

OTDK-dolgozat

Dániel Kovács

2024.



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Artificial Intelligence and Systems Engineering

Efficient incremental verification strategies for transition systems

Scientific Students' Association Report

Author:

Dániel Kovács

Year IV

Advisor:

Milán Mondok,

PhD Student

2024.

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
1.1 The structure of this thesis	1
2 Background	3
2.1 Model checking	3
2.2 Modeling formalisms	4
2.2.1 STS	4
2.2.2 CFA	4
2.2.3 XSTS	5
2.3 SMT solvers	5
2.4 IC3	6
2.4.1 Frames	6
2.4.2 Examining frames	6
2.4.3 Blocking states from a frame	7
2.4.4 Main loop	7
2.4.5 Optimizations	8
2.4.5.1 Strengthening the blocking query	9
2.4.5.2 Removing parts of an expression, using the UNSAT-core	10
2.4.5.3 Propagating blocked states forward	10
2.4.5.4 Initializing the frames differently	11
2.4.5.5 Filtering the expressions	12
2.4.6 Reverse IC3	12
2.5 Theta	13

2.6	Related work	13
3	Efficient incremental verification strategies for transition systems	14
3.1	The Connected version of the algorithm	14
3.1.1	Making the IC3 algorithm more modular	15
3.1.1.1	CheckInitial	15
3.1.1.2	CheckFrameForPredecessors	15
3.1.1.3	TryBlock	15
3.1.1.4	Propagate	16
3.1.1.5	Controlling from the outside	16
3.1.2	Blocking in two directions	16
3.1.2.1	No connection between the two frames	18
3.1.2.2	Connection found between the two frames	18
3.1.2.3	Successfully blocking an expression from F_i	19
3.1.2.4	Unsuccessful blocking of B , looking for successors in RF	19
3.1.2.5	Successfully blocking RB	19
3.1.2.6	Both blockings are unsuccessful, the algorithm managed to find a counterexample	20
3.1.3	Combining Forward IC3 and Reverse IC3	20
3.1.4	Configuration options	21
3.2	Other optimizations	23
3.2.1	Blocking multiple expressions at once	23
3.2.2	Former frames optimization	24
4	Evaluation	25
4.1	Experiment planning	25
4.1.1	Research questions	25
4.1.2	Evaluation environment	25
4.1.3	Evaluation models	26
4.1.4	Parameters of the evaluation	26
4.2	Measurement results	28
4.2.1	RQ1: Effectiveness of the optimizations from the literature	28
4.2.2	RQ2: Effectiveness of my algorithmic contributions	29
4.2.3	RQ3: Comparing the connected IC3 to the reverse IC3 and the forward IC3	29

4.2.4 RQ4: Analysing the configuration options of the connected IC3 algorithm	31
5 Conclusions	33
Bibliography	34
Appendix	36
A.1 Further benchmarking results	36

Kivonat

Az elmúlt évtizedekben az emberiség által használt hardver- és szoftverrendszerek száma és összetettsége meredeken emelkedett. Az ilyen, nagy komplexitású rendszerekbe és a bennük található komponensekbe a tervezés és fejlesztés során könnyen kerülhetnek különféle hibák. Az olyan rendszereket, ahol ezek a hibák akár súlyos következményekkel is járhatnak, biztonságkritikus rendszereknek nevezzük. Biztonságkritikus rendszerek gyakoriak többek közt a vasútipar, a járműipar, a bankszektor vagy akár az energiaipar területén is. Ezekben az iparágakban a legkisebb hiba is emberéletek elvesztését, vagy dollármilliárdokban mérhető károkat okozhat, emiatt fontos, hogy a fejlesztési folyamat során mindent megtegyünk a hibák felderítése és elkerülése érdekében.

Formális verifikáció alkalmazása lehetővé teszi, hogy biztonságkritikus rendszerek helyességéről matematikai módszerekkel érveljünk már a fejlesztés korai fázisai során is. A formális verifikáció egyik módja a modellellenőrzés, amely képes egy rendszer összes lehetséges viselkedését kimerítően felderítve a rendszer hibamentességét bizonyítani, illetve a hibákat megtalálni. Ez egy nagyon számításigényes feladat, mert a vizsgált rendszer változói számának növekedésével a lehetséges állapotainak száma exponenciális mértékben növekedhet. A modellellenőrző algoritmusok hatékonyságának javítása fontos a rendszerek helyességének biztosítása érdekében. A beágyazott rendszerek formális verifikációjának területén az utolsó nagy előrelépést az IC3 algoritmus kidolgozása jelentette, mely a rendszer helyességének ellenőrzését inkrementálisan, számos egyszerű logikai lemma indukciós bizonyításán keresztül végzi. Az IC3 egy korszerű modellellenőrző algoritmus, mely kritikus hardverek és szoftverek alacsony szintű, bitprecíz verifikációjára képes.

Dolgozatomban bemutatok általam kidolgozott új verifikációs stratégiákat és algoritmikus optimalizációkat, melyekkel az IC3 algoritmus még több hardver- és szoftvermodell ellenőrzésére válik alkalmassá. Az általam kidolgozott verifikációs keretrendszer konfigurációs opciói lehetővé teszik, hogy szakértők az IC3 algoritmust a saját területük problémáihoz finomhangolják. A kidolgozott algoritmusokat és optimalizációkat a nyílt forráskódú Theta keretrendszerben implementáltam és valódi ipari példákon értékeltem ki egy részletes mérési kampány során.

Abstract

In recent decades, the number and complexity of hardware and software systems used by humanity have rapidly increased. During the design and development of such high-complexity systems and their components, various errors can easily occur. Systems where these errors could lead to severe consequences are referred to as safety-critical systems. Safety-critical systems are common in the railway, automotive, banking and energy industries. In these sectors, even the smallest error could result in loss of life or cause damage measurable in billions of dollars. This makes it crucial to do everything possible during the development process to detect and avoid errors.

The application of formal verification allows us to mathematically reason about the correctness of safety-critical systems, even in the early phases of development. One method of formal verification is model checking, which can exhaustively explore all possible behaviors of a system to either prove its correctness or find errors. This is a highly computationally demanding task because the number of possible states of the system can increase exponentially with the number of variables. Improving the efficiency of model checking algorithms is essential to ensure system correctness. The last major advancement in the formal verification of embedded systems was the development of the IC3 algorithm, which incrementally verifies system correctness by inductively proving numerous simple logical lemmas. IC3 is a modern model checking algorithm capable of low-level, bit-precise verification of critical hardware and software.

In my thesis, I present novel verification strategies and algorithmic optimizations I have developed, which make the IC3 algorithm applicable for verifying an even wider range of hardware and software models. The configuration options in the verification framework I developed enable experts to fine-tune the IC3 algorithm to the problems specific to their fields. I implemented the algorithms and optimizations in the open-source Theta framework and evaluated them on real-world industrial examples in a detailed measurement campaign.

Chapter 1

Introduction

In recent decades, the number and complexity of hardware and software systems used by humanity have rapidly increased. During the design and development of such high-complexity systems and their components, various errors can easily occur. Systems where these errors could lead to severe consequences are referred to as safety-critical systems. Safety-critical systems are common in the railway, automotive, banking, and energy industries. In these sectors, even the smallest error could result in loss of life or cause damage measurable in billions of dollars. This makes it crucial to do everything possible during the development process to detect and avoid errors.

The application of formal verification allows us to mathematically reason about the correctness of safety-critical systems, even in the early phases of development. One method of formal verification is model checking, which can exhaustively explore all possible behaviors of a system to either prove its correctness or find errors. This is a highly computationally demanding task because the number of possible states of the system can increase exponentially with the number of variables. Improving the efficiency of model checking algorithms is essential to ensure system correctness. The last major advancement in the formal verification of embedded systems was the development of the IC3 algorithm, which incrementally verifies system correctness by inductively proving numerous simple logical lemmas. IC3 is a modern model checking algorithm capable of low-level, bit-precise verification of critical hardware and software.

In my thesis, I present novel verification strategies and algorithmic optimizations I have developed, which make the IC3 algorithm applicable for verifying an even wider range of hardware and software models. The configuration options in the verification framework I developed enable experts to fine-tune the IC3 algorithm to the problems specific to their fields. I implemented the algorithms and optimizations in the open-source Theta framework and evaluated them on real-world industrial examples in a detailed measurement campaign.

1.1 The structure of this thesis

In Chapter 2, I present the theoretical background of my work. In Chapter 3, I present new optimization strategies for the IC3 algorithm, which make the IC3 algorithm applicable for verifying an even wider range of hardware and software models. In Chapter 4, I evaluate my

work on real-world industrial examples in a detailed measurement campaign. In Chapter 5, I conclude my work and suggest new options for further improvement.

Chapter 2

Background

2.1 Model checking

Model checking [11] is a formal method that exhaustively explores every possible behavior of an input model, and then based on the information it gathered, it is capable of determining the model's safety or faultyness. It is capable of finding mistakes in input models, and it is capable of proving, with mathematical certainty, that the input model is free of any errors.

Input models

Because model checking has a formal, precise mathematical definition, it only accepts input models, which have formal semantics as well. *Symbolic Transition System* (STS) [14], *Control Flow Automata* (CFA) [2], and *Extended Symbolic Transition System* (XSTS) [19] are such low-level formalisms.

Programming languages like C or Java are a higher-level formalism, which means that they don't have precise semantics and their direct verification is not efficient. If a model is in a higher-level formalism, and it needs to be formally verified, then it is usually transformed into a lower-level formalism. Programming languages are usually transformed into CFA-s for example.

There are formalisms, which describe hardware models as well. Such formalisms are BTOR2 and AIGER [4]. During the evaluation of my algorithm, I used AIGER models, after I transformed them into STS models.

Requirement property Requirement properties give a formal characterization of our expectations of the model. They are given as input alongside the input models, so the model checker can decide if the model violates the property or not. In this thesis, I use safety properties to differentiate between safe and unsafe models. A safety property defines the set of unsafe states, and the model is safe if no unsafe states are reachable.

State space explosion One of the biggest challenges in model checking is state space explosion. If the number of state variables grows in a model, then the state space starts to grow

exponentially as well. To combat this, model checkers use symbolic techniques, such as the IC3 algorithm [6].

2.2 Modeling formalisms

In this section, I present modeling formalisms that I used.

2.2.1 STS

A *Symbolic Transition System* (STS) [14] represents states, transitions, initial states, and the safe states of a system using first-order logic formulas [16]. In this thesis let $V = \{v_1, v_2, \dots, v_n\}$ notation denote the set of variables in a model. I will use the $V' = \{v'_1, v'_2, \dots, v'_n\}$ notation to indicate the primed variables of a system. A state in the model is a variable assignment to each variable $v \in V$. In [14], $\text{STS} = (\text{Init}, \text{Tran}, \text{Property})$ is a tuple, where

- *Init* is the initial state formula over V , which defines the set of initial states in the model.
- *Tran* describes the transition formula over $V \cup V'$. It tells, that there is a transition step between the current state (V) and the next state (V').
- *Property* is the property of the model over V , which describes every safe state in the model. If a state is not described by the property formula, then it is an unsafe state.

A path is a finite sequence of states $\sigma = v_1, v_2, \dots, v_n$, where v_1 is an initial state, and for which $(v_1, v_2, \dots, v_n) \models \text{Init} \wedge \bigwedge_{1 \leq i \leq n} \text{Tran}$ is true. The path starts in an initial state and the following states satisfy the transition formula.

A state v_i is reachable in the model if a path exists, where v_i is the last state. A model is considered safe if every reachable state satisfies the property formula. In other words, there are no reachable states, where the property formula does not stand.

In this thesis, I will use the notation I to indicate the initial states of the model, T to indicate the transitions of a model, and P to indicate the property of a model.

2.2.2 CFA

Programming languages are designed to be easily read and written by humans. However, they are usually not precise enough to be used in model checking algorithms. They have to be transformed first into lower-level, more formal languages. Control flow automation (CFA) [2] is a formal language, which higher-level programming languages (like C), can be easily transformed into. This transformation can be done automatically see [20].

I define a *Control flow automaton* as a tuple $\langle V, L, l_0, l_{end}, l_{err}, E \rangle$, where:

- $V = \{v_0, v_1, \dots\}$ is the set of *variables*. $\forall v_i \in V$ variable has a D_{v_i} domain, which contains the possible values of v_i ;

- $L = \{l_0, l_1, \dots\}$ is the set of *control locations*. A control location is a possible value for the program counter.
- $l_0 \in L$ is the initial location;
- $l_{end} \in L$ is the ending location;
- $l_{err} \in L$ is the error location;
- $E \subseteq L \times Ops \times L$ is the set of *transitions*. A transition is a directed edge in the CFA, with a starting location, an ending location, and one operation. An operation ($op \in Ops$) can be:
 - An *assumption* statement ($[condition]$). This transition can only fire if the *condition* of the transition evaluates to true in the starting location.
 - A deterministic *assignment* statement ($v_i := expression$). *Expression* is evaluated and the result becomes the new value of v_i .
 - A non-deterministic *havoc* statement (*havoc* v). Variable v_i is assigned a random value from its domain.

A path is a sequence of transitions, which starts from the initial state. $\sigma = (l_0, op_1, l_1), (l_1, op_2, l_2), \dots, (l_{n-1}, op_n, l_n)$. A location l_i is reachable if there exists a path, where l_i is the last state. A CFA model is safe if the error location is unreachable.

CFA-s can be transformed into STS models. During the evaluation of my algorithm, when I used CFA models, I first converted them into STS models.

