Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 2
Software Modeling and Verification

Master's Thesis

Efficient reuse of learnt information for control-flow oriented IC3 algorithms

Thomas Mertens
Student Number: 312690

September 21, 2016

First Reviewer:
Prof. Dr. Thomas Noll
Second Reviewer:
Prof. Dr. Ir. Joost-Pieter Katoen

Advisor:
M. Sc. Tim Lange
Dr. Martin Neuhäußer

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

Abstract

The goal of this thesis is to investigate the effects of efficient information reuse on control flow based IC3 algorithms. Furthermore, the goal is to evaluate the investigated approaches concerning their performance. The subjects of thesis were collected among the already existing approach to improve IC3 and among observations during research on IC3CFA. The first approach reuses computations of further iterations to reduce the effort of re-computations. This approach is extended to achieve a more effective handling of reusing previous computations, so called obligation skipping. In addition, a refinement of the original termination criterion is presented in this thesis. Finally, the pushing technique of the original IC3 algorithm is lifted to the setting of IC3CFA. The theory behind all approaches is described as well as their implementation in an existing framework and evaluated. On basis of the results of this thesis, it can be concluded that the biggest improvement is achieved by obligation reuse combined with obligation skipping. The new termination criteria have only limited effect due to the underlying control flow automaton. Furthermore, the results did not support the expectations that the adaptation of the pushing technique yields improvements.

Contents

List of Figures

1 Introduction

Ensuring the correctness of software is one of the major goals for developers. While some failures may only annoy the user during using the software, others failures may cause high economical costs or even harm people. To prevent those failures verification ensures the absence of logical errors in software. Therefore, verification can improve software development in various design phases. We distinguish between two types of code verification: a semantic check of code and software model checking. A semantic check of a software can be done for example via the Hoare calculus. By using the Hoare calculus, the program or a set of functions of the program can be checked. Therefore, one introduces preconditions and postconditions for each function. For a precondition P, a program Q and the postcondition R following notation is used: $P\{Q\}R$. If the precondition P holds before executing the program Q, the postcondition R must hold as well. In case that the program Q has no precondition one denotes it with $true\{Q\}R$ [Hoa69].
A novel technique of software verification has recently been proposed by Aaron Bradley under the name Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) [Bra11], also known as Property Directed Reachability (PDR) [EMB11]. IC3 is an incremental algorithm to verify invariants of finite transition systems. This new algorithm had major impact on hardware model checking in the last decade. During the last decade several combinations of ART-based approaches combined with CEGAR have been developed. For verifying a property, the algorithm creates boolean clauses which are an over-approximation of reachable states within an upper bound of steps which is incremented stepwise. The created clauses are all stepwise relative inductive, i.e. each state described by the clauses still holds after taking a transition. Applying IC3 on a non-trivial problem causes tens of thousands of SAT queries. Theses queries test if formulas are 1-step reachable. Comparing the complexity of SAT queries executed by IC3 with queries of other common methods one observes that they are of low complexity [Bra11].

Experiments using the low complexity of SAT queries caused by IC3 on hardware model checking have shown that IC3 is superior to any other single solver used in hardware model checking. The new technique has several advantages, first, IC3 does not need to unroll the transition relation for more than one step. In comparison, the bounded model checker unrolls the transition system up to a given bound. Also for k-induction or interpolation the transition system needs to be unrolled for more than just a single step. Second, the reasoning is based on sets of clauses and is driven by the property which is being checked. Finally, the method makes use of the powerful modern SAT solvers, which are able to solve huge numbers of small problems efficiently [CG12]. This algorithm has been shown to be highly effective and is one of the most important algorithms for hardware model checking.

The original algorithm of IC3 has been adapted in multiple ways. The setting of software model checking needs a more advanced symbolic reasoning technique due to the infinite state system. One of the most prominent techniques to achieve an advanced symbolic reasoning is Satisfiability Modulo Theories (SMT). In this technique a first-order formula symbolically represents a set of states while spurious counterexamples are disproven by SMT-solving techniques. A first approach of using the symbolic reasoning is named Tree-IC3 [CG12]. In this approach an Abstract Reachability Tree (ART) is unwinded where each location is related to a control location and is represented by a formula. Using Tree-IC3 makes an adaptation of the relative inductiveness necessary, because the path-wise unwinding of the transition system does not hold for that. In ongoing research, a monolithic transition relation replaced the pre-image computation by predicate abstraction.

One of these adaptations is using IC3 for software model checking on control flow automata. In this lifted IC3 an explicit representation of control flow automata is combined with symbolic reasoning. This approach combines the advantages of an explicit handling of the program's CFA, which represent the program as an automata model, and the relative inductive reasoning of its data space which is represented in a symbolic way. This variant has already been shown to be competitive [LNN15]. Nevertheless, there are still some parts which can be improved. This thesis focuses on the information reuse of already learnt information to improve the algorithm. Not each learnt information can be used in further steps to improve the algorithm. The learnt information has to be filtered to only save those which improve the algorithm in further steps. Most learnt information have to get transformed in some way to be usable and eventually useful in further steps, because in later steps the algorithm would compute highly related information but not an exact copy of the learnt information. The required transformation has to be more efficient than recomputing the information. In case the transformation requires more resources than recomputing the information the reuse does not yield any improvements. This thesis presents four approaches of information reuse for control-flow oriented IC3 algorithms.

After providing the introduction and a short overview about related work, this thesis starts by giving the necessary theoretical background for improving IC3CFA, by introducing a control flow automaton (CFA) and explaining IC3 and IC3CFA itself. Furthermore some other common model checking techniques are also shortly explained in section 2. In section 3 the theoretical ideas of reusing learnt information are described and the new approaches are proven to be correct. Section 4 provides details how the theoretical ideas are realized in an existing tool and which challenges occurred during implementation. The results of performance tests are topic of section 5 which are also compared to some other common model checking techniques. Finally section 6 concludes this thesis.

Related Work

Reusing information during verification of a program has been investigated before. [BW13] presents various approaches of reusing verification results. Those ideas are based on reuse information of previous verification runs. Conditional model checking generates a formula describing a subset of the program's states which satisfy a given condition. This output can be reused in later verification runs such that the already described states are not verified again. Another technique of information reuse is the so called precision reuse, where intermediate results from previous verification runs are used to accelerate further verification runs. These approaches reuse the information of an already executed verification run [BW13]. On the other hand [IG15] reuses information during one verification run. The approach of Ivrii and Gurfinkel is based on the original IC3 algorithm. They try to push a blocked cube to higher levels as far as possible. Their aim is to achieve a safe inductive invariant faster than without their aggressive pushing technique and therefore terminate sooner. The Tree-based IC3 approach makes use of information reuse, it reuses an already expanded part of the Abstract Reachability Tree in further computations to achieve time and memory savings. [EMB11] presents an algorithm, named Property Directed Reachability, which works in an analogous manner as IC3. This algorithm can be implemented in a simplified and faster variant than the original IC3 algorithm. The main reason for the improvements is caused by using interpolants, because interpolation yields a smaller and more general representation [McM03]. The approach of PDR is also used to perform model checking on quantifier free formulas over bitvectors (QF_BV) [WK13].

2 Background

This chapter provides the theoretical background of software model checking via IC3CFA [LNN15]. Given a program, we start by constructing a graph representation of the program, the control flow automaton. Afterwards the original IC3 [Bra11] algorithm is explained, to provide the general idea of IC3CFA which is explained in Sec. 2.3. Finally this section provides a high level description of bounded model checking and counterexample-guided abstraction refinement, as they are used as competitive algorithms in the evaluation.

2.1 Control Flow Automaton

Definition 1 (Cube, Clause and CubeSet)**.** *A cube is defined as conjunction of literals over the subset $X \subseteq Var$. A cube is characterized by its set of literals, i.e. $literals(a \wedge b) = \{a, b\}$. Each literal being a variable or its negation in the propositional case and a theory atom or its negation in case of quantifier-free first-order logic. A clause is defined as negation of a cube, and a cubeset as set of cubes.*

Definition 2 (Operations)**.** *The operations which are used as edge labels in a control flow automaton are defined by the language*

$$\begin{aligned} basic\_op &= assume\ b \mid var := exp \\ exp &= a \mid b \\ a &= var \mid a_1 + a_2 \mid a_1 - a_2 \mid\ ? \\ b &= var \mid b \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid a_1 \circ a_2 \ \textit{with} \circ \in \{=, <, >, \leq, \geq\} \\ op &= basic\_op \mid op_1; op_2 \mid op_1 \mid\mid op_2 \end{aligned}$$

Definition 3 (Weakest Existential Precondition [Lei05])**.** *The predicate transformer, describes whether there is at least one possible path s.t. Q will be* `true`.

P	$wep(P, Q)$
$x := e$	$Q[x \rightarrow e]$
$assume\ b$	$b \wedge Q$
$op_1; op_2$	$wep(op_1, wep(op_2, Q))$
$op_1 \mid\mid op_2$	$wep(op_1, Q) \vee wep(op_2, Q)$

Definition 4 (Control Flow Automaton)**.** *A Control Flow Automaton (CFA) $\mathcal{A}$ is defined as the tuple $\mathcal{A} = (L, G)$, where L is a finite set of locations and G is a set $L \times op \times L$ [GCS11].*

The set L models the program counter l and G represents the edges in the CFA. Comparing the definition of the guarded command language provided in [JB10] with our definition of an operation, one sees that they are highly related to each other and can interpret our definition as a variant of the guarded command language. Following Def. 2 there is either a basic operation, a choice or sequential operation at the highest level. In case the operation is a choice than op_1 or op_2 have to be executed, in case of a sequential operation first op_1 is executed and afterwards op_2. A basic operation can either be an assume or an assignment. First, an assume evaluates the Boolean expression to `true` or `false`. Second, an assignment sets a variable to some expression, which might be a non-deterministic value. In case of a sequential operation which contains an assume followed by some more operations, the following operations are only executed if the assume was evaluated to `true`. The assumes are therefore also called guards for operations.

Example 1 (Semantics of assume). *Let $op = (assume(b); x := 1)$. The variable x is only assigned to 1 if the expression b is evaluated to* `true`. *In case b is evaluated to* `false` *the assignment is not executed.*

To be able to check more restrictive, the programs can also contain assert statements. An assert statement does also contain a Boolean expression which is evaluated to `true` or `false`. In case an assert statement is evaluated to `false` the program is unsafe. For each assertion that has to be verified on an edge, a new edge with the negated guard of the assertion is created which ends in location l_E. This way, one is able to check the violation of an assertion by checking the reachability of l_E in $\mathcal{A}$. After transforming the given program into a CFA, the Large-Block-Encoding [BCG+09] is applied. This way one reduces the amount of locations and edges in a CFA. Due to the application of Large-Block-Encoding one is able to rewrite the assert statements with assumes, therefore the assert statement is omitted in the definition of operations. The set of all used variables in a CFA $\mathcal{A}$ is denoted by Var.

Definition 5 (Functions on a Control Flow Automaton). *To support subsequent definition, we define the following functions on states:*

$$\begin{aligned}
&\textit{Let } l \in L\\
succ(l) &:= \{l_s | (l, op_x, l_s) \in G \wedge op_x \in Ops \wedge l_s \in L\}\\
pred(l) &:= \{l_p | (l_p, op_y, l) \in G \wedge op_y \in Ops \wedge l_p \in L\}\\
succ\_edges(l) &:= \{(l, op_x, l_s) \in G \wedge op_x \in Ops \wedge l_s \in L\}\\
pred\_edges(l) &:= \{(l_p, op_y, l) \in G \wedge op_y \in Ops \wedge l_p \in L\}
\end{aligned}$$

Definition 6 (Special states). *The initial location l_0 of an CFA is characterized by*

$$pred(l_0) = \emptyset, |succ(l_0)| > 0$$

and an error location $l_E \in L$ is characterized by

$$succ(l_E) = \emptyset, |pred(l_E)| > 0$$

Definition 7 (Program). *A sequential program $\mathcal{P}$ is defined as three tuple $\mathcal{P} = (\mathcal{A}, l_0, l_E)$ where $\mathcal{A}$ is the corresponding CFA, l_0 a unique initial location and $l_E \subseteq L$ represents the error locations [BCG+09].*

In the following, the initial location l_0 shown in figures is marked by a green fill, error locations l_E are marked by a red fill, unreachable states are omitted.

Definition 8 (Result of transition functions). *The resulting variable evaluation after taking a CFA edge $(l, op, l_)$ is defined as*

$$\begin{aligned} var' = & \bigwedge(x' = x \mid x \in \{Var \setminus assigned(op)\}) \\ & \wedge (\bigwedge(x' = exp \mid (x := exp) \in op)) \\ \textit{with } assigned(op) = & \{x \mid (x := exp) \in op\} \end{aligned}$$

The tuple (l, c) is a concrete data state where l represents the program counter and c is a function $c : Var \to \mathbb{Z}$ that assigns an integer value to every variable, where $\mathbb{Z}$ is the domain of integer values. All concrete data states in the set are denoted by C. A region is a subset of C and is represented by a first-order formula φ. This formula φ defines a set S of all concrete data states c that models φ.
The strongest postcondition operator SP_{op} defines the concrete semantics of an operation $op \in Ops$. SP_{op} indicates the set consisting of all states that are reachable from the region, given by the formula φ, after executing an operation op. For a given formula φ and an assignment operation $s := e$ we have $SP_{s:=e}(\varphi) = \exists s' : \varphi_{\{s \longmapsto s'\}} \wedge (s = e_{\{s \longmapsto s'\}})$ as well as for an assume operation $assume(p)$ we have $SP_{assume(o)}(\varphi) = \varphi \wedge p$. The notation $s \longmapsto s'$ says that s is replaced by s' [BCG+09].

A path σ is a sequence $\langle(op_0, l_1), ..., (op_{n-1}, l_n)\rangle$ of operations $(\forall i \in \{0, n-1\}.op_i \in op)$ and locations $(l_i \in L\ \forall i \in \{1, n\})$. If G contains edges (l_{i-1}, op_{i-1}, l_i) s.t. there is a path from l_0 to a terminating location $l_t \in L$ σ is a program path. The successive application of the strongest postoperator for path $SP_\sigma(\varphi) = SP_{op_{n-1}}(...SP_{op_0}(\varphi)...)$ defines the concrete semantics for a program path $\sigma = \langle(op_0, l_1), ..., (op_{n-1}, l_n)\rangle$. If $SP_\sigma(true)$ is satisfied the program path σ is feasible. A concrete state (l_i, c_i) of a path σ is reachable if the path σ is feasible and ends in the location l_i such that $c_i \models SP_\sigma(true)$. An arbitrary location l can be reached if there is a reachable concrete state (l, c). In case the location l_E is not reachable the program is safe [BCG+09].

