduced cube. Otherwise, it falls back to the previous cube including the literal p under consideration.

Algorithm 4.3.3 The additional generalisation function of the IC3CFA algorithm.

```
1: function CUBE GENBINARY(F, T, c, support)
2:     if |c| ≤ 1 then
3:         return c ∪ support
4:     else
5:         (l, r) ← split(c)
6:         if relInd(F, e, support ∪ l) then
7:             split(F, T, l, support)
8:         else if relInd(F, e, support ∪ r) then
9:             split(F, T, r, support)
10:        else
11:            l ← split(F, T, l, support ∪ r)
12:            r ← split(F, T, r, support ∪ l)
13:            return l ∪ r
14: end function
```

Algorithm 4.3.3 shows the additional generalisation function $genBinary$, which potentially performs exponentially better than the default one [BM07]. In particular, this function is used for larger cubes. Intuitively, the function splits the given cube c similar to a binary search and checks the relative inductiveness for each part. Thus, this functions allows to drop more than one literal with a single relative inductiveness check.

Example 4.3.3.
We consider the example program P_{true} (Alg. 2.3.1) with the CFA A_{true} (Fig. 2.3.1) again.

i: \ l:	l_{init}	l_1
0	$\top$	$\bot$
1	$\top$	$\{\neg(y \geq x)\}$
2	$\top$	$\{\neg(y \geq x)\}$

Table 4.2.: Resulting frames of the IC3CFA algorithm with generalisation.

Table 4.2 shows the resulting frames of the IC3CFA algorithm with generalisation. We introduce the variables $e_1 = (l_{init}, assume\ y < x, l_1), e_2 = (l_1, y := y - 1, l_1)$ for the predecessor edges of the location l_1.

At the first iteration $i = 1$, we try to block the cube $c_1 = y \geq x \wedge x = 1$ at location l_1. The generalisation of c_1 results in the cube $gen_{(1,l_1)}(c_1) = y \geq x \subseteq c_1$, i.e. $relInd(\top, e_1, y \geq x) = true$ and $relInd(\bot, e_2, y \geq x) = true$. A generalisation to the cube $true$ is not possible, because $relInd(\top, e_1, true)$ is $false$. The generalisation of the first cube c_1 also blocks the second CTI $c_2 = y \geq x \wedge x = 2$, i.e. $y \geq x \subseteq c_2$.

The second iteration $i = 2$ yields the same result, except that the relative inductiveness checks for the cube c_1 are $relInd(\top, e_1, y \geq x) = true$ and $relInd(\{\neg(y \geq x)\}, e_2, y \geq x) = true$. Overall, the algorithm terminates with the result $true$ again.

4.4. Framework

The IC3CFA algorithm is implemented within an existing proprietary model checking framework, which is graphically illustrated in Fig. 4.4.1 [LNN15].

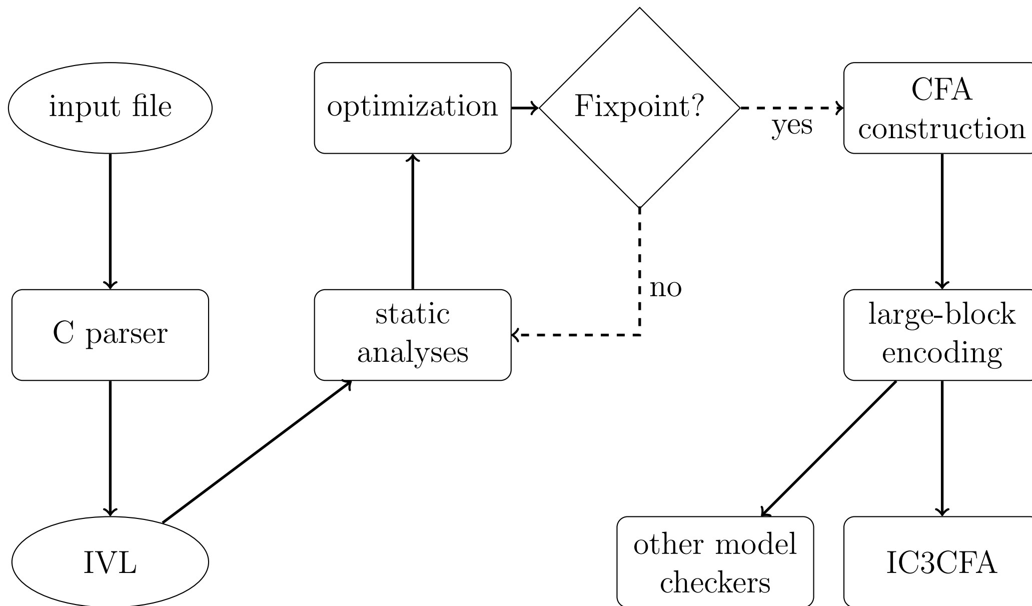

Figure 4.4.1.: Model checking framework.

In this thesis we focus on the verification of C programs, such that the framework takes a C program file as input. The internal C parser translates the program into our own intermediate verification language, short IVL. The representation in terms of IVL makes the program more suitable for further verification analysis. We apply common static program analyses [WCC+95], like expression propagation, needed variables analysis and program slicing. After reaching a fixpoint with respect to these analyses, we convert the resulting IVL program into a CFA labelled with GCL commands (Def. 2.3.1). Additionally, we apply Large-Block Encoding and bisimulation to minimise the resulting CFA [BS92, FV02]. Afterwards, the framework offers a wide range of model checking algorithms including the IC3CFA algorithm. More over, CEGAR ("CounterExample-Guided Abstraction Refinement") [CGJ+00, CGJ+03] and BMC ("bounded model checking") [BCC+03, CBRZ01] are also available in the framework. Similar to IC3CFA, there is another approach, called TreeIC3 [CG12], which is also based on the idea to lift the principles of the original IC3 algorithm to software model checking. However, in contrast to IC3CFA, TreeIC3 makes use of abstract reachability trees (ARTs). Basically, the whole framework is theory unaware and supports also infinite domains, e.g. linear real arithmetic (LRA). We focus on bit-precise analysis, such that we use the theory of bit-vectors (BV). As SMT solver, the framework supports Z3 [MB08] and MathSAT [BCF+08].

5. Generalisation

In this chapter we present several improvements of the original IC3CFA algorithm, which has been introduced in Chapter 4. In general, we try to improve the performance by reducing the number of SMT calls in the IC3CFA generalisation. First, we try to avoid unnecessary SMT calls, where we can already derive the result based on previous computations. Second, a SMT check with respect to a first-order formula scales in worst-case exponentially in the size of the formula [GKSS08]. Thus, we apply different syntactical checks, which can often be divided into sub-tasks and scale better than the alternative SMT check.

5.1. Improved Initialisation

In this section we improve the initialisation of the IC3CFA frames to avoid unnecessary computations in the generalisation.

Definition 5.1.1 (Paths)**.**
Let (L, G, l_{init}, l_E) be a CFA and $n \in \mathbb{N}$. A path π of length $(n+1)$ is a sequence of states $l_0...l_n$, where

- $l_0 = l_{init}$,
- $\forall 0 \leq m \leq n.\ l_m \in L$,
- $\forall 0 \leq m < n.\ (l_m, cmd, l_{m+1}) \in G$.

We introduce the notion of paths in Def. 5.1.1. Intuitively, a path is a sequence of states in the given CFA starting in the initial location l_{init}. Note that a path is only derived by graph analysis, such that it is not necessarily realisable. The set $Paths^{\leq n}$ denotes all paths of the length $(n+1)$ or less, i.e. $Paths^{\leq n} = \{\pi = l_0...l_m \mid m \leq n,\ \pi \textit{ is a path}\}$.

Theorem 5.1.2.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA with location $l' \in L$. It holds that

$$\neg\exists l_0...l' \in Paths^{\leq i} \quad \Rightarrow \quad F_{(i,l')} = \bot$$

Proof.
We prove the theorem by induction over the index i.

- $i = 0 \wedge l' = l_{init}$:

$$\begin{aligned} & l_{init} \in Paths^{\leq 0} \\ \Rightarrow\ & \exists l' \in Paths^{\leq 0} \end{aligned}$$

- $i = 0 \wedge l' \neq l_{init}$:

$$\begin{aligned} & l' \notin Paths^{\leq 0} \\ \Rightarrow\ & \neg\exists l' \in Paths^{\leq 0} \\ \Rightarrow\ & F_{(0,l')} = \bot && \text{(Def. 4.2.2)} \end{aligned}$$

- $i \rightsquigarrow i + 1$:

$$\begin{aligned} & \neg\exists l_0 ... l' \in Paths^{\leq i+1} \\ \Rightarrow\ & \forall (l, cmd, l') \in G.\ \left(\neg\exists l_0 ... l \in Paths^{\leq i}\right) \\ \Rightarrow\ & \forall (l, cmd, l') \in G.\ F_{(i,l)} = \bot && \text{(by hypothesis)} \\ \Rightarrow\ & \forall (l, cmd, l') \in G.\ relInd(F_{(i,l)}, e, true) \\ \Rightarrow\ & F_{(i+1,l')} = \{\neg true\} \\ \Rightarrow\ & F_{(i+1,l')} = \bot \end{aligned}$$

□

Let $i \in \mathbb{N}$ be an index and $l' \in L$ be a location in the CFA. Furthermore, we assume that the shortest path between the location l' and the initial one l_{init} is of length $n \in \mathbb{N}$. The improved initialisation is based on the following observation. Since the state l' is not reachable within less than n steps, $\bot = \{\neg true\}$ is a safe over-approximation for all frames $F_{(m,l')}$ and $m < n$. In consequence, we save unnecessary computations for these frames, e.g. the generalisation of cubes to be blocked in one of these frames. The improved initialisation is formally specified by Th. 5.1.2.

Example 5.1.3.
We consider an example program P_{false}, where the pseudocode is shown in Alg. 5.1.1. The corresponding CFA A_{false} of P_{false} is shown in Fig. 5.1.1.

Algorithm 5.1.1 Pseudocode of the example program P_{false}.

```
1: procedure MAIN(x)
2:     assert(x ≥ 0)
3:     x ← x + 1
4:     while nondetBool do
5:         x ← 4 * x
6:     end while
7:     x ← x − 1
8:     assert(x ≠ 3)
9: end procedure
```

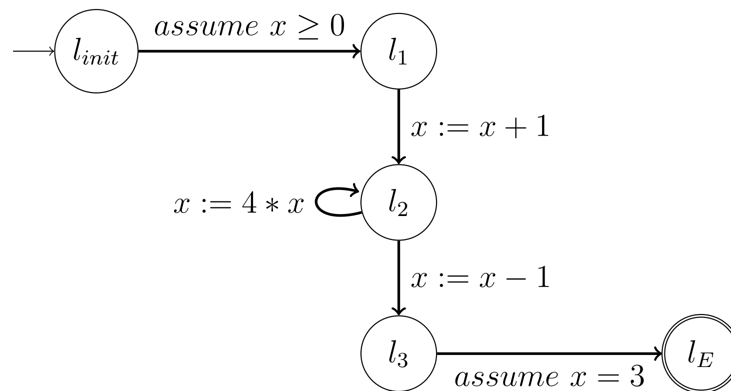

Figure 5.1.1.: The CFA A_{false} of the example program P_{false}.

The application of the IC3CFA algorithm with improved initialisation yields the frames shown in Table 5.1. For example, the location $l_3 \in L$ is reachable with at least 3 steps from the location l_{init}, such that we initialise both frames $F_{(1,l_3)}$ and $F_{(2,l_3)}$ with $\bot$.

i: \ l:	l_{init}	l_1	l_2	l_3
0	$\top$	$\bot$	$\bot$	$\bot$
1	...	...	$\bot$	$\bot$
2	...	...	...	$\bot$
3	...	...	...	...

Table 5.1.: Resulting frames of the IC3CFA algorithm with improved initialisation.

5.2. Predecessor Computation

So far, the predecessors of a cube c with respect to an edge e have been computed by the function $getPredecessor(e, c)$ (Alg. 4.2.2). In this section we consider the concrete implementation of this function. In general, our aim is to present a theory-unaware IC3CFA algorithm. Thus, we introduce the notion of weakest preconditions to compute a safe over-approximation of the entire preimage [Dij76]. Note that the original IC3 algorithm only computes a under-approximation of the entire preimage. Since we consider the blocking of cubes in the IC3CFA algorithm, we also present an optimisation for converting the result into a set of cubes.

Definition 5.2.1 (Weakest Precondition (WP))**.**
Let c be a cube and *cmd* be a GCL command. The weakest precondition $wp(cmd, c)$ is a quantifier-free first-order formula, which specifies all predecessor states of c with respect to *cmd*. More formally, let σ, σ' be two concrete program states, where $\sigma' \models c$. It holds that

$$(\forall \rightarrow .\ < cmd, \sigma > \rightarrow \sigma') \quad \Leftrightarrow \quad \sigma \models wp(cmd, c)$$

where $\rightarrow$ is the execution relation specified in Def. 2.2.3.

The weakest precondition, short WP (Def. 5.2.1), with respect to a GCL command *cmd* and a first-order formula φ has been originally introduced by Dijkstra in [Dij76]. Intuitively, the WP $wp(cmd, \varphi)$ determines the predecessor states, which for all possible executions $\rightarrow$ induced by *cmd* satisfy the formula φ after execution of *cmd*. However, in IC3CFA we have to block all possible predecessors, i.e. predecessor states reachable with at least one execution $\rightarrow$. Thus, we introduce the weakest existential precondition, short WEP, to model this behaviour correctly.

Definition 5.2.2 (Weakest Existential Precondition (WEP))**.**
Let c be a cube and cmd be a GCL command. The weakest existential precondition $wep(cmd, c)$ is a quantifier-free first-order formula, which specifies all predecessor states of c with respect to cmd. More formally, let σ, σ' be two concrete program states, where $\sigma' \models c$. It holds that

$$(\exists \rightarrow . \ < cmd, \sigma > \rightarrow \sigma') \quad \Leftrightarrow \quad \sigma \models wep(cmd, c)$$

where $\rightarrow$ denotes the execution relation of Def. 2.2.3.

Basically, the definition of the WEP (Def. 5.2.2) is equal to the one of the WP, except that the quantifier $\forall$ is replaced by the existential quantification $\exists$. Intuitively, we get all predecessor states, which are realisable via at least one execution $\rightarrow$. The WEP is graphically illustrated in Fig. 5.2.1.

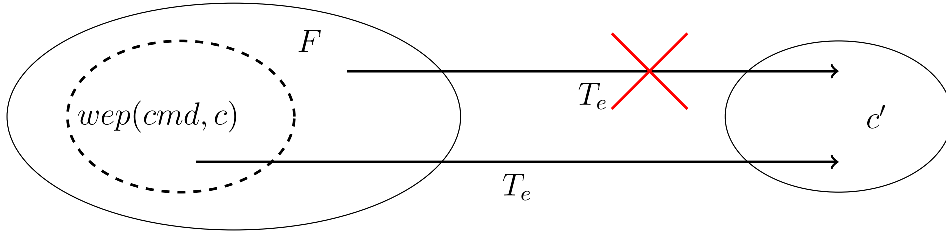

Figure 5.2.1.: Illustration of the WEP of c with respect to the edge $e = (l, cmd, l')$.

Based on our four GCL command types (Def. 2.2.2), the WEP is constructed according to the following rules:

$$\begin{aligned}
wep(assume\ b, \varphi) &= \varphi \wedge b \\
wep(x := a, \varphi) &= \varphi[x \mapsto a] \\
wep(cmd_1; cmd_2, \varphi) &= wep(cmd_1, wep(cmd_2, \varphi)) \\
wep(cmd_1 \Box cmd_2, \varphi) &= wep(cmd_1, \varphi) \vee wep(cmd_2, \varphi)
\end{aligned}$$

The following Example 5.2.3 shows a transformation based on these rules.