2.2.3 XSTS

Extended Symbolic Transition System (XSTS) [19] is a formal language, which is bridging the gap between the higher-level modeling languages used by system engineers like SysML and the lower-level formal languages, which can be checked by model checking algorithms.

XSTS can also be transformed into STS. While evaluating my algorithm, I initially converted the XSTS models into STS models before checking them.

2.3 SMT solvers

During this thesis, I heavily rely on SMT solvers [16] to reason about the satisfiability of first-order logic problems. First-order logic is generally undecidable, but several subsets (background theories) exist, which are decidable in practice. Among other things, these include Integer, Array, and Floating-point theories.

Solving these problems in theory is an NP-hard problem, but there are solvers, which are quite effective against most problems in practice. These solvers include: Z3 [12], Princess[7], Mathsat[9]. During the implementation of my algorithm, I used the Z3 solver.

2.4 IC3

IC3 [6] is a model checking algorithm, that analyzes the system incrementally, using small SMT solver calls, and eventually proves if the given model is safe, meaning that the property always stands in every reachable state, or the transition system is unsafe, meaning that, there is a viable path from the initial states to a faulty state, which violates the property. I based my research on the IC3 algorithm on [21] and [23]

2.4.1 Frames

The IC3 algorithm stores every information it has on the investigated model in frames. F_i means the i -th frame. In F_i we store a conjunction of literals [11], which overapproximate the states reachable in i steps or less from the initial states. It means that F_i contains every state that is reachable in i steps or less than i steps, and there can be other states in it as well.

- In F_0 we store the initial states.
- In F_1 , we store an overapproximation of the states reachable in at most one step.
- In F_2 we store those states, that overestimate the states reachable in two or fewer steps.
- Etc.

While the algorithm is running, we always maintain that $F_i \subseteq F_j$ if $i < j$

2.4.2 Examining frames

When the algorithm examines a frame, it needs to determine, if the current frame intersects with the faulty states or not. It does this by running the SMT query $F_i(x) \wedge \neg P(x)$. If the query returns with SAT and returns an expression, which I will call B , then there are two options.

- At least one state in B is reachable in i steps from the initial states. If this is the case, then the examined model is faulty and the algorithm found a genuine i -length counterexample.
- No state in B is reachable in i steps and B is only in F_i , because of the overapproximation. If this is the case, the algorithm can remove these states from F_i , without the violation of the definition of F_i . In this scenario, it has to remove B from the previous frames too, so $F_i \subseteq F_j$ will still stand for every $i < j$. This is sound to do, because if a state is not reachable in at most j steps then it is not reachable in less than j steps either.

The algorithm must arbitrate, which option is true. The following function, which solves this problem and removes the investigated states if necessary, is called blocking. The function is described in detail in Subsection 2.4.3.

If the blocking was successful, then the algorithm continues to inspect the frame, until it proves that F_i does not intersect with $\neg P$. In that case, the algorithm has proven, that the model can not reach a bad state in i steps. After the inspection of a frame is completed, the algorithm moves on to the next frame and starts to examine it.

2.4.3 Blocking states from a frame

The function gets two input arguments: a set of states (B) described by an expression, which we want to block, and an index, which shows from which frame we want to block them. Together they are called a proof obligation.

If the function is called with the index of 0, then the function can return that the blocking was unsuccessful, because the initial states have been reached. This is the case, where the algorithm has proven that B cannot be blocked from F_i because it is reachable in i steps (meaning the model is unsafe, and the algorithm has found a genuine counterexample, which starts from the initial states and ends in a faulty state).

If in B there is a state that is reachable in i steps, then it has to have a predecessor in F_{i-1} . Let's call the expression, which describes these predecessor states PB . PB exists in this case, because if a state in B is reachable from the initial states in i steps, then there has to be a state, that is reachable in $i - 1$ steps from the initial states, and from that state, a state in B is accessible in one step. PB is reachable in $i - 1$ steps from the initial states and that means, that it is in F_{i-1} , because of the definition of F_{i-1} .

The algorithm tries to find PB , by checking the SMT formula $F_{i-1}(x) \wedge T(x, x') \wedge B(x')$. If the formula is UNSAT, then it means that it is not possible to reach B from F_{i-1} using one transition step, so the function has proven that B has no predecessors in F_{i-1} , and the blocking was successful and it can remove B from F_i and from the previous frames. If it returns SAT and the solver returns PB , then the algorithm has to decide if PB is indeed reachable in $i - 1$ steps. It decides this, by recursively calling the blocking function with the arguments PB and $i - 1$. If it is reachable in $i - 1$ steps, then the blocking will be unsuccessful and the algorithm has proven that B has a predecessor, which is reachable in $i - 1$ steps, meaning that it is reachable in i steps and we can return with this information. If the blocking of PB was successful, then the blocking function has already removed PB from F_{i-1} , which means the algorithm can once again check for other predecessors using the $F_{i-1}(x) \wedge T(x, x') \wedge B(x')$ formula.

This function can end in two ways:

- A blocking of a predecessor was unsuccessful because the initial states have been reached and the function can return with the information that B is reachable in i steps. In this case, the function returns *false*.
- The formula $F_{i-1}(x) \wedge T(x, x') \wedge B(x')$ returns UNSAT, after every predecessor of B was blocked from F_{i-1} and it proves that B is not reachable in i steps. In this case, the function removes B from F_i and from the previous frames and returns *true*.

I provide the pseudo-code of the blocking function in Algorithm 1.

2.4.4 Main loop

During the execution of the algorithm, it first determines if the initial states violate the property or not. If they violate, then it can terminate with the unsafe result.

Algorithm 1 *TryBlock*

Input: An expression, which has to be blocked: B , and an index i

Output: Whether the blocking of B was successful or not

```
1: if  $i$  equals 0 then
2:   return false ▷ the initial states have been reached
3: end
4: while true do
5:   if  $F_{i-1}(x) \wedge T(x, x') \wedge B(x')$  is satisfiable then
6:      $PB \leftarrow$  The satisfying solution, provided by the solver
7:     if not TryBlock( $PB, i - 1$ ) then
8:       return false ▷ the blocking of  $PB$  was unsuccessful
9:     end
10:  else
11:    for  $F_j \in (F_1..F_i)$  do
12:      Add  $\neg B$  to  $F_j$ 
13:    end
14:    return true ▷  $B$  has no predecessors in  $F_{i-1}$  anymore
15:  end
16: end
```

If this is not the case, then the main loop of the algorithm starts. Each iteration examines a frame. If the algorithm finds a counterexample, which means that, the blocking of a set of states was unsuccessful, then the algorithm stops with the result unsafe. If the algorithm manages to block every faulty state from the frame, then it can move on to the next frame and start to inspect it too. Before it moves on to the next frame, it checks if the currently finished frame has become identical to the frame before it. If the two frames are identical, then the algorithm proves, that the model is safe.

F_{i-1} is an overapproximation of the states available in $i - 1$ steps. If F_{i-1} becomes equal with F_i , which is an overapproximation of the states available in i steps, then it means that in F_{i-1} , there are no transitions, which lead outside of F_{i-1} . The algorithm knows that in F_{i-1} there are no faulty states at this point and $F_0 \subseteq F_{i-1}$, because of the definition of the frames, which means that every initial state is in F_{i-1} . Because every initial state is in F_{i-1} and there are no faulty states in it, and there are no outside leading transitions, then it means that the model is safe, because every reachable state is in F_{i-1} and every state in F_{i-1} is safe.

To decide if F_{i-1} equals F_i , the algorithm has to prove that $F_{i-1} \Rightarrow F_i$ and $F_{i-1} \Leftarrow F_i$. $F_{i-1} \subseteq F_i$ proves $F_{i-1} \Leftarrow F_i$, so it only has to prove $F_{i-1} \Rightarrow F_i$. This can easily be done by checking $\neg F_{i-1}(x) \wedge F_i(x)$. If the solver returns UNSAT, then it means that the two frames have become equals.

The pseudo-code of the main loop of the IC3 algorithm can be seen in Algorithm 2.

2.4.5 Optimizations

In this section, I will detail different, already-established optimizations from the literature for the IC3 algorithm, which I implemented in the Theta framework.

Algorithm 2 Main loop of the IC3 algorithm

Input: A symbolic transition system $(I(x), T(x, x'), P(x))$

Output: Whether the transition system is safe

```
1: if  $I(x) \wedge \neg P(x)$  is satisfiable then
2:   return unsafe
3: end
4:  $F_0 \leftarrow I(x)$ 
5:  $F_1 \leftarrow true$ 
6:  $i \leftarrow 1$ 
7: while true do
8:   if  $F_i(x) \wedge \neg P(x)$  is satisfiable then
9:      $B \leftarrow$  The satisfying solution, provided by the solver
10:    if not  $TryBlock(B, i - 1)$  then
11:      return unsafe ▷ failed to block a faulty state
12:    end
13:  else
14:    if  $F_i$  equals  $F_{i-1}$  then
15:      return safe ▷ equal frames found
16:    else
17:       $i \leftarrow i + 1$  ▷ inspection of a frame is finished, move on to the next one
18:       $F_i \leftarrow true$ 
19:    end
20:  end
21: end
```

2.4.5.1 Strengthening the blocking query

When the algorithm is trying to block a (B, i) pair from F_i it checks, if the query $F_{i-1}(x) \wedge T(x, x') \wedge B(x')$ is SAT. This query can be strengthened, by adding $\neg B(x)$ to it. The resulting $F_{i-1}(x) \wedge \neg B(x) \wedge T(x, x') \wedge B(x')$ query is harder to be satisfiable, which means that the blocking process will stop earlier more often.

The algorithm remains correct because, if the strengthened $F_{i-1}(x) \wedge \neg B(x) \wedge T(x, x') \wedge B(x')$ query is UNSAT and the original $F_{i-1}(x) \wedge T(x, x') \wedge B(x')$ query is SAT, then it means that the original query found a predecessor of B in F_{i-1} , which is also in B . Let's call this predecessor in F_{i-1} PB . Because the strengthened query is UNSAT, then that means that B only has predecessors in F_{i-1} , which are a part of B . The transition relation remains the same and $F_{i-2} \subseteq F_{i-1} \subseteq F_i$, so that means, that the query $F_{i-2}(x) \wedge T(x, x') \wedge PB(x')$ can only return states, which are a part of B as well. Using this logic, B will only have predecessors in the initial states in i transition steps, if part of B intersects with the initial states.

If B was created, because the currently inspected frame intersected with the faulty states, then B can not contain initial states because, at the beginning of the algorithm, it has already checked for faulty states in the initial states. If B was created, because it was a predecessor of another set of states, which needed to be blocked, then it also can not intersect with the initial states. If that were the case, then B would be the start of a counterexample, with the length of $k - i$, where k is the index of the currently examined frame in the main loop and i is the index number

of B . At this point, it was already proven, that a counterexample of this length can not exist in this input model.

Because B does not intersect with the initial states, it means that the algorithm can block B from F_i , if it only has predecessors in F_{i-1} , which are part of B . It means that $\neg B(x)$ can be added to the blocking query and the correctness of the algorithm won't change.

2.4.5.2 Removing parts of an expression, using the UNSAT-core

When the algorithm blocks an expression from a frame, it does it because it is not reachable from the initial states in the given steps. The algorithm learns it, after the query $F_{i-1}(x) \wedge \neg B(x) \wedge T(x, x') \wedge B(x')$ becomes UNSAT. Because B is a conjunction of literals, it is entirely possible, that if the algorithm removed one or more literals from B , the solver would still return UNSAT. This would be beneficial, because if the algorithm can remove literals from B , then it will block a larger part of the frame, which will further accelerate the algorithm.

Most SMT solvers (including Z3) can return an UNSAT-core, whenever it returns with the UNSAT result. The UNSAT-core contains a core contradiction to the query. It is possible, that not every literal, which is in B is in the UNSAT-core. If a literal is not in the UNSAT-core, then the algorithm can attempt to remove it from B . However if after the removal, B intersects with the initial states, then the literal can not be removed from B , because the initial states are reachable. The pseudo-code of this process can be seen in Algorithm 3.

Algorithm 3 Removing literals, using the UNSAT-core

```

1: for every  $c$  literal in  $B$  do
2:   if  $c \notin$  UNSAT-core then
3:     Remove  $c$  from  $B$ 
4:     if  $B(x) \wedge I(x)$  is SAT then
5:       Put  $c$  back to  $B$        $\triangleright$  If  $c$  is removed from  $B$ ,  $B$  would intersect with the initial
        states
6:     end
7:   end
8: end

```

2.4.5.3 Propagating blocked states forward

Being able to block more and more states from different frames is crucial, in order to make the algorithm faster and more efficient. If the algorithm can block more states from its frames, then it can lead to a quicker termination.

If the algorithm manages to block a set of states from F_i , then they are not reachable from the initial states in i steps. It is also possible, that they are not reachable in $i + 1$ steps and they can be blocked from F_{i+1} too. This optimization makes it possible to propagate these blocked states and exclude them from other frames as well. Let us call the expression, that describes these states B . If B is blocked from F_i , it means, that F_i contains the expression $\neg B$. If the SMT formula $F_i(x) \wedge T(x, x') \wedge B(x')$ returns UNSAT, then it means, that it is not possible to go from F_i in one step to any state, which is in B . So, we can add $\neg B$ to F_{i+1} too. It is possible, that

once it was not feasible to block B from F_{i+1} , but after a while when the algorithm managed to block more and more states from F_i and F_{i+1} , it became possible. So it might be beneficial, to check this formula from time to time, until the SMT query becomes UNSAT, and the algorithm can block B from F_{i+1} .

