IC3 Software Model Checking

Tim Felix Lange

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2019-02
RWTH Aachen University | Department of Computer Science | February 2019

IC3 Software Model Checking

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Tim Felix Lange, M.Sc. RWTH
aus Viersen

Berichter:	Universitätsprofessor Dr. Ir. Dr. h. c. Joost-Pieter Katoen
	Universitätsprofessor DPhil (Oxon) Georg Weissenbacher
	Dr. rer. nat. Martin R. Neuhäußer

Tag der mündlichen Prüfung: 5. Oktober 2018

Abstract

In times where computers become ever smaller and more powerful and software becomes more complex and advances even deeper into every aspect of our lives, the risk of software misbehaviour and the resulting damage grows dramatically. In order to prevent such erroneous behaviour model checking, a formal verification technique for determining functional properties of information and communication systems, has proven to be highly useful.

For proving mathematical properties, one of the first methods to be taught in schools is induction. With the concept of proving a concrete induction base and a general induction step it is considered a very simple and intuitive, yet powerful proof method. However, for difficult properties finding an inductive formulation can be an extremely hard task. When humans try to solve this problem, they naturally produce a set of smaller, simple lemmas that together imply the desired property. Each of these lemmas holds relative to some subset of previously established lemmas by invoking the knowledge to prove the new lemma.

This incremental approach to proving complex properties using sets of small inductive lemmas was first applied to model checking of hardware systems in the IC3 algorithm and has proven to outperform all known approaches to hardware model checking.

This thesis aims at applying the principles of incremental, inductive verification laid by the IC3 algorithm to software model checking for industrial control software with special attention to the control-flow induced by the program under consideration. For this purpose, basic concepts are introduced and an in-depth explanation of the IC3 algorithm and its different building blocks (search phase, generalization and propagation) is given. Based on these prerequisites the novel IC3CFA algorithm is presented. In this algorithm, the control-flow of the program is explicitly modelled as an automaton, while the variable valuations are handled symbolically, thus using the best of both worlds. Following the search phase of IC3CFA, solutions for applying generalization to IC3CFA are presented and problems arising with propagation are discussed. Finally, the performance of the IC3CFA algorithm and all proposed improvements is extensively evaluated on a set of well-recognised benchmarks. To set these results into a relation, a comparison with other available IC3 software model checking implementations concludes this thesis and underlines the strong potential of the IC3CFA model checking algorithm.

Zusammenfassung

In einer Zeit, in der Computer immer kleiner und leistungsfähiger und ihre Software immer komplexer wird und tiefer denn je unser tägliches Leben durchzieht, wachsen auch die Risiken von Fehlverhalten und die daraus resultierenden Schäden drastisch. Um solch ein unspezifiziertes Verhalten zu verhindern, hat sich das Model Checking, eine formale Verifikationstechnik zur Bestimmung funktionaler Eigenschaften von Informations- und Kommunikationssystemen, als besonders nützlich herausgestellt.

Zum Beweis mathematischer Eigenschaften ist eine der ersten in den Schulen unterrichteten Methoden die Induktion. Mit ihrem Konzept, zunächst eine konkrete Induktionsbasis und anschließend einen abstrakten Induktionsschritt als Stellvertreter für beliebige Schritte zu beweisen, ist sie eine sehr einfache und intuitive, nichts desto trotz aber mächtige Beweistechnik. Dennoch kann es sehr schwierig sein eine induktive Formulierung für eine komplexe Eigenschaft zu finden. Wenn Menschen ein solches Problem zu lösen versuchen, erstellen sie intuitiv eine Menge kleinerer, einfacher Lemmata, welche zusammengenommen die gewünschte Eigenschaft implizieren. Jedes dieser Lemmata gilt relativ zu einer Teilmenge der vorher aufgestellten Lemmata und nutzt deren Wissen, um das neue Lemma zu beweisen.

Dieser inkrementelle Ansatz, komplexe Eigenschaften mit Hilfe einer Menge kleiner, induktiver Lemmata zu beweisen, wurde für das Model Checking von Hardwaresystemen erstmals im IC3 Algorithmus angewendet und hat seitdem sämtliche existierenden Techniken zur Verifikation von Hardware geschlagen.