Example 5.2.3.
Let $c = (x = 3)$ be a cube with a single literal. We compute the WEP of c in DNF with regard to the GCL command

$$cmd_{ex} = assume\ x \geq 0;\ x := x + 1;\ (x := 4 * x) \Box (assume\ true);\ x := x - 1.$$

Note that the WEP is computed backwards with respect to the given GCL command. The symbol $\Rightarrow$ indicates the application of the sequence rule.

The computation of $wep(cmd_{ex}, c)$ yields

$$\begin{aligned}
& wep(x := x-1, x=3) = (x-1=3) && \text{(assign)}\\
\Rightarrow\ & wep(cmd_{ex}, x=3) = \\
& wep(assume\ x \geq 0;\ x := x+1;\ (x := 4*x)\Box(assume\ true), x-1=3)
\end{aligned}$$

$$\begin{aligned}
& wep((x := 4*x)\Box(assume\ true), x-1=3) = && \text{(choice/ assign/ assume)}\\
& 4*x-1=3 \vee x-1=3 \\
\Rightarrow\ & wep(assume\ x \geq 0;\ x := x+1;\ (x := 4*x)\Box(assume\ true), x-1=3) = \\
& wep(assume\ x \geq 0;\ x := x+1, 4*x-1=3 \vee x-1=3)
\end{aligned}$$

$$\begin{aligned}
& wep(x := x+1, 4*x-1=3 \vee x-1=3) = && \text{(assign)}\\
& 4*(x+1)-1=3 \vee x+1-1=3 \\
\Rightarrow\ & wep(assume\ x \geq 0;\ x := x+1, 4*x-1=3 \vee x-1=3) = \\
& wep(assume\ x \geq 0, 4*(x+1)-1=3 \vee x+1-1=3)
\end{aligned}$$

$$\begin{aligned}
\Rightarrow\ & wep(assume\ x \geq 0, 4*(x+1)-1=3 \vee x+1-1=3) = && \text{(assume)}\\
& (4*(x+1)-1=3 \vee x+1-1=3) \wedge x \geq 0
\end{aligned}$$

Definition 5.2.4 (Split)**.**
Let (L, G, l_{init}, l_E) be a CFA and *cmd* be a GCL command. We define a function $split(cmd)$, which translates the choice command into a set of GCL commands, i.e.

$$split(cmd) = \begin{cases} split(cmd_1) \cup split(cmd_2) & \text{if } cmd = (cmd_1 \Box cmd_2)\\ \{c_1; c_2 \mid i \in \{1,2\},\ c_i \in split(cmd_i)\} & \text{if } cmd = (cmd_1; cmd_2)\\ \{cmd\} & \text{otherwise} \end{cases}$$

Given a GCL command *cmd* with a choice command, this command *cmd* can be split into independent sub-commands without a choice (Def. 5.2.4). It holds that

$$(l, cmd, l') \in G \quad \Leftrightarrow \quad \{(l, cmd', l') \mid cmd' \in split(cmd)\} \subseteq G.$$

Intuitively, an edge $e = (l, cmd, l') \in G$, which is labelled with a GCL command *cmd* including a choice, is separated into multiple edges between the same locations. Each new edge contains no choice command any more, such that it models a deterministic, sequential behaviour.

Definition 5.2.5 (Disjunctive Normal Form (DNF))**.**
Let φ be a quantifier-free first-order formula. The formula φ is in disjunctive normal form, if it is of the form

$$c_1 \vee ... \vee c_n$$

where $n \in \mathbb{N}$ and $c_1, ..., c_n$ are cubes.

In principle, the computation of the WEP yields a first-order formula φ, which is not necessarily a cube. Since the IC3CFA algorithm tries to block cubes, we have to convert the formula φ into possibly multiple cubes. Therefore, we compute the disjunctive normal form (Def. 5.2.5) of φ, such that we get a disjunction of cubes [DP02]. Note that any first-order formula can be converted into DNF, however the resulting DNF is in worst-case exponential in the size of the formula [DP02]. In addition, experimental results have proven that the computation of the DNF itself also requires a significant amount of time, especially for larger and complex formulas.

Given a GCL command *cmd* without a choice and a cube c, the computation of the WEP $wep(cmd, c)$ always yields a single cube (Lemma 5.2.6).

Lemma 5.2.6.
Let cmd be a GCL command without a choice and c be a cube. It holds that

$$dnf(wep(cmd, c)) = wep(cmd, c)$$

where $dnf(\varphi)$ converts the given quantifier-free first-order formula φ into DNF according to the rules presented in [DP02].

Proof.
Note that the GCL command *cmd* contains no choice. We prove the lemma by structural induction over *cmd* without the choice case.

- $cmd = (assume\ b)$:

$$\begin{aligned} & dnf(wep(assume\ b, c)) \\ = \ & dnf(c \wedge b) \\ = \ & c \wedge b && \text{(Def. 2.2.1)} \\ = \ & wep(assume\ b, c) \end{aligned}$$

- $cmd = (x := a)$:

$$\begin{aligned} & dnf(wep(x := a, c)) \\ = \quad & dnf(c[x \mapsto a]) \\ = \quad & c[x \mapsto a] \\ = \quad & wep(x := a, c) \end{aligned}$$

- $cmd = (cmd_1; cmd_2)$:

$$\begin{aligned} & dnf(wep(cmd_1; cmd_2, c)) & \\ = \quad & dnf(wep(cmd_1, wep(cmd_2, c))) & \\ = \quad & wep(cmd_1, wep(cmd_2, c)) & \text{(by hypothesis)} \\ = \quad & wep(cmd_1; cmd_2, c) & \end{aligned}$$

□

We extend the previous lemma to also incorporate the GCL choice command. Intuitively, Th. 5.2.7 states that the WEP of a choice $cmd_1 \Box cmd_2$ results in the disjunction of two cubes.

Theorem 5.2.7.
Let cmd be a GCL command and c be a cube. It holds that

$$dnf(wep(cmd, c)) = \bigvee \{wep(cmd', c) \mid cmd' \in split(cmd)\}$$

where $dnf(\varphi)$ converts the given quantifier-free first-order formula φ into DNF according to the rules presented in [DP02].

Proof.

$$\begin{aligned} & dnf(wep(cmd, c)) & \\ = \quad & dnf(\bigvee \{wep(cmd', c) \mid cmd' \in split(cmd)\}) & \\ = \quad & \bigvee \{dnf(wep(cmd', c)) \mid cmd' \in split(cmd)\} & \\ = \quad & \bigvee \{wep(cmd', c) \mid cmd' \in split(cmd)\} & \text{(Lemma 5.2.6)} \end{aligned}$$

□

Overall, we avoid computing the DNF of a first-order formula in the IC3CFA algorithm. Instead, we derive an equivalent set of cubes based on the structure of the GCL command.

Example 5.2.8.
Let $c = (x = 3)$ be a cube again. Based on the Example 5.2.3, we have computed

$$\varphi = wep(cmd_{ex}, c) = (4 * (x+1) - 1 = 3 \vee x + 1 - 1 = 3) \wedge x \geq 0.$$

By applying the function $dnf(\varphi)$, we transfer the resulting formula φ into DNF [DP02].

$$\begin{aligned} & dnf(wep(cmd_{ex}, x = 3)) \\ = \quad & dnf((4 * (x+1) - 1 = 3 \vee x + 1 - 1 = 3) \wedge x \geq 0) \\ = \quad & (4 * (x+1) - 1 = 3 \wedge x \geq 0) \vee (x + 1 - 1 = 3 \wedge x \geq 0) \end{aligned}$$

In comparison, we apply the function $split(cmd_{ex})$ to the GCL command cmd_{ex}.

$$\begin{aligned} split(cmd_{ex}) = \{ & assume\ x \geq 0;\ x := x + 1;\ x := 4 * x;\ x := x - 1, \\ & assume\ x \geq 0;\ x := x + 1;\ assume\ true;\ x := x - 1\} \end{aligned}$$

The resulting partial GCL commands $cmd' \in split(cmd_{ex})$ are independent of the given cube c, such that they are usually cached. For all partial commands cmd', we compute the WEP $wep(cmd', c)$.

$$\begin{aligned} & wep(assume\ x \geq 0;\ x := x + 1;\ x := 4 * x;\ x := x - 1, x = 3) \\ = \quad & 4 * (x+1) - 1 = 3 \wedge x \geq 0 \\ & wep(assume\ x \geq 0;\ x := x + 1;\ assume\ true;\ x := x - 1, x = 3) \\ = \quad & x + 1 - 1 = 3 \wedge x \geq 0 \end{aligned}$$

Overall, we get the same result for both computations.

$$\begin{aligned} & dnf(wep(cmd, c)) \\ = \quad & (4 * (x+1) - 1 = 3 \wedge x \geq 0) \vee (x + 1 - 1 = 3 \wedge x \geq 0) \\ = \quad & \bigvee \{wep(cmd', c) \mid cmd' \in split(cmd)\} \end{aligned}$$

5.3. Relative Inductiveness

In the previous section we have introduced the WEP to compute the predecessors of a given cube (Sec. 5.2). In this section we introduce an alternative relative inductiveness check based on the WEP. Experimental results have proven that this alternative check is advantageous, when using SMT solvers with caching.

Definition 5.3.1 (Alternative Relative Inductiveness)**.**
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA with edge $e = (l, cmd, l') \in G$. Given the frame $F_{(i-1,l)}$ and a cube c, we define a function $relIndAlt(F_{(i-1,l)}, e, c)$, such that

- $l \neq l'$:

$$relIndAlt(F_{(i-1,l)}, e, c) \quad \Leftrightarrow \quad unsat(F_{(i-1,l)} \wedge wep(cmd, c))$$

- $l = l'$:

$$relIndAlt(F_{(i-1,l)}, e, c) \quad \Leftrightarrow \quad unsat(F_{(i-1,l)} \wedge \neg c \wedge wep(cmd, c))$$

The cube c is called relative inductive to the frame $F_{(i-1,l)}$ and the edge e, iff the function $relInd(F_{i-1,l}, e, c)$ evaluates to true.

The formal definition of the alternative relative inductiveness is shown in Def. 5.3.1. So far, the standard relative inductiveness check (Def 4.2.1) has intuitively verified that there is a transition T_e from the given frame $F_{(i-1,l)}$ to a successor state in c'. Instead the alternative one first computes the exact preimage of the given cube c and then checks, if this preimage intersects with the frame $F_{(i-1,l)}$. We show that both relative inductiveness checks yield the same result.

Lemma 5.3.2.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA with edge $e = (l, cmd, l') \in G$. Furthermore, let c be a cube and $F_{(i-1,l)}$ be a frame. It holds that

$$relIndAlt(F_{(i-1,l)}, e, c) \quad \Leftrightarrow \quad relInd(F_{(i-1,l)}, e, c)$$

Proof.

- $l \neq l'$:

$$\begin{aligned} & relIndAlt(F_{(i-1,l)}, e, c) \\ \Leftrightarrow \quad & unsat(F_{(i-1,l)} \wedge wep(cmd, c)) \\ \Leftrightarrow \quad & unsat(F_{(i-1,l)} \wedge T_e \wedge c') && \text{(Def. 5.2.2)} \\ \Leftrightarrow \quad & relInd(F_{(i-1,l)}, e, c) \end{aligned}$$

- $l = l'$:

$$\begin{aligned}
& relIndAlt(F_{(i-1,l)}, e, c) \\
\Leftrightarrow \quad & unsat(F_{(i-1,l)} \wedge \neg c \wedge wep(cmd, c)) \\
\Leftrightarrow \quad & unsat(F_{(i-1,l)} \wedge \neg c \wedge T_e \wedge c') && \text{(Def. 5.2.2)} \\
\Leftrightarrow \quad & relInd(F_{(i-1,l)}, e, c)
\end{aligned}$$

□

As mentioned before, the alternative relative inductiveness check might be preferable for SMT solvers with caching. Let *cmd* and *cmd'* be two different GCL commands of the edges $e \in G$ and $e' \in G$, which yield the same WEP with respect to a given cube c, i.e. $wep(cmd, c) = wep(cmd', c)$. If we apply the alternative relative inductiveness $relIndAlt(F_{(i-1,l)}, e, c)$ and $relIndAlt(F_{(i-1,l)}, e', c)$ for both commands, the second query is derived from the cached result, since both requests are syntactically equivalent. Instead, the standard relative inductiveness check can not rely on caching, because both requests $relInd(F_{(i-1,l)}, e, c)$ and $relInd(F_{(i-1,l)}, e', c)$ are syntactically different.

Example 5.3.3.
We consider the example program P_{true} (Alg. 2.3.1) with the CFA A_{true} (Fig. 2.3.1) again. The CFA A_{true} contains the self loop $e = (l_1, y := y - 1, l_1) \in G$. Furthermore, let $c = y \geq x$ be the cube to be checked at location l_1, where $wep(y := y - 1, y \geq x) = y - 1 \geq x$. Given the frame $F_{(i-1,l)} = \top$, we get

$$\begin{aligned}
& relIndAlt(F_{(i-1,l)}, e, c) \\
= \quad & unsat(\neg(y \geq x) \wedge y - 1 \geq x) \\
= \quad & true \\
= \quad & unsat(\neg(y \geq x) \wedge y' = y - 1 \wedge y' \geq x) \\
= \quad & relInd(F_{(i-1,l)}, e, c)
\end{aligned}$$

5.4. Assumes

In this section we consider GCL assumes (Def. 2.2.2) in detail. Basically, we can drop a literal of a cube with respect to an edge $e \in G$, if this literal is implied by an assume command along this edge e.

First, we have to introduce a lemma, which states the distributivity of the WEP (Def. 5.2.2) concerning the conjunction of cubes (Lemma 5.4.1).

Lemma 5.4.1.
Let cmd be a GCL command without a choice. Given two cubes c_1, c_2, it holds that

$$wep(cmd, c_1 \wedge c_2) \quad \Leftrightarrow \quad wep(cmd, c_1) \wedge wep(cmd, c_2)$$

Proof.
Note that the GCL command *cmd* contains no choice. We prove the Lemma 5.4.1 by structural induction over GCL command *cmd* without the choice case.

- $cmd = (assume\ b)$:

$$\begin{aligned} & wep(assume\ b, c_1 \wedge c_2) \\ = \quad & (c_1 \wedge c_2) \wedge b \\ = \quad & (c_1 \wedge b) \wedge (c_2 \wedge b) \\ = \quad & wep(assume\ b, c_1) \wedge wep(assume\ b, c_2) \end{aligned}$$

- $cmd = (x := a)$:

$$\begin{aligned} & wep(x := a, c_1 \wedge c_2) \\ = \quad & (c_1 \wedge c_2)[x \mapsto a] \\ = \quad & c_1[x \mapsto a] \wedge c_2[x \mapsto a] \\ = \quad & wep(x := a, c_1) \wedge wep(x := a, c_2) \end{aligned}$$

- $cmd = (cmd_1; cmd_2)$:

$$\begin{aligned} & wep(cmd_1; cmd_2, c_1 \wedge c_2) \\ = \quad & wep(cmd_1, wep(cmd_2, c_1 \wedge c_2)) \\ = \quad & wep(cmd_1, wep(cmd_2, c_1) \wedge wep(cmd_2, c_2)) && \text{(by hypothesis)} \\ = \quad & wep(cmd_1, wep(cmd_2, c_1)) \wedge wep(cmd_1, wep(cmd_2, c_2)) && \text{(by hypothesis)} \\ = \quad & wep(cmd_1; cmd_2, c_1) \wedge wep(cmd_1; cmd_2, c_2) \end{aligned}$$

□

Based on the previous lemma, we can proof the following Th. 5.4.2. Intuitively, if a literal $p \in g$ of the generalisation g is implicitly guaranteed by a GCL assume in cmd , we can drop this literal p without violating the relative inductiveness. However, the formal definition in Th. 5.4.2 is based on the WEP, such that the WEP $wep(cmd, p)$ of the literal p is implied by the WEP $wep(cmd, true)$ of the empty cube $true$. Note that given a GCL command cmd' without a choice command, the WEP $wep(cmd', p)$ yields a cube.