In most IC3 implementations, including mine, this propagation happens after the algorithm manages to exclude any faulty states from a frame, and it prepares to move to the next one. The implementation of this is quite simple, and intuitive. The algorithm checks every frame from F_1 to F_i and inspects if a set of states, which are blocked from the frame can be blocked from the next frame as well. It is also possible, that during the propagation two frames became equal, and the algorithm can stop, with the safe result. The pseudo-code of the propagation can be seen in Algorithm 4. If the optimization is turned on, then this code replaces the code from lines 14-19 in Algorithm 2.

Algorithm 4 Propagate

```

1: for  $F_j \in (F_1..F_i)$  do
2:   for  $\neg B \in F_j$  do
3:     if  $F_j(x) \wedge T(x, x') \wedge B(x')$  is UNSAT then
4:       Add  $\neg B$  to  $F_{j+1}$ 
5:     end
6:     if  $F_j$  equals  $F_{j+1}$  then
7:       return safe
8:     end
9:   end
10: end
11:  $i = i + 1$  ▷ inspection of a frame is finished, move on to the next one
12:  $F_i = \text{true}$ 

```

2.4.5.4 Initializing the frames differently

The IC3 algorithm initializes every frame with true, which means the whole state space. This can be wasteful, so it could be beneficial to initialize every frame to be more specific. This optimization initializes every newly created frame with P .

To be able to initialize F_1 with P , the algorithm first needs to prove that no faulty states are reachable in 1 step. This can be done by checking the SMT call $I(x) \wedge T(x, x') \wedge \neg P(x')$. If the solver returns SAT, then the algorithm has found a 1-step counterexample. If the solver returns UNSAT, then the algorithm can initialize F_1 with P , because it is proven, that no faulty states can be reached in one transition step. After this initialization, the main loop of the algorithm can start.

If this optimization is enabled, then the algorithm will not look for intersections between the currently examined frame and $\neg P$. Instead, it will check, if the faulty states are reachable from the currently examined frame. It does this by checking the SMT formula $F_i(x) \wedge T(x, x') \wedge \neg P(x')$. If the solver returns SAT and returns with B , then the algorithm has to block B from the frame, because it could be a second to last step in a possible counterexample. If it manages to block B and every other predecessor of $\neg P$, from F_i , then the algorithm managed to prove, that in

$i + 1$ steps no faulty states are reachable. After this, it can initialize F_{i+1} with P , and start to examine it.

2.4.5.5 Filtering the expressions

When a blocking $F_i(x) \wedge T(x, x') \wedge B(x')$ SMT query is satisfiable, then the solver returns an expression PB . PB satisfies the query, but it assigns a value to every variable, which was in the query. It is entirely possible, that a variable v_k -s value does not matter in PB , meaning that v_k can take any value in its domain and PB would still satisfy the blocking query. This optimization filters this PB expression so every variable, which can take any value in its domain, will be filtered out. This means, that the algorithm will block a larger part of the frame, which can accelerate the algorithm.

The algorithm checks this condition by checking if setting v_k to any different value than it was assigned in PB can make the query unsatisfiable. If it finds that all assignments of v_k satisfy the query, then the variable did not influence the result of the original blocking query, and it can be filtered out of PB . If the query is UNSAT for any assignment, then v_k cannot be removed from PB because it influences the query. Using this optimization, the algorithm will check every variable in PB and examine if they can be left out.

2.4.6 Reverse IC3

The IC3 algorithm starts from the initial states, and through continuous overestimations of the forward reachable states, it tries to determine whether the current model is faulty or not. If it finds a counterexample, then it finds a viable path from the initial states to a state that violates the given property, concluding that the model is faulty. It is also possible that the algorithm finds two identical frames, which proves that the model is safe.

It is also possible to go backward and start from the states that violate the property and then try to reach the initial states. It means that for the initialization of F_0 , $\neg P$ will be used, and in this *reverse* IC3 the goal is to reach I or prove that it is unreachable. Of course, this means that the reverse IC3 can not use forward transition relations, and the transitions have to be reversed before the algorithm starts.

In [21] is shown, that when the reverse version of the algorithm is used alongside the forward version, it complements it, meaning that there are some problems, that the forward IC3 algorithm can not solve in a reasonable amount of time, and the reverse version can, and vice versa, there are problems, which the forward version can solve and the reverse version can not in the given timeframe. It means that the best approach, which solves the most amount of problems is if the two algorithms are used alongside each other, which gives the best possible chance to prove a model's safety or faultiness.

2.5 Theta

Theta [24] is "a generic, modular and configurable model checking framework developed at the Critical Systems Research Group of Budapest University of Technology and Economics, aiming to support the design and evaluation of abstraction refinement-based algorithms for the reachability analysis of various formalisms. The main distinguishing characteristic of Theta is its architecture that allows the definition of input formalisms with higher-level language front-ends, and the combination of various abstract domains, interpreters, and strategies for abstraction and refinement. Theta can both serve as a model checking backend and also includes ready-to-use, stand-alone tools" [13].

I implemented the algorithm and my optimizations to it in the Theta framework.

2.6 Related work

Bounded Model Checking One (BMC) [10] is a model checking algorithm, which unrolls the cycles in the input model up to a given bound and increases this bound until it finds a counterexample. One weakness of this algorithm is that it can't prove the safety of models, only their faultyness.

K-induction [22] complements BMC by being able to stop with a safe result. It realizes that the model is safe by proving the inductivity of the transition formula and the property.

IMC [18] also complements the BMC algorithm. It tries to overestimate the state space using interpolation in an attempt to find a fixed point where it can prove that the model is correct.

In [17], Lange et al. show a method to more efficiently apply the IC3 algorithm to software models by maintaining reachability information specifically for each control location. It does not have one global frame list, but instead a separate frame list for every control location.

In [21], et al. show a method to connect the forward and the reverse versions of the IC3 algorithm. One difference compared to my approach is that the variation presented in [21] switches between the two directions using a timer function. In my version, the algorithm switches after a given number of blocking attempts. The information exchange between the two directions is also different. [21]'s version passes proof obligations between the two versions. Meanwhile, my approach uses the overapproximations maintained by the nested IC3 algorithms as safety properties in the opposing direction's nested IC3. I will explain my approach in Chapter 3.

[15] presents a reformulation of IC3 that separates reachability checking from induction reasoning. This makes it possible to integrate IC3 and k-induction into a combined algorithm.

[23] presents an implementation of the forward IC3 algorithm, which uses implicit predicate abstraction to handle variables with large domains. This can significantly reduce the state space of the model to counter the state-space explosion problem.

Chapter 3

Efficient incremental verification strategies for transition systems

In Chapter 2, I introduced the IC3 algorithm and its several optimizations from the literature. In this chapter, I will present my theoretic contributions and introduce novel algorithmic optimizations, which will further improve the efficiency of the algorithm, and make it possible to verify an even wider range of hardware and software models.

3.1 The Connected version of the algorithm

In 2.4 I introduced the IC3 algorithm and in 2.4.6 I showed, that the algorithm can start from the other side of the state space as well: starting from the faulty states and trying to reach the initial states using the reversed transitions in an effort to prove a model's safety. The two versions of the algorithm complement each other, meaning that a portfolio approach using both of these algorithms can be the best option if the goal is to solve the most amount of problems. Using the portfolio approach, we start both algorithms at the same time and wait for one of them to finish.

In [21] is shown a way to connect these two algorithms together and use them in a combined approach in an effort to improve the algorithm's efficiency further. [21] executes both algorithms alternatively and connects the two, by sharing proof obligations between the two versions. My approach is different from it because of two modifications, which fundamentally change the nature of the connection.

- The variant presented in [21] switches between the two directions, using a timer function, meaning that the algorithm changes directions after a predefined time. In my version, the algorithm switches after a given number of blocking attempts.
- The information exchange between the two directions is also different. [21]'s version passes proof obligations between the two versions. Meanwhile, my approach uses the overapproximations maintained by the nested IC3 algorithms as safety properties in the opposing direction's nested IC3. I will explain my approach in detail below.

Firstly, in 3.1.1 I will show how I dismantled the algorithm in an effort to make it more modular and versatile. The refactoring makes it possible to reach into the algorithm from the outside, modify certain variables, and stop and restart it at will, after each blocking. Then I will describe in detail in 3.1.2 and 3.1.3, how I connected the versions of the algorithm together, using the modular functions. Lastly, I will show configuration options in 3.1.4, which will enable experts to fine-tune the IC3 algorithm to their fields.

During this explanation, I will use every optimization, which was described in 2.4.5

3.1.1 Making the IC3 algorithm more modular

As a preliminary for introducing the connected IC3, I will first describe the functions of the modular IC3. Then I will show how the algorithm can be controlled from the outside using these functions.

3.1.1.1 CheckInitial

Initialises F_0 and checks if F_0 intersects with $\neg P$. It also checks if the faulty states are reachable from the initial states in one transition step. This check is needed, because the algorithm initializes the frames differently as it was described in 2.4.5. The function returns *true* if it found any faulty states in the initial states, or if a faulty state is reachable in one step. It returns *false* otherwise. The pseudo-code of this function can be seen in Algorithm 5.

Algorithm 5 CheckInitial

```

1: if  $I(x) \wedge \neg P(x)$  is satisfiable then
2:   return true ▷ The initial states intersect with the faulty states
3: end
4: if  $I(x) \wedge T(x, x') \wedge \neg P(x')$  is satisfiable then
5:   return true ▷ A faulty state is reachable in 1 step from the initial states
6: end
7: return false ▷ Initial checks completed

```

3.1.1.2 CheckFrameForPredecessors

It gets a *target* expression, and a frame index i as its input. It checks if the target expression is reachable from the frame in one transition step. If it is reachable, then it returns an expression, from which the target expression is reachable in one transition step. If from the current frame, the target expression is not reachable in one transition step, then it returns an empty expression. The pseudo-code of this algorithm can be seen in Algorithm 6.

3.1.1.3 TryBlock

Tries to block a B expression from the i -th frame. It returns *true* if the blocking was successful, and it managed to block the expression. It returns *false* if the blocking was unsuccessful. The pseudo-code of this function is the same, as it was in Algorithm 1.

Algorithm 6 CheckFrameForPredecessors

Input: A *target* expression, and a frame index i

Output: A predecessor of *target* or an empty expression

```
1: if  $F_i(x) \wedge T(x, x') \wedge \text{target}(x')$  is satisfiable then  
2:    $B \leftarrow$  A satisfying expression provided by the SMT solver  
3:   return  $B$   $\triangleright$  The target expression is reachable from the frame in one transition step  
4: end  
5: return an empty expression
```

3.1.1.4 Propagate

Tries to propagate blocked states forward. If it finds two identical frames during the propagation, then it returns *true*. Otherwise, it returns *false*. The pseudo-code of this algorithm can be seen in Algorithm 7.

Algorithm 7 Propagate

```
1: for  $F_j \in (F_1..F_i)$  do  
2:   for  $\neg B \in F_j$  do  
3:     if  $F_j(x) \wedge T(x, x') \wedge B(x')$  is UNSAT then  
4:       Add  $\neg B$  to  $F_{j+1}$   
5:     end  
6:     if  $F_j$  equals  $F_{j+1}$  then  
7:       return true  
8:     end  
9:   end  
10: end  
11:  $i = i + 1$   $\triangleright$  inspection of a frame is finished, move on to the next one  
12:  $F_i = \text{true}$   
13: return false
```

3.1.1.5 Controlling from the outside

In Algorithm 8, I show how the IC3 algorithm can be controlled from the outside using the modular functions described before. With this code-segment, I demonstrate that the original IC3 algorithm can be reconstructed using the modular functions of the dismantled algorithm. The behavior of the IC3 algorithm does not change from this modification.

3.1.2 Blocking in two directions

Using the modular version of IC3, which I defined in 3.1.1, I will show how I connected the two versions of the algorithm together.

In this chapter, I will use the notation F_i to indicate a frame of the forward-stepping IC3 and I will use the notation RF_i to indicate a frame of the reverse-stepping version of IC3. I will also use illustrations to explain the algorithm's inner work (like Figure 3.2). In the illustrations, I use ovals to indicate the states, which are contained in a specific frame. In the illustrations $(F_0, F_1...F_i)$ represent the forward frames, and $(RF_0, RF_1...RF_j)$ represent the reverse frames. Because of the frames definition F_0 equals I and RF_0 equals $\neg P$.

Algorithm 8 Original IC3 reconstructed from the modular functions

Input: A symbolic transition system $(I(x), T(x, x'), P(x))$

Output: Whether the transition system is safe

```
1: IC3Checker  $\leftarrow$  a new IC3 checker using  $(I(x), T(x, x'), P(x))$ 
2: if IC3Checker.CheckInitial() then
3:   return unsafe
4: end
5: while true do
6:    $B \leftarrow$  IC3Checker.CheckFrameForPredecessors( $\neg P$ , IC3.currentFrameindex)
7:   if  $B$  is not empty then
8:     if not IC3Checker.TryBlock( $B$ , IC3.currentFrameindex) then
9:       return unsafe
10:    end
11:  else
12:    if IC3Checker.Propagate() then
13:      return safe
14:    end
15:  end
16: end
```

When the normal algorithm tries to refine a frame, it checks whether the faulty states are reachable or not from the current frame. If they are not reachable, then it moves on to the next frame. If they are reachable, then it tries to block the predecessors of the faulty states from the frame. If it succeeds, then it tries to find another set of states to block. If the blocking fails, then a faulty state is reachable from the initial states and the model is faulty. On the other hand, in the connected version of IC3, the algorithms don't use the property of the system. Instead, the forward-stepping version uses the overapproximation maintained by the reverse-stepping version (one of the reverse frames), and the reverse version uses the overapproximation maintained by the forward-stepping version.