Die vorliegende Arbeit zielt darauf ab, die von IC3 benutzen Methoden auf die Verifikation von Software für industrielle Steuerungsanlagen anzuwenden. Hierbei wird besonderes Augenmerk auf die Ausnutzung des vom Programm vorgegebenen Kontrollflusses gelegt. Zu diesem Zweck werden zunächst grundlegende Konzepte eingeführt und der IC3 Algorithmus mit seinen verschiedenen Phasen (Suchphase, Generalisierung und Propagation) im Detail erläutert. Auf dieser Grundlage wird der neue IC3CFA Algorithmus vorgestellt, welcher den Kontrollfluss explizit als Automaten modelliert, während die Variablenbelegungen symbolisch dargestellt werden, sodass die Vorteile aus beiden Welten vereint werden können. Nachfolgend wird die Leistungsfähigkeit des Algorithmus, sowie aller vorgestellten Verbesserungen ausführlich anhand eines anerkannten Satzes von Referenzprogrammen evaluiert. Um die erzielten Ergebnisse in einen Kontext zu setzen und die Leistungsfähigkeit des vorgestellten IC3CFA Algorithmus zu unterstreichen, schließt ein Vergleich zu anderen, bestehenden Implementierungen von IC3 Verifikationsalgorithmen für Software diese Arbeit ab.

Acknowledgements

Being a Ph.D. student and writing a dissertation takes several years in which you get in contact with many different people that provide insights, help, ideas, feedback and other ways which have helped shape this thesis. While being grateful to all of them, I like to dedicate a few lines to the most important of these people:

First of all I thank the Siemens AG for making this work possible in the first place. In particular, I thank Martin Neuhäußer, who has been a continuous source of help, guidance and inspiration from the first day I worked in the project to the very last days of writing my dissertation. Whenever there were any uncertainties or problems ranging from the most theoretical proofs to the most practical compiler tweaks, Martin has been there to provide help regardless of time and place.

I also want to express my deep gratitude to Joost-Pieter Katoen and Thomas Noll for all their guidance and feedback to papers and in particular this thesis, many of which came back so in-depth and fast that it often seemed somewhat superhuman. Thank you Joost-Pieter for being the kind of boss that many, many people would wish for, but only few have the luck to experience.

But even with a great boss, work life is nothing without excellent colleagues. Thankfully, I was lucky enough to find both, so thank you Christina, Matthias, Tim, Souy, Chris, Benni, Jip, Federico, Friedrich, Gereon, Stef, Johanna, Marcel, Jera for all the great time and many awesome discussions, thanks to Elke, Birgit and Arnd for all the organizational help, and thanks to Sabrina, Nils and Flo for our amazing conference trips. But, in particular, I would like to thank Christoph for countless meetings at the breakfast club and my office mates Harold, Philipp and Sebastian for all the laughs we've shared over the years.

Of course, designing something like IC3 Software Model Checking is not a one-man show and so besides the great help from the people at Siemens, I'd like to thank all students who have shared parts of my journey.

And because no journey is possible without a start, I'd like to thank my parents who have always supported me to go my own way, regardless of the direction life would take me. Thank you mum and dad for all the love and support over the past 30 years.

Last but not least, I'd like to thank my wife Sina, possibly the only one who has experienced every single up and down over the past years and who has always been there to support me.

Contents

Chapter 1

Introduction

A loud crashing noise from shattering aluminium echoed through the car factory building and startled up the workers for a short moment, before they returned back to work. What had happened was that the robot arm picking up a wheel to mount it to the car had lifted the wheel, moved it towards the car, but half way smashed the wheel into the ground with full power. While this accident might sound frightful, it did not draw much attention, since this situation had occured multiple times in the previous weeks, but up to now no one was able to find the cause, partially due to the fact that these incidents only occurred very sporadically and never at the same production line twice.

It was only after this fifth incident over a time of more than three months, that an investigator noticed the connection between all these incidents: Each incident happened shortly after one specific control engineer had serviced the controller of the respective production line that was about to fail. A deeper investigation revealed that this control engineer had modified the control software on his computer because he thought some variable initializations would be unnecessary and therefore deleted the respective part of the control software.

Whenever the engineer would then connect his computer to the controller, it would detect a modified version of the control software and would automatically upload the new and faulty software to the controller. However, as it turned out, the missing variable initializations caused a misbehaviour in certain border-line cases that made the robot arm drive the wheel unit into the ground. Interestingly, a large test suite for the control software existed and all regression tests passed successfully. Thankfully, nobody was injured, the error was found and only some dents in the floor remind of these incidents.

1.1 Programmable Logic Controllers

While the out-of-control robot arms only caused a small damage, it is not hard to see that these incidents could have ended totally different if a worker had stood under the robot arm. And just like this robot arm, many other machines in various scenarios are able to cause danger for life and limb of people interacting with these machines.