Theorem 5.4.2.
Let g be the generalisation of a cube c at index $i \in \mathbb{N}$ and location $l' \in L$ with respect to the edge $e \in G$, i.e. $gen(i, l')(c, e) = g$. Given a literal $p \in g$ of the generalisation g, then it holds that for all $cmd' \in split(cmd)$

$$wep(cmd', p) \subseteq wep(cmd', true) \quad \Rightarrow \quad relInd(F_{(i-1,l)}, e', c \setminus \{p\})$$

where the function $split(cmd)$ is specified by Def. 5.2.4.

Proof.
Let $\widehat{c} = c \setminus \{p\}$ be the reduced cube. We prove the relative inductiveness of $\widehat{c}$ with respect to every partial edge $e' = (l, cmd', l')$, where $cmd' \in split(cmd)$. We further distinguish between edge e being a self-loop or not.

- $l \neq l'$:

$$\begin{aligned}
& relInd(F_{(i-1,l)}, e', \widehat{c} \wedge p) & \\
\Rightarrow\ & relIndAlt(F_{(i-1,l)}, e', \widehat{c} \wedge p) & \text{(Lemma 5.3.2)} \\
\Rightarrow\ & unsat(F_{(i-1,l)} \wedge wep(cmd', \widehat{c} \wedge p)) & \\
\Rightarrow\ & unsat(F_{(i-1,l)} \wedge wep(cmd', \widehat{c}) \wedge wep(cmd', p)) & \text{(Lemma 5.4.1)} \\
& \textit{Since } wep(cmd', p) \subseteq wep(cmd', true), & \\
& \textit{we get } wep(cmd', true) \Rightarrow wep(cmd', p). & \\
\Rightarrow\ & unsat(F_{(i-1,l)} \wedge wep(cmd', \widehat{c}) \wedge wep(cmd', true)) & \\
\Rightarrow\ & unsat(F_{(i-1,l)} \wedge wep(cmd', \widehat{c} \wedge true)) & \text{(Lemma 5.4.1)} \\
\Rightarrow\ & unsat(F_{(i-1,l)} \wedge wep(cmd', \widehat{c})) & \\
\Rightarrow\ & relIndAlt(F_{(i-1,l)}, e', \widehat{c}) & \\
\Rightarrow\ & relInd(F_{(i-1,l)}, e', \widehat{c}) & \text{(Lemma 5.3.2)}
\end{aligned}$$

- $l = l'$:

$$
\begin{aligned}
& \mathit{relInd}(F_{(i-1,l)}, e', \widehat{c} \wedge p) \\
\Rightarrow\quad & \mathit{relIndAlt}(F_{(i-1,l)}, e', \widehat{c} \wedge p) && \text{(Lemma 5.3.2)} \\
\Rightarrow\quad & \mathit{unsat}(F_{(i-1,l)} \wedge \neg(\widehat{c} \wedge p) \wedge \mathit{wep}(\mathit{cmd}', \widehat{c} \wedge p)) \\
\Rightarrow\quad & \mathit{unsat}(F_{(i-1,l)} \wedge \neg(\widehat{c} \wedge p) \wedge \mathit{wep}(\mathit{cmd}', \widehat{c}) \wedge \mathit{wep}(\mathit{cmd}', p)) \\
& && \text{(Lemma 5.4.1)} \\
& \mathit{Since}\ \mathit{wep}(\mathit{cmd}', p) \subseteq \mathit{wep}(\mathit{cmd}', \mathit{true}), \\
& \mathit{we\ get}\ \mathit{wep}(\mathit{cmd}', \mathit{true}) \Rightarrow \mathit{wep}(\mathit{cmd}', p). \\
\Rightarrow\quad & \mathit{unsat}(F_{(i-1,l)} \wedge \neg(\widehat{c} \wedge p) \wedge \mathit{wep}(\mathit{cmd}', \widehat{c}) \wedge \mathit{wep}(\mathit{cmd}', \mathit{true})) \\
\Rightarrow\quad & \mathit{unsat}(F_{(i-1,l)} \wedge \neg\widehat{c} \wedge \mathit{wep}(\mathit{cmd}', \widehat{c}) \wedge \mathit{wep}(\mathit{cmd}', \mathit{true})) \\
\Rightarrow\quad & \mathit{unsat}(F_{(i-1,l)} \wedge \neg\widehat{c} \wedge \mathit{wep}(\mathit{cmd}', \widehat{c} \wedge \mathit{true})) && \text{(Lemma 5.4.1)} \\
\Rightarrow\quad & \mathit{unsat}(F_{(i-1,l)} \wedge \neg\widehat{c} \wedge \mathit{wep}(\mathit{cmd}', \widehat{c})) \\
\Rightarrow\quad & \mathit{relIndAlt}(F_{(i-1,l)}, e', \widehat{c}) \\
\Rightarrow\quad & \mathit{relInd}(F_{(i-1,l)}, e', \widehat{c}) && \text{(Lemma 5.3.2)}
\end{aligned}
$$

□

We show an example application of the previous theorem in the following (Example 5.4.3).

Example 5.4.3.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA. Furthermore, let $e = (l, \mathit{cmd}, l') \in G$ be an edge, where $\mathit{cmd} = \mathit{assume}(x \geq 0); x := x + 1$ be a GCL command. Note that the command cmd implies $x \geq 1$ and it holds that $\mathit{split}(\mathit{cmd}) = \mathit{cmd}$. We consider the cube $c = (x \geq 1 \wedge y \geq 1) = \{x \geq 1, y \geq 1\}$ to be blocked at location l'. Given the frame $F_{(i-1,l)} = \{\neg y \geq 1\}$, it holds that $\mathit{relInd}(F_{(i-1,l)}, e, c) = \mathit{true}$. We compute $\mathit{wep}(\mathit{cmd}, \mathit{true}) = x \geq 0$. The cube c contains the literal $x \geq 1 \in c$, such that we get $\mathit{wep}(\mathit{cmd}, x \geq 1) = x \geq 0$. Since

$$\mathit{wep}(\mathit{cmd}, x \geq 1) = \{x \geq 0\} \sqsubseteq \{x \geq 0\} = \mathit{wep}(\mathit{cmd}, \mathit{true}),$$

we remove the literal $x \geq 1$ from the cube c, such that $\widehat{c} = c \setminus \{x \geq 1\} = y \geq 1$. The second literal $y \geq 1$ can not be dropped, because

$$\mathit{wep}(\mathit{cmd}, y \geq 1) = \{y \geq 1\} \not\sqsubseteq \{x \geq 0\} = \mathit{wep}(\mathit{cmd}, \mathit{true}).$$

Overall, the resulting cube $\widehat{c}$ is also relative inductive with respect to the edge e, i.e. $\mathit{relInd}(F_{(i-1,l)}, e, \widehat{c}) = \mathit{true}$.

5.5. Predecessor Cubes

In this section we present an approach to compute the (edge-based) generalisation of a given cube c based on a predecessor cube. Therefore, we first show that the relative inductiveness of the cube c can be derived from a predecessor cube.

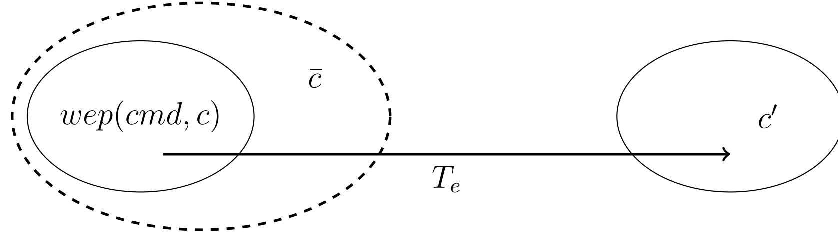

Figure 5.5.1.: Graphical representation of a predecessor cube $\widehat{c} \supseteq wep(cmd, c)$.

More precise, if the predecessor frame under consideration contains a negated cube $\neg\bar{c}$, which blocks at least the WEP $wep(cmd, c)$ with respect to the edge $e = (l, cmd, l') \in G$, then the original cube c is relative inductive to e. The intuitive notion is graphically illustrated in Fig. 5.5.1. Respectively, the formal definition is shown in the following Lemma 5.5.1. Note that this lemma is also applied to the relative inductiveness checks in the main algorithm of IC3CFA algorithm (Alg. 4.2.2). We can save a SAT call, every time we find a predecessor cube $\bar{c}$ characterised by Lemma 5.5.1.

Lemma 5.5.1.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA with edge $e = (l, cmd, l') \in G$. Furthermore, let c be a cube to be blocked at location $l' \in L$ and index i. Given the frame $F_{(i-1,l)}$, it holds that

$$\exists cube\ \bar{c}.\ \neg\bar{c} \in F_{(i-1,l)} \land \bar{c} \subseteq wep(cmd, c) \quad \Rightarrow \quad relInd(F_{(i-1,l)}, e, c)$$

Proof.

$$\begin{aligned} & \exists cube\ \bar{c}.\ \neg\bar{c} \in F_{(i-1,l)} \land \bar{c} \subseteq wep(cmd, c) \\ \Rightarrow\ & \exists cube\ \bar{c}.\ unsat(F_{(i-1,l)} \land \bar{c}) \land \bar{c} \subseteq wep(cmd, c) \\ \Rightarrow\ & unsat(F_{(i-1,l)} \land wep(cmd, c)) \\ \Rightarrow\ & relIndAlt(F_{(i-1,l)}, e, c) && \text{(Def. 5.3.1)} \\ \Rightarrow\ & relInd(F_{(i-1,l)}, e, c) && \text{(Lemma 5.3.2)} \end{aligned}$$

□

Based on a found predecessor cube $\bar{c}$, we also derive the generalisation $g \subseteq c$ of the original cube c. Therefore, we apply the inverse function wep^{-1} of the WEP. More precise, we define the function $wep^{-1}(cmd, c)$ for a GCL command cmd without a choice as follows:

$$\begin{aligned}
wep^{-1}(assume\ b, c) &= c \setminus \{b\} \\
wep^{-1}(x := a, c) &= c[a \mapsto x] \\
wep^{-1}(cmd_1; cmd_2, c) &= wep^{-1}(cmd_2, wep^{-1}(cmd_1, c))
\end{aligned}$$

Note that we consider GCL commands without a choice, which can be computed by the function *split* (Def. 5.2.4). The following Lemma 5.5.2 states, that the inverse function $wep^{-1}(cmd, c)$ of the WEP is monotonic.

Lemma 5.5.2.
Let cmd be GCL command without a choice and $c, \bar{c}$ be two cubes. It holds that wep^{-1} is monotonic, i.e.

$$\bar{c} \subseteq c \quad \Rightarrow \quad wep^{-1}(cmd, \bar{c}) \subseteq wep^{-1}(cmd, c).$$

Proof.
Note that the GCL command *cmd* contains no choice. We prove lemma by structural induction over GCL command *cmd* without the choice case.

- $cmd = (assume\ b)$:

$$\begin{aligned}
& \bar{c} \subseteq c \\
\Rightarrow\ & \bar{c} \setminus \{b\} \subseteq c \setminus \{b\} \\
\Rightarrow\ & wep^{-1}(assume\ b, \bar{c}) \subseteq wep^{-1}(assume\ b, c)
\end{aligned}$$

- $cmd = (x := a)$:

$$\begin{aligned}
& \bar{c} \subseteq c \\
\Rightarrow\ & \bar{c}[a \mapsto x] \subseteq c[a \mapsto x] \\
\Rightarrow\ & wep^{-1}(x := a, \bar{c}) \subseteq wep^{-1}(x := a, c)
\end{aligned}$$

- $cmd = (cmd_1; cmd_2)$:

$$
\begin{aligned}
& \bar{c} \subseteq c \\
\Rightarrow\quad & wep^{-1}(cmd_1, \bar{c}) \subseteq wep^{-1}(cmd_1, c) && \text{(by hypothesis)} \\
\Rightarrow\quad & wep^{-1}(cmd_2, wep^{-1}(cmd_1, \bar{c})) \sqsubseteq wep^{-1}(cmd_2, wep^{-1}(cmd_1, \bar{c})) \\
& && \text{(by hypothesis)} \\
\Rightarrow\quad & wep^{-1}(cmd_1; cmd_2, \bar{c}) \subseteq wep^{-1}(cmd_1; cmd_2, c)
\end{aligned}
$$

□

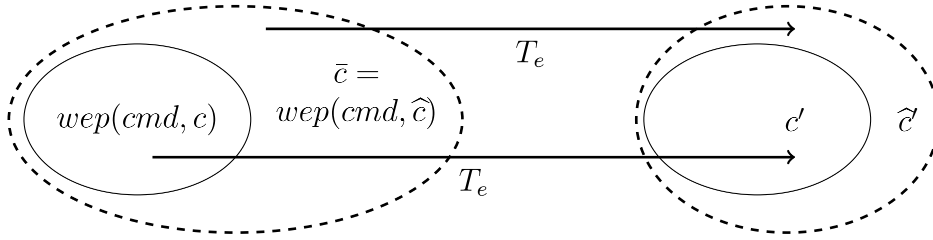

Figure 5.5.2.: Graphical representation of a generalisation $\widehat{c}$ based on the predecessor cube.

We assume in the following that we have found a predecessor cube $\bar{c} \subseteq wep(cmd, c)$, where cmd is a GCL command without a choice. There always exists a cube $\widehat{c}$, such that it holds $\bar{c} = wep(cmd, \widehat{c})$. In fact, we compute $\widehat{c}$ by applying the inverse function $wep^{-1}(cmd, \bar{c}) = \widehat{c}$. It follows that the cube $\widehat{c} \subseteq c$ is a generalisation of the original cube c. Figure 5.5.2 illustrates the relation between the different cubes $c, \bar{c}$ and $\widehat{c}$. The whole approach is formalised by the following Th. 5.5.3.

Theorem 5.5.3.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA with edge $e = (l, cmd, l') \in G$. The cube c has to be blocked at location $l' \in L$ and index i. Given the frame $F_{(i-1,l)}$, it holds that

$$
\begin{aligned}
& \forall cmd' \in split(cmd).\ \neg wep(cmd', \widehat{c}) \in F_{(i-1,l)} \wedge wep(cmd', \widehat{c}) \subseteq wep(cmd', c) \\
\Rightarrow\quad & gen_{(i,l')}(c, e) = \widehat{c}
\end{aligned}
$$

where $\widehat{c}$ is a cube and the function $split(cmd)$ is given by Def. 5.2.4.

Proof.
Let $\widehat{c}$ be a cube and *cmd* be a GCL command without a choice

$$\begin{aligned}
& \neg wep(cmd, \widehat{c}) \in F_{(i-1,l)} \wedge wep(cmd, \widehat{c}) \subseteq wep(cmd, c) & \\
\Rightarrow\ & relInd(F_{(i-1,l)}, e, \widehat{c}) \wedge wep(cmd, \widehat{c}) \subseteq wep(cmd, c) & \text{(Lemma 5.5.1)} \\
\Rightarrow\ & relInd(F_{(i-1,l)}, e, \widehat{c}) \wedge \widehat{c} \subseteq c & \text{(Lemma 5.5.2)} \\
\Rightarrow\ & gen_{(i,l')}(c, e) = \widehat{c} &
\end{aligned}$$

□

The application of the previous theorem is shown in the following Example 5.5.4.