Every forward frame is an overapproximation of I and every reverse frame is an overapproximation of $\neg P$. The goal of this optimization is to advance from both ends of the state space in an effort to prove a model's safety or faultiness faster.

I will explain this new blocking procedure step by step. Let us assume that the forward-stepping version is at F_i and tries to refine it. In the base version of IC3 using the property optimization, the algorithm looks for predecessors of $\neg P(x)$ in F_i . In the connected version, however, it uses a frame (RF_j) from the reverse direction instead of $\neg P(x)$. RF_j contains an overapproximation of the states, which are reachable from the faulty states in j steps or less, using the reverse transitions. In other words, it contains an overapproximation of the states, from which the faulty states are reachable in j steps or less, using the forward directional transitions. The algorithm will use the CheckFrameForPredecessors function of the forward IC3, with the arguments (RF_j, i) because it looks for a predecessor of RF_j in the i -th frame.

3.1.2.1 No connection between the two frames

If the $F_i(x) \wedge T(x, x') \wedge RF_j(x')$ query returns UNSAT, then it means, that the forward-stepping algorithm managed to block every state from F_i , from which a state in RF_j is reachable (see Figure 3.1). In this case, the algorithm has proven that in $i + 1$ steps or less, it can not reach a faulty state from the initial states.

The proof of this is quite intuitive. If in $i + 1$ steps it can not reach a state in RF_j , which is an overapproximation of the faulty states, then it can not reach a faulty state in $i + 1$ steps.

If the forward-stepping version can not reach a faulty state in $i + 1$ steps, then it can move on to F_{i+1} and start to examine it too.

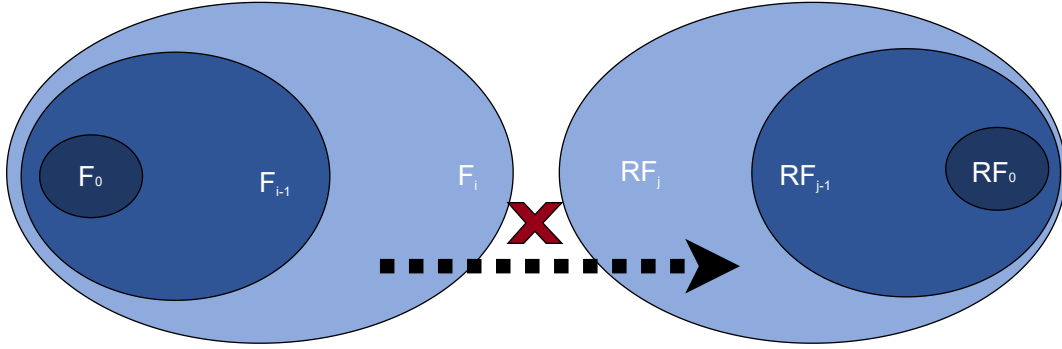


Figure 3.1: The SMT solver did not find a connection between F_i and RF_j . This means, that no faulty states are reachable in $i + 1$ steps from the initial states, so the algorithm can move on to F_{i+1} .

3.1.2.2 Connection found between the two frames

If the solver decides that the SMT query $F_i(x) \wedge T(x, x') \wedge RF_j(x')$ is SAT and returns the predecessor states B from F_i , then the algorithm tries to block B from F_i , using the TryBlock function (see Figure 3.2).

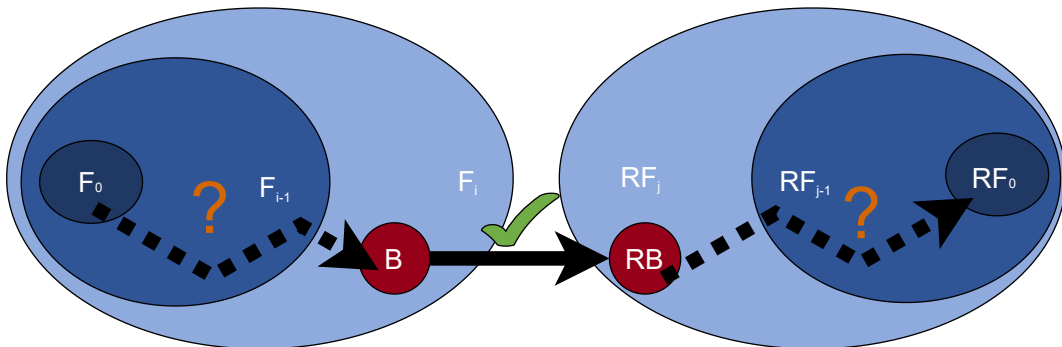


Figure 3.2: The solver found a connection between F_i and RF_j . If B is reachable from the initial states in i steps, and RF_0 is reachable from RB in j steps, then the algorithm has found a counterexample.

3.1.2.3 Successfully blocking an expression from F_i

If B is successfully blocked from F_i , then the algorithm continues, and F_i has been refined. This can be seen in Figure 3.3.

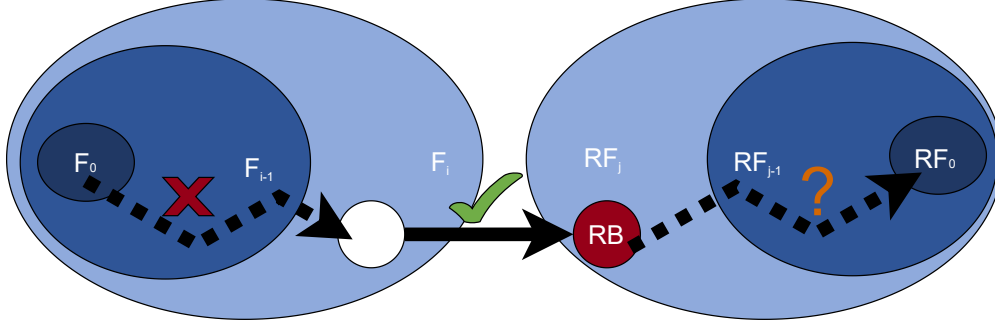


Figure 3.3: B is not reachable from the initial states in i steps, so it can be blocked from F_i

3.1.2.4 Unsuccessful blocking of B , looking for successors in RF

If the TryBlock function returns *false*, because B is reachable in i steps from the initial states, then it doesn't mean automatically that the model is faulty because RF_j is an overapproximation. In this case, the algorithm has to block every successor of B from F_j . At least one successor state must exist because the solver previously found a connection between F_i and RF_j . The algorithm finds these states, by calling the CheckFrameForPredecessor function of the reverse IC3, with the input arguments (B, j) . This will return predecessors of B , from the RF_j , using the reverse transitions (see Figure 3.4). This means, that the function will return the successors of B , from RF_j using the forward transitions. Let us call these successor states RB .

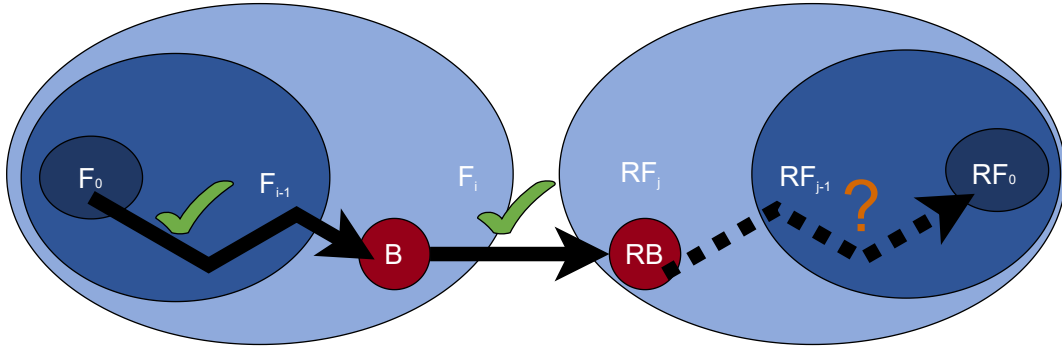


Figure 3.4: The algorithm finds the successor states of B in RF_j .

3.1.2.5 Successfully blocking RB

The algorithm must block RB . It calls the TryBlock function of the reverse IC3 with the arguments (RB, j) because it wants to block RB from the j -th frame of the reverse IC3. If it manages to block RB , then RF_j has been refined, and the connection between F_i and RF_j has

been weakened. (See Figure 3.5) In this case, the algorithm looks for another successor of B and tries to block it, until there are no successors of B in RF_j , or one blocking is unsuccessful.

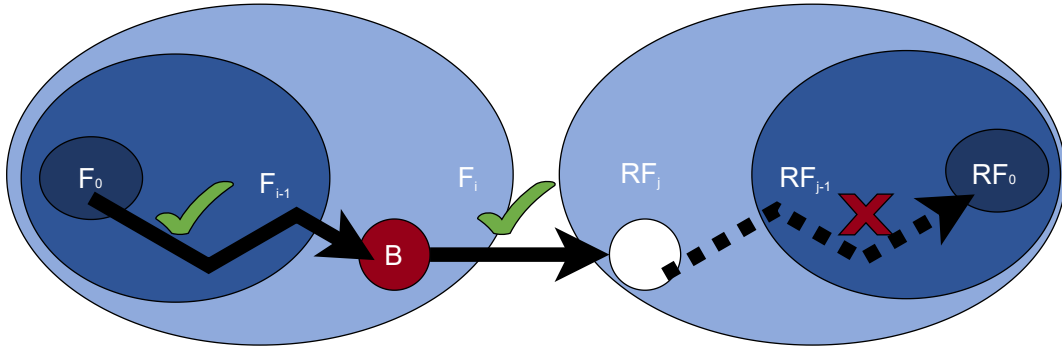


Figure 3.5: The solver found a connection between F_i and RF_j . B is reachable in i steps, meaning it cannot be blocked from F_i . However, RB can be blocked from RF_j .

3.1.2.6 Both blockings are unsuccessful, the algorithm managed to find a counterexample

If the algorithm doesn't manage to block RB from RF_j (see Figure 3.6), then it means that the algorithm managed to find a counterexample, and the model is faulty. The forward-stepping version has found the first i steps, one step between the two frames has been found by the CheckFrameForPredecessors function, and the reverse-stepping version has found the last j steps.

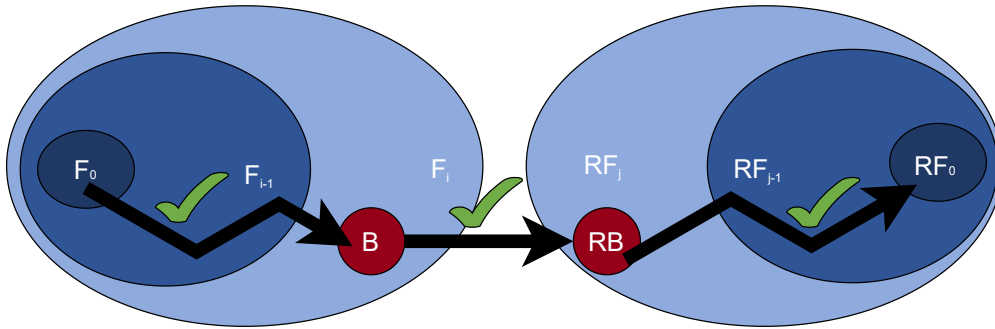


Figure 3.6: B is reachable in i steps from the initial states and $\neg P$ is reachable in j steps from RB . This means, that the algorithm managed to find a counterexample.

3.1.3 Combining Forward IC3 and Reverse IC3

In this subsection, I will detail the mechanism of the ConnectedIC3 algorithm and explain it in detail, using the blocking process, which I have introduced in 3.1.2, and the dismantled, modular functions of IC3, which I have introduced in 3.1.1. The pseudo-code for this algorithm can be seen in Algorithm 9.

First, the algorithm initializes the Forward and Reverse IC3 algorithms and checks the initial conditions with the CheckInitial function. It only needs to check the forward or the reverse IC3's CheckInitial function, because both of them use the same SMT queries, $I(x) \wedge \neg P(x')$ and $I(x) \wedge T(x, x') \wedge \neg P(x')$. If the algorithm finds a counterexample, then it returns with *unsafe*. If it doesn't, then the main loop of the algorithm can begin.

During the loop, the algorithm checks with the CheckFrameForPredecessors function, if from the current forward frame, a reverse frame is reachable. I will describe it in detail in 3.1.4, which reverse frame the algorithm uses here.

If it is reachable, then the algorithm must block B from F_i or its successor RB from F_j . This blocking process has been described in 3.1.2.

If the CheckFrameForPredecessors function has not found any connection between F_i and RF_j , then it is proven that from F_i no faulty states are reachable in one transition step, so the algorithm can call the propagate function on the Forward IC3 algorithm.

If during the propagation two equal frames have been found, then the algorithm can stop with a *safe* result. The reasoning behind this is the same as it was in the normal IC3 algorithm. Let us assume that F_i is identical to F_{i-1} after the algorithm has proven that from F_i , a frame from the other direction is not reachable. The reverse frame contains RF_0 because of its definition. RF_0 equals $\neg P$, so F_i does not contain any faulty states. The rest of the proof is the same, as it was in 2.4.4 The same thing is true if the reverse IC3 finds two identical frames.

After the algorithm inspects the forward part, then it moves on to the reverse part. It performs the same steps as before, just with switched directions.