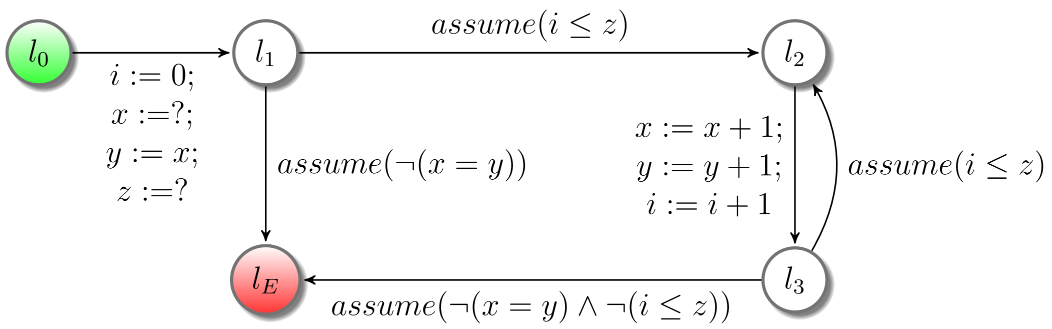
Definition 9 (Counterexample trace). *A counterexample trace is defined as sequence $c_0, \ldots, c_n$ of data regions with $d_i = (l_i, c_{l_i}) \forall i < n$ s.t.*

$$\begin{aligned} d_0 &= (l_0, c_{l_0}) \\ d_n &= (l_E, c_{l_E}) \\ \langle \ldots, d_i, d_{i+1}, \ldots \rangle &\Rightarrow (l_i, op_i, l_{i+1}) \in G \end{aligned}$$

Listing (2.1) Simple add test

```
int x, y, z;
int main(){
    x = ?;
    y = x;
    z = ?;
    for(int i = 0; i <= z; i++) {
        x++;
        y++;
    }
    assert(x == y);
    return 0;
}
```

(a) Sample code


l_0
l_1
l_2
l_3
l_E
$i := 0;$ $x :=?;$ $y := x;$ $z :=?$
$assume(i \leq z)$
$assume(\neg(x = y))$
$x := x + 1;$ $y := y + 1;$ $i := i + 1$
$assume(i \leq z)$
$assume(\neg(x = y) \wedge \neg(i \leq z))$

(b) Control Flow Automaton

Figure 2.1: Sample program

Sample program As mentioned in the introduction of this chapter, programs are transformed into CFAs. The transformation will be performed on the following example code (Lst. 2.1). The code checks if variables x and y have the same value after incrementing the variables a and y by one in each iteration of the loop. The amount of loop executions is given by a non-deterministic value which is assigned to variable z. The first step of transforming code into a CFA is creating an initial location. Second, the program lines are read one by one. For each assignment a new edge to the next location is created. For if statements the CFA is divided into branches which are merged at the end of the if statement. Loops are also represented by branches in the CFA, where one branch ends in the start location of the branch and the other branch proceeds to the next location in the program. In this example this can be seen at location l_2. The resulting CFA after applying Large-Block-Encoding contains five locations and 6 edges, see Fig. 2.1b.

2.2 IC3

The algorithm decides for a given transition system S whether it satisfies the safety property P in all reachable states, i.e. P is *S-invariant* or not. The presented algorithm was originally constructed to perform Boolean hardware model checking. The algorithm's strategy is similar to human's strategy when solving non-trivial problems. IC3 produces lemmas, which are represented as clauses, that are relative inductive to previous lemmas and stepwise assumptions. A subset of generated lemmas comprises a one-step inductive strengthening of P. During the execution of IC3 thousands of SAT queries are executed, for which a solver tries to find a model which satisfies the query, or to compute a counterexample. In case a counterexample exists the checked formula is unsatisfiable.

Definition 10 (Transition system). *A transition system (TS) S consists of two propositonal logic formulas [Bra11]*

1. *$I(\bar{x})$ representing the initial condition of S,*
2. *$T(\bar{x}, \bar{i}, \bar{x}')$ representing the transition relation over a set of variables $\bar{i}$, internal state variables $\bar{x}$ and the next state internal state variables $\bar{x}'$ [CGP99].*

Definition 11 (Invariant [Bra11]). *A safety property $P(\bar{x})$ asserts that only P-states are reachable. P is invariant for S iff only $P-$ states are reachable. If P is not invariant, there exists a counterexample trace $s_0, s_1, \cdots, s_k$ s.t. $s_k \not\models P$.*

To prove or disprove P being S-invariant, the algorithm incrementally refines and extends the sequence of formulas which represents an overapproximation of reachable states in at most i steps. The sequence is of the form $F_0 = I, F_1, F_2, \cdots, F_k$.

The algorithm can be divided into an outer and an inner loop, which are also known as propagation and blocking phase. The outer loop iterates over the steps k, searching for states in F_k that can reach a state $\neg P$ in one step, a so called counterexample to induction (CTI). In case there was no CTI discovered, IC3 tries to propagate the learnt clauses forward to F_{k+1}. The purpose of the blocking phase is deciding whether the CTI is reachable from I. Therefore F_{k+1} is extended by new clause. The extension of F_i, for refining the over approximation for i-step reachability, always satisfies the equations of Def. 12.

Definition 12 (IC3 invariants [Bra11]).

$$\begin{aligned} I &\Rightarrow F_0 \\ F_i &\Rightarrow F_{i+1} \quad \textit{, for } 0 \leq i < k \\ F_i &\Rightarrow P \qquad \textit{, for } 0 \leq i \leq k \\ F_i \wedge T &\Rightarrow F'_{i+1} \quad \textit{, for } 0 \leq i < k \end{aligned}$$

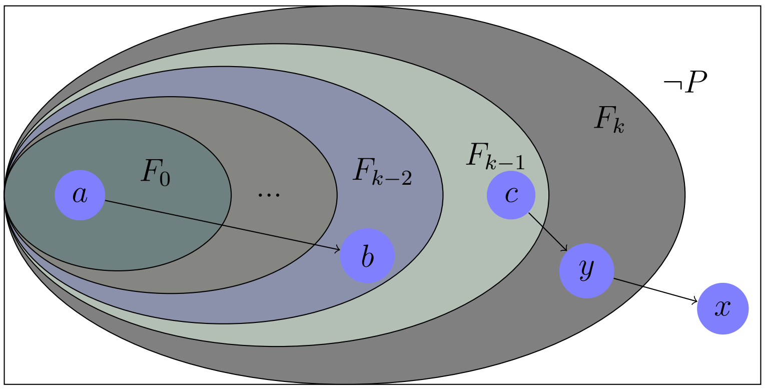


Figure 2.2: Graphical representation of invariants

Definition 13 (Relative inductiveness [Bra11]). *Let T_S be a transition relation, formula φ is inductive relative to another formula ψ if*

$$\varphi \wedge T_S \wedge \psi \Rightarrow \varphi'$$

is valid.

Figure 2.3 gives a graphical idea of relative inductiveness. As shown in the left rectangle the point m is satisfying the formula $\phi \wedge \psi$. After applying the transition relation T_S the resulting node m is still satisfying the formula ϕ, but not satisfying ψ anymore.

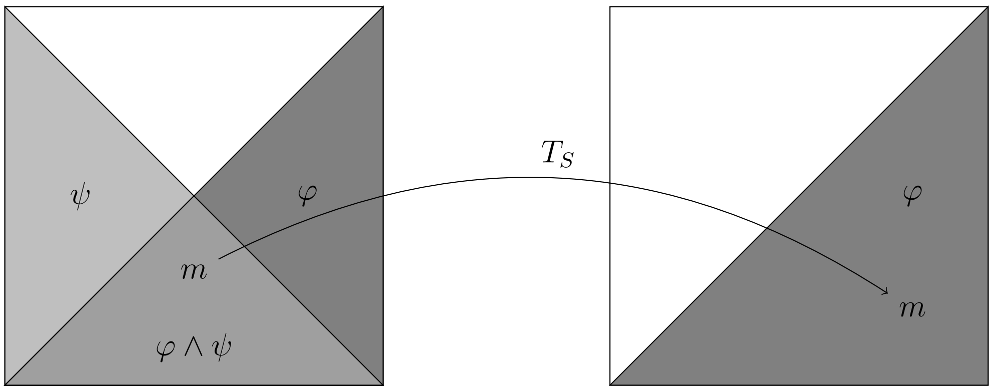


Figure 2.3: Inductive relative

The algorithm Alg. 1 starts with the initial checks. First the existence of a 0-step counterexample is checked by the satisfiability query $I \wedge \neg P$, if the result is `false` the query $I \wedge T \wedge \neg P$ is solved to check the existence of a 1-step counterexample. If both queries are not satisfiable the sequence of formulas is initialized. $F_0 = I$ and all other F_i are initialized to assume that P is indeed an invariant for S, i.e. $F_i = P$ for $i > 0$. The sets of clauses are initialized to empty. Each F_i can be interpreted as formula of the form $P \wedge \bigwedge clauses(F_i)$. Each major iteration starts with calling $strengthen(k)$, afterwards $propagateClauses$ propagates the clauses forward to $F_1, F_2, \cdots, F_{k+1}$.

Algorithm 1 The main function [Bra11]

```
function PROOF
    if $sat(I \wedge \neg P)$ or $sat(I \wedge T \wedge \neg P)$ then
        return false
    else
        $F_0 := I$
        $clauses(F_0) := \emptyset$
        for all $i > 0$ do
            $F_i := P$
            $clauses(F_i) := \emptyset$
        end for
        for $k := 1$ to ... do
            if not strengthen(k): then
                return false
            propagateClauses(k)
            if $clauses(F_i) = clauses(F_{i+1})$ for some $1 \leq i \leq k$ then
                return true
        end for
end function
```

The *strengthen* function (Alg. 2) iterates until the formula F_k excludes all states that can reach a P violating state in a single step. Assume s is a state which is one transition away from a state which does not satisfy the property P. This state is eliminated within two steps, first it is inductively generalized to some F_i by calling $inductivelyGeneralize(s, k-2, k)$ (Alg. 4). This functions produces the inductive generalization to some F_i. In case $min < 1$, s can possibly have an initial state as predecessor. Second pushing for a generalization at level k by calling $pushGeneralization(\{(n+1, 2)\}, k)$. After this call it is ensured that F_k excludes the state s.

The *propagateClauses* function (Alg. 3) checks for each $c \in F_i$ if the query $F_i \wedge T \wedge c'$ is satisfiable. If so, c is not added to F_{i+1}, in case of unsatisfiability c is

Algorithm 2 The strengthen function [Bra11]

```
function STRENGTHEN(k:level)
    try:
        while $sat(F_k \wedge T \wedge P')$ do
            s := the predecessors extracted from the witness
            n := inductivelyGeneralize(s,k-2,k)
            pushGeneralization ({(n + 1, s)} , k )
        end while
        return true
    exceptCounterexample
    return false
end function
```

Algorithm 4 Stepwise-relative inductive generalization [Bra11]

```
1: function inductivelyGeneralize(s : state, min : level, k : level))
2:     if $min < 0$ and $sat(F_0 \land T \land \neg s \land s')$ then
3:         raise Counterexample
4:     for $i := max(1, min + 1)$ to $k$ do
5:         if $sat(F_i \land T \land \neg s \land s')$ then
6:             generateClause(s,i-1,k)
7:             return i-1
8:     end for
9:     generateClause(s,k,k)
10:    return k
11: end function
12:
13: procedure generateClause(s : state, i : level, k : level)
14:     $c :=$ subclause of $\neq s$ that is inductive to $F_i$
15:     for $j := 1$ to $i + 1$ do
16:         $clauses(F_j) := clauses(F_j) \cup \{c\}$
17:     end for
18: end procedure
```

added to F_{i+1}. F_i is an inductive strengthening of P, proving P's invariance, if for any i $F_i = F_{i+1}$ holds. In general this function tries to extend the trace with a new formulas, i.e. each formula's clause is check whether it holds in the next step formula. Through this approach one tries to yield a more inductive state, which might satisfy the termination criterion $F_i = F_{i+1}$ for some i after *propagateClauses*.

Algorithm 3 The propagateClauses function [Bra11]

```
1: procedure propagateClauses(k:level)
2:     for $i := 1$ to $k$ do
3:         for each $c \in clauses(F_i)$ do
4:             if not $sat(F_i \land T \land c')$ then
5:                 $clauses(F_{i+1}) := clauses(F_{i+1}) \cup c$
6:         end for
7:     end for
8: end procedure
```

The main part of pushing inductive generalization to higher levels is handled in the *pushGeneralization* function (Alg. 5). Inside this function it is checked whether the state s is inductive relative to F_i. If so the inductive generalization is applied to its F_i-state predecessors. The most complicated part of this function is to define a termination criterion even if there are cyclic predecessors, i.e. $l \in \bigcup_{l_p \in pre(l)} pre(l_p)$. The termination is guaranteed by the fact that a set of pairs (i, s) is maintained s.t. each pair $(i, s) \in states$ represents the knowledge that s is inductive to F_{i-1} and

Algorithm 5 The pushGeneralization function [Bra11]

```
procedure PUSHGENERALIZATION(states : (level, state) set, k : level)
    while true do
        $(n, s)$ := choose from state, minimizing n
        if $n > k$ then
            return
        if $sat(F_n \wedge T \wedge s')$ then
            p:=the predecessor extracted from the witness
            m:= inductivelyGeneralize (p,n-2,k)
            $states := states \cup \{(m+1, p)\}$
        else
            m:= inductivelyGeneralize (s,n,k)
            $states := states \setminus \{(n, s)\} \cup \{(m+1, s)\}$
    end while
end procedure
```

that F_i excludes s. The loop always selects the pair (n, s) such that n is minimal w.r.t to the available pairs. There cannot be a state in *states* that is a predecessor of s at level n.

Definition 14 (Delta Encoding [Sud13], [EMB11]). *To avoid duplicated cubes in the union of all frames for one location one uses delta encoding. The delta frame $F_{\Delta(i,l)}$ contains only cubes appearing last in $F_{(i,l)}$.*

$$F_{\Delta(i,l)} := F_{i,l} \setminus F_{i+1,l}$$

2.3 IC3CFA

This section provides a detailed overview of the IC3 algorithm adopted to the control flow representation of a program. A naive way applying IC3 to software model checking is to use an additional variable *pc* to encode the control flow. The variable *pc* represents the program location. In [Bra11] the author describes the similarity how IC3 and humans analyse a system. Both are producing a set of lemmas s.t. each of them holds relative to a previous one. By following this strategy all lemmas imply the whole property. The IC3CFA approach follows this idea as well. A given control flow automaton already represents the possible execution steps in the program.

Definition 15 (Data region [LNN15]). *A data region is represented by a quantifier-free first-order formula s over Var and consists of all variable assignments σ satisfying s, i.e., $\{\sigma \mid \sigma \models s\}$.*

Definition 16 (Region [LNN15]). *A region $r = (l, s)$ is defined as a pair consisting of location $l \in L$ and a data region s. Given such a region $r = (l, s)$, the corresponding formulas are given by the set $\{\phi \mid \phi \equiv (pc = l \wedge s)\}$. The formulas for the negated region are defined as $\{\phi \mid \phi \equiv \neg(pc = l \wedge s)\}$.*

Algorithm 6 Outer loop [LNN15]

```
function PROVE
    if $l_0 = l_E$ or $((l_0, op, l_E) \in G$ and $sat(T_{l_0 \to l_E}))$ then
        return false
    initialize frames
    for $k = 1$ to ... do
        if not $STRENGTHEN(k)$ then
            return false
        propagate
        if termination then
            return true
    end for
end function
```

The possible transitions for a region $r = (l, s)$ can be reduced to the set of available ones from l in the program $\mathcal{P}$. The reduction is a result of the explicit representation of edges. Due to the reduced transition set one creates smaller queries for a solver, which avoids high solver times or solver timeouts in worst case. The static check for existing 0-step or 1-step counterexample reduces the size of solver queries as well. Additionally, these checks avoid initial and error conditions. Bradley uses global frames to construct a sequence of frames $F_0, \cdots, F_k$ [Bra11]. In the setting using a CFA for program representation one uses location-local frames $F_{(i,l)}$ for each $l \in (L \setminus l_E)$. Each of these frames is interpreted as set of data regions which are reachable in at most i steps at location l.