Example 5.5.4.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA. Furthermore, let $e = (l, y := y-1, l') \in G$ be an edge, where $split(y := y-1) = y := y-1$. Let $c = y \geq x \wedge x = 1$ be a cube to be generalised at index i and location $l' \in L$. We first compute the WEP $wep(y := y-1, c) = y-1 \geq x \wedge x = 1$ of the original cube c. Given the frame $F_{(i-1,l)} = \{\neg y - 1 \geq x\}$, we obtain

$$\bar{c} = wep(y := y-1, \widehat{c}) = \{y-1 \geq x\} \subseteq \{y-1 \geq x, x = 1\} = wep(y := y-1, c).$$

Based on the the found predecessor cube $\bar{c}$, we compute the generalisation

$$gen_{(i,l')}(c, e) = \widehat{c} = wep^{-1}(y := y-1, \bar{c}) = (y-1 \geq x)[y-1 \mapsto y] = y \geq x.$$

5.6. Generalisation Context

In this section we present a so called generalisation context. Given a cube c to be generalised with respect to an edge $e \in G$, we try to derive a generalisation based on previous generalisations of the same cube c.

Definition 5.6.1 (Generalisation Context)**.**
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA. The generalisation context GC_i at index i is a set of four tuples

$$GC_i \subseteq 2^{(c,e,F,g)}$$

where $g \subseteq c$ is a generalisation of cube c, $e = (l, cmd, l') \in G$ is an edge and F is a frame. It holds that

$$(c, e, F, g) \in GC_i \quad \Leftrightarrow \quad \exists j \leq i.\ gen_{(j,l')}(c, e) = g \wedge F = F_{(j-1,l)}$$

The formal definition of the generalisation context in terms of a 4-tuple is shown in Def. 5.6.1. Each generalisation g of a cube c at index $j \in \mathbb{N}$ and edge e is stored

as one generalisation context (c, e, F, g) available in all sets GC_i $(i \geq j)$. Note that in fact, we use a least recently used cache for the generalisation contexts, such that we omit older and less probable generalisations. The intuitive notion of the generalisation context is graphically illustrated in Fig. 5.6.1.

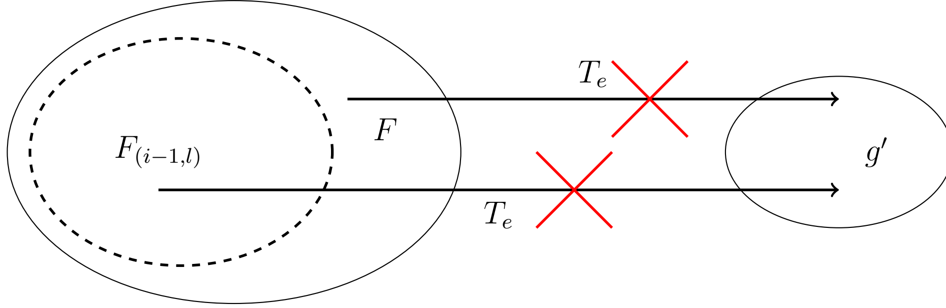

Figure 5.6.1.: Intuitive notion of the generalisation context.

Given a generalisation context $(c, e, F, g) \in GC_i$, if we encounter the cube c again at index $i \in \mathbb{N}$, we might reuse the previously computed generalisation g. Therefore, the frame $F_{(i-1,l)}$ of the predecessor location $l \in L$ has to be smaller compared to the frame F in the generalisation context. This condition is formally specified by Th. 5.6.2, which is based on the idea presented in [Mue15].

Theorem 5.6.2.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA with edge $e = (l, cmd, l') \in G$. Furthermore, let c be a cube to be generalised at index i and at location l'. Given the frame $F_{(i-1,l)}$, it holds that

$$(c, e, F, g) \in GC_i \wedge F_{(i-1,l)} \sqsubseteq_F F \quad \Rightarrow \quad gen_{(i,l')}(c, e) = g$$

Proof.

$$\begin{aligned}
& (c, e, F, g) \in GC_i \wedge F_{(i-1,l)} \sqsubseteq_F F \\
\Rightarrow\ & \exists j \leq i.\ (gen_{(j,l')}(c, e) = g \wedge F = F_{(j-1,l)}) \\
\Rightarrow\ & relInd(F_{(j-1,l)}, e, g) \\
& \textit{Since } F_{(i-1,l)} \sqsubseteq_F F = F_{(j-1,l)},\ \textit{let } F_{(i-1,l)} := F_{(j-1,l)} \cup F_{\triangle}. \\
\Rightarrow\ & relInd(F_{(j-1,l)} \cup F_{\triangle}, e, g) \\
\Rightarrow\ & relInd(F_{(i-1,l)}, e, g) \\
\Rightarrow\ & gen_{(i,l')}(c, e) = g
\end{aligned}$$

□

So far, we might reuse a generalisation g based on the generalisation context (c, e, F, g), if we encounter the exact same cube c again. We present a new generalisation context, where we encounter a similar cube $\widehat{c}$. More precise, the cube $\widehat{c}$ has to be a syntactical superset of the generalisation g, i.e. $g \subseteq \widehat{c}$. If $\widehat{c}$ is a syntactical superset and the frame condition $F_{(i-1,l)} \sqsubseteq_F F$ still holds, then we also get g as generalisation of $\widehat{c}$. In consequence, we avoid even more computations compared to the standard generalisation context.

Theorem 5.6.3.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA with edge $e = (l, cmd, l') \in G$. Furthermore, let $\widehat{c}$ be a cube to be generalised at index i and at location l'. Given the frame $F_{(i-1,l)}$, it holds that

$$(c, e, F, g) \in GC_i \wedge F_{(i-1,l)} \sqsubseteq_F F \wedge g \subseteq \widehat{c} \quad \Rightarrow \quad gen_{(i,l')}(\widehat{c}, e) = g$$

Proof.

$$\begin{aligned} & (c, e, F, g) \in GC_i \wedge F_{(i-1,l)} \sqsubseteq_F F \wedge g \subseteq \widehat{c} \\ \Rightarrow\ & \exists j \leq i.\ (gen_{(j,l')}(c, e) = g \wedge F = F_{(j-1,l)}) \\ \Rightarrow\ & gen_{(i,l')}(c, e) = g && \text{(Th. 5.6.2)} \\ & \textit{Since } g \subseteq \widehat{c}, \textit{ g is also a generalisation of } \widehat{c}. \\ \Rightarrow\ & gen_{(i,l')}(\widehat{c}, e) = g && \text{(Def. 4.3.1)} \end{aligned}$$

□

Theorem 5.6.3 shows the formal definition of the new generalisation context. In the following we show an example regarding the standard and new generalisation context (Example 5.6.4), where we also point out the difference between both.

Example 5.6.4.
Let $c = p_1 \wedge p_2$ be a cube to be blocked at location l' and index i. Furthermore, let $e = (l, cmd, l') \in G$ be the edge from the only predecessor location l. The corresponding frame of location l is $F_{(i-1,l)} = \{\neg c_1, \neg c_2\}$. We assume that the generalisation context GC_i only contains the tuple $(c, e, \{\neg c_1\}, p_1) \in GC_i$, i.e. the cube c has been previously generalised to literal p_1. Overall, we obtain

$$\begin{aligned} (c, e, \{c_1\}, p_1) \in GC_i \wedge F_{(i-1,l)} \sqsubseteq_F \{c_1\} \quad &\Rightarrow \quad gen_{(i,l')}(c, e) = p_1 \\ (\widehat{c}, e, \{c_1\}, p_1) \in GC_i \wedge F_{(i-1,l)} \sqsubseteq_F \{c_1\} \wedge \{p_1\} \subseteq \{p_1, p_2\} \quad &\Rightarrow \quad gen_{(i,l')}(c, e) = p_1 \end{aligned}$$

However, if we encounter a similar cube $\widehat{c} = p_1 \wedge p_3$ at location l' and index i, we get

$$\frac{(\widehat{c}, e, \{c_1\}, p_1) \in GC_i \wedge F_{(i-1,l)} \sqsubseteq_F \{c_1\} \wedge \{p_1\} \subseteq \{p_1, p_3\} \quad \Rightarrow \quad gen_{(i,l')}(\widehat{c}, e) = p_1}{(\widehat{c}, e, \widehat{F}, \widehat{g}) \notin GC_i}$$

5.7. Minimal Generalisation

In the previous section we have introduced the generalisation contexts, which are used to derive a generalisation based on the previous computations. In this section we determine a minimal generalisation based on these generalisation contexts. A minimal generalisation is formally given in terms of a cube and contains possibly multiple literals, which are definitely part of the final generalisation.

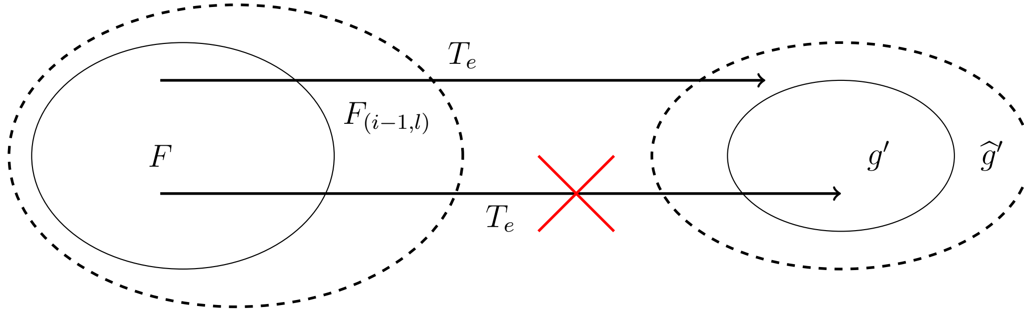

Figure 5.7.1.: Intuitive notion of the minimal generalisation.

The intuitive notion of the minimal generalisation is illustrated in Fig. 5.7.1. Let (c, e, F, g) be a generalisation context, which has been introduced in Def. 5.6.1. Since the original cube c has been previously generalised to the cube $g \subseteq c$, all real subsets $\widehat{g} \subset g$ are not possible generalisations any more. Otherwise, c would have been previously generalised to the cube $\widehat{g}$. In consequence, every subset $\widehat{g}$ is not relative inductive with respect to the edge $e = (l, cmd, l') \in G$ and the frame F. Intuitively, there is a transition T_e with respect to the edge e from the frame F to a state in $\widehat{g}'$. If we encounter a greater frame $F_{(i-1,l)}$ compared to F, there is also a transition T_e to the cube $\widehat{g}$, such that $\widehat{g}$ is not relative inductive. Thus, the minimal generalisation, i.e. the set of necessary literals, is given by the previous generalisation g. Theorem 5.7.1 shows the formal definition of the minimal generalisation based on the generalisation context.

Theorem 5.7.1.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA, where $e = (l, cmd, l') \in G$. Let c be a cube

to be generalised at location $l' \in L$. Furthermore, we assume that the generalisation g is minimal with respect to the cube size. It holds that

$$\begin{aligned} & (c,e,F,g) \in GC_i \wedge F \sqsubseteq_F F_{(i-1,l)} \wedge \widehat{g} \subset_C g \\ \Rightarrow\ & gen_{(i,l')}(c,e) \neq \widehat{g} \\ \Rightarrow\ & g \subseteq gen_{(i,l')}(c,e) \subseteq c \end{aligned}$$

Proof.

$$\begin{aligned} & (c,e,F,g) \in GC_i \wedge F_{(i-1,l)} \sqsubseteq_F F \wedge \widehat{g} \subset g \\ \Rightarrow\ & \exists j \leq i.\ (gen_{(j,l')}(c,e) = g \wedge F = F_{(j-1,l)}) \\ \Rightarrow\ & \exists j \leq i.\ (gen_{(j,l')}(c,e) \neq \widehat{g} \wedge F = F_{(j-1,l)}) \\ \Rightarrow\ & \neg relInd(F_{(j-1,l)}, e, \widehat{g}) \\ & \textit{Since } F = F_{(j-1,l)} \sqsubseteq_F F_{(i-1,l)},\ \textit{let } F_{(j-1,l)} := F_{(i-1,l)} \cup F_{\triangle}. \\ \Rightarrow\ & \neg relInd(F_{(i-1,l)} \cup F_{\triangle}, e, \widehat{g}) \\ \Rightarrow\ & \neg relInd(F_{(i-1,l)}, e, \widehat{g}) \\ \Rightarrow\ & gen_{(i,l')}(c,e) \neq \widehat{g} \end{aligned}$$

□

Note that in the corresponding proof we assume that the previous generalisation $g \subseteq c$ is minimal with respect to the cube size. In certain limited cases, this assumption is not fulfilled. Then, the resulting generalised cube might contain more literals than necessary, i.e. the resulting generalisation is also not minimal. However, the final generalisation remains correct, because we definitely block the original cube c.

Example 5.7.2.
Let $c = p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$ be a cube to be blocked at location l' and index i. Furthermore, let $e = (l, cmd, l') \in G$ be the edge from the only predecessor location l. The corresponding frame of location l is $F_{(i-1,l)} = \{\neg c_1\}$. We assume that the generalisation context GC_i only contains the tuple $(c, e, \{\neg c_1, \neg c_2\}, p_1 \wedge p_2) \in GC_i$, such that we get

$$\{\neg c_1, \neg c_2\} \sqsubseteq_F F_{(i-1,l)} \quad \Rightarrow \quad \{p_1, p_2\} \subseteq gen_{(i,l')}(c,e) \subseteq \{p_1, p_2, p_3, p_4, p_5\}.$$

For example, we generalise the cube c to $gen_{(i,l')}(c,e) = p_1 \wedge p_2 \wedge p_3$, while $gen_{(i,l')}(c,e) = p_1$ is not possible any more.

5.8. Ordering

In this section we consider the predecessor computation introduced in Sec. 5.2, again. Let $e = (l, cmd, l') \in G$ be an edge between the locations $l, l' \in L$, where the GCL command *cmd* contains a choice. If we compute the WEP (Def. 5.2.2) of a certain cube c with respect to the edge e, we get a set of predecessor cubes as result. Regarding the IC3CFA generalisation, this opens the question, in which order we should process these predecessor cubes. Different experimental result have shown that a set ordering according to the cube size in ascending order yields the best results. The following Example 5.8.1 gives an intuition, why this order is preferable.

Example 5.8.1.
Let (L, G, l_{init}, l_E) be a CFA with edge $e = (l, cmd, l') \in G$. We assume that there are three cubes $c_1 = p_1 \wedge p_2$, $c_2 = p_1 \wedge p_3 \wedge p_4$ and $c_3 = p_1 \wedge p_3 \wedge p_5$, which have to be generalised at location $l' \in L$ and index $i \in \mathbb{N}$. Furthermore, there is a possible generalisation p_1, which blocks all these cubes, i.e.

$$gen_{(i,l')}(c_1) = gen_{(i,l')}(c_2) = gen_{(i,l')}(c_3) = p_1$$

If the given cubes are sorted according to their cube size, we start by generalising the cube c_1. We execute the generalisation function $gen(i, l', c_1)$ (Def. 4.3.1), such that we require only two SAT calls for each predecessor edge $(l, cmd, l') \in G$ to obtain the desired generalisation p_1. In contrast, if we start with another cube c_2 or c_3, we would require one additional SAT call per predecessor edge $(l, cmd, l') \in G$ for the same result.