I provide the pseudo-code of the Connected IC3 algorithm in Algorithm 9. In the pseudo-code i is the index of the currently inspected forward frame. j is the index of the reverse frame, which is checked for successors, by the Forward IC3. k is the index of the currently inspected reverse frame. l is the index of the forward frame, which is checked for successors, by the Reverse IC3. The algorithm always maintains that $i \geq l$ and $k \geq j$.

3.1.4 Configuration options

My version of the Connected IC3 algorithm can be configured through 4 parameters:

- The offset of the reverse frame can be set. If the offset is 0, then the forward IC3 will look for predecessors, using the currently inspected reverse frame. If the offset is a , then the algorithm will use the $(currentReverseIndex - a)$ -th reverse frame to look for intersections. This means that in Algorithm 9 j will be equal to $k - a$. If the algorithm uses a frame closer to the currently inspected frame, it is more likely to find connections between the frames, which can be beneficial, but it can mean more irrelevant connections, which slows down the algorithm;
- The *offset* of the *forward* frame can also be set, which sets which forward frame will be used by the Reverse IC3;

Algorithm 9 Connected IC3

Input: A symbolic transition system $(I(x), T(x, x'), P(x))$

Output: Whether the transition system is safe

```
1: ForwardIC3  $\leftarrow$  a new IC3 checker using  $(I(x), T(x, x'), P(x))$ 
2: ReverseIC3  $\leftarrow$  a new IC3 checker using  $(\neg P(x), T(x', x), \neg I(x))$ 
3: if ForwardIC3.CheckInitial() then
4:   return unsafe
5: end
6: while true do
7:    $B \leftarrow$  ForwardIC3.CheckCurrentForPredecessors( $RF_j, i$ )
8:   if  $B$  is not empty then
9:     if not ForwardIC3.Tryblock( $B, i$ ) then
10:      while  $B$  has successors in  $RF_j$  do
11:         $RB \leftarrow$  ReverseIC3.CheckFrameForPredecessors( $B, j$ )
12:        if not ReverseIC3.Tryblock( $RB, j$ ) then
13:          return unsafe
14:        end
15:      end
16:    end
17:  else
18:    if ForwardIC3.propagate() then
19:      return safe
20:    end
21:  end
22:   $RB \leftarrow$  ReverseIC3.CheckFrameForPredecessors( $F_l, k$ )
23:  if  $RB$  is not empty then
24:    if not ReverseIC3.Tryblock( $RB, k$ ) then
25:      while  $PB$  has successors in  $F_l$  do
26:         $B \leftarrow$  ForwardIC3.CheckFrameForPredecessors( $RB, l$ )
27:        if not ForwardIC3.Tryblock( $B, l$ ) then
28:          return unsafe
29:        end
30:      end
31:    end
32:  else
33:    if ReverseIC3.propagate() then
34:      return safe
35:    end
36:  end
37: end
```

- The algorithm’s *forward* and *reverse step sizes* can be set too. If the forward step size parameter of the algorithm is b , then the main algorithm switches to the reverse version after it calls the TryBlock function b times and blocks those expressions from the forward frames, or blocks successors of them from the reverse frames. If the forward step size is large and the reverse step size is small, then the Connected IC3 algorithm behaves more like a Forward IC3 algorithm. If the reverse step size is larger and the forward step size is smaller, then the algorithm behaves more like a Reverse IC3.

In the pseudo-code in Algorithm the forward and the reverse step size is 1 because the algorithm switches directions after one blocking. The offset of the reverse frame is $k - j$, and the forward frame’s is $i - l$.

By changing these parameters, the algorithm can be tailored to specific models, increasing the likelihood of solving them.

3.2 Other optimizations

In this section, I will detail two other optimizations, which I have developed to further improve the efficiency of the IC3 algorithm and make it applicable for verifying an even wider range of hardware and software models.

3.2.1 Blocking multiple expressions at once

When the algorithm tries to block a B expression from a frame, it asks the SMT solver for a predecessor of B in the previous frame. If it gets one, then it tries to block it from the previous frame. If the blocking of the predecessor was successful, then it asks the solver for another predecessor and the blocking process continues. If B has many predecessors, then this process can take a long time. My optimization tries to block them together at once. This can dramatically reduce the number of function calls, which can have a positive effect on the runtime of the algorithm.

I have modified the TryBlock function in the following way to achieve this. After the solver returns a predecessor PB_1 of B from the previous frame, the algorithm does not immediately try blocking PB_1 but instead stores PB_1 and then adds the expression $\neg PB_1(x)$ to the solver and asks for another predecessor. In the normal algorithm, this only happens after PB has been blocked from the previous frame, by a TryBlock function call and $\neg PB_1$ has been added to F_{i-1} . The algorithm asks for predecessors until the solver returns UNSAT, meaning there are no more predecessors of B in the previous frame. Then the algorithm must block every predecessor of B from the previous frame. It does this by calling the TryBlock function with $PB_1 \vee PB_2 \dots \vee PB_j$. This means that the recursive TryBlock function has to block every predecessor of B from the previous frame simultaneously. If the blocking of every predecessor was successful, then the function can block B from the frame and return true. If it was unsuccessful, then at least one of the predecessors can be reached from the initial states, which means that the function has to return false because the blocking was unsuccessful. I provide the pseudo-code for this modified function in Algorithm 10.

Algorithm 10 Modified TryBlock to block more states at once

Input: An expression, which is needed to be blocked: B , and an index i

Output: Whether the blocking of B was successful or not

```
1: if  $i$  equals 0 then
2:   return false ▷ the initial states have been reached
3: end
4: while  $B$  has predecessors in  $F_{i-1}$  do
5:    $PB_j \leftarrow$  The satisfying solution, provided by the solver
6:   Adding  $\neg PB_j(x)$  to the solver.
7: end
8: if not  $TryBlock(PB_1 \vee PB_2 \dots \vee PB_j, i - 1)$  then
9:   return false ▷ a predecessor can be reached from the initial states in  $i - 1$  steps
10: else
11:   for  $F_j \in (F_1..F_i)$  do
12:     Add  $\neg B$  to  $F_j$ 
13:   end
14:   return true ▷  $B$  has no predecessors in  $F_{i-1}$  anymore
15: end
```

3.2.2 Former frames optimization

This optimization tries to make the blocking process smoother and avoid unnecessary function calls during the blocking process.

During the blocking process, the algorithm tries to find out if the expression it got is reachable from the initial states or not in the given number of steps. If the blocking process finds a predecessor of this expression in the previous frame, then it tries to block this predecessor from that frame. Let us assume that the process tries to block an expression from F_i . If the blocking process found a predecessor in F_{i-1} , which is in F_{i-2} too, then it is unnecessary to check this predecessor because it is not reachable from the initial states in $i - 1$ steps. It is not reachable in $i - 1$ steps, because it is in F_{i-2} , and if it were a part of a counterexample of this length the algorithm would have found it by now.

The algorithm can remove these states from the process altogether by adding $\neg F_{i-2}(x)$ to the SMT query that looks for predecessors, i.e., $F_{i-1}(x) \wedge \neg B(x) \wedge T(x, x') \wedge B(x')$. This ensures, that the algorithm will not try to block these states, which are already proven to be unreachable in $i - 1$ steps. After this modification, the SMT query becomes $\neg F_{i-2}(x) \wedge F_{i-1}(x) \wedge \neg B(x) \wedge T(x, x') \wedge B(x')$.

Chapter 4

Evaluation

In this chapter, I evaluate the algorithmic optimizations from the literature presented in Chapter 2. I also evaluate my algorithmic contributions in a detailed measurement campaign, to prove its effectiveness and efficiency. In Section 4.1 I introduce the experiment plans, and in 4.2 I analyze the results of the experiments.

4.1 Experiment planning

This experiment aims to analyze the different algorithmic optimizations and variations of the IC3 algorithm. During this experiment, I executed several diverse variations of the algorithm on different highly complex input models.

4.1.1 Research questions

My aim during this experiment was to answer the following questions:

RQ1 How do the optimizations presented in Chapter 2 impact the performance of the IC3 algorithm?

RQ2 Do my algorithmic contributions to the IC3 algorithm manage to improve its effectiveness?

RQ3 Is the connected IC3 I presented capable of solving more models compared to its forward and reverse counterparts?

RQ4 How do the configuration options presented in 3.1.4 affect the connected IC3 algorithm's performance?

4.1.2 Evaluation environment

I implemented the IC3 algorithm and its optimizations in the open-source Theta tool. The evaluation ran in the BME Cloud on the Ubuntu 22.04 LTS operating system with the JDK version 17. The virtual machine had 8 cores and had 8192 MiB RAM. I used the BenchExec [3] tool to provide reliable and accurate measurements. It is a state-of-the-art benchmarking tool,

which is also used at major competitions such as the Satisfiability Modulo Theories Competition (SMT-COMP) [8], the Software Verification Competition (SV-COMP) [1], and the Termination Competition. The time limit of each measurement was 300s.

4.1.3 Evaluation models

I evaluated the different variations of the IC3 algorithm using STS models from the 2017 edition of the Hardware Model Checking Competition. These are highly complex, difficult hardware models. The attributes of the models can be seen in Table 4.1 organized into categories [5].

Category	6s	bj	bob	intel	oski	other	pdt	tip
Models	5	3	9	1	1	3	7	16
Minimum vars	115	384	91	295	327052	68	1	1
Maximum vars	95866	922	18570	295	327052	28945	3571	5110
Average vars	57044.02	717.67	6037.33	295	327052	9648.33	1128	1505.88

Table 4.1: The attributes of the input models

I evaluated the algorithm on simple CFA and XSTS models as well, after I transformed them into STS models. However, because they have values with large domains, the algorithm is currently not efficient at solving these models. In the future, I plan to use abstraction to solve this problem and make the algorithm able to solve more CFA and XSTS models.

4.1.4 Parameters of the evaluation

The parameters of the evaluation can be seen in Table 4.2. It can be seen, that these parameters yield a lot of different combinations and it is not realistic to try out all of those. Instead, for each research question, I fixed a subset of the parameters to values that were deemed optimal based on previous experiences, while leaving other parameters free. The overview of this design can be seen in Table 4.3.

I selected the configurations for the benchmarks in the following way:

- During the evaluation of RQ1 the Boolean parameters of the optimizations from the literature were left free (enabled or disabled), while my optimizations were set to a fixed value. The Direction parameter was also left free, meaning I tested every Boolean parameter with every possible directional value.
- During the evaluation of RQ2, the optimizations from the literature were enabled, while my optimizations were left free (enabled or disabled). The Direction parameter was also left free, meaning I tested every Boolean parameter with every possible directional value.
- During the evaluation of RQ3, every Boolean parameter was fixed (all optimizations were enabled except for Blocked), and the directional parameter was left free.

Parameter name	Type	Description
Direction	Enum	Direction of the IC3 algorithm. Possible values: Forward, Reverse, Connected
NotB	Boolean	Strengthen the blocking query, by adding $\neg B(x)$ to it
UnSat	Boolean	Remove expressions from the UNSAT-core
Propagate	Boolean	Propagate blocked states forward
Filter	Boolean	Generalize the predecessors of the blocking query
Property	Boolean	Initialize the frames using P
Former	Boolean	Strengthen the blocking query by adding $\neg F_{i-2}(x)$ to it
Blocked	Boolean	Block the predecessors of B together
ForwardStep	Non-negative integer	It sets the forward step size of the connected IC3
ReverseStep	Non-negative integer	It sets the reverse step size of the connected IC3
ForwardFrameOffset	Non-negative integer	It sets the offset of the forward frame
ReverseFrameOffset	Non-negative integer	It sets the offset of the reverse frame

Table 4.2: The variables of the evaluation. The NotB, UnSat, Propagate, Filter, and Property optimizations are from the literature. The Former and Blocked optimizations were presented by me in Chapter 3. The ForwardStep, ReverseStep ForwardFrameOffset, and ReverseFrameOffset are the integer parameters of the connected IC3.

Parameter category	RQ1	RQ2	RQ3	RQ4
Direction	Forward, Reverse, Connected	Forward, Reverse, Connected	Forward, Reverse, Connected	Connected
Optimizations from the literature	True or False	True	True	True
My optimizations	Former=True, Blocked=False	True or False	Former=True, Blocked=False	Former=True, Blocked=False
Integer parameters of the connected IC3	Fixed values (1,1,1,1)	Fixed values (1,1,1,1)	Fixed values (1,1,1,1)	Many combinations

Table 4.3: The configurations that I used to answer the various research questions.

- During the evaluation of RQ1, RQ2, and RQ3 the ForwardStep, ReverseStep, ForwardFrameOffset, and the ReverseFrameOffset parameters were set to 1.
- During the evaluation of RQ4, the Boolean parameters were fixed (all optimizations were enabled except for Blocked), the direction parameter was set to Connected, and the four integer parameters were left free. The ForwardStep parameter had values $\{1, 2, 10\}$, the ReverseStep parameter had values $\{1, 2, 10\}$, the ForwardFrameOffset had values $\{0, 1, 2\}$ and the ReverseFrameOffset had values $\{0, 1, 2\}$.

4.2 Measurement results

In this section, I detail the results of the measurement campaign. I visualised the results of the measurements in two different kinds of diagrams, *pairwise scatterplots* and *heatmaps*.

Pairwise comparisons For analyzing Boolean parameters, I used pairwise scatterplots. Every point in these diagrams indicates two model checking measurement results, whose configurations only differ in one parameter. One where the optimization was turned on and one where it was not. The x coordinate of the point indicates a result’s runtime, when the optimization was turned on, and the y coordinate value indicates when it was not.