However, with a certain level of safety required, the question is how this can be ensured. Historically, testing has been a popular approach to test software systems for errors under certain inputs. But while testing can reveal errors, it is never able to guarantee the absence of errors, because testing all possible combinations of input values for any meaningful program is infeasible: Consider a program with 5 integer inputs, each 16 bits in size, the number of input combinations is $\left(2^{16}\right)^5 = 65536^5$.

In order to fill this gap, formal verification aims at inspecting all possible executions, without actually executing the program. Together with a given formal specification, verification is able to determine for all possible behaviour of the system, whether this behaviour is according to specification or not. This way unintended behaviour can be reported to the user in terms of a concrete example where the software under inspection does not behave as specified, i.e. a *counterexample* to the specification. But even more important, most verification algorithms are able to indicate that there does not exist any behaviour in the system that violates the specification and therefore unintended behaviour, with respect to the given specification, can be ruled out.

As such, formal verification is a very useful tool in order to ensure safety under all possible scenarios and has already been successfully employed in many domains, such as the *Maeslantkering* storm surge barrier [Kars, 1996] protecting the city of Rotterdam from flooding. However, often the verification approach in these scenarios is tailored specifically towards this specific project and can not be applied to other projects. This is obviously very cost-intensive and prevents the application of formal verification to small and medium scale projects.

Let us reconsider the setting of the out-of-control robot arm. Apart from such industrial production lines, many other applications, such as traffic lights, elevators, escalators, waterworks, chemical plants and millions of other industrial applications that need to be controlled by software use *Programmable Logic Controllers* (PLC), which are special computers that are custom-tailored for control engineering applications. As such, the application of verification to PLC code would allow it to be used in a variety of domains, which are more likely to expose dangers to safety than other systems, such as personal computers.

1.2 Approach

While programs for personal computers are designed to execute a set of instructions and terminate, the execution model of PLCs is different. Because controlling e.g. a chemical plant is no task that should terminate, but rather a continuous reaction to sensor inputs using actuators, a PLC program is executed in a cyclic manner, i.e. inputs are read, the program computes the new values of the actuators and writes the output. Typical cycle times, depending on the computing power of the PLC and the program length, can be as fast as a few milliseconds.

A verification framework for PLC code would have to consider arbitrarily many cycles, but checking this arbitrarily long sequence is not feasible. As such, the verification framework uses abstraction on the input values to restrict the inputs as little as possible and only consider a single execution of the program. By abstracting the inputs, it is able to not only find counterexamples in the first cycle, but in any arbitrary cycle. In addition, considering only one execution of the PLC program brings the program closer to software for personal computers and enables the use of existing approaches for the verification of software for personal computers, as well as much larger sets of benchmarks to evaluate the performance of the verification framework.

For this reason, the thesis focuses on a novel verification framework for software verification of general programs, such as e.g. C programs, and is implemented and evaluated on a set of benchmarks from the international software verification competition. Only for some minor details the differences between C and PLC programs will be highlighted and the resulting decisions will be influenced by the peculiarities of PLC programs.

1.3 Outline

The remaining chapters of this thesis are structured as follows:

- Chapter 2 starts with an introduction to logics and satisfiability, in particular *propositional logic* and *Boolean satisfiability*, which are needed for Chapter 3, as well as *first-order logic* and *satisfiability modulo theories*, which is required for Chapter 4. Furthermore it covers an overview of *model-checking*, and different types of *properties*, which leads to *symbolic* *model-checking*. The chapter concludes with basics about the PLC programs that are considered in the remainder of the thesis.

- Chapter 3 focuses on the incremental, inductive verification algorithm *IC3* and starts with a description of a prior version of inductive verification, called *finite-state inductive strengthening*, which illustrates the idea of inductive verification, followed by the incremental extension IC3. The IC3 algorithm consists of a *main search phase* and two important extensions, namely *generalization* and *propagation*, which are considered separately.
- Chapter 4 presents the lifting of IC3 (Chapter 3) to software and is structured analogously to Chapter 3. It starts with previous approaches and the remainder of the chapter presents the lifting of IC3 to control-flow automata (IC3CFA), representing input programs. The presentation of the main search phase is followed by adaptations and improvements of generalization and a discussion about propagation. The chapter concludes with a comparison to other IC3-style verification algorithms.
- In Chapter 5 an implementation of IC3CFA is explained and used to evaluate the results of all ideas presented in Chapter 4. The chapter concludes with a short overview of some industrial experiences with the application of the presented implementation to the verification of PLC code.

1.4 Prior Publications

Parts of this thesis have been published in prior work. The following gives an overview where the work in this has been published.