5.9. Multiple Predecessors

In this section we consider the generalisation of a given cube c at a location with multiple predecessor edges more detailed. Since we compute the union of all edge-based generalisations (Def. 4.3.1), we introduce an optimisation avoiding unnecessary computations.

Lemma 5.9.1.
Let $i \in \mathbb{N}$ and (L, G, l_{init}, l_E) be a CFA. Let c be cube to be generalised at location $l' \in L$. It holds that

$$\forall e = (l, cmd, l') \in G.\ gen_{(i,l')}(c, e) \subseteq gen_{(i,l')}(c) \subseteq c$$

Proof.

$$\begin{aligned} & \forall e \in G.\ gen_{(i,l')}(c,e) \subseteq gen_{(i,l')}(c) && \text{(Def. 4.3.2)} \\ & gen_{(i,l')}(c) \subseteq c && \text{(Def. 4.3.1)} \\ \Rightarrow\ & \forall e \in G.\ gen_{(i,l')}(c,e) \subseteq gen_{(i,l')}(c) \subseteq c \end{aligned}$$

□

According to Lemma 5.9.1, each edge-based generalisation $gen_{(i,l')}(c,e)$ is a syntactical subset of the entire (location-based) generalisation $gen_{(i,l')}(c)$ at index $i \in \mathbb{N}$ and location $l' \in L$, i.e. $gen_{(i,l')}(c,e) \subseteq\subseteq gen_{(i,l')}(c)$. In consequence, every edge-based generalisation $gen_{(i,l')}(c,e)$ determines a set of necessary literals with regard to the resulting generalisation. If we assume that we have already computed the generalisations with respect to other edges, we do not try to derive a generalisation smaller than the previous edge-based generalisations. The following Example 5.9.2 shows the application of the Lemma 5.9.1.

Example 5.9.2.
Let $A_{true} = (L, G, l_{init}, l_E)$ (Fig. 2.3.1) be the CFA of the example program P_{true} (Alg. 2.3.1). Furthermore, let $c_1 = y \geq x \wedge x = 1$ be the cube to be blocked at index $i = 1$ and location $l_1 \in L$. According to the generalisation in Example 4.3.3, we get $gen_{(1,l_1)}(c_1) = y \geq x$ as result. If we take a closer look at the generalisation, we start by generalising according the edge $e_1 = (l_{init}, assume\ y < x, l_1)$. The relative inductiveness check $relInd(\top, e_1, y \geq x)$ yields $true$, while $relInd(\top, e_1, true)$ yields $false$. In consequence, we get the generalisation $gen_{(1,l_1)}(c_1, e_1) = y \geq x$ with respect to the edge e_1. If we start generalising with respect to the second edge $e_2 = (l_1, y := y - 1, l_1)$, we already know that

$$gen_{(1,l_1)}(c_1, e_1) = \{y \geq x\} \subseteq gen_{(1,l_1)}(c_1, e_2) \subseteq \{y \geq x, x = 1\} = c_1.$$

The generalisation $gen_{(1,l_1)}(c_1, e_2) = true$ is not possible any more, such that we can save the relative check $relInd(\top, e_2, true)$. We only have to check, if $y \geq x$ is also relative inductive to the edge e_2, i.e. $relInd(\top, e_2, y \geq x) = true$. Overall, we get the same result $gen_{(1,l_1)}(c_1) = y \geq x$, but have saved one relative inductiveness check.

5.10. Interpolation

So far, the IC3CFA algorithm has been restricted to a syntactical generalisation $g \subseteq c$ with respect to the original cube c. In this section we weaken this restriction,

such that we also allow a semantic generalisation g, i.e. $c \Rightarrow g$. This semantic generalisation g is computed via interpolation in the SMT solver.

First, we consider the example program $Jain1$ (Alg. 5.10.1), which illustrates the usefulness of a semantic generalisation. The corresponding CFA is shown in Fig. 5.10.1.

Algorithm 5.10.1 Pseudocode of the program $Jain1$.

1: **procedure** MAIN
2: $\quad y \leftarrow 1$
3: $\quad$ **while** $true$ **do**
4: $\quad\quad assert(y \neq 0)$
5: $\quad\quad y \leftarrow y + 2 * nondetInt$
6: $\quad$ **end while**
7: **end procedure**

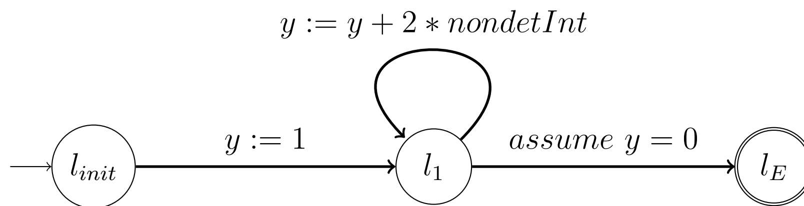

Figure 5.10.1.: The CFA of the program $Jain1$.

The given program $Jain1$ is obviously correct, because we only add even values to the odd initial valuation 1 of the variable y. Thus, the assertion $assert(y \neq 0)$ always holds. However, the standard IC3CFA algorithm is not able to prove the correctness, because it derives a potentially infinite sequence of predecessors based on the CTI $y = 0$. In consequence, the resulting frames never reach a fixpoint.

Definition 5.10.1 (Interpolants)**.**
Let A, B be two (quantifier-free) first-order logic formulas, where $A \wedge B$ is unsatisfiable. An interpolant I for A and B satisfies

- $A \Rightarrow I$,
- $I \Rightarrow \neg B$,
- $Var(I) \subseteq Var(A) \cap Var(B)$,

where $Var(\varphi)$ is the set of variables contained in the first-order logic formula φ.

An interpolant I is formally introduced in Def. 5.10.1 [Cra57, McM03]. Regarding the IC3CFA generalisation, we choose the formulas A, B as follows:

- $l \neq l'$:

$$unsat(\underbrace{F_{(i-1,l)} \wedge T_e}_{B} \wedge \underbrace{c'}_{A})$$

- $l = l'$:

$$unsat(\underbrace{F_{(i-1,l)} \wedge \neg c \wedge T_e}_{B} \wedge \underbrace{c'}_{A})$$

Based on the chosen formulas A and B, we compute an interpolant I, which is a semantic generalisation of the cube c, i.e. $c \Rightarrow I$. Simultaneously, the interpolant I is still relative inductive with respect to the predecessor edge $e \in G$ and the corresponding frame $F_{(i-1,l)}$. The graphical representation of the interpolant I is shown in Fig. 5.10.2.

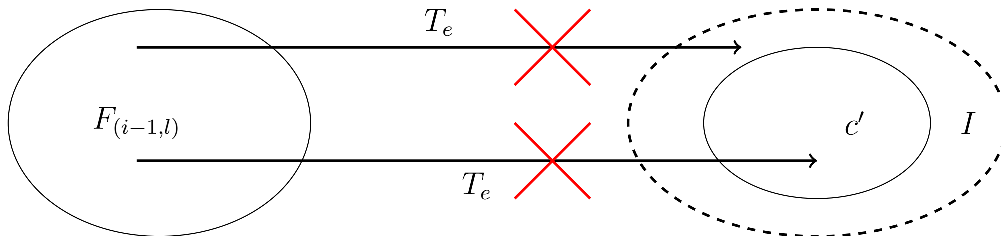

Figure 5.10.2.: Graphical representation of the interpolant I.

The IC3CFA algorithm with interpolation is able to prove the correctness of the program $Jain1$ (Example 5.10.2).

Example 5.10.2.
We consider the example program $Jain1$ (Alg. 5.10.1) with the CFA in Fig. 5.10.1.

i: \ l:	l_{init}	l_1
0	$\top$	$\bot$
1	$\top$	$\{\neg(y\%2 = 0)\}$
2	$\top$	$\{\neg(y\%2 = 0)\}$

Table 5.2.: Resulting frames of the IC3CFA algorithm with interpolation.

Table 5.2 shows the resulting frames of the IC3CFA algorithm with interpolation. The CTI $c = (y = 0)$ has to be blocked at the location l_1. To compute the interpolant I with respect to the edge $e_1 = (l_{init}, y := 1, l_1)$, we choose $A = c' = (y' = 0)$ and $B = F_{(0,l_{init})} \wedge T_{e_1} = (y' = 1)$. We get the interpolant $I = (y'\%2 = 0)$, which satisfies

$$(y' = 0) \Rightarrow (y'\%2 = 0) \quad \text{and} \quad (y'\%2 = 0) \Rightarrow \neg(y' = 1).$$

The interpolant $I = (y'\%2 = 0)$ is also relative inductive to the second edge $e_2 = (l_1, y := y + 2 * nondetInt, l_1)$, i.e.

$$relInd(F_{(0,l_1)}, e_2, y\%2 = 0) = relInd(F_{(1,l_1)}, e_2, y\%2 = 0) = true$$

Overall, we get $gen_{(1,l_1)}(c) = gen_{(2,l_1)}(c) = (y\%2 = 0)$, such that the algorithm terminates with the result *true*.

5.11. CTGs

There exists an extended generalisation function for the original IC3 algorithm, which has been proposed in [HBS13]. This extended generalisation is based on so-called counterexamples to generalisations, short CTGs. Given a cube c to be generalised, we try to drop a literal $p \in c$ and check the relative inductiveness of the reduced cube $\widehat{c} = c \setminus \{p\}$. If the cube $\widehat{c}$ is not satisfied, there are possibly multiple predecessor cubes. These predecessor cubes are called CTGs, because they prevent us from generalising c to $\widehat{c}$. However, the extended generalisation tries to recursively block all CTGs, such that a generalisation to $\widehat{c}$ is possible, again [HBS13]. Regarding the IC3 algorithm, the extended generalisation improved the performance of several programs. Thus, we have lifted the approach to the IC3CFA algorithm and the given CFA. The corresponding pseudocode of the extended

IC3CFA generalisation is shown in Alg. 5.11.1.

Algorithm 5.11.1 The extended IC3CFA generalisation function with CTGs.

1: **function** CUBE GENCTG($F, e, c, support, i$)
2: $res \leftarrow c \cup support$
3: **for** $p \in c$ **do**
4: $tmp \leftarrow res \setminus \{p\}$
5: **if** $relInd(F, e, tmp)$ **then**
6: $res \leftarrow tmp$
7: **else if** $i > 1$ **then**
8: $blocked \leftarrow true$ ▷ try to block CTGs
9: **for** $\widehat{e} = (\widehat{l}, cmd, l) \in G$ **do** ▷ $e = (l, cmd, l')$
10: $\widehat{c} \leftarrow getPredecessor(e, c)$ ▷ compute CTG
11: **if** $relInd(F_{(i-2,\widehat{l})}, \widehat{e}, \widehat{c})$ **then**
12: $\widehat{g} \leftarrow genMic(F_{(i-2,\widehat{l})}, \widehat{e}, \widehat{c}, true)$ ▷ generalise $\widehat{c}$
13: $F_{(i-1,l)} \leftarrow F_{(i-1,l)} \cup \{\neg\widehat{g}\}$
14: **else**
15: $blocked \leftarrow false$
16: **end for**
17: **if** blocked **then**
18: $res \leftarrow tmp$
19: **end for**
20: **return** res
21: **end function**

However, different experimental results have shown, that the additional SAT calls are too expensive. Furthermore, the effect of a better generalisation is only minimal, because it is focused on a single location in the CFA. Let n be the number of predecessor edges of the location under consideration. The extended generalisation requires in worst-case $(1 + n)$ SAT calls per literal p. Instead, the standard generalisation *genMic* (Alg. 4.3.2) always requires only a single SAT call per literal. Respectively, the binary generalisation *genBinary* (Alg. 4.3.3) requires in average even less than one SAT call per literal.

5.12. New Generalisation Algorithm

Based on the presented improvements in this chapter, we consider the new IC3CFA generalisation algorithm.

Algorithm 5.12.1 New generalisation algorithm of the IC3CFA algorithm.

```
 1: function CUBE GEN(i, l', c)
 2:     res ← true
 3:     for e = (l, cmd, l') ∈ G do
 4:         res ← res ∪ genEdge(i, l', c, e, res)
 5:     end for
 6:     return res
 7: end function
 8: function CUBE GENEDGE(i, l', c, e, prev)
 9:     (found, gen) ← findGenContext(i, c, e)
10:     if found then
11:         return gen
12:     else
13:         (found, gen) ← findPredecessor(i, c, e)
14:         if found then
15:             return gen
16:         else
17:             necessary ← prev ∪ findMinGeneralisation(i, c, e)
18:             cand ← c \ necessary
19:             cand ← removeAssumedLiterals(cand, e)
20:             if |cand| > 4 then
21:                 res ← genBinary(F_(i−1,l), e, cand, necessary)    ▷ e = (l, cmd, l')
22:             else
23:                 res ← genMic(F_(i−1,l), e, cand, necessary)       ▷ e = (l, cmd, l')
24:             if interpolate then
25:                 res ← interpolate(i, res, e)
26:             addGenContext(i, res, e)
27:             return res
28: end function
```

The pseudocode of the new generalisation function *gen* is shown in Alg. 5.12.1. Similar to the original IC3CFA generalisation (Def. 4.3.1), the function takes a cube c to be generalised at index $i \in \mathbb{N}$ and location $l \in L$ as input. Basically, we first try to derive the generalisation g syntactically. Therefore, we call the function $findGenContext$ to lookup previous generalisations (Sec. 5.6). Additionally, we

try to derive a generalisation g from a predecessor cube based on the approach presented in Sec. 5.5 ($findPredecessor$). If both approaches are not successful, we compute a necessary subset $necessary \subseteq c$ of original cube c. This necessary subset contains literals, which are definitely contained in the final generalisation g. The cube $necessary$ is derived from the given cube $prev$ based on the approach presented in Sec. 5.9 and the function $findMinGeneralisation$ (Sec. 5.7). The remaining literals $c \setminus necessary$ are called candidates. Again, we try to reduce these candidates by a syntactical analysis of the GCL command of the edge e, which is done in the function $removeAssumedLiterals$ (Sec. 5.4). For the resulting candidates, we call the common generalisation functions $genMic$ (Alg. 4.3.2) and $genBinary$ (Alg. 4.3.3) depending on the cube size. Finally, we potentially interpolate the resulting generalisation (Sec. 5.10). Note that, although not explicitly stated, we also make use of the improved initialisation (Sec. 5.1), the alternative relative inductiveness check (Sec. 5.3) and the clauseset ordering (Sec. 5.8) in the generalisation.

6. Evaluation

The new generalisation algorithm of IC3CFA, which has been presented in the previous Chapter 5, is realised in about 800 lines of functional OCaml code [LDF+14]. The implementation is embedded into the theory-unaware verification framework introduced in Section 4.4.

In this chapter we evaluate the performance of the IC3CFA implementation including our new generalisation algorithm (Sec. 5.12.1), which we also refer to as new IC3CFA algorithm. Therefore, we first take a closer look to the improvements compared to the IC3CFA algorithm with old generalisation (Sec. 6.1). Second, we measure the performance in comparison to other state-of-the-art model checking tools (Sec. 6.2).

For all experimental results we use the benchmark programs specified in Appendix A. Each program is executed with a timeout of 1800 seconds and a memory limit of 4GB. Furthermore, each program is executed on a single CPU core with 2.1GHz.

The score is determined as follows: Since proving the correctness of a program is usually considered to be much harder than deriving a counterexample, every successfully solved safe program increases the score by two. Respectively, a successfully solved unsafe program increases the score only by one. Overall, there are 100 safe and 50 unsafe programs in the benchmark set (Appendix A), such that the maximum score is 250.