- If the point is *on the $x = y$ diagonal* it means, that the optimization has not had any effect on the runtime of the algorithm on that input model, because the runtime of the algorithm did not change when the optimization was turned on.
- If it is *below the diagonal*, which means its x coordinate is larger than its y coordinate, then the optimization slowed down the algorithm, because it had a longer runtime, when the optimization was turned on.
- If it is *above the diagonal*, which means its y coordinate is larger than its x coordinate, then the optimization accelerated the algorithm, because it had a quicker runtime, when the optimization was turned on.

If a point’s x or y coordinate is 300, then it means that the algorithm timed out, and it did not manage to solve the input model, with that particular configuration.

Heatmaps Whenever I wanted to compare multiple configurations at once, I used heatmaps. Every cell in the heatmap signals the success rate of an algorithm in a category. The row of the cell indicates, which version of the IC3 algorithm achieved these results, and the column of the cell indicates the category of models. The color of the cells indicates, how successful a measurement configuration was in a given category.

4.2.1 RQ1: Effectiveness of the optimizations from the literature

In this subsection, I will answer Research Question 1, based on the measurement results. The diagrams describing the results can be found in the appendix. As can be seen in Figure A.1.3

and Figure A.1.2, turning on the NotB and the Filter optimizations did not have a positive or negative effect on the algorithm’s runtime. This means, that in these input models, the strengthened query did not stop the blocking process many times. It is also true for these input models, that generalizing the predecessors of the blocking query did not yield any results. It can be interesting to check in the future if this is the case on other input models as well.

As can be seen in Figure A.1.1 and Figure A.1.4 the UnSat, and the Propagate optimizations had a major positive effect on the runtime of the algorithm. This was true for all three directions and every different category as well. This means, that these optimizations are very powerful. There were a lot of timeouts as well, when one of these optimizations was turned off.

As can be seen in Figure A.1.5 initializing the frames differently, has not provided major results. There were some models where it was effective at reducing the runtime, but it’s nothing conclusive.

Summary The measurements indicate that the UnSat and the Propagate optimizations provide a major advantage, while the NotB, Filter, and Property optimizations require further experiments.

4.2.2 RQ2: Effectiveness of my algorithmic contributions

In this subsection, I will answer Research Question 2, based on the measurement results. The diagrams describing the results can be found in the appendix. As can be seen in Figure A.1.6 filtering out the expressions from F_{i-2} did not have an effect on the runtime of the algorithm. The extra expression added to the blocking query did not slow down the algorithm, but it seems, that it had not managed to stop many unnecessary TryBlock calls, which would have accelerated the algorithm.

As can be seen in Figure A.1.7 blocking multiple expressions at once did not accelerate the algorithm. There were some input models, where it slowed down the runtime of the algorithm, meaning that this optimization only brought disadvantages. It could be an interesting question if this is the case for other models as well.

Summary Based on these results, it seems that the Former optimization didn’t affect the algorithm’s performance on my benchmark set. The Blocked optimization did not perform well on these input models. My assumption for this is, that it lists too many expressions when examining a frame, which slows down the algorithm. This requires further inquiry.

4.2.3 RQ3: Comparing the connected IC3 to the reverse IC3 and the forward IC3

In this subsection, I will answer Research Question 3, based on the measurement results. In Figure 4.1 the comparison of the reverse, forward, and connected IC3 algorithms can be seen. Recall that the rows and columns of the diagram correspond to configurations and model categories, respectively. For example, the cell in the first row and the second column indicates

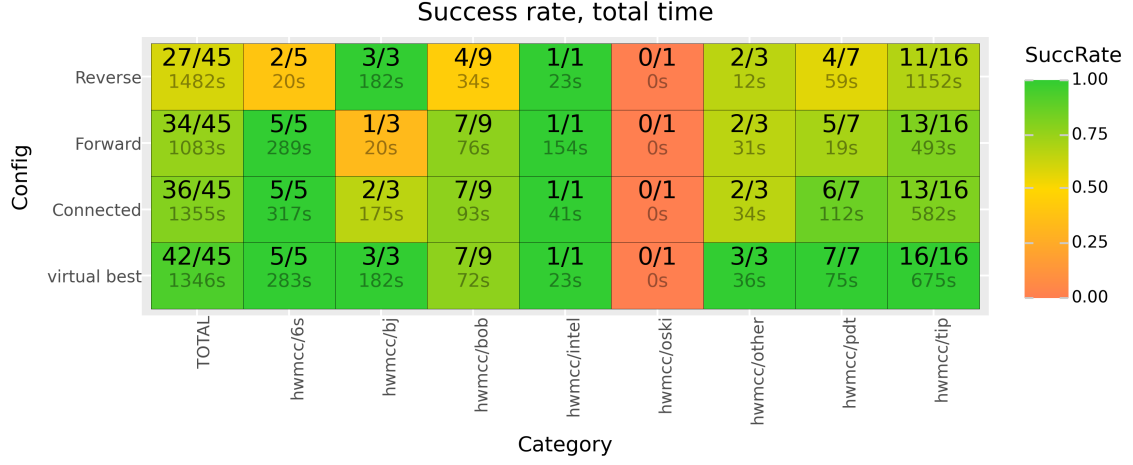


Figure 4.1: Comparing the different directional variations of the IC3 algorithm

the success rate of the reverse IC3 in the 6s category. The first column summarizes the results of a version, and the fourth row in the table indicates the virtual best configuration. This configuration handpicks the best result from the three variations for every input model.

It can be seen, that every algorithm's success rate was highly dependent on the input models. For example, in the bj category, the reverse version solved every problem, while the forward and the connected were not able to do this. However, there were categories where the reverse version severely underperformed compared to its counterparts. The tip category also brought an interesting result, where no algorithms could solve every problem, but every problem was solved by at least one algorithm.

It can be stated that in the case of certain input models, the reverse performed better and in the case of certain models the forward performed better. This happens because in some models the counterexample, which proves the model's unsafeness, or two equal frames, which proves the model's safety, can be more easily found by starting from the initial states. In other cases, they can be found more easily starting from the faulty states.

The connected version provides a middle ground. It is able to solve problems, more suited for the forward version, and problems more suited for the reverse version as well. Sometimes it is able to solve input models, which are not solvable by the forward and the reverse versions as well. An example of this can be seen in the section dedicated to Research Question 4.

Summary The measurements indicate that, on one hand, the forward and reverse IC3 algorithms complement each other well (they perform well on different subsets of the input models) and that also my combined IC3 algorithm can integrate them well, solving the highest amount of input models. This means that my combined IC3 can serve as an optimal default setting for a model checker tool, having the highest chance to solve the input models it receives.

4.2.4 RQ4: Analysing the configuration options of the connected IC3 algorithm

In this subsection, I will answer Research Question 4, based on the measurement results. To evaluate the effect of the configurable parameters of the connected algorithm, I first modified the ForwardStep and the ReverseStep parameters. The resulting heatmap can be seen in Figure 4.2. Every configuration of the connected IC3 algorithm can be identified on the heatmap by four numbers in this order: the ForwardStep, ReverseStep, ForwardFrameOffset, and ReverseFrameOffset. For example, the 1_1_1_1 configuration is the base version, where every parameter is set to 1 and the 10_1_1_1 version is one, where the ForwardStep parameter has been set to 10. If Figure 4.2 is compared to Figure 4.1, then there are several interesting discoveries. Firstly the connected 10_1_1_1 version produces almost identical results to the forward IC3. It is not that surprising, because in this connected version, the algorithm mostly refines the forward frame and only deals with the reverse frame a fraction of its runtime. Due to similar reasons, the 1_10_1_1 version produces very similar results to the reverse IC3. The 2_1_1_1 and the 1_2_1_1 versions produce results between the two extremes, which is not surprising as well.

This means, that the connected IC3 algorithm can be made more similar to the forward version and can be made more similar to the reverse version, by changing its parameters. Obviously, it largely depends on the model, which parameter combination is the most effective.

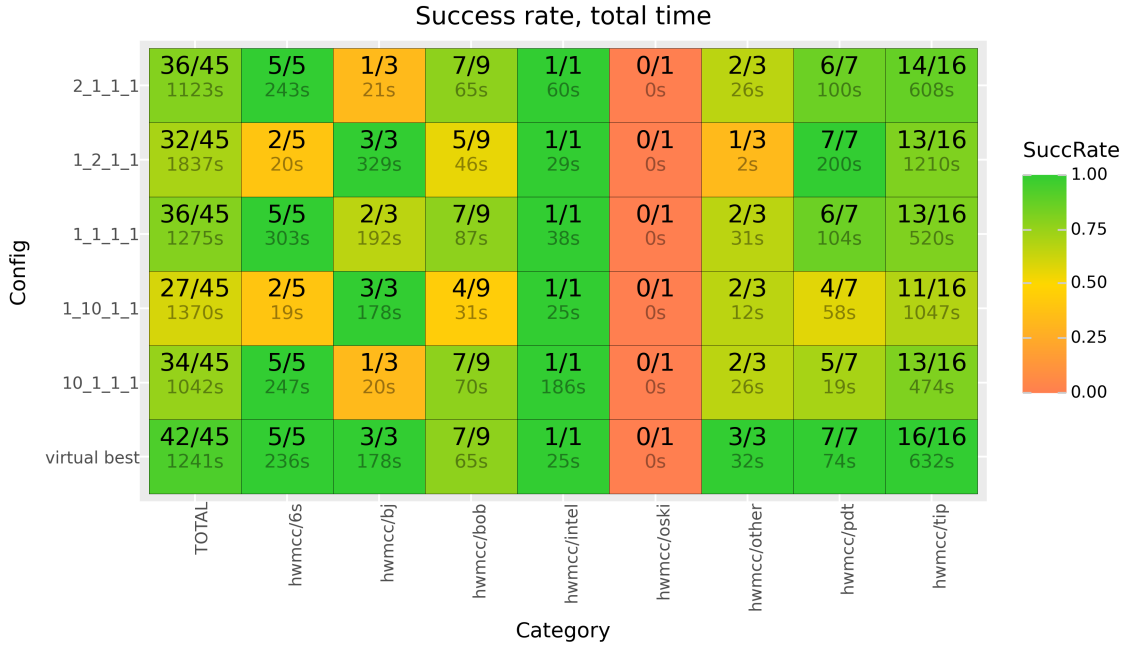


Figure 4.2: Results of the modification of the ForwardStep and the ReverseStep parameters.

Figure 4.3 shows the results of changing the offset parameters as well. There are two main takeaways from this figure. Firstly, changing the offset parameters can yield diverse results based on the input models. Secondly, in the fourth column of the first row, there is a model, which was solved by the connected version 2_1_0_0, which was not solvable by the forward and reverse versions of the IC3 algorithm. This indicates that my connected version of the IC3 is capable of solving problems that are not solvable by the forward and the reverse IC3 algorithms.

It is also true that changing the parameters can result in the solving of an input model, which was previously unsolvable.

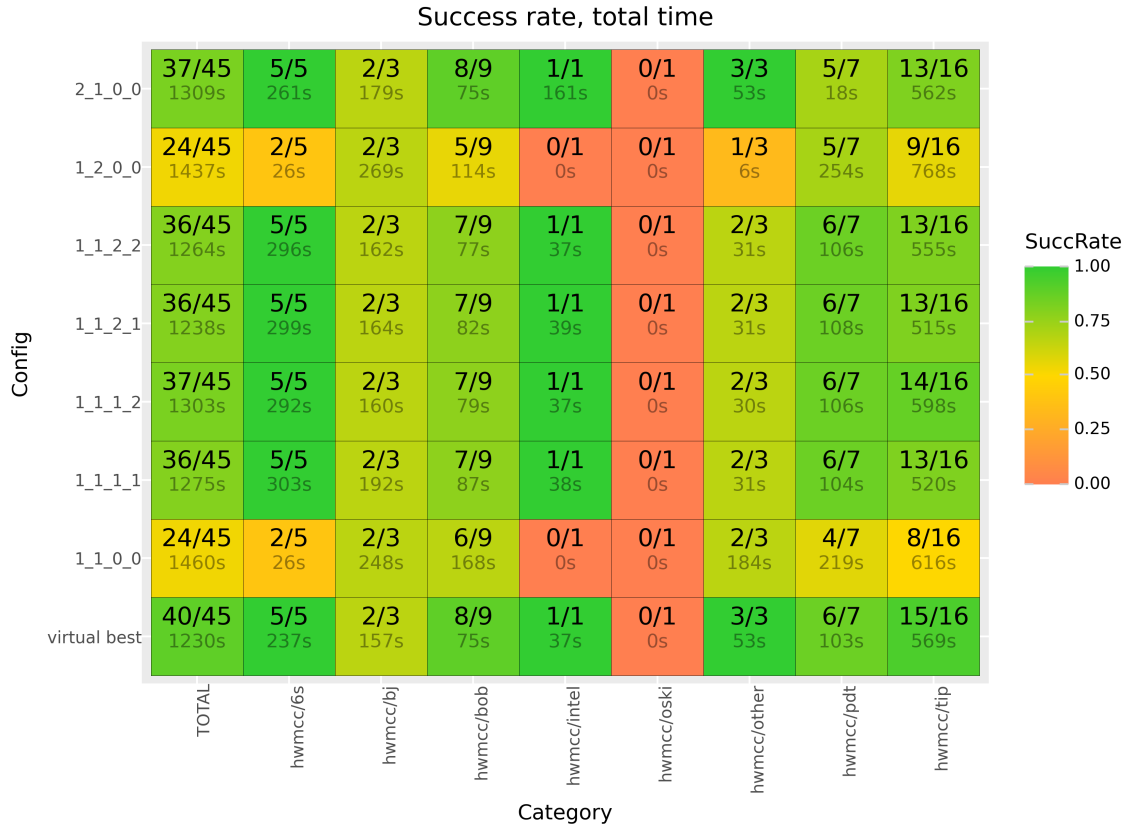


Figure 4.3: Results of the modification of the ForwardFrameOffset and the ReverseFrameOffset parameters.