Definition 17 (Transition formula [LNN15])**.** *The transition formula between two locations l_1 and l_2 is defined as:*

$$T_{l_1 \to l_2} \begin{cases} (pc = l_1) \wedge op \wedge (pc' = l_2) & , if (l_1, op, l_2) \in G \\ \texttt{false} & , otherwise \end{cases}$$

The outer loop of IC3CFA (Alg. 6) is quite similar to the original function *PROOF* (see Alg. 1). It starts with the initial checks to find 0-step or 1-step counterexamples. The corresponding reachability queries are $I \wedge \neg P$ for 0-step and $I \wedge T_{l_0 \to l_E} \wedge \neg P'$ for a 1-step counterexample if there exists an edge $(l_0, t, l_E) \in G$. If those two initial checks are passed, i.e. there is no 0-step or 1-step counterexample, the frames for $i = 0$ and $i = 1$ are initialized. The frame $F_{(0,l_0)}$ is set to `true` and the frames $F_{(0,l)}$ are set to `false` for $l \in (L \setminus l_0)$ because these locations are not initial locations and are therefore not reachable from l_0 in zero steps, the frames $F_{i,l}$ with $i > 0$ are initialized with `true` $\forall l \in (L \setminus l_E)$. After initializing the frames, the algorithm starts the main loop with frame limit k. Goal of the main loop is to strengthen the new frame set. In case the blocking phase succeeds the propagation phase tries to push the new information forward. The termination check is performed at the end of the inner loop. The original criterion $F_i = F_{i+1}$ has to be adapted to the new setting of location-local frames. Therefore the algorithm terminates if $F_{(i,l)} = F_{(i+1,l)}$ holds for some i and all $l \in (L \setminus l_E)$.

Algorithm 7 Strengthening [LNN15]

```
1: function STRENGTHEN($k$: int)
2:     for $l : L$ do
3:         if $sat(F_{(k,l)} \wedge T_{l \to l_E})$ then
4:             $s :=$ predecessor data region
5:             Q.add($k, l, s$)
6:     end for
7:     if not $BACKWARDBLOCK(Q)$ then
8:         return false
9:     return true
10: end function
```

The *Strengthen* function (see. Alg. 7) works also similar to the original function in IC3. The most important difference is the guard of the while loop. In this setting it has to be adapted to the query if there exists a $l \in L$ s.t. $e = (l, t, l_E) \in G$ and $T_{l \to l_E}$ is satisfiable under $F_{(k,l)}$. In case there is an edge that satisfies the query there exists a so called counterexample to induction (CTI). Using weakest preconditions (WP) is a safe but not best performing overapproximation of CTI states. The results of the WP computation is analysed in Alg. 8.

Definition 18 (Egde-realtive inductiveness [LNN15])**.** *Given a CFA A and locations $l_1, l_2 \in l$ and a formula ϕ is edge-relative inductive to another formula ψ if*

$$\psi \wedge \phi \wedge T_{l_1 \to l_2} \Rightarrow \phi'$$

The inner loop mainly follows the idea presented in [EMB11]. Their approach introduces an explicit priority queue which handles the different obligations, which is called ObligationQueue. They are ordered in an ascending order, i.e. the proof obligation with minimal index is at the top. The parameters of the *backwardblock* function are in detail the frame index i, a location $l \in L$ and a data region s. As mentioned in [EMB11] the initial proof obligations are added to the ObligationQueue Q. The obligation with the lowest index will be popped first. In case the index of the obligation is equal to 0 one can immediately stop and provide a counterexample, because l has to be an initial location. In case l not being an initial location ($l \neq l_0$), the previous proof obligation would have included the frame $F_{(0,l)}$, which is `false` for every $l \in L \setminus \{l_0\}$. In case $i \neq 0$ it has to be checked whether the region $\neg(l, s)$ is edge-relative to $F_{(i-1,l_p)}$ $\forall l_p \in pre(l)$ by solving on of the queries depending on whether $l_p = l$ or not.

$$r_1 = (l_p, \hat{s}) = (l_1, s_1), \neg r_2 = \neg(l, s) = \neg(l_2, s_2)$$
$$s_1 \wedge T_{l_1 \to l_2} \Rightarrow \neg s_2', \text{if } l_2 \neq l_1$$
$$s_1 \wedge \neg s_2 \wedge T_{l_1 \to l_2} \Rightarrow \neg s_2', \text{if } l_2 = l_1$$

Definition 19 (Generalization)**.** *Let c be a cube which is relative inductive to $F_{(i,l)}$,*

Algorithm 8 Inner loop [LNN15]

```
function BACKWARDBLOCK($Q$:ObligationQueue)
    while $| Q |> 0$ do
        $(i, l, s) = Q.pop$                                    ▷ $i$:int, $l$:location, $s$ :data region
        if $i = 0$ then
            return false
        else
            for each $l_p$, s.t. $(l_p, op, l) \in F$ do
                if $l_p = l$ and $sat(F_{(i-1,l_p)} \wedge \neg s \wedge T_{l_p \to l} \wedge s')$ then
                    generate predecessor $c$ of $s$
                    add $(i-1, l_p, c)$ and $(i, l, s)$ to $Q$
                else if $l_p \neq l$ and $sat(F_{(i-1,l_p)} \wedge T_{l_p \to l} \wedge s')$ then
                    generate predecessor $c$ of $s$
                    add $(i-1, l_p, c)$ and $(i, l, s)$ to $Q$
                else
                    compute generalization $s_{gen}$ of $s$
                    block $s_{gen}$ in frames $F_{(j,l)}$ for $0 \leq j \leq i$
            end for
    end while
    return true
end function
```

the generalization of c is defined as minimal subset c_{gen}, s.t.:

$$literals(c_{gen}) \subseteq literals(c)$$
$$c \text{ relative inductive to } F_{(i,l)} \Rightarrow c_{gen} \text{ relative inductive to } F_{(i,l)}$$

In case the region $\neg(l, s)$ is indeed inductive edge-relative to all predecessors l_p to $F_{(i-1,l_p)}$, s_{gen} can be blocked in all frames $F_{(j,l)}, 0 \leq j \leq i$. In case the region is not inductive edge-relative then there must exists a predecessor of l_p which can reach (l, s). To expand this path, a new proof obligation is created. Therefore one computes the WP c of s w.r.t. the transition formula and adds the new obligation $(i-1, l_p, c)$ to the ObligationQueue. The old obligation (i, l, s) is also reinserted into the ObligationQueue for later inspection after the predecessors are checked. If the ObligationQueue is empty, the inner loop terminates and returns `true`. In case there exists an obligation at level $i = 0$, there is a counterexample and the inner loop returns `false`.

Sample execution

The next paragraph describes a sample execution of IC3CFA on the program which was already provided as example for the CFA transformation (see Fig. 2.1). At the end of the loop both variables are assumed to have the same value.

Static pre-checks At first the algorithm starts with the 0-step and 1-step counterexample checks. Both are not satisfied because $l_0 \neq l_E$ and $(l_0, op, l_E) \notin G$.

k=1 At the beginning the frames are initialized to $F_{(0,l_0)} = \texttt{true}$ and $F_{(0,l_x)} = \texttt{false}\ \forall l_x \in (L \setminus l_0)$ and $F_{(1,l)} = \texttt{true}\ \forall l \in L$. Now, the algorithm starts with $k = 1$ and searches of CTIs. There are two locations s.t. $(l, op, l_E) \in G$, namely l_1 and l_3, therefore the ObligationQueue $Q_{init(1)}$ contains two proof obligations (i, l_x, c) s.t. $l_x \in L$.

$$Q_{init(1)} = \{\underbrace{(1, l_1, (x \neq y))}_{q_1^1}; \underbrace{(1, l_3, (x \neq y \wedge z \leq i))}_{q_1^2}\}$$

The minimal entry q_1^1 is popped from the ObligationQueue and has $i \neq 0$, so we start by checking the inductivity. The resulting query for the inductiveness check $F_{(0,l_0)} \wedge (x' =? \wedge y' = x \wedge i' = 0 \wedge z' =?) \wedge x' \neq y'$, which is unsatisfiable, due to the cube $x' \neq y'$, therefore $x = y$ is added to $F_{(1,l_2)}$. Next, q_1^2 is inspected concerning inductivity, by checking the query $F_{(0,l_2)} \wedge (x' = x + 1 \wedge y' = y + 1 \wedge i' = i + 1) \wedge z \leq i' \wedge x' = y'$, which is unsatisfiable due to $F_{(0,l_2)} = \texttt{false}$. Therefore $\texttt{true}$ is blocked at $F_{(1,l_3)}$ which yields $F_{(1,l_3)} = \texttt{false}$.

l : / i :	l_0	l_1	l_2	l_3
0	`true`	`false`	`false`	
1		$x = y$		`false`

Figure 2.4: Delta frames after 1st iteration

k=2 The initial ObligationQueue is the same than in the iteration before, only the entries index differ. Each following iteration starts with an initial ObligationQueue of the form:

$$Q_{init(k)} = \{\underbrace{(k, l_1, (x \neq y))}_{q_k^1}; \underbrace{(k, l_3, (x \neq y \wedge z \leq i))}_{q_k^2}\}$$

Again the minimal proof obligation (q_2^1) is checked for inductivity. The query is $F_{(1,l_0)} \wedge (x' =? \wedge y' = x \wedge i' = 0 \wedge z' =?) \wedge x' \neq y'$ not satisfiable and therefore $F_{(2,l_1)} = (x = y)$. Now, the minimal obligation in Q is q_2^2 and its inducitivtiy query is $F_{(1,l_2)} \wedge (x' = x + 1 \wedge y' = y + 1 \wedge i' = i + 1) \wedge z \leq i' \wedge x' \neq y'$. In this case the SAT solver returns satisfiable and the ObligationQueue is extended by the predecessors of l_3, q_2^2 remains in the ObligationQueue to be checked after handling the predecessor obligations. The new obligation is $(1, l_2, (z \leq i + 1 \wedge x + 1 \neq y + 1))$. Next, this proof obligation is under inspection. The predecessor which does not satisfies the query is l_1. Therefore the generalization of $z \leq i + 1 \wedge x + 1 \neq y + 1$ is blocked at $F_{(1,l_2)}$, which yields $F_{(1,l_2)} = \texttt{false}$. Again q_2^2 is checked for inductivity. Due to the changes in $F_{(1,l_2)}$, the query is unsatisfiable now and causes changes in $F_{(2,l_3)} = \texttt{false}$.

i : \ l :	l_0	l_1	l_2	l_3
0	`true`	`false`		
1			`false`	
2		$x = y$		`false`

Figure 2.5: Delta frames after 2nd iteration

k=3 Again the initial ObligationQueue is $Q_{init(3)}$, the corresponding query for minimal obligation q_3^1, is unsatisfiable and therefore the frame changes to $F_{(3,l_1)} = (x = y)$. Next, the obligation q_3^2 is checked concerning inductivity. The resulting query is $F_{(2,l_2)} \wedge (x' = x + 1 \wedge y' = y + 1 \wedge i' = i + 1) \wedge z \leq i' \wedge x' \neq y'$, which is satisfiable (see iteration $k = 2$). Therefore $(2, l_2, (z \leq i+1 \wedge x+1 \neq y+1))$ is added to Q. Checking this obligation yields changes of $F_{(2,l_2)} = (x + 1 \neq y = 1)$. Now, q_3^2 is checked again and due to the changes in$F_{(2,l_2)}$ the query is unsatisfiable, therefore $F_{(3,l_3)} = (x = y)$.

i : \ l :	l_0	l_1	l_2	l_3
0	`true`	`false`		
1			`false`	
2			$x + 1 = y + 1$	`false`
3		$x = y$		$x = y$

Figure 2.6: Delta frames after 3rd iteration

k=4, k=5 Iteration k=4 and k=5 are nearly identical, only the obligations index is increased by 1 respectively by 2 compared to iteration $k = 3$. In both iterations the frames $F_{(i,l_x)} \mid i < k \wedge l_x \in \{l_1, l_2, l_3\}$ s.t. $F_{(i,l_x)} \neq$`false` are shifted to the next index $F_{(i,l_x)} \rightsquigarrow F_{(i+1,l_x)}$. The result of the SAT queries remains unchanged from those in iteration $k = 3$. At the end of the fifth iteration the termination criterion of the algorithm is satisfied. Thus the algorithm terminates after five iterations with the verification result `true`.

i : \ l :	l_0	l_1	l_2	l_3
0	`true`	`false`		
1			`false`	
2				`false`
3				
4			$x + 1 = y + 1$	`false`
5		$x = y$		$x = y$

Figure 2.7: Delta frames after 5th iteration

2.4 Bounded Model Checking

Bounded Model Checking (BMC) unrolls the program up to a given bound k, i.e. BMC only checks the first k steps in the program, thus b bounded. If there is a counterexample in this part of the program such that the property is violated, the bounded model checker finds this counterexample. If BMC does not find a counterexample within the first k steps of the program, one can not say that the program is safe or not. Due to the given bound, the BMC will not find a counterexample with a path length greater than k. By using BMC the returned counterexample is the minimal one that violates the property, if there exists a violation up to bound k. The main disadvantage of BMC is caused by the given bound. Therefore, BMC cannot proof the correctness of a program in general [BCC+03]. Another technique of checking if a program violates or satisfies a property is topic of the next section.

2.5 Counterexample-Guided Abstraction Refinement

The counterexample-guided abstraction refinement is one of the used ways to model check a program in this thesis. Counterexample-guided abstraction refinement checks if a program abstraction contains any counterexample to prove that at least one assertion is violated. Before checking the existence of a counterexample the abstraction of the program has to be constructed. The abstraction of a program $\mathcal{P}$ is called $\mathcal{P}'$. The abstraction function for a program $\mathcal{P}$ is a surjective function $\alpha : D \rightarrow D'$ where D is the set of all possible concrete states in $\mathcal{P}$ and D' the abstract domain of states in $\mathcal{P}'$. After constructing the abstraction, the algorithm checks the existence of a path to the state l_E, a so called counterexample for a given assertion. If there does not exist a counterexample, the program does not violate the assertion and can be called safe. In case that there is a counterexample, one distinguishes between real and spurious counterexamples. A real counterexample is feasible in the concrete program. Having a real counterexample, the program is called unsafe [CGJ+00].
Second, a counterexample can be spurious, i.e. the counterexample is not feasible in the concrete program. In this case the abstraction of the program has to be refined. During the refinement process the locations, which are responsible for creating the spurious counterexample, have to be found. The equivalence classes which are separated from the abstracted states have to be modified as well. After the abstraction is refined, the assertion is checked again. As long as the counterexample is spurious, one cannot say if the program is safe or unsafe. To decide if a program is safe or not, it has to be ensured, that the found counterexample is not a result of a coarse abstraction of the program. The process of refining a program might not terminate in all cases.
Figure 2.8 illustrates the behaviour of the counterexample-guided abstraction refinement algorithm.