6.1. Performance

In this section we evaluate the results of the new IC3CFA generalisation to other settings of IC3CFA algorithm. Note that the detailed experimental results of the new IC3CFA algorithm are shown in Appendix B. We compare the results to IC3CFA with the old generalisation function (Sec. 4.3.1), which we further denote by IC3CFAOld.

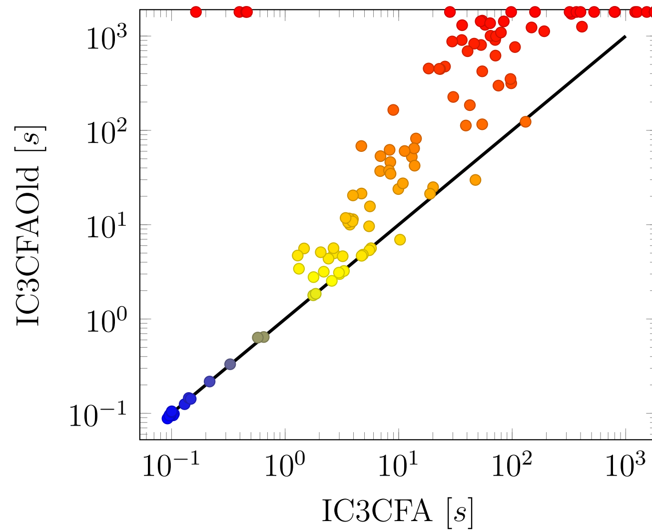

Figure 6.1.1.: Graphical comparison with IC3CFAOld.

Regarding the runtime, the scatter plot in Fig. 6.1.1 shows that we, i.e. our new IC3CFA algorithm, outperform IC3CFAOld for almost all benchmark programs. Intuitively, each dot above the diagonal line indicates that we are faster compared to the reference tool. In addition, the dots at the top represent successfully solved programs by IC3CFA, which the reference tool can not solve. Vice versa, the dots below the diagonal or at the right side indicates the opposite, respectively. Note that both axis in the scatter plot use a logarithmic scale.

Algorithm	# solved	t solved	score	memory
IC3CFAOld	101 / 150	29200 s	165	30820 MB
IC3CFA	117 / 150	10700 s	194	14230 MB

Table 6.1.: Comparison with IC3CFAOld.

Table 6.1 shows the overall results for all programs. Based on the new generalisation, we can solve more 16 programs, while using only the half amount of memory. Note that the memory is given in terms of MB and the runtime is measured in seconds. In the following, we consider possible reasons for a better performance of the new IC3CFA algorithm compared to the old one.

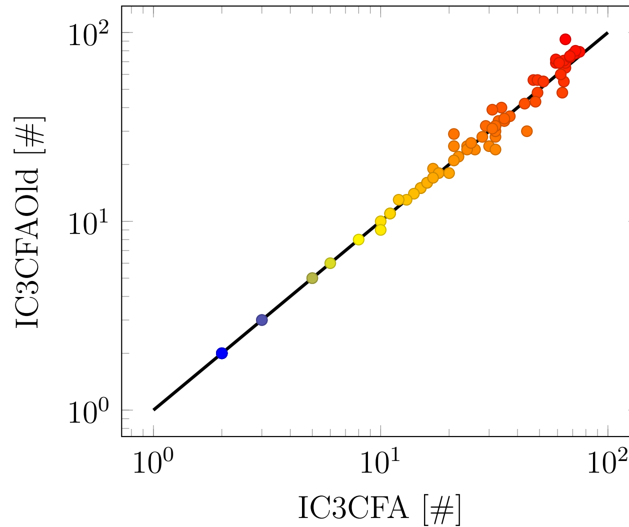

Figure 6.1.2.: Number of iterations compared to IC3CFAOld.

Fig. 6.1.2 shows the number of iterations, i.e. the maximum frame index k, required to solve the given program. Thereby, we only consider the programs solved by the new IC3CFA algorithm as well as IC3CFAOld. Basically, a reduced number of iterations indicates a faster convergence of the derived frames, which might yield a better performance. However, according to Figure 6.1.2, there is only a minor difference between both algorithms with respect to the number of iterations. Furthermore, there are also programs, where IC3CFAOld requires less iterations than our new IC3CFA algorithm.

Since the number of iterations does not significantly differ between both algorithms, we take a closer look at the required number of SMT calls in the generalisation. In general, we have tried to replace SMT calls by syntactical checks, which potentially perform better than the respective SMT calls.

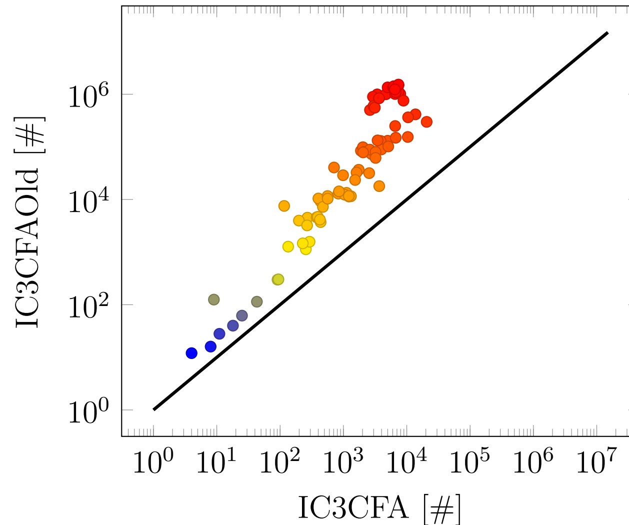

Figure 6.1.3.: Number of SMT calls in generalisation compared to IC3CFAOld.

Regarding the benchmark programs solved by IC3CFA and IC3CFAOld, a comparison of the SMT calls between both IC3CFA algorithms is shown in Fig. 6.1.3. We note that our new IC3CFA algorithm requires significantly less SMT calls for each program compared to IC3CFAOld. In best-case, IC3CFA reduces the number of SMT calls by factor 306, i.e. 2,924 SMT calls instead of 895,207 ones. Even in worst-case, the number of SMT calls is reduced by factor two compared to the old IC3CFA algorithm. Overall, the new IC3CFA algorithm requires 249,000 SMT calls for all programs under consideration, while IC3CFAOld requires 24,700,000 for the same programs. This yields a reduction of factor 67 on average per program. In consequence, the reduced number of SMT calls in the generalisation leads to an improved performance of the new IC3CFA algorithm.

In principle, the generalisation is not necessary for the IC3CFA algorithm. Thus, we also compare our new IC3CFA generalisation to the standard IC3CFA algorithm without generalisation, i.e. IC3CFANoGen.

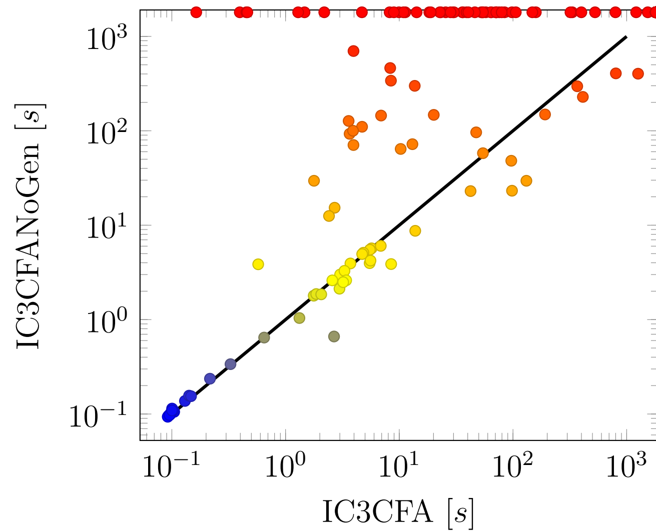

Figure 6.1.4.: Graphical comparison with IC3CFANoGen.

The experimental results are illustrated in Fig. 6.1.4. We observe that our new IC3CFA algorithm is able to solve significantly more programs, which is indicated by the dots at the top.

Algorithm	# solved	t solved	score	memory
IC3CFANoGen	65 / 150	4630 s	101	8620 MB
IC3CFA	117 / 150	10700 s	194	14230 MB

Table 6.2.: Comparison with IC3CFA without generalisation.

The overall results in Table 6.2 support this observation. In total, the IC3CFA algorithm with new generalisation can solve 52 programs more than the one without generalisation. Respectively, the score is also 93 points higher. Note that regarding the programs also solved by IC3CFANoGen, there is no clear trend between both variants, i.e. both algorithms solve specific programs faster compared to the other one. Furthermore, both IC3CFA variants require roughly the same amount of memory with respect to the number of solved programs. Overall, we conclude that our new IC3CFA algorithm is advantageous, because it solves significantly more programs than IC3CFANoGen.

6.2. Comparison

In this section we compare IC3CFA including our new generalisation algorithm to other state-of-the-art model checkers. First, we consider the tool TreeIC3 [CG12], which also tries to lift the idea of the original IC3 algorithm to software model checking.

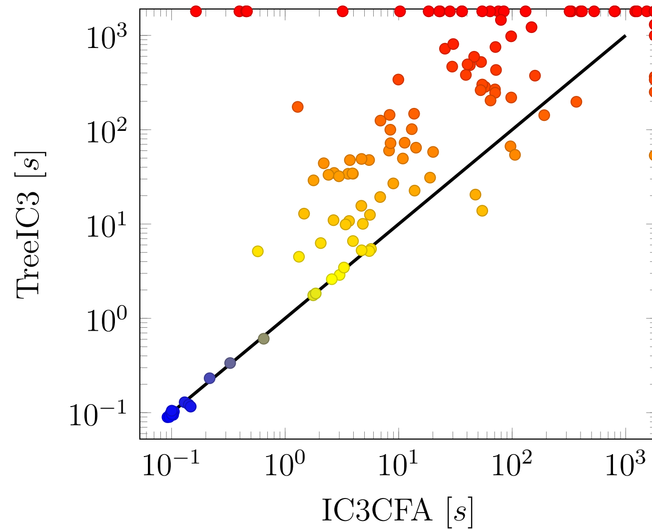

Figure 6.2.1.: Graphical comparison with TreeIC3.

Figure 6.2.1 shows a graphical comparison of the runtime between our new IC3CFA algorithm and TreeIC3 for the given benchmark programs (Appendix A).

Algorithm	# solved	t solved	score	memory
TreeIC3	96 / 150	17400 s	153	10360 MB
IC3CFA	117 / 150	10700 s	194	14230 MB

Table 6.3.: Comparison with TreeIC3.

The overall performance values are shown in Tab. 6.3. The IC3CFA algorithm is able to solve 21 programs more than TreeIC3, while simultaneously being faster on most programs. Although, there are some programs, which TreeIC3 can solve faster, IC3CFA requires overall 6,700 seconds less time compared to TreeIC3.

Furthermore, the score indicates that IC3CFA can prove the correctness of 20 safe programs more compared to TreeIC3, such that the overall score is 41 points higher. Regarding the amount of memory, the IC3CFA algorithm requires about 4 GB RAM more to solve the 21 programs in comparison to TreeIC3.

Second, we compare our new IC3CFA algorithm to bounded model checking (BMC) [BCC+03, CBRZ01], where the experimental results are shown in the following Fig. 6.2.2.

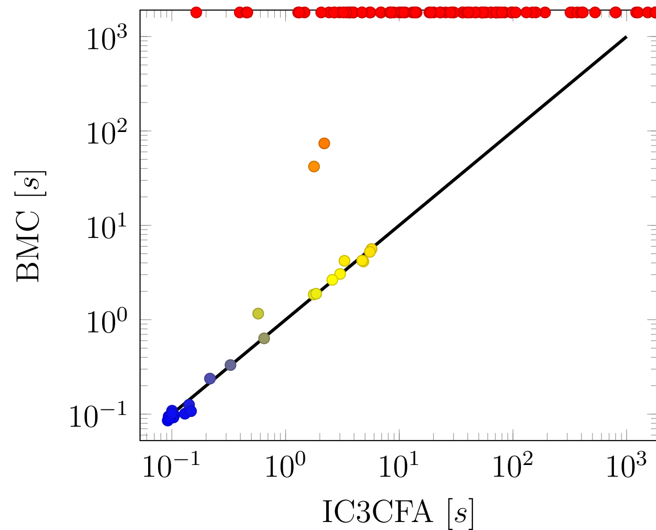

Figure 6.2.2.: Graphical comparison with BMC.

Algorithm	# solved	t solved	score	memory
BMC	28 / 150	150 s	34	2980 MB
IC3CFA	117 / 150	10700 s	194	14230 MB

Table 6.4.: Comparison with BMC.

Based on Fig. 6.2.2 and Table 6.4, IC3CFA clearly outperforms the standard BMC approach. Since BMC is only able to provide counterexamples for unsafe programs, it solves 28 of the 150 programs leading to a score of 34. Note that there are 6 safe programs, where the correctness is already derived from the computed CFA. The new IC3CFA algorithm solves 89 programs more, especially it proves

the correctness of 77 safe programs. In consequence, the overall score of IC3CFA is 160 points higher compared to the one of BMC.

The experimental results of CEGAR ("CounterExample-Guided Abstraction Refinement") compared to our new IC3CFA algorithm are illustrated in detail in the following Fig. 6.2.3.

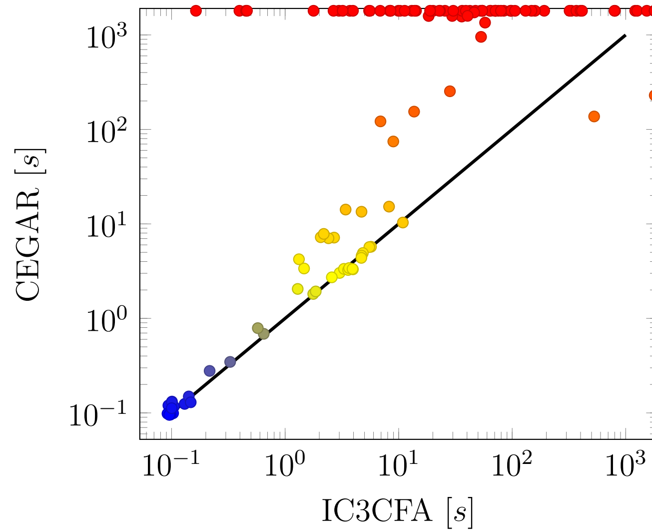

Figure 6.2.3.: Graphical comparison with CEGAR.

CEGAR is a state-of-the-art model checking algorithm based on iterative abstraction refinements [CGJ+00, CGJ+03]. We note that IC3CFA outperforms CEGAR on almost all benchmark programs, in particular it solves 62 programs more in comparison to CEGAR.

Algorithm	# solved	t solved	score	memory
CEGAR	55 / 150	11500 s	79	7380 MB
IC3CFA	117 / 150	10700 s	194	14230 MB

Table 6.5.: Comparison with CEGAR.

Respectively, the score of IC3CFA in Table 6.5 is 115 pointer higher compared to the one of CEGAR. Furthermore, the new IC3CFA algorithm proves the correctness

of 117 programs in 10,700 seconds, while CEGAR requires 11,500 seconds to solve only 55 programs.

Finally, we compare our new IC3CFA algorithm to the CPAchecker [BK11], which is state-of-the-art one of the fastest model checkers for verifying the given set of benchmark programs (Appendix A) [Bey15].