Summary The measurement results show that my configuration options have a significant effect on the performance of the connected IC3 algorithm on the different model categories. The configuration options allow users to set the balance between the forward and reverse components of the algorithm (in extreme cases, the connected IC3 can even have the same performance as a simple forward or reverse IC3). This allows users to fine-tune the algorithm to their specific problems. Modifying the configurable parameters also allowed the algorithm to solve a model that was not solvable by the forward or the reverse IC3 algorithms.

Chapter 5

Conclusions

My primary goal during this work was to make the IC3 algorithm applicable for verifying an even wider range of hardware and software models, which makes it easier and quicker to detect and avoid errors as quickly as possible during the development of safety-critical systems.

My theoretical contributions are the following:

- I processed the recent literature on the IC3 algorithm and *several optimizations* of it, which I presented in Chapter 2.
- I presented a *novel* way of *connecting* the forward and reverse IC3 algorithms and sharing information between them. Furthermore, I made the IC3 algorithm more *configurable*, allowing experts to fine-tune it to the problems specific to their fields.
- I presented two further *novel algorithmic optimizations*. My first optimization strengthened the blocking query, while my second optimization enhanced the blocking process to block more expressions at once.

During my work, I made the following practical contributions:

- I implemented both the algorithms found in the literature and the algorithms and optimizations presented by me in the *open-source Theta framework*.
- I conducted a systematic benchmarking campaign to evaluate and compare the algorithms and optimizations found in the literature and presented by me. I ran the evaluation in a detailed benchmarking campaign on highly complex input models from the 2017 Hardware Model Checking Competition.

Future work In the future, I plan to implement the connected IC3 presented in [21] and compare it to my own version. I also plan to combine the location-aware IC3 presented in [17] with my connected version. Furthermore, I want to use abstraction-refinement techniques, to enable the algorithm to better handle variables with large domains (e.g., integers).

Bibliography

- [1] D. Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In *Proc. TACAS (3)*, LNCS 14572, pages 299–329. Springer, 2024. DOI: 10.1007/978-3-031-57256-2_15.
- [2] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 146–162, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [3] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019.
- [4] Armin Biere. The aiger and-inverter graph (aig) format version 2007. 10. 12. 2007.
- [5] Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking competition 2017. In Daryl Stewart and Georg Weissenbacher, editors, *Formal Methods in Computer-Aided Design, FMCAD 2017, Vienna, Austria, October 02-06, 2017.*, page 9. IEEE, 2017.
- [6] Aaron R Bradley. Understanding ic3. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 1–14. Springer, 2012.
- [7] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free presburger arithmetic. *Journal of Automated Reasoning*, 47(4):341–367, 2011.
- [8] Martin Bromberger. 19th international satisfiability modulo theories competition (smt-comp 2024). <https://smt-comp.github.io/2024/>, 2024. Accessed: 2024-10.
- [9] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.
- [10] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19:7–34, 2001.
- [11] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018. ISBN 3319105744.

- [12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [13] FTSRG. Theta framework GitHub repository. <https://github.com/ftsrg/theta>, 2021. Accessed: 2024-10.
- [14] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. In *Formal Techniques for Distributed Objects, Components and Systems*, volume 9688 of *Lecture Notes in Computer Science*, pages 158–174. Springer, 2016. DOI: 10.1007/978-3-319-39570-8_11.
- [15] Dejan Jovanović and Bruno Dutertre. Property-directed k-induction. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD ’16, page 85–92, Austin, Texas, 2016. FMCAD Inc. ISBN 9780983567868.
- [16] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 3540741046.
- [17] Tim Lange, Martin R Neuhäuser, Thomas Noll, and Joost-Pieter Katoen. Ic3 software model checking. *International Journal on Software Tools for Technology Transfer*, 22(2): 135–161, 2020.
- [18] Kenneth L McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*, pages 1–13. Springer, 2003.
- [19] Milán Mondok. Efficient formal verification of component-based engineering models, 2022. Master’s Thesis, Budapest University of Technology and Economics.
- [20] Gyula Sallai. Development of a verification compiler for C programs, 2016. Bachelor’s Thesis, Budapest University of Technology and Economics.
- [21] Tobias Seufert. *On safety verification using PDR and Reverse PDR*. PhD thesis, Dissertation, Universität Freiburg, 2023, 2023.
- [22] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.
- [23] Tamás Tegzes. Applying incremental, inductive model checking to software, 2018. Bachelor’s Thesis, Budapest University of Technology and Economics.
- [24] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102257.

Appendix

A.1 Further benchmarking results

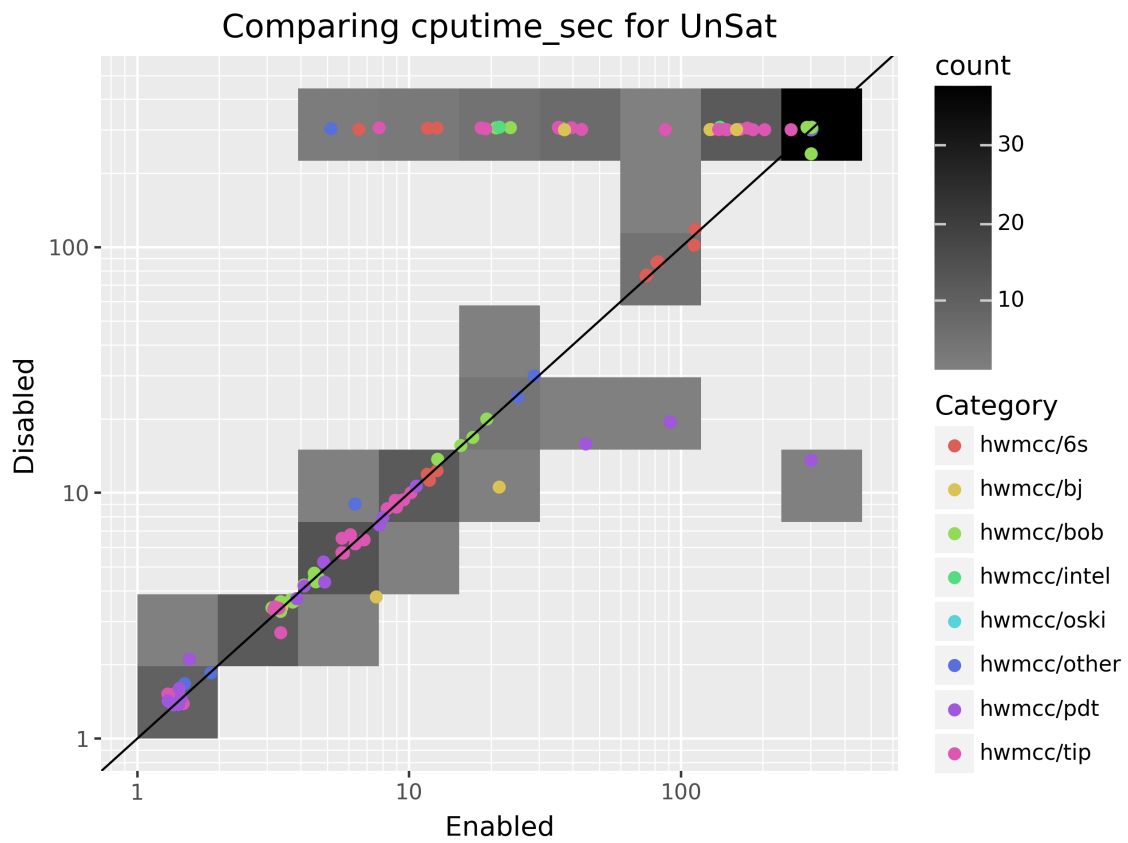


Figure A.1.1: Results of the UnSat optimization.

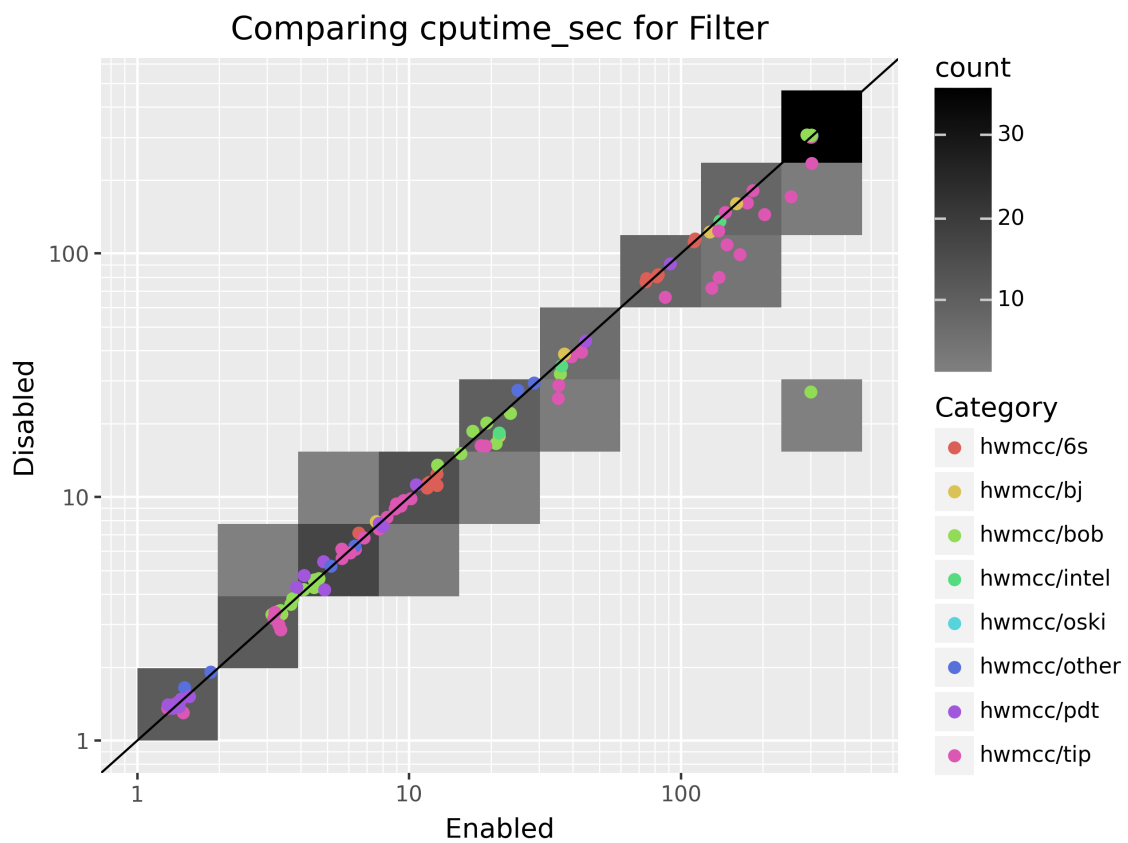


Figure A.1.2: Results of the filter optimization.

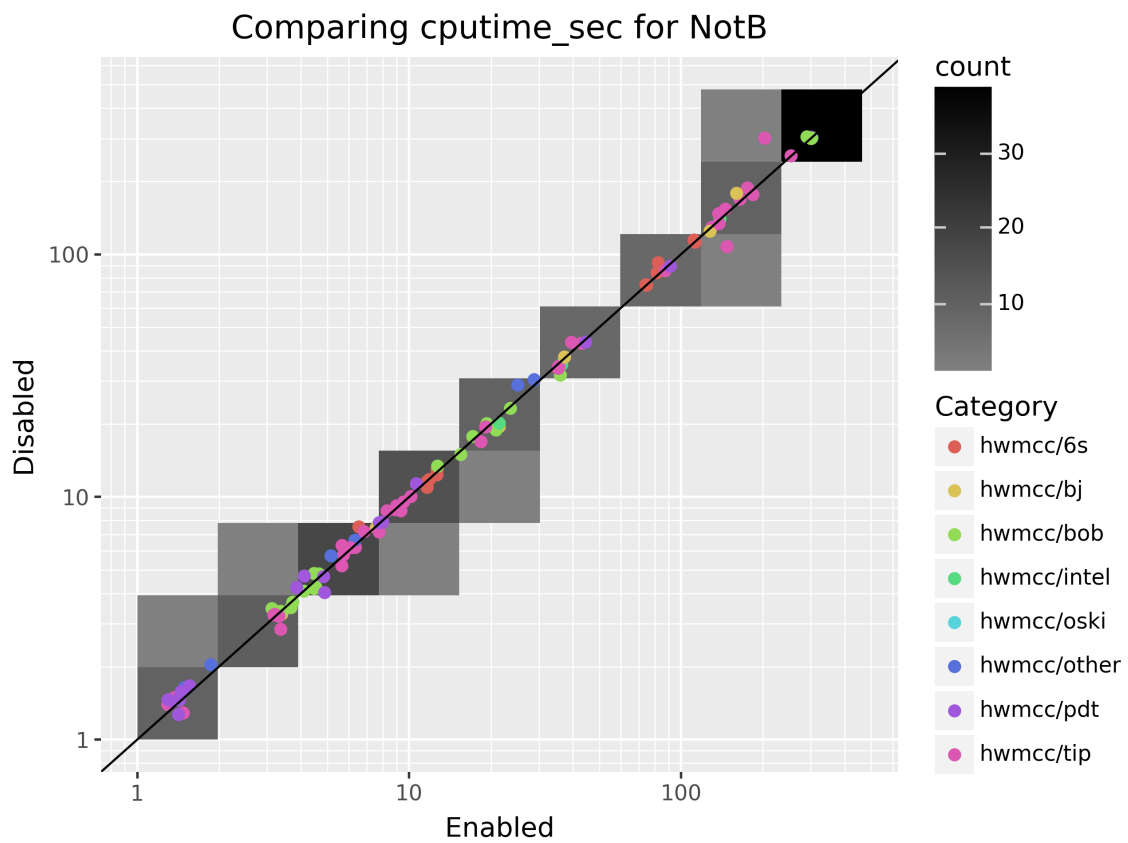


Figure A.1.3: Results of the NotB optimization.