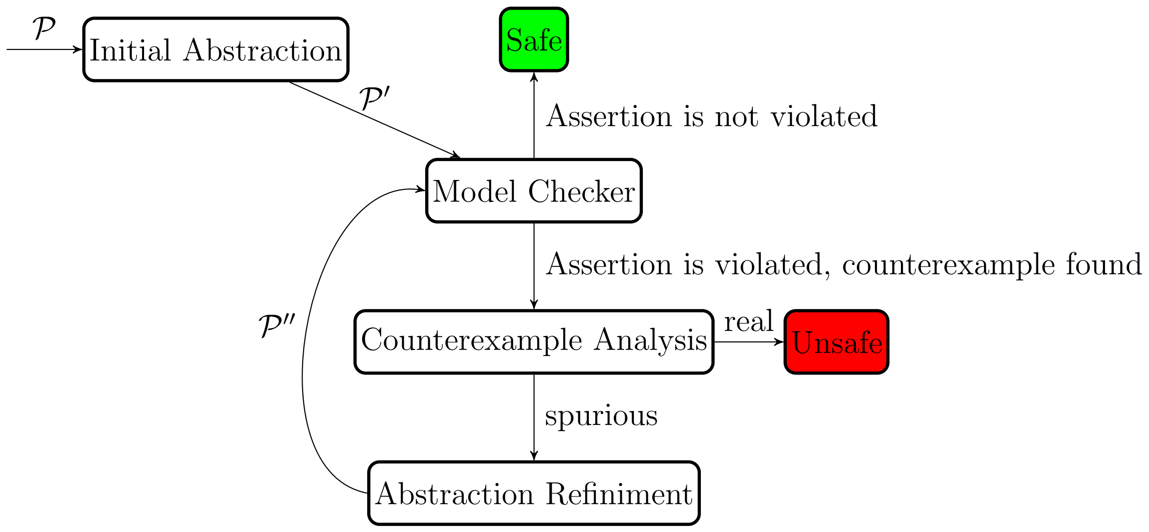


Figure 2.8: Counterexample Guided Abstraction Refinement

3 Information reuse

This section provides the theoretical ideas of information reuse when using IC3CFA. Basically it is divided into four sections. First the reuse of computed obligations is motivated and described. Second, the idea of skipping SAT queries of inductivity checks is presented. Afterwards a new termination criterion which is based on observations made during obligation reuse is shown and proven. Finally, an adaptation of the pushing technique of IC3 is described.

3.1 Obligation reuse

Due to the incremental approach of IC3CFA, each iteration starts with the CTIs and the steps are bounded by the index of the iteration k. Therefore, one recomputes obligations of earlier iterations. This offers potential for improvements of the algorithm, through reusing already learnt information. The complexity of each iteration grows exponentially due to branchings in the program.

IC3CFA starts each inner loop iteration with computing the CTIs and adding them to the ObligationQueue (see Alg. 2). These obligations are checked and eventually the predecessors of these CTIs are expanded for k steps. Therefore, the first idea was to create the CTI obligations only once in the first iteration and reuse them as initial ObligationQueue in each following iteration, w.r.t to the incremented index of each obligation.

Definition 20. *An obligation is represented as three tuple (i, l, c) where i indicates the level of the obligation, $l \in L$ defines the location and c is a cube describing a data region.*

Lemma 1. *The initial ObligationQueue for each inner loop is of the form:*

$$Q_{init(k)} = \{(k, l_p, wp(op, \texttt{true})) \mid (l_p, op, l_E) \in G\}$$

Proof 1 (Proof of lemma 1)**.** *Following Alg. 2 the CTI obligations at level k are computed at the beginning of each iteration. Executing IC3CFA does not change the program's CFA, therefore the predecessors of an error location do not change and the frame initialization is impartial of reusing obligations, which yields the same result of $sat(F_{(k,l)} \wedge T_{l_p \to l_E})$.*

Through our work we discovered the fact that not only the initial CTI obligations, even others obligation, e.g. the obligations for the CTI predecessors were recomputed in further iterations. Therefore, we extended our approach to reuse every obligation which has been computed once. To be able to use the learnt obligations of iteration i in the next iteration $i+$ the index of each obligation has to be incremented by one.

For the following lemma we define that the set of all computed obligations during an iteration i which is initialized with an ObligationQueue $Q_{init(i)}$ is denoted by $seen(Q_{init(i)})$.

Lemma 2. *Each $q \in Q_k$ would be computed in iteration k when starting the inner loop with $Q_{init(k)}$, i.e. $Q_k \subseteq seen(Q_{init(k)})$.*

Proof 2 (Proof of Lem. 2)**.** *Proof by induction:*

$$\begin{aligned}
k = 1 :& \textit{as shown in Lem. 1 the initial ObligationQueue contains only CTI obligation}\\
&\Rightarrow \forall q \in Q_1, \exists \hat{q} \in seen(Q_{init(1)}) \text{ s.t. } q = \hat{q}\\
&\Rightarrow Q_1 \subseteq seen(Q_{init(1)})\\
k \rightsquigarrow k+1 :& q \in Q_k \textit{ with } q = (i, l_p, c)\\
&\Rightarrow \exists \bar{q} \in Q_k \textit{ with } \bar{q} = (i+1, l, \bar{c}) \text{ s.t. } l_p \in pre(l)\\
&\Rightarrow F_{(i+1,l_p)} \wedge T_{l_p \rightarrow l} \wedge \neg \bar{c}' \textit{ was satisfiable}\\
&\bar{q} \in seen(Q_{init(k)}) \textit{ by induction base, let } l_p \in pre(l)\\
&\Rightarrow F_{(i+1,l_p)} \wedge T_{l_p \rightarrow l} \wedge \neg \bar{c}' \textit{ is satisfied due to same frame initializazion}\\
&\Rightarrow q \textit{ is added to } seen(Q_{init(k)})\\
&\Rightarrow Q_k \subseteq seen(Q_{init(k)})
\end{aligned}$$

Lemma 2 ensures that only those obligations are in the ObligationQueue which are computed when running the algorithm without reuse of obligations. Therefore, the initial ObligationQueue for each iteration with $k > 1$ contains more than only the CTI obligations. By this approach one reduces the effort for exploring a path up to a given bound, which is only increment by one, in comparison to the previous iteration. The predecessor computations are also reduced, because they are already in the ObligationQueue. The effort for computing the whole ObligationQueue each iteration top-down might grow up to exponentially on complex programs due to branchings. As consequence of Lem. 2 we can formulate the following theorem of the monotonic ObligationQueue size.

Theorem 1 (Monotonic ObligationQueue size)**.** *The ObligationQueue after an iteration k contains at least all elements of the ObligationQueue before executing the iteration, i.e. $Q_k \subseteq Q_{k+1}$.*

Using the obligation reuse changes the order of SAT queries compared to the standard ObligationQueue initialization. When reusing obligations one starts at the bottom of paths, which were explored in previous iterations. The standard algorithm always starts at the predecessors of error locations and explores the path step by step heading the initial location for to k steps. Therefore, one recursively follows in general, if no new obligation is added, the transition direction heading the error locations. In detail that means, when applying reuse let the minimal element be given by $q_k^{min_1} = (i, l_x, c_x)$ than one reaches an element $q_k^{min_2} = (j, l_y, c_x)$

which is minimal after eventually recursively exploring the predecessors, except a counterexample can be extracted, s.t. $j \geq i$.

After executing an iteration which does not extract a counterexample the resulting ObligationQueue of reused obligations has to be modified, to use it as initial ObligationQueue in the next iteration. The level index of each obligation has to be incremented by one.

Reusing obligations does not affect any parts of the termination criteria, because it only reduces the computations of obligations which were already computed in previous iterations. Therefore, the termination, first having a minimal element of the ObligationQueue with $i = 0$ or second having empty delta frames for all $l \in L \backslash L_E$ for some i will remain unchanged.

Example 2. *To demonstrate the most significant changes during execution of the algorithm we explain the introductory example in Fig. 2.1 again, but now with obligation reuse. We will omit the explanation of unchanged parts. The static pre-checks, frame and the first ObligationQueue initialization remain unchanged. The first change occurs in iteration $k = 1$, when the minimal element of the ObligationQueue is popped and checked for inductivity. When blocking a cube, the original obligation, in this case the old minimal element, is re-added to the ObligationQueue as new copy of the original obligation which is marked as reused now. Afterwards the last unmarked obligation is inspected. After querying the SAT solver, the generalization of its cube is blocked and the obligation is also re-added to the ObligationQueue as reused obligation. At the end of the iteration the ObligationQueue is cleaned-up and prepared for the next iteration. Thus iteration $k = 2$ starts with a similar ObligationQueue in which each obligation is only modified at its level index, which is increment by one. In this iteration one of the initial obligation returns satisfiable after querying the SAT solver, so the predecessor obligations are added to the ObligationQueue as well as the original obligation. Both are not marked as reused because nothing was blocked yet. Next, the new obligation is check for inductivity and returns unsatisfiable, therefore blocking takes place and the obligation is re-added to the ObligationQueue with a reuse marking. The two following obligations cause also a cube blocking and afterwards they are re-added to the ObligationQueue. Iteration $k = 3$ is the first iteration in which the initial ObligationQueue differs compared to the execution in Sec. 2.3. In this iteration the initial ObligationQueue contains three obligations, two CTI obligations and one predecessor obligation. Now, the minimal element is checked and returns unsatisfiable, same happens to the both CTI obligations. Compared to the original algorithm this iteration does not need any predecessor computation and all SAT-queries return unsatisfiable. Iteration $k = 4$ and $k = 5$ just differ in an increment level index. The frames remain unchanged compared to the execution in Sec. 2.3.*

3.2 Skipping

Solving a SAT-query comes along with high time and memory requirements. The solving time increases immediately when having choice operators in the query.

Therefore, nearly each avoided SAT-query yields time and memory savings. The handing of obligations is done in the inner loop of the algorithm (see. Alg. 8), while each obligation causes at least one SAT query. In case the first SAT-query returns satisfiable the predecessors of the location are taken into account. These obligations are added to the ObligationQueue as well as the original obligation, s.t. it will be checked after the predecessor are handled. These obligations will be handled in the following iterations and will also cause SAT-queries. During our research we found an indicator to reduce the number of SAT-queries in the inner loop.

An obligation which has been seen in the previous iteration can be directly passed to the blocking phase if in iteration $i-2$ the frames of all predecessors of the obligation's location remained unchanged. Thus the query of this obligations yields the same result as in iteration $i-1$. Therefore we can directly block the cubes of $F_{\Delta(i-1,l)}$ without querying the SAT-solver.

Definition 21. *An obligation is called old obligation if has been seen in the previous iteration.*

Each obligation in $seen(Q_{init(k)})$ yields exactly one unsatisfiable query, because only in this case the obligation is removed from the ObligationQueue, which has to be empty at the end of the inner loop by definition [LNN15].

Lemma 3 (Blocking without SAT-query)**.** *For any $q^j \in Q_k$ with $q^j = (i,l,c)$, s.t. $(i-1,l,c) \in seen(Q_{init(k-1)})$, the generalization of c can be block at $F_{(i,l)}$ if $\forall l_p \in pre(l)$: $F_{\Delta(i-2,l_p)} = \emptyset$ without causing a SAT-query.*

Proof 3 (Proof of lemma 3)**.** *Let $q_j \in Q_k$ with $q_j = (i,l,c)$ s.t. $(i-1,l,c) \in seen(Q_{init(k-1)})$ and $\forall l_p \in pre(l)$: $F_{\Delta(i-2,l_p)} = \emptyset = true$*

$$\Rightarrow \bigwedge_{l_p \in pre(l)} (F_{(i-2,l_p)} \wedge T_{l_p \to l} \wedge c') \textit{ is unsatisfiable because } (i-1,l,c) \in seen(Q_{init(k-1)})$$

$$\Rightarrow \bigwedge_{l_p \in pre(l)} (F_{(i-2,l_p)} \wedge T_{l_p \to l} \wedge c') \overset{Def.14}{=} \bigwedge_{l_p \in pre(l)} ((F_{(i-1,l_p)} \wedge F_{\Delta(i-2,l_p)}) \wedge T_{l_p \to l} \wedge c')$$

$$\overset{F_{\Delta(i-2,l_p)}=\emptyset}{=} \bigwedge_{l_p \in pre(l)} ((F_{(i-1,l_p)} \wedge true) \wedge T_{l_p \to l} \wedge c') = \bigwedge_{l_p \in pre(l)} (F_{(i-1,l_p)} \wedge T_{l_p \to l} \wedge c')$$

$$\Rightarrow \bigwedge_{l_p \in pre(l)} (F_{(i-2,l_p)} \wedge T_{l_p \to l} \wedge c') \overset{sat}{\equiv} \bigwedge_{l_p \in pre(l)} (F_{(i-1,l_p)} \wedge T_{l_p \to l} \wedge c')$$

$$\Rightarrow \bigwedge_{l_p \in pre(l)} (F_{(i-1,l_p)} \wedge T_{l_p \to l} \wedge c') \textit{ is also unsatisfiable}$$

The observation of blocking without SAT-query can be used at two positions in the algorithm, first before starting the inner loop and second during the inner loop when checking the predecessors of an obligations locations. The details will be provided in Sec. 4.3.

3.3 Termination

Through analysing the behaviour of IC3CFA with reusing obligations we made the observation, that the ObligationQueue is lifted at some moments during executing the algorithm. A lifted ObligationQueue is caused by the minimal element in the previous iterations. This obligation caused the query $F_{(i-1,l_p)} \wedge T_{l_p \to l} \wedge c'$ which was unsatisfiable, because otherwise the predecessors were taken into account. We were able to reduce it to the fact that the subset $T_{l_p \to l} \wedge c$ of the original is responsible for the unsatisfiability of the query. Therefore the approach of a new termination criterion is built on top of reusing obligations.

Definition 22 (Lifted ObligationQueue)**.** *Let Q_k be an ObligationQueue at the beginning of iteration k with minimal element $q^k_{min} = (i, l, c)$. Q_k is lifted if $i > 2$.*

The effect of a lifted ObligationQueues occurs several times during execution, but it does not hold from a specific point until the end of the algorithm in general. That means, that we might have a lifted ObligationQueue in iteration k for the next n following iterations, however after $n + 1$ iterations the ObligationQueue is not lifted any more. The described behaviour of having a lifted ObligationQueue for some iterations is illustrated by the following figure.

minimal element	(a, l_x, c_x)	$(a+1, l_x, c_x)$	$\cdots$	$(a+k, l_x, c_x)$	$(2, l_x, c_x)$
iteration	k	$k+1$	$\cdots$	$k+n$	$k+n+1$
remark	$a > 2$				$n \geq 0$

Figure 3.1: Graphical representation of a lifted queue

During the sequence of lifted iterations, it might also be the case, that no new obligations are added to the ObligationQueue. In this case the obligations are only incremented at their level index by one for each iteration. If no new obligations were added, we talk about so called stable iterations.

Definition 23 (Stable iteration)**.** *Two consecutive iterations i and $i + 1$ are called stable, which is denoted by $Q_i \simeq Q_{i+1}$ iff*

$$\{(j + 1, l, c) | (j, l, c) \in Q_i\} = Q_{i+1}$$

Having stable iterations means that the iterations k and $k + 1$ have computed the same set of obligations with respect to the incremented index of the obligation in iteration $k + 1$. When having two successive iterations which are both lifted we deduced the obvious fact, that there cannot be a valid counterexample.