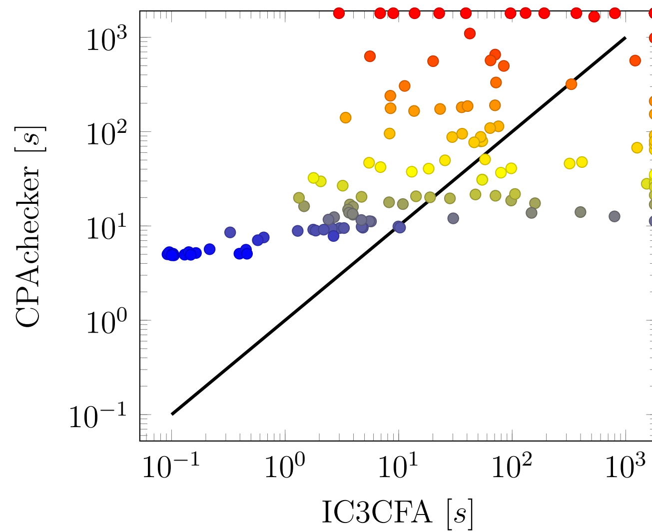

Figure 6.2.4.: Graphical comparison with CPAchecker.

Algorithm	# solved	t solved	score	memory
CPAchecker	120 / 150	12680 s	193	44000 MB
IC3CFA	117 / 150	10700 s	194	14230 MB

Table 6.6.: Comparison with CPAchecker.

According to the resulting performance values in Table 6.6, the CPAChecker is able to solve three programs more than our new IC3CFA algorithm. However, IC3CFA can prove the correctness of more safe programs, such that the overall score is one point higher. Regarding the runtime performance and the amount of memory, IC3CFA is about 2,000 seconds faster and requires only one third of the memory compared to CPAchecker. The detailed evaluation in Fig. 6.2.4 shows that IC3CFA is faster regarding the runtime, when verifying more than 50 percent of the

programs. However, there are also programs, which CPAchecker solves anyway or verifies faster compared to IC3CFA. In consequence, we observe that IC3CFA and CPAchecker are so-called orthogonal approaches, i.e. both algorithms have their advantages with respect to the other one. Overall, the experimental results have proven that our new IC3CFA algorithm is competitive to other state-of-the-art model checking tools.

7. Conclusion

7.1. Summary

In this thesis we have presented a new generalisation function for the IC3CFA algorithm. Our approach has been based on the idea to lift the principles of the original IC3 algorithm to software model checking. Furthermore, we have tried to exploit the program structure given in terms of the control flow automaton (CFA). With regard to the generalisation, we have mainly replaced SMT calls by equivalent syntactical checks, which perform faster or scale better according to the experimental results. More precise, we have made the following contributions to the new IC3CFA generalisation.

The improved initialisation of the frames avoids unnecessary computations in the generalisation (Sec. 5.1). Regarding the predecessor computation, we have presented a new approach to convert the predecessors into DNF based on the structure of the GCL command (Sec. 5.2). Furthermore, we have shown an alternative relative inductiveness check, which is advantageous in combination with a SMT solver with caching (Sec. 5.3). Given a CFA edge labelled with a GCL command, we have extended the idea to remove literals already implied by the GCL assumes (Sec. 5.4). We have also presented an approach to derive the generalisation based on the predecessor cubes (Sec. 5.5). In addition, we have introduced the so-called generalisation context. This context is used to deduce the generalisation based on the previous generalisations (Sec. 5.6). Analogously, we have introduced the concept of minimal generalisation based on the generalisation context, which determines a necessary sub-cube of the final generalisation (Sec. 5.7). Regarding a set of cubes to be blocked at a certain location, we have proposed a new ordering of these cubes to avoid SMT calls (Sec. 5.8). Furthermore, we have reduced the required number of SMT calls in the generalisation with regard to multiple predecessor edges (Sec. 5.9). To compute semantic generalisations in the IC3CFA algorithm, we have made use of interpolation (Sec. 5.10). We have also tried to lift the idea of counterexamples to generalisation (CTGs) from the original IC3 algorithm to IC3CFA (Sec. 5.11). Finally, we have proposed a new IC3CFA generalisation algorithm, which includes the presented improvements (Sec. 5.12).

Based on the contributions, we have implemented a new IC3CFA algorithm including our generalisation into the existing model checking framework (Sec. 4.4). Note that our implementation is also fully theory-unaware. We have evaluated the performance of the new IC3CFA algorithm based on the given set of benchmark programs (Appendix A). The new IC3CFA algorithm can solve significantly more programs in comparison to the old IC3CFA generalisation, while simultaneously being faster on most programs. The experimental results have also proven that the new IC3CFA algorithm outperforms other model checking approaches, like TreeIC3 and CEGAR. Finally, we have shown that the new IC3CFA algorithm is competitive to the state-of-the-art model checking tool CPAchecker, where both approaches are orthogonal to each other.

7.2. Future Work

Regarding the future development of our new IC3CFA algorithm, we propose three possible approaches.

First, we might consider the development of further syntactical checks in the generalisation algorithm. Since the experimental results have shown that the syntactical checks often perform faster or scale better, we potentially solve more programs by further reducing the number of SMT calls. Analogously, we might consider the reduction of SMT calls in the main IC3CFA algorithm, such that we try to reduce the number of SMT calls for reachability.

Second, in this thesis we have mainly focused on the reduction of SMT calls. However, we might also consider applying more and simpler SMT calls to derive better generalisations. Following this idea, we have presented a first approach with counterexamples to generalisation (CTGs) in Sec. 5.11. However, this approach has not improved the performance. Nevertheless, there might be other approaches or further developments with a positive effect on the overall runtime.

Third, we might consider the integration of IC3CFA into a so-called CPA framework, similar to the one in the model checking tool CPAchecker. The CPAchecker yields state-of-the-art a competitive performance with respect to our new IC3CFA algorithm, while using an orthogonal approach. It is based on the idea to combine the verification algorithm and static program analyses, where this idea has been originally proposed in [BHT07]. Thus, a combination of our new IC3CFA algorithm and the CPA framework might turn out to be very effective, because it ideally combines the advantages of both approaches, i.e. IC3CFA and CPAchecker.

Bibliography

[All70] F. E. Allen. Control flow analysis. In *Proc. of ACM*, volume 5, pages 1–19. ACM, 1970.

[BCC+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[BCF+08] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *Proc. of CAV*, pages 299–303. Springer, 2008.

[BCG+09] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. of FMCAD*, pages 25–32. IEEE, 2009.

[Bey15] D. Beyer. Software verification and verifiable witnesses. In *Proc. of TACAS*, pages 401–416. Springer, 2015.

[BHT07] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. of CAV*, pages 504–518. Springer, 2007.

[BK08] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[BK11] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. of CAV*, pages 184–190. Springer, 2011.

[BKW10] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. of FMCAD*, pages 189–198. FMCAD Inc, 2010.

[BM07] A. R. Bradley and Z. Manna. Checking safety by inductive generalization of counterexamples to induction. In *Proc. of FMCAD*, pages 173–180. IEEE, 2007.

[Bra11] A. R. Bradley. SAT-based model checking without unrolling. In *Proc. of VMCAI*, pages 70–87. Springer, 2011.

[BS92] A. Bouali and R. De Simone. Symbolic bisimulation minimisation. In *Proc. of CAV*, pages 96–108. Springer, 1992.

[CBRZ01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CG12] A. Cimatti and A. Griggio. Software model checking via IC3. In *Proc. of CAV*, pages 277–293. Springer, 2012.

[CG15] W. Conradie and V. Goranko. *Logic and Discrete Mathematics: A Concise Introduction.* John Wiley & Sons, 2015.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*, pages 154–169. Springer, 2000.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT Press, 1999.

[CK90] C. C. Chang and H. J. Keisler. *Model theory*, volume 73. Elsevier, 1990.

[Cra57] W. Craig. Linear reasoning. a new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(03):250–268, 1957.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dij76] E. W. Dijkstra. *A discipline of programming*, volume 1. Englewood Cliffs: prentice-hall, 1976.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to lattices and order.* Cambridge University Press, 2002.

[Fit90] M. Fitting. First-order logic. In *First-Order Logic and Automated Theorem Proving*, pages 97–125. Springer, 1990.

[FV02] K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.

[GKSS08] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[GS93] D. Gries and F. B. Schneider. Boolean expressions. In *A Logical Approach to Discrete Math*, pages 25–40. Springer, 1993.

[HBS13] Z. Hassan, A. R. Bradley, and F. Somenzi. Better generalization in IC3. In *Proc. of FMCAD*, pages 157–164, 2013.

[Kel76] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[LDF+14] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, pages 1–586, 2014.

[Lee61] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, pages 346–365, 1961.

[Lei05] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[LNN15] T. Lange, M. R. Neuhäußer, and T. Noll. IC3 software model checking on control flow automata. In *Proc. of FMCAD*, pages 97–104. FMCAD Inc, 2015.

[MB08] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, pages 337–340. Springer, 2008.

[McM03] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. of CAV*, pages 1–13. Springer, 2003.

[Mon08] D. Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *Proc. of LPAR*, pages 243–257. Springer, 2008.

[Mue15] S. Mueller. Evaluating control-flow based inductive model checking algorithms. Master's thesis, RWTH Aachen, 2015.

[Smu95] R. M. Smullyan. *First-order logic*. Courier Corporation, 1995.

[Sof15] Software verification competition. `http://sv-comp.sosy-lab.org/2015/`, 2015. Accessed: 2016-04-01.

[WCC+95] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2):69, 1995.

[WGI07] C. Wang, A. Gupta, and F. Ivancic. Induction in CEGAR for detecting counterexamples. In *Proc. of FMCAD*, pages 77–84. IEEE, 2007.

[WHM13] C. M. Wintersteiger, Y. Hamadi, and L. De Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.

A. Benchmark Programs

cimatti-cav12	
array_max-1_true-unreach-call.c	s3_srvr_14.cil_true-unreach-call.c
array_max-2_true-unreach-call.c	s3_srvr_15.cil_true-unreach-call.c
array_max-3_true-unreach-call.c	s3_srvr_16.cil_true-unreach-call.c
array_max-4_true-unreach-call.c	s3_srvr_2.cil_false-unreach-call.c
array_max-5_true-unreach-call.c	s3_srvr_2.cil_true-unreach-call.c
bist_cell_true-unreach-call.c	s3_srvr_3.cil_true-unreach-call.c
bubblesort-1_true-unreach-call.c	s3_srvr_4.cil_true-unreach-call.c
bubblesort-2_true-unreach-call.c	s3_srvr_6.cil_true-unreach-call.c
bubblesort-3_true-unreach-call.c	s3_srvr_7.cil_true-unreach-call.c
bubblesort_init-1_true-unreach-call.c	s3_srvr_8.cil_true-unreach-call.c
bubblesort_init-2_true-unreach-call.c	s3_srvr_9.cil_true-unreach-call.c
bubblesort_init-3_true-unreach-call.c	simple_array_inversion-1_true-unreach-call.c
cdaudio_simpl1.cil_64_false-unreach-call.c	simple_array_inversion-2_true-unreach-call.c
cdaudio_simpl1.cil_false-unreach-call.c	simple_array_inversion-3_true-unreach-call.c
diskperf_simpl1.cil_64_false-unreach-call.c	simple_array_inversion-4_true-unreach-call.c
floppy_simpl3.cil_64_false-unreach-call.c	simple_array_inversion-5_true-unreach-call.c
floppy_simpl3.cil_false-unreach-call.c	simple_array_inversion-6_true-unreach-call.c
floppy_simpl4.cil_64_false-unreach-call.c	simple_array_inversion-7_true-unreach-call.c
floppy_simpl4.cil_false-unreach-call.c	simple_array_inversion-8_true-unreach-call.c
kbfiltr_simpl1.cil_false-unreach-call.c	simple_array_inversion-9_true-unreach-call.c
kbfiltr_simpl2.cil_false-unreach-call.c	token_ring.10_true-unreach-call.c
kbfiltr_simpl2_wrong.cil_false-unreach-call.c	token_ring.11_true-unreach-call.c
kundu-1_false-unreach-call.c	token_ring.12_true-unreach-call.c
kundu-2_false-unreach-call.c	token_ring.13_true-unreach-call.c
kundu_true-unreach-call.c	token_ring.1_true-unreach-call.c
mem_slave_tlm.1_true-unreach-call.c	token_ring.2_true-unreach-call.c
mem_slave_tlm.2_true-unreach-call.c	token_ring.3_true-unreach-call.c
mem_slave_tlm.3_true-unreach-call.c	token_ring.4_true-unreach-call.c
mem_slave_tlm.4_true-unreach-call.c	token_ring.5_true-unreach-call.c
mem_slave_tlm.5_true-unreach-call.c	token_ring.6_true-unreach-call.c
pc_sfifo_1_true-unreach-call.c	token_ring.7_true-unreach-call.c
pc_sfifo_2_true-unreach-call.c	token_ring.8_true-unreach-call.c
pc_sfifo_3_true-unreach-call.c	token_ring.9_true-unreach-call.c
pipeline_false-unreach-call.c	toy-1_false-unreach-call.c
pipeline_true-unreach-call.c	toy-2_false-unreach-call.c
s3_clnt_1.cil_false-unreach-call.c	toy_true-unreach-call.c
s3_clnt_1.cil_true-unreach-call.c	transmitter.10_false-unreach-call.c
s3_clnt_2.cil_false-unreach-call.c	transmitter.11_false-unreach-call.c
s3_clnt_2.cil_true-unreach-call.c	transmitter.12_false-unreach-call.c
s3_clnt_3.cil_false-unreach-call.c	transmitter.13_false-unreach-call.c
s3_clnt_3.cil_true-unreach-call.c	transmitter.1_false-unreach-call.c
s3_clnt_4.cil_false-unreach-call.c	transmitter.2_false-unreach-call.c
s3_clnt_4.cil_true-unreach-call.c	transmitter.3_false-unreach-call.c
s3_srvr_1.cil_false-unreach-call.c	transmitter.4_false-unreach-call.c
s3_srvr_1.cil_true-unreach-call.c	transmitter.5_false-unreach-call.c
s3_srvr_10.cil_true-unreach-call.c	transmitter.6_false-unreach-call.c
s3_srvr_11.cil_true-unreach-call.c	transmitter.7_false-unreach-call.c
s3_srvr_12.cil_true-unreach-call.c	transmitter.8_false-unreach-call.c
s3_srvr_13.cil_true-unreach-call.c	transmitter.9_false-unreach-call.c

Table A.1.: Benchmark programs used in [CG12].

svcomp15
bitvector/byte_add_1_true-unreach-call.c
bitvector/byte_add_2_true-unreach-call.c
bitvector/byte_add_false-unreach-call.c
bitvector/gcd_1_true-unreach-call.c
bitvector/gcd_2_true-unreach-call.c
bitvector/gcd_3_true-unreach-call.c
bitvector/gcd_4_true-unreach-call.c
bitvector/interleave_bits_true-unreach-call.c
bitvector/jain_1_true-unreach-call.c
bitvector/jain_2_true-unreach-call.c
bitvector/jain_4_true-unreach-call.c
bitvector/jain_5_true-unreach-call.c
bitvector/jain_6_true-unreach-call.c
bitvector/jain_7_true-unreach-call.c
bitvector/modulus_true-unreach-call.c
bitvector/num_conversion_1_true-unreach-call.c
bitvector/num_conversion_2_true-unreach-call.c
bitvector/parity_true-unreach-call.c
bitvector/s3_clnt_1_false-unreach-call.BV.c.cil.c
bitvector/s3_clnt_1_true-unreach-call.BV.c.cil.c
bitvector/s3_clnt_2_false-unreach-call.BV.c.cil.c
bitvector/s3_clnt_2_true-unreach-call.BV.c.cil.c
bitvector/s3_clnt_3_false-unreach-call.BV.c.cil.c
bitvector/s3_clnt_3_true-unreach-call.BV.c.cil.c
bitvector/s3_srvr_1_alt_true-unreach-call.BV.c.cil.c
bitvector/s3_srvr_1_true-unreach-call.BV.c.cil.c
bitvector/s3_srvr_2_alt_true-unreach-call.BV.c.cil.c
bitvector/s3_srvr_2_true-unreach-call.BV.c.cil.c
bitvector/s3_srvr_3_alt_true-unreach-call.BV.c.cil.c
bitvector/s3_srvr_3_true-unreach-call.BV.c.cil.c
bitvector/soft_float_1_true-unreach-call.c.cil.c
bitvector/soft_float_2_true-unreach-call.c.cil.c
bitvector/soft_float_3_true-unreach-call.c.cil.c
bitvector/soft_float_4_true-unreach-call.c.cil.c
bitvector/soft_float_5_true-unreach-call.c.cil.c
bitvector/sum02_true-unreach-call.c
bitvector-regression/implicitfloatconversion_false-unreach-call.c
bitvector-regression/implicitunsignedconversion_false-unreach-call.c
bitvector-regression/implicitunsignedconversion_true-unreach-call.c
bitvector-regression/integerpromotion_false-unreach-call.c
bitvector-regression/integerpromotion_true-unreach-call.c
bitvector-regression/pointer_extension2_false-unreach-call.c
bitvector-regression/pointer_extension3_false-unreach-call.c
bitvector-regression/pointer_extension_false-unreach-call.c
bitvector-regression/pointer_extension_true-unreach-call.c
bitvector-regression/recHanoi03_unsafe.c
bitvector-regression/signextension2_false-unreach-call.c
bitvector-regression/signextension2_true-unreach-call.c
bitvector-regression/signextension_false-unreach-call.c
bitvector-regression/signextension_true-unreach-call.c
bitvector-loops/diamond_false-unreach-call2.c
bitvector-loops/overflow_false-unreach-call1.c

Table A.2.: Benchmark programs of the software verification competition [Sof15].