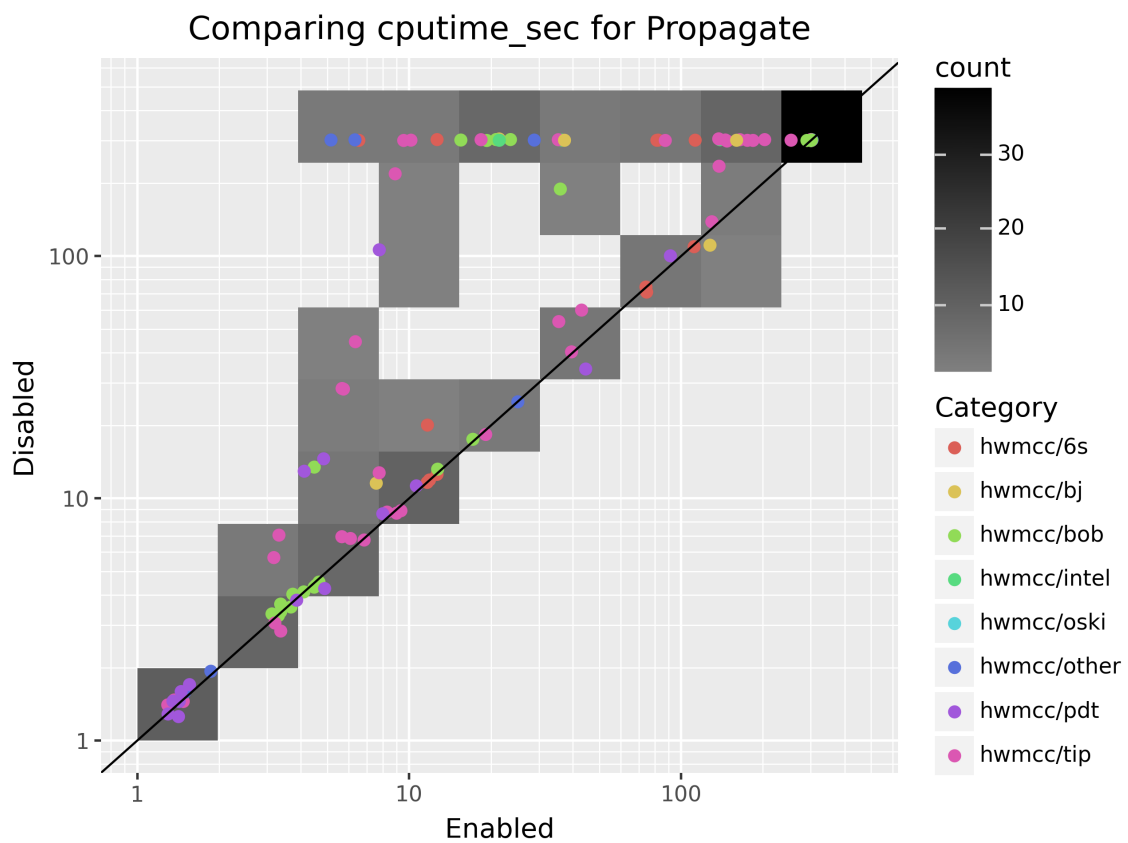


Figure A.1.4: Results of the propagate optimization.

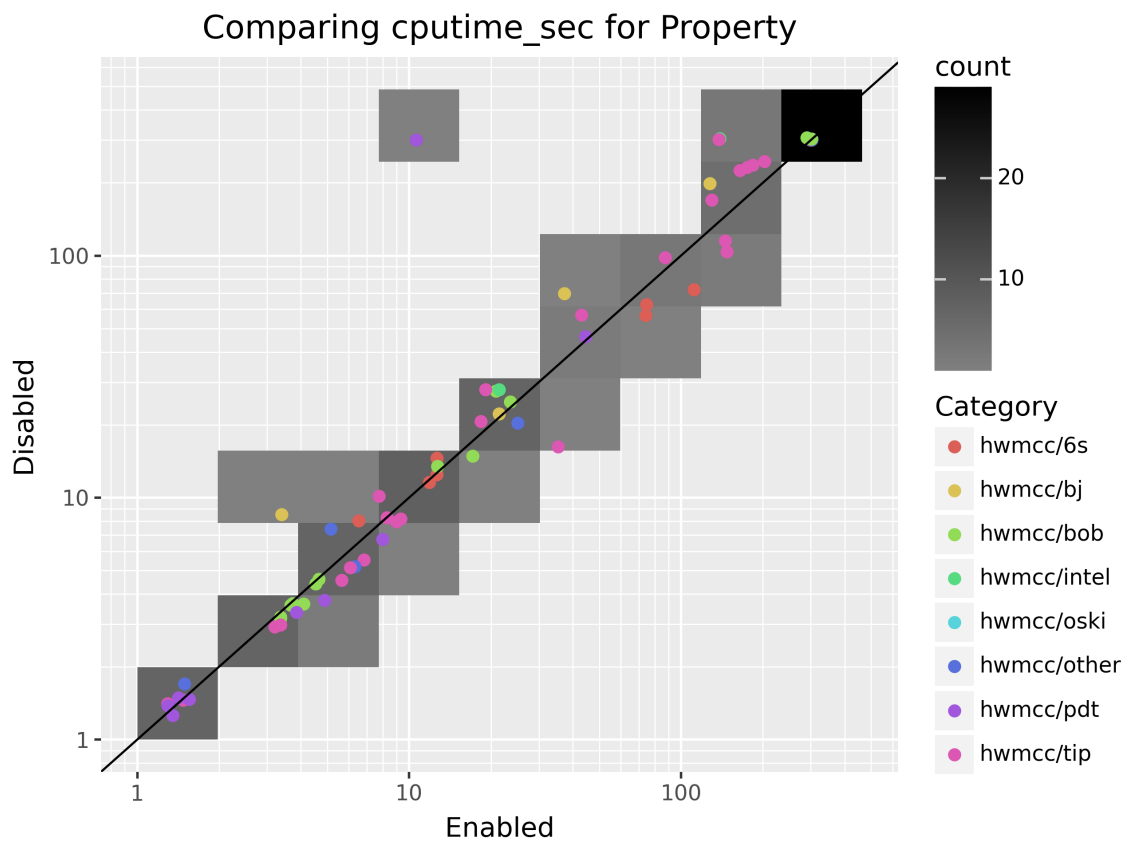


Figure A.1.5: Results of the Property optimization.

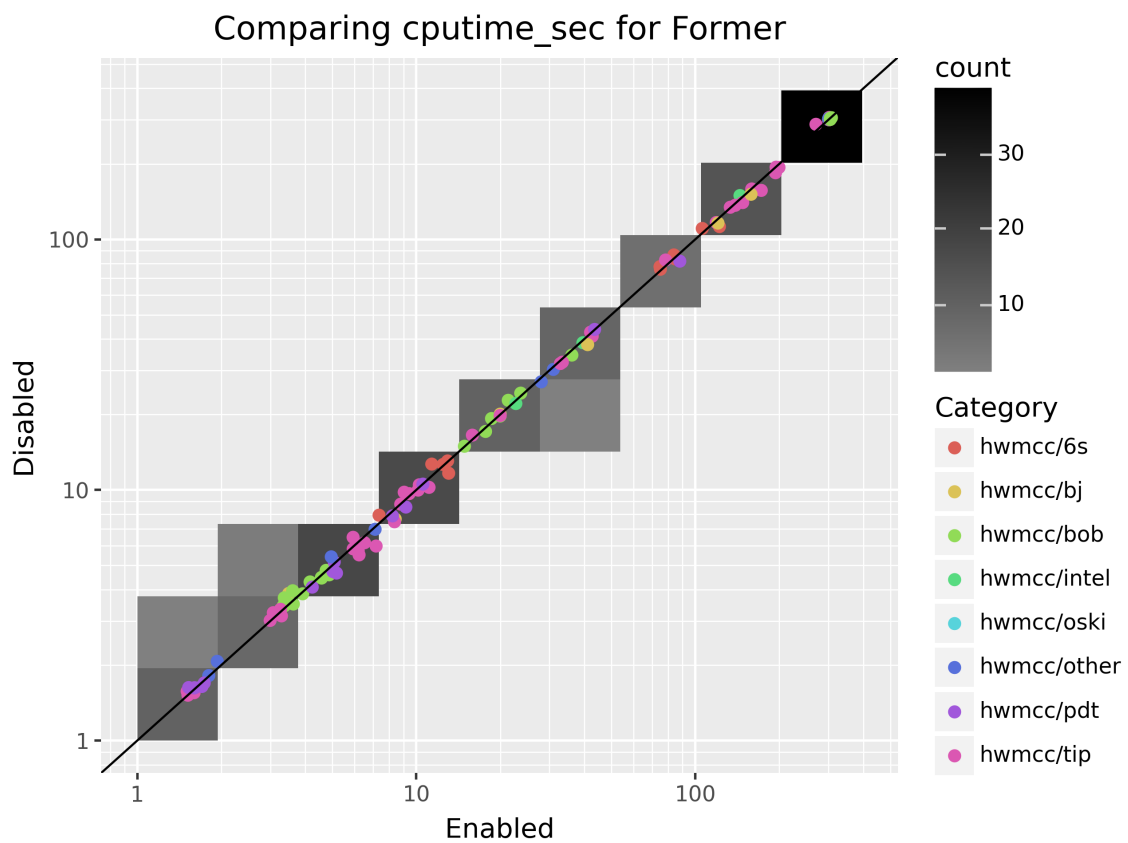


Figure A.1.6: Results of the Former optimization.

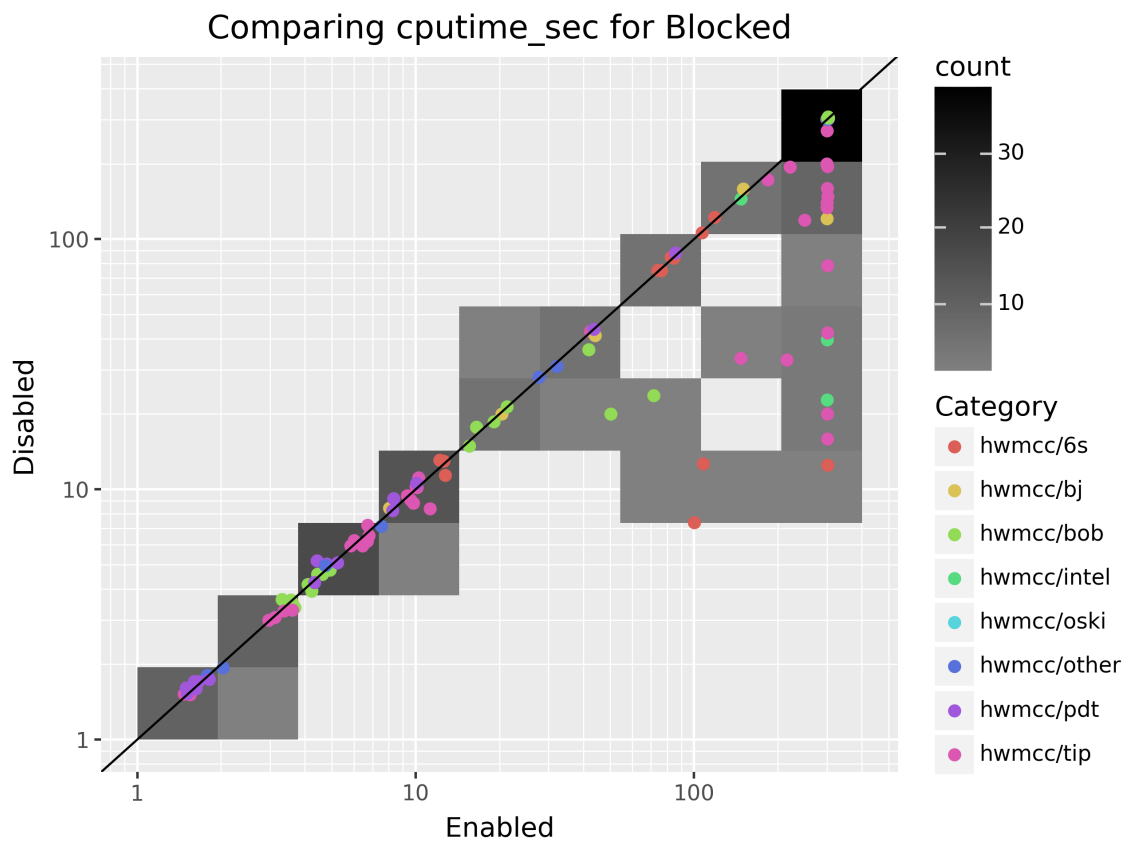


Figure A.1.7: Results of the Blocked optimization.