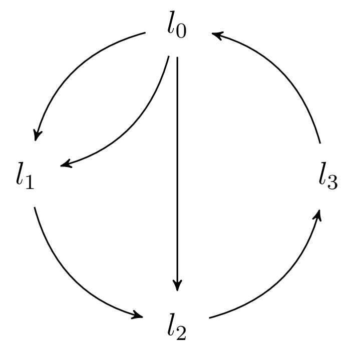
Lemma 4 (Preventing counterexample)**.** *In case that no new obligation is inserted into a lifted ObligationQueue there can never be a counterexample at the end of iteration k.*

Proof 4 (Proof of Lem. 4)**.** *Let Q_k be a lifted ObligationQueue with minimal element q^k_{min}*

$$
\begin{aligned}
q^k_{min} = (i, l, c) \textit{with } i > 2 & \\
&\textit{execute next iteration k+1}\\
&\Rightarrow Q_k = Q_{k+1}\\
&\Rightarrow q^{k+1}_{min} = (i+1, l, c)\\
&\not\Rightarrow Q_{k+1} \textit{ contains a counterexample trace, becaue } q^{k+1}_{min} \neq (1, l, c)
\end{aligned}
$$

But even some successive stable iterations with a lifted ObligationQueue do not imply a terminating state in general. During our research we observed, that valid programs, i.e. programs that do not have a property violation, sometimes yield a sequence of stable iterations. We counted the number of the longest sequence of stable iteration and observed the fact, that the number of stable iterations after which changes occur is always smaller than the length of all cycles in the CFA. We found the algorithm of Johnson [Joh75] to calculate elementary cycles in a directed graph, which is a valid model for abstracting a CFA. The algorithm computes all elementary cycles of a CFA with time bound $\mathcal{O}((|L|+|E|)(|c|+1))$ and space bound $\mathcal{O}(|L|+|E|)$ where $|c|$ denotes the number of elementary cycles.

Definition 24 (Elementary cycle [Joh75])**.** *An elementary cycle is defined as cycle in which each location of the cycle is only visited once.*



Elementary cycles:
$\{l_0, l_2, l_3\}$, $\{l_0, l_1, l_2, l_3\}$ and $\{l_0, l_1, l_2, l_3\}$

$sum(elementary\_cycles)$
$= 3 + 4 + 4 = 11$

$sum(unique\_elementary\_cycles)$
$= 3 + 4 = 7$

Figure 3.2: Graphical representation of elementary cycles

In case the number of stable iterations becomes greater than the length of all elementary cycles there will be no changes in further iterations. In case there would be changes in further iterations, these changes would have occurred earlier because there was a sequence of stable iterations. Having more stable iterations than the length of all elementary cycles means that during these stable iterations no combination of nested cycles was able to cause any changes, therefore there will be no changes in further iterations.

In the following we will explain two approaches which are highly related, but have different upper bound for the amount of stable iterations before terminating if the original termination criterion ($F_{\Delta(i,l)} = \emptyset \; \forall l \in L$ for some i) was not satisfied yet.

New criterion

In our first variant of the new termination criterion we calculate the sum of the length of all elementary cycles. These elementary cycles are computed by the algorithm of Johnson [Joh75]. In addition to this calculation we count the number of successive stable iterations, in case this sum becomes greater than the sum of the length of all elementary cycles, IC3CFA can terminate and return `true`. There will be no changes in further iterations until the original termination criterion is satisfied.

Lemma 5 (New Termination Criterion)**.** *IC3CFA can terminate and return* `true` *if the number of stable iterations is greater than the sum of the length of all elementary cycles in the CFA.*

Proof 5 (Proof of Lem. 5)**.**
Using the definition of a stable iteration (see Def. 23) we count the number of successive stable iterations starting in iteration j denoted by n. In case the ObligationQueue is lifted in iteration j and the number of stable iterations n is greater than the sum of the lengths of elementary cycles, denoted by e, the algorithm can terminate and return `true`*. Having n successive stable iterations means that the results of the satisfiability checks do not differ within these n iterations. In case at least one result differs a new obligation would be added to the ObligationQueue and the stable iteration condition would be violated. Assume the stable iteration condition holds for n iterations, then there were no new predecessors reachable within the last n iterations. If now n is greater than e all combinations of nested cycles would have been reachable in the last n iterations. In these iterations were no changes in the ObligationQueue so there will also be no change in further iterations starting in iteration $j+n+1$. In case there would be changes in iteration $j+n+h$ the changes would have already occurred in iteration $j+h$, because there was no change in n iterations starting from iteration j.*

$$\begin{tikzcd}
j & j+1 & \cdots & j+n-1 & j+n \arrow[llll, bend right] & j+n+1 \arrow[llll, bend left]
\end{tikzcd}$$

Now, we have defined a safe upper bound for stable iterations, which might become significantly big. The reason for a big sum of the length of all elementary cycles can be found in the limited number of operations in an edge label. The original Large-Block-Encoding summarizes all edges as far as possible, i.e. possibly until each location is related to three edges, one ingoing and outgoing and one self-loop. This summarization of edges as far as possible causes complex solver queries, s.t. typically the solver is unable to solve the query. Therefore, the number of operations is limited to achieve a more efficient summarization which is also suitable for the solver. Due to this some locations are connected by two or more parallel edges because the summarization of those edges would violate the operation limit. These parallel edges affect the sum of the length of all elementary cycles, because those cycles are taken into account more than once (see. Fig. 3.2).

Greedy criterion

As mentioned in the previous subsection the new criterion, the sum of the length of all elementary cycles in a CFA is usually much higher than the number of required iterations. Therefore, the new criterion does not have an effect on much problems. Because, usually the programs contain parallel edges in the elementary cycles, s.t. the sum of all elementary cycles grows exponentially by the number of parallel edges in a CFA. Due to this fact the new criterion yields improvements on only a small subset of programs. Therefore, we redefined the upper bound to the sum of the length of all unique elementary cycles, i.e. cycles which contain parallel edges are only taken into account once.

Lemma 6 (Greedy Termination Criterion)**.** *IC3CFA can terminate and return* `true` *if the number of stable iterations is greater than the sum of the length of all unique elementary cycles in the CFA.*

Proof 6 (Proof of Lem. 6)**.** *This proof follows the idea of the previous proof. The upper bound of Proof 5 is reduced to the length of all unique elementary cycles. In case the number of stable iterations becomes greater than the new upper bound we are still able to explore all nested cycles. Having parallel edges in a cycle do not affect the distance between locations which is calculated by the number of needed steps. Using the sum of the length of all unique elementary cycles does not restrict the reachability of locations.*

$$\begin{tikzcd} l_1 \arrow[r, bend left] \arrow[r, bend right] & l_2 & & l_1 \arrow[r] & l_2 \end{tikzcd}$$

In both subgraphs the location l_1 reaches location l_2 in single step, therefore the upper bound of stable iterations can be reduced to the sum of the length of all unique elementary cycles.

3.4 Pushing

Pushing of learnt lemmas is one of the key points of the original IC3 algorithm. This approach was not realized in our setting of IC3CFA yet, therefore we have decided to enrich our algorithm by this key point of the original algorithm. We had to make some adaptation to be able to integrate pushing into our setting. For checking if the learnt lemma c is pushable to F_{i+1} a satisfiability check is performed. c is pushable to F_{i+1} if $(F_i \wedge T \wedge \neg c')$ is unsatisfiable. Transferring the original definition by

[Bra11] yields only a pushing to frames restricted to edges. Which means c be only be pushed to $F_{i+1,l}$ if the specific edge is taken. Pushing c to F_{i+1} means enriching the clauses of F_{i+1} by c [Bra11].

Definition 25 (Pushing on edges).

$$c \textit{ pushable to } F_{(i+1,l)} \textit{ on edge } e = (l_p, op, l) \in G \Leftrightarrow F_{(i,l_p)} \wedge T_{l_p \to l} \wedge c' \textit{ is unsatisfiable}$$

By pushing we get the opportunity to insert clauses in iteration k into empty frames $F_{(k+1,l)}$ before starting iteration $k+1$. In theory it might happen that iterations can skip the blocking phase due to pushing. Furthermore one achieves a better strengthening of the frames which yields a better guidance for path construction or stronger context for obligation blocking [Sud13]. Finally, one might achieve a terminating state earlier through pushing, because the frames might become equal faster through this approach.

To lift the original approach to our setting of performing IC3 on a control flow automaton we have to change the query for the satisfiability check. Instead of taking only the frame of the previous iteration into account we have to check the satisfiability for each predecessor of the location. Therefore, the satisfiability query is extended by a disjunction over each predecessor taking its frame and the transition function into account. Due to the fact that the cubes are implicitly negated when adding them to a frame we do not have to negate c in the satisfiability query.

Lemma 7 (Pushing in IC3CFA).

$$c \textit{ is pushable to } F_{(i+1,l)} \Leftrightarrow \forall e \in \textit{pre\_edges}(l) \; c \textit{ is pushable to } F_{(i+1,l)} \textit{ on edge } e$$

Proof 7 (Proof of Lem. 7).

$$\begin{aligned} c \textit{ is pushable to } F_{(i+1,l)} &\Leftrightarrow \forall e \in \textit{pre\_edges}(l) \; c \textit{ is pushable to } F_{(i+1,l)} \textit{ on edge } e \\ &\Leftrightarrow \forall e \in \textit{pre\_edges}(l) \; F_{(i,l_p)} \wedge T_{l_p \to l} \wedge c' \textit{ is unsatisfiable} \\ &\Leftrightarrow \Big(\bigvee_{e \in \textit{pre\_edges}(l)} (F_{(i,l_p)} \wedge T_{l_p \to l}) \wedge c' \Big) \textit{ is unsatisfiable} \end{aligned}$$

As mentioned in previous sections a satisfiability check is expensive and requires exponential time and memory with an increasing formula complexity, we searched for indicators to be able to perform a push without causing a satisfiability check. In case the location where we want to push has only one predecessor the satisfiability check formula shrinks already because the expensive choice operator in the formula is cleared out. In case the transition formula does not assign any of the cube literals we do not have to prime the cube which should be pushed. This reduces the effort as well, furthermore we are able to skip the satisfiability check for this push request.

Lemma 8 (Pushing without SAT-query). *A cube c can be pushed at $F_{i+1,l}$ without querying the SAT-solver if $pre(l) = \{l_p\}$ and $assigned(T_{l_p \to l}) \cap literals(c) = \emptyset$.*

Proof 8 (Proof of Lem. 8). *Let $F_{(i,l_p)}$ be an arbitrary frame, c be a cube, $T_{l_p \to l}$ the translation relation with*

(1) $pre(l) = \{l_p\}$ and
(2) $assigned(T_{l_p \to l}) \cap literals(c) = \emptyset$.

$$\begin{aligned}
c \textit{ is pushable to } F_{(i+1,l)} &\Leftrightarrow \forall l_p \in pre(l)\textit{: } \left(\bigvee (F_{(i,l_p)} \wedge T_{l_p \to l}) \wedge c'\right) \textit{ is unsatisfiable} \\
&\overset{(1)}{\Rightarrow} F_{i,l_p} \wedge T_{l_p \to l} \wedge c' \textit{ is unsatisfiable} \\
&\overset{(2)}{\Rightarrow} F_{i,l_p} \wedge T_{l_p \to l} \wedge c \textit{ is unsatisfiable} \\
&\neg c \in F_{i,l_p} \textit{ because } l \in succ(l_p) \\
&\Rightarrow \textit{ query is unsatisfiable}
\end{aligned}$$

Pushing without causing a satisfiability check reduces the number of satisfiability queries and preserves unnecessary satisfiability checks which might require a lot of time and memory. Omitting the satisfiability check is not the only improvement which can be realized when using pushing, in case we combine the previous approaches of obligation reuse and obligation skipping in the outer loop we are also able to reduce the number of obligations for the next iteration. Pushing a cube c at $F_{(i+1,l)}$ makes the obligation for location l at level $i+1$ in the next iteration superfluously if the cube of the obligation is already covered by c, i.e. c is subset of the obligations cube. Checking this obligation and potentially its predecessors will not yield a frame which is stronger after blocking the obligation's cube. Through pushing c to $F_{(i+1,l)}$ we have already achieved a stronger frame than before and adding a superset of c to the frame does not yield any stronger frame.

Lemma 9 (Pruning)**.** *If c was pushed at $F_{(i,l)}$ all obligations which were derived from obligations with $q = (i, l, \hat{c})$ can be omitted in the next iteration iff $\hat{c}$, $c \subseteq \hat{c}$*

Proof 9 (Proof of Lem. 9)**.** *Let $F_{p(i,l)}$ the frame before c was pushed.*

$$\begin{aligned}
F_{(i,l)} := F_{p(i,l)} \wedge \neg c &\Rightarrow \neg c \in F_{(i,l)} \\
&\Rightarrow F_{(i,l)} \subseteq F_{p(i,l)} \wedge \neg\hat{c} \\
&\Rightarrow \textit{blocking } \hat{c} \textit{ yields no change} \\
&\Rightarrow \textit{predecessors of } l \textit{ not reachable without reuse}
\end{aligned}$$

4 Implementation

This section provides implementation details of the approaches which are explained in theory in Sec. 3. First we give a general overview of the framework in which the implementation was done in. Following the structure of the previous section we start with the obligation reuse. Afterwards skipping and the new termination criteria are described. Finally this section deals with the implementation of pushing.

4.1 Framework

The approaches were integrated in an existing model checking framework, improving the IC3CFA algorithm. The framework is at the moment able to handle C programs. First the programs are translated into the C Intermediate Language (CIL) [NMRW02]. Afterwards this CIL-Code is translated by a self-written parser into an intermediate verification language (IVL) [LNN15]. Before performing the model checking itself, the IVL is optimized. For optimization we apply widely used methods like program slicing, expression propagation, bisimulation minimization and Steensgaard's pointer analysis [Ste96]. The underlying bit-precise memory model supports limited pointer operations, including record-field as well as array-element addressing. The optimizations are performed until a fixpoint is reached, afterwards the control flow graph of the program is constructed. The labels are in guarded command language, which is similar to the definition of operations (see Def. 2) in Sec. 2. The use of GCL provides an efficient way to construct the weakest precondition [FS01],[Lei05]. Our tool supports Z3 and MathSAT as SMT solvers. A high-level structure of the framework is shown in the figure below.

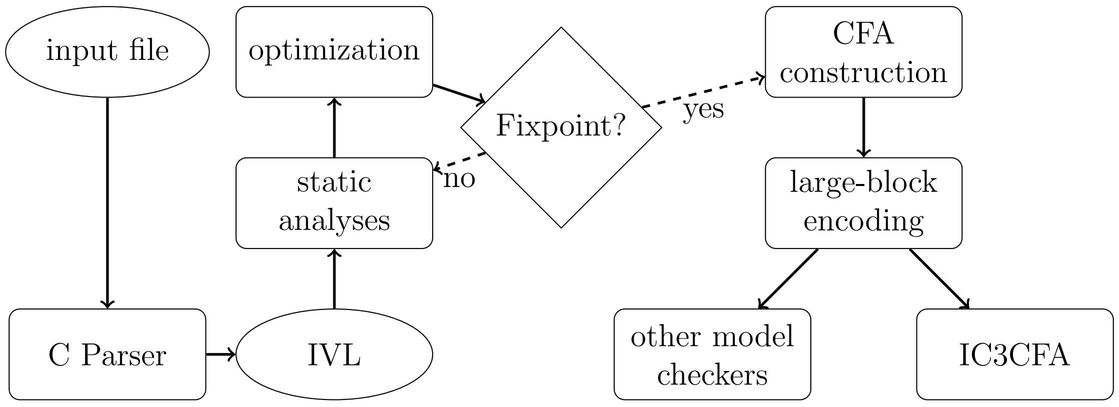


Figure 4.1: Framework [LNN15]

4.2 Obligation reuse

The theoretical idea of obligation reuse is provided in Sec. 3.1, this section focuses the implementation of this approach. Basis for realizing this is the original IC3CFA algorithm, as it was presented in Sec. 2.3. In general the changes are not significant, however each of the changes will be discussed in the following. The changes are marked with red in the pseudo code to make clear which parts are adapted. The ObligationQueue is represented by a priority queue where the obligations are ordered in ascending order w.r.t. their level index. Obligations which are marked as reused have a lower priority as obligations which are not marked as reused. To realize the reuse marking we added a new boolean flag to the representation of an obligation. Therefore, the ordering of obligations is as follows:

$$\begin{array}{rl}
(i, l, c, \texttt{false}) \preceq (i, l, c, \texttt{true}) & \forall i, j \\
(i, l_x, c_x, \texttt{false}) \preceq (j, l_y, c_y, \texttt{false}) & \Rightarrow i < j \\
 & \lor\, i = j \land l_x < l_y \\
 & \lor\, i = j \land l_x = l_y \land |literals(c_x)| < |literals(c_y)| \\
 & \text{else arbitrary order} \\
(i, l, c, \texttt{true}) \preceq (j, l, c, \texttt{true}) & \Rightarrow i < j \\
 & \lor\, i = j \land l_x < l_y \\
 & \lor\, i = j \land l_x = l_y \land |literals(c_x)| < |literals(c_y)| \\
 & \text{else arbitrary order}
\end{array}$$

Change 1 The ObligationQueue only needs to be initialized with the CTI obligations once at the beginning of IC3CFA. Therefore, we moved the computation of these obligations to a point before starting the inner loop. The initial CTI obligations start at level 1 and their reuse flag is initialized by `false` as it will be done for each obligation is added to the ObligationQueue unless q_{min} causes an immediate blocking of its cube, in this case the reused flag is set to `true`.