B. Detailed Experimental Results

The following table shows the detailed experimental results of our new generalisation algorithm for IC3CFA with optimal settings.

- Program: name of the benchmark program
- status: true (safe)/ false (unsafe)/ Timeout/ Memory (Out of Memory)/ Error
- t: required time in seconds
- SAT: required time in the SMT solver
- # SAT: number of SMT queries
- k: required number of IC3CFA iterations, i.e. maximum frame index
- memory: required amount of memory in MB

Program	status	t	SAT	# SAT	k	memory
array_max-1_true-unreach-call	true	13	7.656	1708	24	65
array_max-2_true-unreach-call	true	76	48.84	4971	29	115
array_max-3_true-unreach-call	true	330	215.7	13688	34	307
array_max-4_true-unreach-call	true	1200	750.8	32913	42	848
array_max-5_true-unreach-call	Timeout	1800				1612
bist_cell_true-unreach-call	true	0.33				56
bubblesort-1_true-unreach-call	true	5.5	4.236	466	30	56
bubblesort-2_true-unreach-call	true	42	34.93	2070	64	74
bubblesort-3_true-unreach-call	true	800	730.4	16385	111	273
bubblesort_init-1_true-unreach-call	true	3.7	2.584	420	32	55
bubblesort_init-2_true-unreach-call	true	98	83.28	4448	65	101
bubblesort_init-3_true-unreach-call	true	1300	1.146	23806	119	415
cdaudio_simpl1_64_false-unreach-call	false	5.7	0	2		58
cdaudio_simpl1_false-unreach-call	false	5.5	0.004000	2		58
diskperf_simpl1_64_false-unreach-call	false	3.0	0	2		54
floppy_simpl3_64_false-unreach-call	false	1.8				62
floppy_simpl3_false-unreach-call	false	1.9				62
floppy_simpl4_64_false-unreach-call	false	4.8	0.1120	45	3	54
floppy_simpl4_false-unreach-call	false	4.7	0.09200	46	3	54
kbfiltr_simpl1_false-unreach-call	false	0.65				56
kbfiltr_simpl2_false-unreach-call	false	3.3	0.02800	10	2	56
kbfiltr_simpl2_wrong_false-unreach-call	false	2.6	0.02000	9	2	56
kundu-1_false-unreach-call	false	4.7	2.292	1020	16	52
kundu-2_false-unreach-call	false	4.0	1.688	764	15	56
kundu_true-unreach-call	true	320	162.3	23096	85	524
mem_slave_tlm.1_true-unreach-call	Timeout	1800				3132
mem_slave_tlm.2_true-unreach-call	Timeout	1800				3134
mem_slave_tlm.3_true-unreach-call	Timeout	1800				2736
mem_slave_tlm.4_true-unreach-call	Timeout	1800				2725
mem_slave_tlm.5_true-unreach-call	Timeout	1800				2829
pc_sfifo_1_true-unreach-call	true	1.5	0.5200	357	17	51
pc_sfifo_2_true-unreach-call	true	14	6.624	1764	31	81
pc_sfifo_3_true-unreach-call	true	1.3	0.4960	178	21	58
pipeline_false-unreach-call	false	530	198.3	21069	121	526
pipeline_true-unreach-call	Timeout	1800				804

Program	status	t	SAT	# SAT	k	memory
s3_clnt_1_false-unreach-call	false	3.6	0.8120	345	15	50
s3_clnt_1_true-unreach-call	true	6.9	1.800	657	33	54
s3_clnt_2_false-unreach-call	false	4.0	1.040	385	16	54
s3_clnt_2_true-unreach-call	true	8.4	3.104	916	37	55
s3_clnt_3_false-unreach-call	false	3.7	0.9360	342	15	52
s3_clnt_3_true-unreach-call	true	8.3	2.612	820	35	58
s3_clnt_4_false-unreach-call	false	3.9	0.8880	339	15	49
s3_clnt_4_true-unreach-call	true	14	3.976	1154	35	73
s3_srvr_1_false-unreach-call	false	2.7	0.1680	78	11	57
s3_srvr_1_true-unreach-call	true	53	3.324	864	65	98
s3_srvr_10_true-unreach-call	true	58	3.412	1333	68	101
s3_srvr_11_true-unreach-call	true	36	2.900	1139	65	84
s3_srvr_12_true-unreach-call	true	71	2.924	1106	64	95
s3_srvr_13_true-unreach-call	true	36	2.648	1070	59	95
s3_srvr_14_true-unreach-call	true	71	3.524	1281	68	108
s3_srvr_15_true-unreach-call	true	54	2.920	1342	73	84
s3_srvr_16_true-unreach-call	true	64	3.140	1188	70	108
s3_srvr_2_false-unreach-call	false	2.4	0.1200	62	11	59
s3_srvr_2_true-unreach-call	true	18	1.568	699	47	71
s3_srvr_3_true-unreach-call	true	30	1.828	759	59	80
s3_srvr_4_true-unreach-call	true	26	1.648	726	49	79
s3_srvr_6_true-unreach-call	true	72	3.100	1276	72	96
s3_srvr_7_true-unreach-call	true	84	4.320	1546	75	108
s3_srvr_8_true-unreach-call	true	53	2.940	1270	71	107
s3_srvr_9_true-unreach-call	true	64	4.236	1588	72	121
simple_array_inversion-1_true-unreach-call	true	1.3	0.5080	273	18	51
simple_array_inversion-2_true-unreach-call	true	2.1	0.9120	412	20	56
simple_array_inversion-3_true-unreach-call	true	3.4	1.456	622	26	53
simple_array_inversion-4_true-unreach-call	true	5.6	2.504	937	32	54
simple_array_inversion-5_true-unreach-call	true	6.9	2.484	1088	32	56
simple_array_inversion-6_true-unreach-call	true	14	5.220	1865	44	68
simple_array_inversion-7_true-unreach-call	true	97	34.84	5707	63	120
simple_array_inversion-8_true-unreach-call	true	190	53.64	7526	62	229
simple_array_inversion-9_true-unreach-call	true	370	94.97	11553	71	380
token_ring.10_true-unreach-call	Timeout	1800				1620
token_ring.11_true-unreach-call	Timeout	1800				1622
token_ring.12_true-unreach-call	Timeout	1800				1694
token_ring.13_true-unreach-call	Timeout	1800				1297
token_ring.1_true-unreach-call	true	4.7	1.744	648	21	57
token_ring.2_true-unreach-call	true	160	82.56	10047	53	279
token_ring.3_true-unreach-call	true	1500	766.2	55400	83	1661
token_ring.4_true-unreach-call	Timeout	1800				2664
token_ring.5_true-unreach-call	Timeout	1800				2404
token_ring.6_true-unreach-call	Timeout	1800				2456
token_ring.7_true-unreach-call	Timeout	1800				2473
token_ring.8_true-unreach-call	Timeout	1800				1706
token_ring.9_true-unreach-call	Timeout	1800				1953
toy-1_false-unreach-call	false	98	53.26	9161	32	200
toy-2_false-unreach-call	false	71	38.16	6376	32	136
toy_true-unreach-call	Timeout	1800				1938

Program	status	t	SAT	# SAT	k	memory
transmitter.10_false-unreach-call	Timeout	1800				1859
transmitter.11_false-unreach-call	Timeout	1800				1824
transmitter.12_false-unreach-call	Timeout	1800				2594
transmitter.13_false-unreach-call	Timeout	1800				2831
transmitter.1_false-unreach-call	false	0.57	0.05600	29	6	52
transmitter.2_false-unreach-call	false	2.2	0.5800	290	10	42
transmitter.3_false-unreach-call	false	9.9	3.164	1243	16	59
transmitter.4_false-unreach-call	false	30	12.54	4112	22	103
transmitter.5_false-unreach-call	false	150	67.16	15699	28	330
transmitter.6_false-unreach-call	false	400	172.4	35380	33	935
transmitter.7_false-unreach-call	Timeout	1800				3189
transmitter.8_false-unreach-call	Timeout	1800				2787
transmitter.9_false-unreach-call	Timeout	1800				2717
byte_add_1_true-unreach-call	true	54	39.88	3174	49	128
byte_add_2_true-unreach-call	true	110	68.92	4895	43	193
byte_add_false-unreach-call	false	1.8	0.6560	257	11	46
gcd_1_true-unreach-call	true	47	47.06	68	8	85
gcd_2_true-unreach-call	true	10	10.07	31	5	65
gcd_3_true-unreach-call	true	20	19.84	40	5	72
gcd_4_true-unreach-call	true	800	794.7	577	13	180
interleave_bits_true-unreach-call	true	54	50.92	427	32	79
jain_1_true-unreach-call	true	0.16	0.03200	9	3	47
jain_2_true-unreach-call	true	0.40	0.1960	9	3	56
jain_4_true-unreach-call	true	0.45	0.2440	9	3	55
jain_5_true-unreach-call	Timeout	1800				93
jain_6_true-unreach-call	true	0.46	0.2600	9	3	56
jain_7_true-unreach-call	Timeout	1800				837
modulus_true-unreach-call	true	3.0	2.812	12	2	48
num_conversion_1_true-unreach-call	true	2.7	1.880	276	17	54
num_conversion_2_true-unreach-call	true	410	339.5	15127	24	246
parity_true-unreach-call	true	130	125.0	1245	35	71
s3_clnt_1_false-unreach-call	false	8.2	2.232	719	21	63
s3_clnt_1_true-unreach-call	Timeout	1800				1819
s3_clnt_2_false-unreach-call	false	11	4.192	1135	31	70
s3_clnt_2_true-unreach-call	true	39	14.74	2928	48	97
s3_clnt_3_false-unreach-call	false	11	3.220	883	14	70
s3_clnt_3_true-unreach-call	true	9.0	2.376	870	25	59
s3_srvr_1_alt_true-unreach-call	Timeout	1800				167
s3_srvr_1_true-unreach-call	true	80	6.876	1940	68	121
s3_srvr_2_alt_true-unreach-call	true	23	2.872	598	52	85
s3_srvr_2_true-unreach-call	true	23	2.868	600	52	80
s3_srvr_3_alt_true-unreach-call	true	46	9.252	925	61	95
s3_srvr_3_true-unreach-call	true	40	3.436	924	61	85
soft_float_1_true-unreach-call	true	3.2	2.536	187	10	57
soft_float_2_true-unreach-call	true	19	18.10	298	13	40
soft_float_3_true-unreach-call	Timeout	1800				134
soft_float_4_true-unreach-call	true	8.5	7.328	409	12	39
soft_float_5_true-unreach-call	true	28	27.18	418	21	47

Program	status	t	SAT	# SAT	k	memory
sum02_true-unreach-call	Timeout	1800				983
implicitfloatconversion_false-unreach-call	false	0.13	0.02000	2		39
implicitunsignedconversion_false-unreach-call	false	94				36
implicitunsignedconversion_true-unreach-call	true	92				36
integerpromotion_false-unreach-call	false	0.10				37
integerpromotion_true-unreach-call	true	0.10				37
pointer_extension2_false-unreach-call	false	0.10	0.004000	2		38
pointer_extension3_false-unreach-call	false	98	0.004000	2		37
pointer_extension_false-unreach-call	false	97	0	2		37
pointer_extension_true-unreach-call	true	95	0	1		37
recHanoi03_unsafe	Memory	900				4000
signextension2_false-unreach-call	false	0.14				48
signextension2_true-unreach-call	true	0.15				48
signextension_false-unreach-call	false	0.10				37
signextension_true-unreach-call	true	0.10				37
diamond_false-unreach-call2	false	0.22	0.01200	3	2	44
overflow_false-unreach-call1	Timeout	1800				386

Table B.1.: Detailed experimental results of the benchmark programs.

Index

A
abstract reachability tree . . . 31
alternative relative inductiveness . . 42
assume . . . 45

B
bit-vector theory . . . 20, 31
bmc . . . 69
bounded model checking . . . 31

C
cegar . . . 31, 70
cfa . . . 10
clauseset . . . 6
counterexample . . . 15, 24
cpachecker . . . 71
ctg . . . 59
cti . . . 15, 26
cube . . . 6

D
distributivity . . . 44
dnf . . . 39

E
expression . . . 7

F
first-order formula . . . 5
first-order logic . . . 5
formal verification . . . 1
frame . . . 21
frame relation . . . 22
frame sequence . . . 22
framework . . . 30

G
gcl . . . 7
gcl semantics . . . 8
generalisation . . . 28
generalisation context . . . 50

I
ic3 . . . i, 13
ic3 generalisation . . . 18, 19
ic3 invariants . . . 14
ic3 pseudocode . . . 16
ic3cfa . . . 21
ic3cfa generalisation . . . 27
ic3cfa invariants . . . 24
ic3cfa new generalisation . . . 61
ic3cfa no generalisation . . . 66
ic3cfa pseudocode . . . 25
improved initialisation . . . 34
inductive strengthening . . . 14
inductiveness . . . 14
interpolant . . . 58
interpolation . . . 57
ivl . . . 31

L
large-block encoding . . . 10
least upper bound . . . 19
linear real arithmetic . . . 20, 31
literal . . . 6

M
minimal generalisation 53
model 5
model checking 1
model checking process 1
multiple predecessors 55

N
new generalisation context 52

O
ordering 55

P
path 33
predecessor cube 47
priming 5
program state 5
program variable 5
proof obligation 17, 26
propositional property 14

R
relative inductiveness 14, 23

S
score 65
smt 20
static program analysis 31

T
testing 1
transition formula 10
transition system 13
treeic3 . . . . . 31, 68

W
weakest precondition 36
wep 36
wep function 37