The theoretical aspects of Section 4.2, related to the IC3CFA algorithm without generalization, are the result of fruitful discussions with Martin Neuhäußer and Thomas Noll and I implemented these on top of a framework developed by Martin Neuhäußer and his colleagues. The results were published in:

T. Lange, M. R. Neuhäußer, and T. Noll (2015). "IC3 Software Model Checking on Control Flow Automata". In: *FMCAD*. IEEE, pp. 97–104.

The subsequent work on generalization in IC3CFA is mainly the result of the master thesis

F. Prinz (2016). "Generalisation methods for control-flow oriented IC3 algorithms". Master thesis. RWTH Aachen University which I supervised and assisted to implement. Some of these results, in particular *split, predecessor cubes, WEP-based inducttvity and generalization and generalization caching* have subsequently been published in:

T. Lange, F. Prinz, M. R. Neuhäußer, T. Noll, and J.-P. Katoen (2018). "Improving Generalization in Software IC3". In: *SPIN*. LNCS. To be published. Springer.

The core work of investigating propagation in IC3CFA, as presented in Section 4.4 is based on the master thesis of

T. Mertens (2016). "Efficient reuse of learnt information for control-flow oriented IC3 algorithms". Master thesis. RWTH Aachen University

which I supervised and assisted to implement. While not related to propagation, the obligation reuse as presented on page 112, has also been a result of this master thesis.

Some earlier work published in: T. Lange, M. R. Neuhäußer, and T. Noll (2013). "Speeding Up the Safety Verification of Programmable Logic Controller Code". In: *Haifa Verification Conference*. Vol. 8244. Lecture Notes in Computer Science. Springer, pp. 44–60

contains aspects of the static program minimizations that are subject of Section 5.1.1.

1.5 Contributions

This thesis contributes to the state-of-the-art in IC3-style software verification theoretically, as well as practically. Our main contributions are the following:

- We present the IC3CFA algorithm, in particular the main search phase, which lifts the way IC3 works on hardware systems to control-flow automata, a very common encoding for programs, in the most straightforward way. We prove the correctness of the algorithm and highlight the equivalences between IC3 and IC3CFA.
- Based on the IC3CFA search phase, we present a number of optimizations to the search phase that go beyond the simple lifting of IC3.
- We introduce a simple generalization for IC3CFA and present the challenges to be tackled when applying generalization. Again, we start with a basic approach that tries to mimic the technique employed in IC3 as close

as possible and highlight the main differences that have to be considered, especially with respect to multiple explicit predecessors state sets.

- Following standard generalization, we introduce improvements enabled by the exact predecessor computations that has to applied in IC3CFA and how it is used to modify the generalization of cubes.
- In addition, we present novel ways of handling generalizations in efficient ways, most notably the caching of generalizations in varying contexts and how we can use cached results, even though they appeared in a different situation, thus exceeding the spectrum of standard caching.
- Apart from our novel techniques for generalization in IC3CFA, we also consider known generalization approaches such as unsatisfiable cores and interpolation, and discuss their application to IC3CFA.
- The propagation phase, a vital part of the IC3 algorithm, apart from the search phase and generalization, is considered and we discuss the problems that arise when trying to lift propagation to IC3CFA.
- In addition to these theoretical contributions, we implemented a prototypical verification framework with all presented algorithms and improvements and evaluated the performance of all proposed methods in an isolated fashion on a large set of benchmarks.

Chapter 2

Preliminaries

In a world where information and communication systems have become ubiquitous, and software interacts with multiple aspects of our daily live, software failures pose problems more than ever. As a result, many of such systems need a reliable way to ensure the absence of errors, such as formal verification of software. In order to advance the state-of-the-art in that direction, this thesis presents a way to apply the so-called *IC3* algorithm, as explained in Chapter 3 to the domain of software systems, presented in Chapter 4. In order to agree on a common understanding, we define the basic theoretical concepts of model-checking and our notion of software, necessary for our presentation in Chapter 4, in the remainder of this chapter. In order to pave the way towards efficient, state-of-the-art model-checking, we start with defining logic in Section 2.1, in particular *propositional logic*, *first-order logic* and *first-order theories*, which will be used later as our basic units of reasoning. We continue in Section 2.2 with the *satisfiability* of formulas in these logics and presenting ways of automated decision procedures for the satisfiability of formulas in propositional logic and first-order theories. These tools, called satisfiability (SAT) *solvers*, will allow us to formulate questions in terms of logical formulas and determine the answer. There exists a variety of such solvers that all have strengths and weaknesses for particular problem domains, such that we can reuse existing software and choose a solver depending on our needs. Building on top of logics and their satisfiability, we give a brief outline of formal verification with *model-checking*. This technique verifies a system by abstracting it to a theoretical model that can be checked for violations of the requirements, given as a property of the model. We conclude this chapter by defining a modelling formalism for software programs that is tailored towards facilitating our software model-checking algorithm presented in Chapter 4. The presented formalism is of specific use due to its succinctness, that allows us to define small sets of rules for verification. Furthermore, we define our formalism using different layers that distinguish control and data flow. This separation of concerns will later allow us to simplify our reasoning about those programs by ignoring irrelevant parts.