Change 2 The inner loop, represented by the function *backwardblock*(...), does not only return whether a counterexample was detected but also the ObligationQueue after executing the inner loop. Therefore the result is now defined as tuple of $(Queue, boolean)$. The returned ObligationQueue will be used for the next iteration after performing a clean-up of the ObligationQueue.

Change 3 The *clean* function (see Alg. 10) increments the index and sets the reuse flag to `false` for each obligation in the ObligationQueue. Updating the ObligationQueue is necessary, because otherwise the next iteration will not yield any new results, and the algorithm will terminate misleadingly due to empty delta frames.

Algorithm 9 Outer loop [LNN15]

```
function PROVE
    if $l_0 = l_E$ or $((l_0, op, l_E) \in G$ and $sat(T_{l_0 \to l_E}))$ then
        return false
    initialize frames
    for $l : L$ do                                              ▷ Change 1
        if $sat(F_{(k,l)} \wedge T_{l \to l_E})$ then
            $s :=$ predecessor data region
            Q.add$(1, l, s, false)$
    end for
    for $k = 1$ to ... do
        (result,Q) := $BACKWARDBLOCK(Q)$                        ▷ Change 2
        if not result then
            return false
        propagate
        if termination then
            return true
        $CLEAN(Q)$                                              ▷ Change 3
    end for
end function
```

Algorithm 10 cleaner

```
function CLEAN($Q$:ObligationQueue)                             ▷ Change 3
    while $| Q | > 0$ do
        $(i, l', s, r) = Q.pop$
        add $(i + 1, l', s, false)$ to $Q_{result}$
    end while
    return $Q_{result}$
end function
```

Change 4 When applying the reuse of obligations, it is proven, that the ObligationQueue size is monotonic for each iteration. Therefore, an empty ObligationQueue does not appear during executing the inner loop. This causes a significant change to the termination of the inner loop, returning `false` in case the minimal obligation has level 1 remains unchanged. In case the reuse flag of the minimal obligation is set to `true` all following obligations will also be reused due to the ascending order in the ObligationQueue. Therefore, we leave the inner loop by returning `true` and the ObligationQueue which is equivalent to the termination in case of an empty without obligation reuse.

Change 5 In case of a satisfiable inductivity check the predecessors of location l are taken into account for the next inductivity check. Therefore their obligations are added to the ObligationQueue, obviously these obligations are fresh and are not marked as reused, i.e. $r :=$ `false`. Even the obligation q which caused the satisfiable

Algorithm 11 Inner loop [LNN15]

```
function BACKWARDBLOCK($Q$:ObligationQueue)
    while $| Q | > 0$ do
        $(i, l, s, r) = q = Q.pop$          ▷ $i$:int, $l$:location, $s$ :data region, $r$ :boolean
        if $i = 0$ then
            return ($Q$,false)                                    ▷ Change 2
        else if r then                                            ▷ Change 4
            return ($Q$,true)                     ▷ minimal element is reused
        else
            for each $l_p$, s.t. $(l_p, op, l) \in F$ do
                if $l_p = l$ and $sat(F_{(i-1,l_p)} \wedge \neg s \wedge T_{l_p \to l} \wedge s')$ then
                    generate predecessor $c$ of $s$
                    add $(i-1, l_p, c, false)$ and $(i, l, s, false)$ to $Q$   ▷ Change 5
                else if $l_p \neq l$ and $sat(F_{(i-1,l_p)} \wedge T_{l_p \to l} \wedge s')$ then
                    generate predecessor $c$ of $s$
                    add $(i-1, l_p, c, false)$ and $(i, l, s, false)$ to $Q$   ▷ Change 5
                else
                    compute generalization $s_{gen}$ of $s$
                    block $s_{gen}$ in frames $F_{(j,l)}$ for $0 \leq j \leq i$
                    add $(i, l, s, true)$ to Q                    ▷ Change 6
            end for
        end while
        return ($Q$,true)                                         ▷ Change 2
    end function
```

inductivity check is reinserted into the ObligationQueue, but due to the unsatisfied condition that an obligation is added as reuse causes an immediate blocking, q is also added to the ObligationQueue with $r :=$ `false`.

Change 6 In case that the inductivity check for an obligation returns unsatisfiable the generalization of the cube is blocked, we add this obligation with a reused marking to the ObligationQueue.

The functionality of the *strenghthen* function which part of the origrinal IC3CFA algorithm (see Sec. 2.3) is integrated in the outer loop.

4.3 Skipping

When using the obligation reuse one checks obligations which might have been checked in previous iterations. As already mentioned in Sec. 3.2 there are indicators for skipping an obligation or at least leaving some predecessors out of scope for inductivity checks in the inner loop. Therefore, we implemented the concept of skipping at two positions in the algorithm. First, checking each obligation in the already existing ObligationQueue being skipable, second one taken these predecessors into account which have changed in the previous iteration.

Algorithm 12 Outer loop [LNN15]

```
function PROVE
    if $l_0 = l_E$ or $((l_0, op, l_E) \in G$ and $sat(T_{l_0 \to l_E}))$ then
        return false
    initialize frames
    for $l : L$ do
        if $sat(F_{(k,l)} \wedge T_{l \to l_E})$ then
            $s :=$ predecessor data region
            Q.add$(1, l, s, false, false)$                    ▷ Change 1
    end for
    for $k = 1$ to ... do                                     ▷ Change 2
        for each $q = (i, l, s, r, o) \in Q$ do
            delete $q$ from $Q$
            if $skipable(q)$ then
                block $F_{\Delta(i-1,l)}$ at frame $F_{(i,l)}$
                add $(i, l, s, true, o)$ to $Q_{skip}$
            else
                add $(i, l, s, false, o)$ to $Q_{skip}$
        end for
        $Q := Q_{skip}$
        (result,Q) $:= BACKWARDBLOCK(Q)$
        if not result then
            return false
        propagate
        if termination then
            return true
        $CLEAN(Q)$
    end for
end function
```

Change 1 For being able to perform the check in the inner loop we needed to add another attribute to our obligation representation. Therefore, we extended the obligation by a Boolean flag for representing the old attribute. The CTI obligations are inserted into the ObligationQueue as shown with obligation reuse, extended with the old flag, which is set to `false`, because the obligation was never marked as reused before.

Change 2 Before executing the inner loop, we check each obligation in the ObligationQueue if it this skippable in the current iteration. Therefore, the function *skippable* is called for each obligation, it checks whether the obligation is old and the delta frames for all predecessors at level $i - 2$ are empty. An obligation is old if it has been marked as reuse in previous iterations. In case both conditions are satisfied, the obligation can be skipped and we add $F_{\Delta(i-1,l)}$ to $F_{(i,l)}$. A skipped obligation will not be handled in the inner loop for this iteration again, therefore we

Algorithm 13 Inner loop [LNN15]

```
1:  function BACKWARDBLOCK($Q$:ObligationQueue)
2:      while $| Q |> 0$ do
3:          $(i, l', s, r, old) = q = Q.pop$
4:                                        ▷ $i$:int, $l'$:location, $s$ :data region, $r, old$ :boolean
5:          if $i = 0$ then
6:              return ($Q$,false)
7:          else if r then
8:              return ($Q$,true)                         ▷ minimal element is reused
9:          else
10:             for each $l$, s.t. $(l, op, l') \in G$ do
11:                 if $(\text{old} = \text{true} \wedge F_{\Delta(i-2,l)} = \emptyset)$ then       ▷ Change 3
12:                     goto line 19
13:                 if $l = l'$ and $sat(F_{(i-1,l)} \wedge \neg s \wedge T_{l \to l'} \wedge s')$ then
14:                     generate predecessor $c$ of $s$
15:                     add $(i-1, l, c, false, false)$ and $(i, l', s, false, false)$ to $Q$
16:                                                               ▷ Change 1
17:                 else if $l \neq l'$ and $sat(F_{(i-1,l)} \wedge T_{l \to l'} \wedge s')$ then
18:                     generate predecessor $c$ of $s$
19:                     add $(i-1, l, c, false, false)$ and $(i, l', s, false, false)$ to $Q$
20:                                                               ▷ Change 1
21:                 else
22:                     compute generalization $s_{gen}$ of $s$
23:                     block $s_{gen}$ in frames $F_{(j,l')}$ for $0 \leq j \leq i$
24:                     add $(i, l', s, true, true)$ to Q                 ▷ Change 4
25:             end for
26:     end while
27:     return ($Q$,true)
28: end function
```

mark it as reused and add it to a new ObligationQueue Q_{skip}. In case an obligation cannot be skipped we directly add it to the ObligationQueue Q_{skip}.

Change 3 We also implemented the skipping criterion in the inner loop, because it might be the case that an obligation was not skippable before executing the inner loop, but some predecessors changed during handling 'smaller' obligations s.t. the delta frames might become empty. For that reason, we added a check before the SMT-solver is involved. The check is similar to the implementation of the function *skippable*. The main difference is, that not the whole obligation is skipped, rather single predecessors are not taken into account for checking the inductivity.

Change 4 When adding an obligation which is marked as reused to the ObligationQueue we have to set the old flag to true. Hence in following iterations we see that this obligation has been marked as reused in previous iterations and perform

Algorithm 14 Check skipable obligation

```
1: function SKIPABLE($q$:Obligation)
2:     $q = (i, l, s, r, old)$    ▷ $i$:int, $l'$:location, $s$ :data region, $r$ :boolean, $old$ :boolean
3:     frame := $\emptyset$
4:     if $\neg old$ then return false
5:     for each $l_p \in pre(l)$ do
6:         $frame := frame \cup F_{\Delta(i-2,l_p)}$
7:     end for
8:     return ($frame == \emptyset$)    ▷ $\Leftrightarrow \forall l_p \in pre(l)$: $F_{\Delta(i-1,l)} = \emptyset$
9: end function
```

the skipping check for the obligation (outer loop) or at least for the predecessors of the obligation's location in the inner loop.

4.4 Termination

We implemented our new termination criterion on top of the already existing implementation with reuse and skipping. There were only some minor changes necessary to integrate the new criterion in the algorithm. For being able to determine stable iterations with obligation reuse and skipping we have to refine definition 23.

Definition 26 (Stable iteration (refined))**.** *Two consecutive iterations k and $k+1$ are called stable (denoted by $Q_k \simeq Q_{k+1}$) iff*

$$\begin{aligned} \forall q = (i, l, c, r, o) \in Q_k,\ \exists \hat{q} \in Q_{k+1} \\ s.t.\ \hat{q} = (i+1, l, c, true, o) \\ and\ |Q_k| = |Q_{k+1}| \end{aligned}$$

After refining the definition of stable iterations we will describe the changes in the algorithm in detail. The inner loop of the algorithm remains unchanged, therefore we omit the pseudo code.

Change 1 At the beginning of the algorithm we calculate the sum of the lengths of the elementary cycles in the CFA A. In case we use the greedy criterion we calculate the sum of unique elementary cycles. We execute the algorithm for calculating elementary cycles after the initial checks of IC3CFA are passed, because if one of the initial conditions is violated the inner loop will not be executed anymore and the algorithm terminates immediately. After calculating the cycle-sum the frames and the ObligationQueue are initialized.

Change 2 At the beginning of each outer loop we have to save the current content of the ObligationQueue, because at the end of the iteration we want to check whether we have a stable iteration. After saving the ObligationQueue at the beginning of the outer loop, the checks for skippable obligation are performed and afterwards the inner loop is executed.

Algorithm 15 Outer loop

```
1: function PROVE
2:     if $l_0 = l_E$ or $((l_0, op, l_E) \in G$ and $sat(T_{l_0 \to l_E}))$ then return false
3:     cycles := sum(elementary_cycles(A)) OR: sum(unique_elementary_cycles(A))
4:                                                                 ▷ Change 1
5:     initialize frames
6:     for $l : L$ do
7:         if $sat(F_{(k,l)} \wedge T_{l \to l_E})$ then
8:             $s$ := predecessor data region
9:             Q.add$(1, l, s, false, false)$
10:    end for
11:    for $k = 1$ to ... do
12:        $Q_{before} := Q$                                       ▷ Change 2
13:        for each $q = (i, l, s, r, o) \in Q$ do
14:            delete $q$ from $Q$
15:            if $skippable(q)$ then
16:                block $s$ in frames $F_{(j,l')}$ for $0 \leq j \leq i$
17:                add $(i, l, s, true, o)$ to $Q_{skip}$
18:            else
19:                add $(i, l, s, false, o)$ to $Q_{skip}$
20:        end for
21:        $Q := Q_{skip}$
22:        (result,Q) := $BACKWARDBLOCK(Q)$
23:        if not result then return false
24:        propagate
25:        (stable,termination) := termination_check $(Q_{before}, Q, cycles, stable, k)$
26:                                                                 ▷ Change 3
27:        if termination then return true
28:        $CLEAN(Q)$
29:    end for
30: end function
```

Change 3 The details for the termination check of IC3CFA were not provided yet, but it will be integrated in the new termination check as well. First, the termination check is more complex than before, because it also manages the sequence length of stable iterations. Therefore, we have to pass the ObligationQueue before executing the current iteration, saved in Q_{before}, the ObligationQueue at the end of the iteration k in Q, the cycle-sum, the length of stable iterations up to the current iteration, and the iteration index to the termination check function. First, we check whether the original criterion, empty delta frames at some level $1 < i < k$, holds. In case there is no level s.t. all delta frames are empty, we will perform our new approach, therefore the queues are compared. In case both queues are equal, i.e. each obligation that is in the ObligationQueue after the iteration was already in the ObligationQueue before the iteration, the value of *stable* is incremented by one. In

Algorithm 16 Termination check

```
1: function TERMINATION_CHECK($Q_{before}$, Q:ObligationQueue; cycles, stable, k:int)
2:     for i=1 to k do
3:         if $\forall l \in L : F_{(i,l)} = F_{(i+1,l)}$ then return true
4:     end for
5:     if $Q_{before} \simeq Q$ then                              ▷ Following Def. 26
6:         stable:=stable+1
7:     else
8:         stable:=0
9:     return (stable > cycles)
10: end function
```

the following we check, whether the sequence of stable iterations is greater than the cycle-sum, if so we will terminate at this point, if not the next iteration will the executed.

4.5 Pushing

The adaptation of the original pushing of IC3 is also implemented on top of the already existing improvements of obligation reuse, skipping and new termination criterion. For making pushing applicable to our setting, we have to introduce two new functions which are explained in this section. Furthermore, we integrate a new global variable which describes the filter for push requests. The variable is an enumeration of $\{NONE, PREDECESSOR, ASSIGN, RESTRICTIVE\}$. For achieving most effects, we decided to execute the pushing at the end of each iteration, because at this point each frame is already extended by the newly blocked cubes. For being able to deal with push requests we also added a new queue for these push requests, we denote the new PushQueue with Q_P. The push requests are ordered by their level index, location index and cube size.

$$
\begin{aligned}
(i, l_x, c_x) \preceq (j, l_y, c_y) \qquad & \Rightarrow i < j \\
& \lor i = j \land l_x < l_y \\
& \lor i = j \land l_x = l_y \land |literals(c_x)| < |literals(c_y)| \\
& \text{else arbitrary order}
\end{aligned}
$$

Change 1 In case the ObligationQueue is pseudo-empty, i.e. the minimal element of the ObligationQueue is marked as reused, we execute the pushing using the PushQueue. The PushQueue is created during the iteration (see Change 2). The real pushing is done by the function *push*. This function iterates over the queue of push requests. In case the restrictive filter is enabled we are able to block c without any SAT-query at $F_{(i+1,l)}$, otherwise we have to check if the formula $F_{(i,l_p)} \land T_{l_p \to l} \land c'$ is unsatisfiable for all predecessors l_p of l. If so, we can block c at $F_{i+1,l}$, if not c cannot blocked.

Algorithm 17 Push

```
function CREATE_PUSHS($s$:cube, $l$:location, $i$:int, $Q_P$:PushQueue)
    successors:=succ(l)
    for each $l_s \in successors$ do
        if filter=NONE then
            add $(i, l_s, c)$ to $Q_P$
        else if filter=PREDECESSOR then
            if length($pre(l_s) = 1$) then
                add $(i, l_s, c)$ to $Q_P$
        else if filter=ASSIGN then
            label := $T_{l \to l_s}$
            if $assigned(label) \cap literals(c) = \emptyset$ then
                add $(i, l_s, c)$ to $Q_P$
        else                                        ▷ filter=RESTRICTIVE
            label := $T_{l \to l_s}$
            if length($pre(l_s) = 1$ && $assigned(label) \cap literals(c) = \emptyset$ then
                add $(i, l_s, c)$ to $Q_P$
    end for
end function
```

Change 2 In case we are able to block the cube c at location l with index i we extend the PushQueue with new requests. For extending the PushQueue we call the function *create_pushs* which first computes the successors of the given location l. In case no push filter was selected we add a push request for each successor to the PushQueue. This yields a high number of requests for pushing and the maximal amount of pushs, but we observed that several request were not pushable or caused solver timeouts. Therefore, we introduced more restrictive filters for creating push request. First, we added a filter s.t. push request is only created if the successor has one predecessor, which is obvious l. By applying this filter, we reduced the number of satisfiability checks already significantly, especially by applying Large-Block-Encoding before executing IC3CFA many locations have more than one predecessor. But still many of the push requests are not pushable yet. To achieve a higher rate of successful push requests, we implemented another filter. In this case a push request is only created if the intersection of the assigned variables along the edges with the literals of the cube which should be pushed is empty. That means in detail, that the edge does not modify any of the cube literals. With this filter we still allow several predecessors for the push locations. The restrictive filter is a combination of the previous filters. When enabling this filter, a push request is created if both conditions are satisfied. That means the push location has only one predecessor and any literals of the cube are assigned in the edge. In this case we are able to push c without causing a satisfiability check.

We have also implemented a variant which supports the pruning of obligations, therefore we had to change the structure of the ObligationQueue from a priority queue to a tree-like structure, s.t. one is able to prune a subtree with minimal ef-

Algorithm 18 Create_Pushs

```
1: function PUSH($Q_P$:PushQueue)
2:     while $| Q_P |> 0$ do
3:         $(i, l, c) = p = Q_P.pop$                    ▷ $i$:int, $l$:location, $c$ :cube
4:         if push_filter=ALL then
5:             goto line 8
6:         for each $l'$, s.t. $(l', t, l) \in G$ do
7:             if sat($F_{(i,l')} \wedge T_{l' \to l} \wedge c'$) then
8:                 goto line 2
9:         end for
10:        block $c$ at $F_{i+1,l}$
11:    end while
12: end function
```

fort. This implementation was not competitive, because we had to keep a copy of the ObligationQueue in each iteration to be able to realise obligation reuse. Obligation reuse is necessary to prune a subtree, because without reuse the ObligationQueue would be empty at the end of the inner loop. Therefore, this approach only works when enabling the obligation reuse, due to this we have to introduce a copy of the ObligationTree, because the former order of obligations w.r.t to the reuse flag is not applicable any longer. Therefore, we are not able to keep normal and reuse obligations in a single instance of the tree. Due to this we have to insert the predecessor of a satisfiable obligation into two ObligationTrees, the working ObligationTree and the copy ObligationTree which is responsible for reusing obligations in the next iteration. This fact causes already inefficiency. For the checking whether a subtree can be pruned we have to check all obligations for location l at level i whether the pushed cube is subset of the obligation's cube. Only if all obligations satisfy this condition the subtree can be pruned.

Algorithm 19 Inner loop [LNN15]

```
1: function BACKWARDBLOCK($Q$:ObligationQueue)
2:     while $\mid Q \mid > 0$ do
3:         $(i, l, s, r, old) = q = Q.pop$            ▷ $i$:int, $l$:location, $s$ :data region, $r, old$ :boolean
4:         if $i = 0$ then
5:             return ($Q$,false)
6:         else if r then
7:             push($Q_P$)                            ▷ Change 1
8:             return ($Q$,true)                      ▷ minimal element is reused
9:         else
10:            for each $l_p$, s.t. $(l_p, op, l) \in G$ do
11:                if (old = true $\wedge$ $F_{\Delta(i-2,l_p)} = \emptyset$) then
12:                    goto line 19
13:                if $l_p = l$ and $sat(F_{(i-1,l_p)} \wedge \neg s \wedge T_{l_p \to l} \wedge s')$ then
14:                    generate predecessor $c$ of $s$
15:                    add $(i-1, l_p, c, false, false)$ and $(i, l, s, false, false)$ to $Q$
16:                else if $l_p \neq l$ and $sat(F_{(i-1,l_p)} \wedge T_{l_p \to l} \wedge s')$ then
17:                    generate predecessor $c$ of $s$
18:                    add $(i-1, l_p, c, false, false)$ and $(i, l, s, false, false)$ to $Q$
19:                else
20:                    compute generalization $s_{gen}$ of $s$
21:                    block $s_{gen}$ in frames $F_{(j,l)}$ for $0 \leq j \leq i$
22:                    add $(i, l, s, true, true)$ to Q
23:                    $Q_P$:=create_pushs($c, l, i, Q_P$)   ▷ Change 2
24:            end for
25:    end while
26:    return ($Q$,true)
27: end function
```

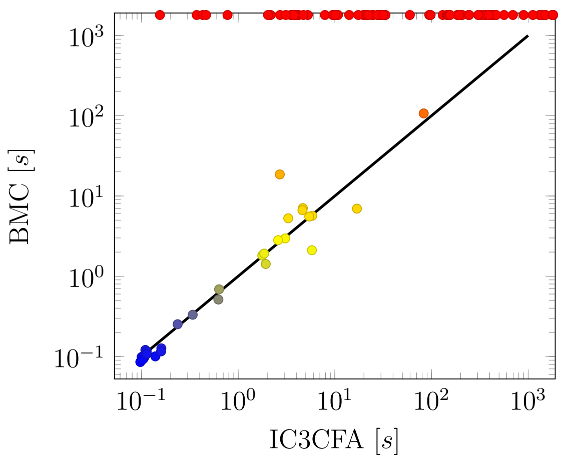
5 Evaluation

For evaluation of the implemented approaches we used a set of 150 programs which are subsets of the benchmarks in [CG12] and of the software verification competition benchmarks. 49 of 150 programs contain a bug. All experiments have been executed on a cluster using one core per instance, running at 2.1 GHz. The memory limit was set to 3 GB and a time limit of 1800 seconds. For scoring the results we used the tool benchexec [Bey16], each verification is scored by a value of -32 up to $+2$. A reported result unknown is scored with 0 points, a correct false result with $+1$, in case the algorithm reports a violation on a program which satisfies the property, the score is -16. Proving the correctness of a satisfied program is scored with $+2$, reporting a wrong proof is scored with -32 [Bey16].

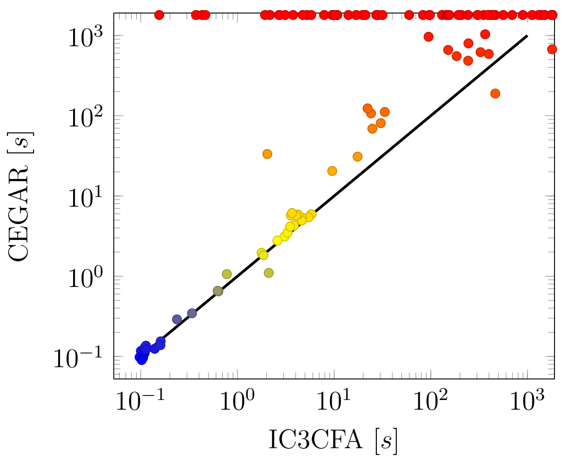
This section provides the performance evaluation of different configurations and approaches of our implementation. First, we compare IC3CFA to common used algorithms like BMC and CEGAR. Second each of our approaches is evaluated compared to the original IC3CFA algorithm and eventually with some other approaches. For each comparison we provide a scatter plot, which shows the cpu time against the named algorithm on a logarithmic scale. A node above the diagonal line means that the algorithm which is listed on the x axis performs better than the algorithm listed on the y axis with respect to the cpu time. Memory requirements in MB are shown in the table which also provides the absolute scores and the number of solved programs as well as the runtime in seconds. All shown algorithms do not report any false results. The number of shown satisfiability checks are those that have been executed. A cache for satisfiability checks is integrated in our project, that means an undefined number of satisfiability checks, depending on their memory requirements, are held in the cache and the result is taken from there without causing a real satisfiability check at the solver. Therefore, a reduction of satisfiability checks could be a higher than the difference shows.

5.1 BMC and CEGAR

We start our evaluation with a comparison of IC3CFA to other common model checkers. BMC with a default bound of 1,000 steps was able to solve 31 programs, most of them are false programs. The valid programs were solved on the underlying CFA structure, because BMC is not able to proof complex true programs, otherwise. In those cases, the CFA is empty or contains no error locations due to the performed optimizations. The algorithm required 178 seconds and 2 GB of memory in total to solve the programs. Overall BMC achieved a score of 37 points. We also tested CEGAR on the set of programs and it was able to solve 52 of 150 programs in 7,215 with a required memory of 5.7 GB. 23 of the 52 solved programs are valid ones. The small amount of memory can be professed by the abstraction of the program which is usually coarse. CEGAR achieved a score of 75 points. Finally we executed the standard IC3CFA algorithm as described in Sec. 2.3. By using this algorithm, we are able to solve 110 of 150 programs, where 73 of them are valid programs. There we yield a score of 182 points, which is the best in this comparison. IC3CFA requires most memory compared to the two other approaches because for each frame and iteration there exists a frame.



(a) Comparison of BMC vs IC3CFA



(b) Comparison of CEGAR vs IC3CFA

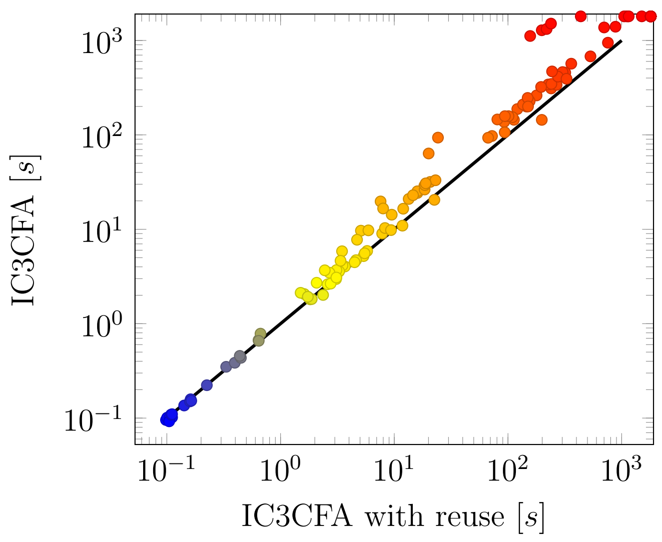
tool	score	solved	t solved	memory
BMC	37	31/150	178	2,085
CEGAR	75	52/150	7,215	5,703
IC3CFA	182	110/150	18,170	20,866

Figure 5.1: Comparison of IC3CFA vs CEGAR and BMC

It is quite obvious that the original algorithm of IC3CFA is already more competitive than the other common model checking approaches like BMC or CEGAR.

5.2 Obligation reuse

Following the structure of approaches to improve the original algorithm by using learnt information we now focus on obligation reuse. By applying reuse, we are able to solve 116 out of 150 programs, which is 6 more than without reuse. Every program that was already solvable by the original algorithm is still solvable by this new approach of obligation reuse. The total score is improved by 11 points compared to the original algorithm. In addition, we save about 2,000 seconds on our benchmark set before the programs are solved. Nearly all programs, except for one, are solved in less time. One argument for time savings is the smaller amount of predecessor computation, because we use the obligations of the previous iteration, therefore we do not need to unroll the path in each iteration again. The necessary transformation, increment the obligation's level, is therefore more efficient than recomputing them in each further iteration. This argument also holds for the fact of saving about 4 GB of memory during executing the set of benchmark programs. A significant decrease can be observed on the amount of needed satisfiability checks. The original algorithm needs about 710,000 checks, by using obligation reuse the algorithm only needs about 430,000 checks. The reason for that reduction can also be found in the fact, that the paths are not explored in each iteration beginning at the CTIs. With an increasing iteration index the amount of reachable locations might grow up to exponentially due to branching.

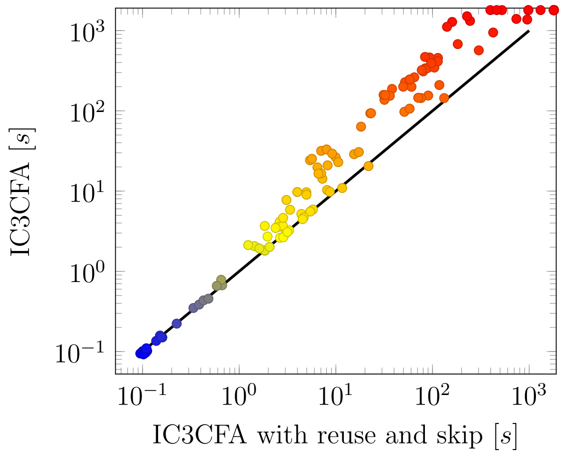


tool	score	solved	t solved	memory
IC3CFA	182	110/150	18,170	20,866
IC3CFA with reuse	193	116/150	16,132	16,264

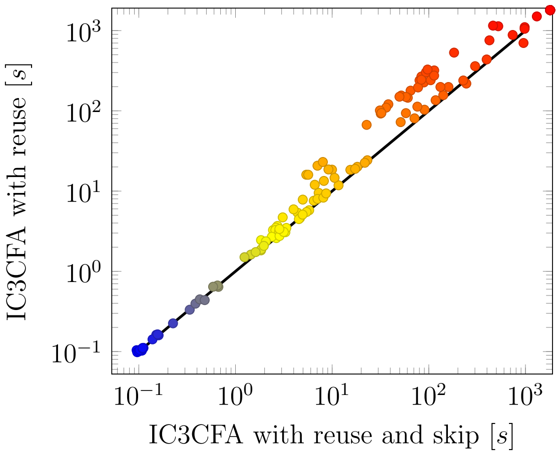
Figure 5.2: Comparison of IC3CFA vs IC3CFA with reuse

5.3 Obligation reuse with skipping

Our second approach was to skip an obligation already in the outer loop if the predecessor frames remained unchanged in the previous iteration. We have tested this approach compared with the original IC3CFA algorithm as well as with the already improved algorithm with obligation reuse. Compared to the original algorithm we are able to solve 6 programs in addition and safe nearly 6,000 seconds, which is nearly one third of the time. The final score for IC3CFA with reuse and skipping is 193 points. Not only the time can be reduced significantly, the satisfiability checks can also be reduced. The original algorithm needs about 710,000 checks, where this approach needs only 410,000, which is a saving of about 40%. In comparison with the improved algorithm with obligation reuse, this approach solves no new program and yields the same score. Focusing the memory requirements both need the same amount of memory, but this approach saves about 1,200 seconds compared to obligation reuse. The number of satisfiability checks can be reduced by 20,000, that means that over all programs this approach is able to skip about 20,000 obligations in the outer loop. Our fear of producing too much effort by checking each obligation for being skipable did not become true. Rather the reduction of satisfiability checks outperforms the additional checks at the beginning of each iteration. An advanced variant of this approach being able to skip a whole iteration causes too restrictive conditions. Checking these conditions is less efficient than not skipping the whole iteration.



(a) Comparison of IC3CFA vs IC3CFA with reuse and skip



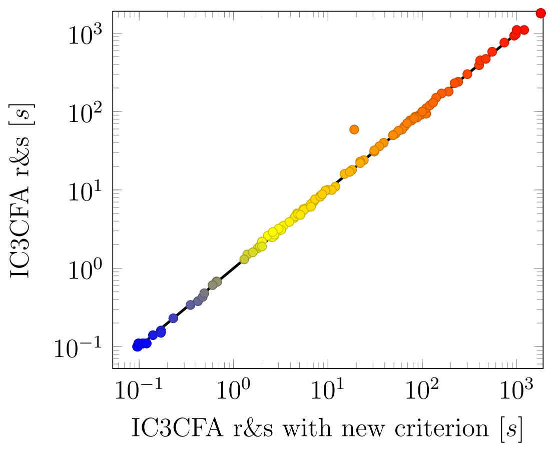
(b) Comparison of IC3CFA with reuse vs IC3CFA with reuse and skip

tool	score	solved	t solved	memory
IC3CFA	182	110/150	18,170	20,866
IC3CFA with reuse	193	116/150	16,132	16,264
IC3CFA with reuse and skip	193	116/150	10,407	14,986

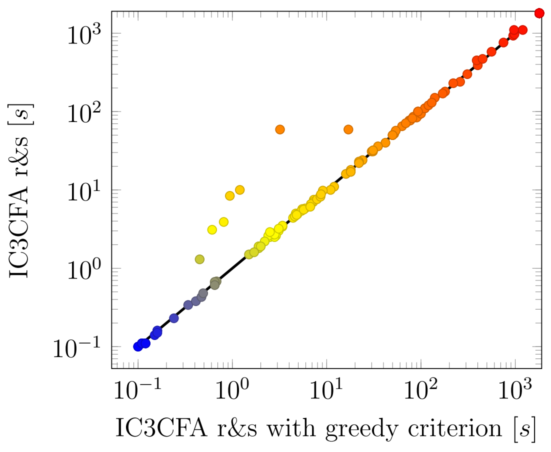
Figure 5.3: Comparison of IC3CFA (with reuse) vs IC3CFA with reuse and skip

5.4 Obligation reuse with skipping and termination

The next approach which should improve IC3CFA not only in theory but also in practical use is the refined termination criterion. As already described we have found two upper bounds for stable iterations in a row, which are both based on elementary cycles in the CFA. Both criteria are implemented on top of the improvements of obligation reuse and skipping of obligations. Therefore, we made benchmarks for both criteria only against the improved implementation of IC3CFA. We first tested the new criteria, i.e. the upper bound is the sum of all elementary cycles. The improvements are not as significant as expected. We achieved the same score in nearly equivalent time while the number of satisfiability checks decreases due to less executed iterations. We inspected structure of the underlying CFAs more detailed and found the reason why the sum of all elementary cycles is often higher than the needed iterations. The number of operations in an edge label is limited by the implementation of Large-Block-Encoding that means there are several parallel edges in the cycles. Therefore, the sum of all elementary cycles grows significantly. Due to this reason we refined the upper bound to the sum of the length of all unique elementary cycles. This implementation yields a better improvement of IC3CFA than the previous approach. We still achieve 193 points at same time requirements than before but the number of satisfiability checks can be reduced by about 14,000 compared to the original termination criteria. The improvements are still limited by the fact that nearly all locations are part of a cycle.



(a) Comparison of IC3CFA r&s vs IC3CFA r&s with new criterion



(b) Comparison of IC3CFA r&s vs IC3CFA r&s with greedy criterion

tool	score	solved	t solved	memory	# SAT	# k
IC3CFA r&s	193	116/150	10,407	14,986	408,194	3,887
IC3CFA r&s new	193	116/150	10,335	15,129	404,926	3,862
IC3CFA r&s greedy	193	116/150	10,176	14,953	394,872	3,727

Figure 5.4: Comparison of IC3CFA r&s vs IC3CFA r&s with termination criteria

5.5 Pushing

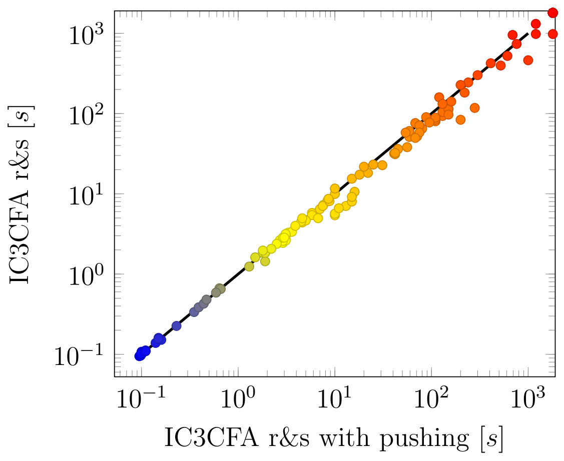
Our last approach was the adaptation of the original pushing technique to the setting of IC3CFA. We started the implementation with a primitive variant in which after blocking a cube a push request was created for each successor of the location at the cube was blocked at. The main problem of this strategy was that the number of satisfiability checks was increased significantly which causes more inefficiently than improvements.

filter	checked	successful	rate
NONE	3.451.204	2.865.003	83%
ASSIGN	2,353,298	2,076,557	88%
PREDECESSOR	1,000,217	983,060	98%
RESTRICTIVE	799.974	799,974	100%

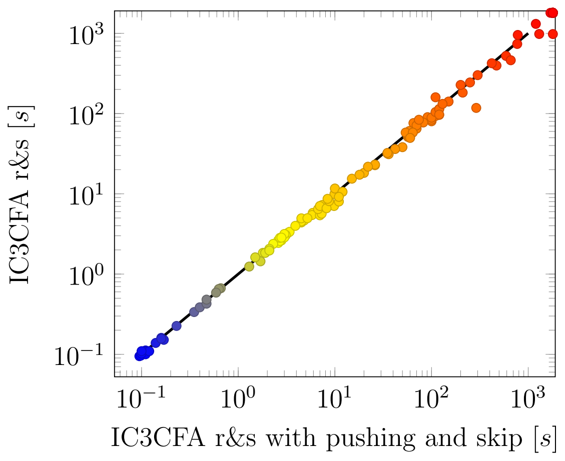
Figure 5.5: Comparison of various push filter strategies

In case no filter is enabled the algorithm creates about 3.4 billion push requests, about 83% of those requests are pushable. Enabling the filter which creates a request only if the cube literals are not assigned in the edge label, reduces the number of checked request to 2.3 billion, where 88% are successfully. Still we check nearly 200,000 request which are rejected. Filtering only those requests which have one predecessor yields a rate of 98% successful requests. The best rate is achieved by the restrictive filter which creates only those requests which have one predecessor and none of the cube literals is assigned in the edge label. The following benchmarks are made with the restrictive filter strategy. We compare two variants of pushing, first we do not skip the satisfiability check for pushing, which we will omit in the second configuration. Pushing without omitting the satisfiability check for the push request achieves already a higher score than the improved variant of IC3CFA. Both configurations, pushing and improved, solve the same number of programs, but one invalid program was not solvable with pushing but with the improved variant. Thus this pushing variant is able to solve one valid program which was not solvable before. Due to the benchmark scoring the score value is different despite the same number of solved programs. Furthermore both variants need more time to solve the same amount of programs. Enabling pushing without omitting the satisfiability checks causes obviously a significant higher number of checks. In case one omits the satisfiability check the total number of checks decreases to less than 40,000 while the improved variant still needs nearly 41,000 checks. The deviation can be explained by the fact that due to pushing some obligations might become skippable and do not cause any satisfiability checks in the inner loop.

Finally, we have done some experiments concerning the pruning of obligations in case of a successive push request. Therefore, the data structure of the Obligation-Queue was changed from a priority queue to a tree like structure. The consequences concerning the implementation are described in Sec. 4.5, we will now present a comparison of the old data structure without the functionality of pruning and the new



(a) Comparison of IC3CFA r&s vs IC3CFA r&s with pushing

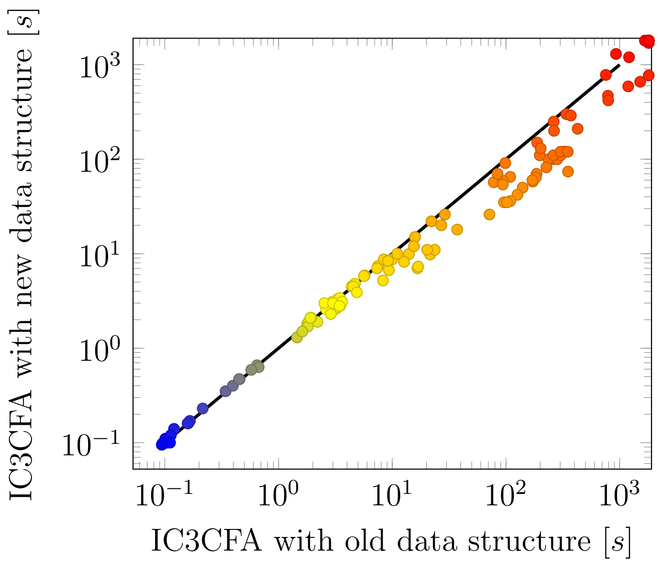


(b) Comparison of IC3CFA r&s vs IC3CFA r&s with pushing and skip

tool	score	solved	t solved	memory	# SAT
IC3CFA r&s	193	116/150	10,407	14,986	408,194
IC3CFA r&s with pushing	194	116/150	12,776	18,751	576,036
IC3CFA r&s with pushing & skip	194	116/150	11,916	15,733	396,158

Figure 5.6: Comparison of IC3CFA r&s vs IC3CFA r&s with pushing

structure which provides the functionality of pruning. We executed both variants with the same configuration and enabled pruning on the new structure. The new data structure is roughly 45% slower overall program than the old structure, furthermore it requires about 1 GB of memory more than before. The checks for pruning obligations are not outperformed by the effect of a pruned subtree in the ObligationTree. Furthermore, the number of valid programs which are solved decreases, as one can be seen on the score points.



tool	score	solved	t solved	memory
old structure	194	116/150	11,916	15,733
new structure	191	115/150	16,681	16,506

Figure 5.7: Comparison of different data structures managing obligations

6 Conclusion

The aim of this thesis was to efficiently reuse information which have already been learnt to improve IC3CFA. Therefore, we first integrated the reuse of obligations, such that once computed obligations are reused in the following iterations. Due to the reuse of obligations we were able to reduce the computation of predecessors as well as the required time to solve the programs. In addition, we achieved an improvement such that we are able to solve more programs than the original IC3CFA approach. On top of obligation reuse we implemented a skipping of satisfiability checks if the conditions are known to be satisfied a priori. This approach reduced the time requirement as well. Furthermore, the satisfiability checks were decreased by a significant amount. By proceeding our research, we found the effect of a lifted ObligationQueue and derived some new termination criteria from this fact. The effect of the new termination criteria was not as expected due to several cycles in the program containing all locations except initial and error locations. The reason for that can be found in the optimizations which are performed before the actual model checking takes place. Therefore, the upper bound of stable iterations was higher than the executed iterations of IC3CFA, but still on some programs we achieved improvements. Finally, we adapted the pushing technique of the original IC3 algorithm to our implementation of IC3CFA. While the naive adaptation of this technique was no improvement in our setting we refined some criteria of this technique. First we integrated a filter s.t. we reduced the number of push requests significantly and after more research being able to create only those request which can be performed without causing a satisfiability check. We also developed the concept of pruning obligations after a successful push request and expected improvements of this theoretical fact. The necessary change of the data structure managing the obligations did not outperformed the improvements of pruning obligations.

In future one could do more research on finding more indicators for reducing satisfiability checks, because these checks are expensive and consume significant time in model checking algorithms. Reducing the number of satisfiability checks improves the algorithm unless the effort the checking to omit the satisfiability check becomes too complex. Refining the upper bound of stable iterations to determine a terminating state before reaching a level with empty delta frames can also be part of future work. Combined with refining the upper bound one could try to reduce the length of cycles in the CFA as well. Finding a strategy to achieve a more minimized CFA could yield a lower upper bound for stable iterations, on the other hand a more minimized CFA causes more complex satisfiability checks for the solver. Thus one has to find a good balance between minimizing the CFA and still achieving not to complex satisfiability checks. Finally finding a competitive data structure combining the approach of obligation reuse with obligation skipping and pushing with pruning of obligations could be done in future work, because in theory the idea of pruning seems promising.

Bibliography

[BCC$^+$03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[BCG$^+$09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. *CoRR*, abs/0904.4709, 2009.

[Bey16] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses report on sv-comp 2016. In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*, pages 887–904, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[BW13] Dirk Beyer and Philipp Wendler. *Reuse of Verification Results*, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[CG12] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 277–293, Berlin, Heidelberg, 2012. Springer-Verlag.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 125–134, Austin, TX, 2011. FMCAD Inc.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 193–205, New York, NY, USA, 2001. ACM.

[GCS11] Arie Gurfinkel, Sagar Chaki, and Samir Sapra. Efficient predicate abstraction of program summaries. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 131–145, Berlin, Heidelberg, 2011. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[IG15] Alexander Ivrii and Arie Gurfinkel. Pushing to the top. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 65–72, Austin, TX, 2015. FMCAD Inc.

[JB10] Ivan Jager and David Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University CyLab, February 2010.

[Joh75] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[Lei05] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, March 2005.

[LNN15] Tim Lange, Martin R. Neuhäußer, and Thomas Noll. IC3 software model checking on control flow automata. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 97–104, 2015.

[McM03] K. L. McMillan. *Interpolation and SAT-Based Model Checking*, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[Sud13] Martin Suda. Triggered clause pushing for IC3. *CoRR*, abs/1307.4966, 2013.

[WK13] T. Welp and A. Kuehlmann. Qf bv model checking with property directed reachability. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 791–796, March 2013.