2.1 Logic

The origins of *logic* in its modern perception stem from the ancient greek, where the word *logos* meant *thought* or *reason*. In its general interpretation, logic considers the systematic study of the *form* of *valid inference*. This contains three important aspects of logic: The central concept of the *logical form* says that the validity of some abstract argument is determined by the logical form of the argument, not its content. In particular, the *validity* of the argument is determined by the meaning, the *semantics*, of the sentences of which the argument consists. Lastly, an *inference* consists of two propositions p and q that are asserted individually, in the form p *therefore* q. One of the most prominent inferences in common parlance may be *cogito ergo sum*, *I think; therefore, I am* by the French philosopher René Descartes.

In the course of history, logic has been studied in philosophy since the ancient times, later, since the mid of the 19th century, in mathematics and in its most recent forms in (theoretical) computer science. In this sense, logics such as propositional logic, outlined later in this section, that are main foundations for modern computer science are as old as the 3rd century BC.

We will, however, omit the wide field of logics and rather focus on a few, specific aspects and logics that will be of use for subsequent theoretical and practical application. We will start in Section 2.1.1 with *propositional logic*, which offers basic understanding and reasoning about binary systems. We extend these concepts in Section 2.1.2 with quantifiers, and functions over arbitrary domains to *first-order logic*. Section 2.1.2 completes this section with a number of *first-order theories* that can be considered instantiations of first-order logic over concrete domains.

2.1.1 Propositional Logic

As the name indicates, *propositional logic*, the oldest of the presented logics, studies formal systems over propositions, i.e. statements that are either *true* or *false*. In its modern interpretation in theoretical computer science, these propositions can be related directly to bits that have states 1 and 0. In [Boole, 1853], the British mathematician and philosopher, George Boole published a systematization and foundation to the principles of Aristotle's logic, which later became the modern propositional logic, sometimes also referred to as *Boolean logic*. In the remainder of this section we will outline the main aspects of propositional logic, starting with its syntax, i.e. the logical form of propositional arguments. We continue, giving the syntactic constructs a meaning by defining its semantics and conclude by ways to normalize propositional formulas with some widely used normal forms.

Definition 2.1 (Syntax propositional logic [Bradley and Manna, 2007b])**.** The syntax of propositional logic (PL, shortly) is defined by the following grammar:

$$p := true \mid false \mid x \mid \neg p \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid p_1 \Rightarrow p_2 \mid p_1 \Leftrightarrow p_2$$

Given Definition 2.1, we can identify three terminal symbols, also called *atoms*, which are the truth symbols *true* and *false* and *propositional variables* denoted by symbols like x, y, z, of which a countably infinite set is assumed to exist. Apart from these terminal symbols, PL is defined over unary *negation* operator $\neg$ and the binary connectives *conjunction* $\wedge$, *disjunction* $\vee$, *implication* $\Rightarrow$ and *iff* $\Leftrightarrow$. The left argument of the implication is called the *antecedent* and the right the *consequent*. A *literal* is defined as an atom α or its negation $\neg\alpha$. A *formula* is a literal or the application of a connective. Whenever operator precedence is clear from the context, we will avoid unnecessary parentheses. To simplify chains of conjunction or disjunction operators, we define n-ary conjunction and disjunction later.

Example 2.1. Consider the natural statement ”the sun is shining”. Depending on place, time and weather, this statement might be *true* or *false*, i.e. we say that ”the sun is shining” is a variable, called *sun*. If we now want to express a condition when we can leave the house without a jacket, we could say that ”the sun is shining” *and* ”it is warm outside” (*warm*), i.e. $sun \wedge warm$. Furthermore, we could express that ”the sun is shining” *implies* that it is *not raining*, i.e. $sun \Rightarrow \neg rain$. The most noticeable difference between natural language and the connectives in propositional logic is that *or* in natural language is usually meant exclusive, e.g. ”the sun is shining” *or* ”it is raining”, i.e. $sun \vee rain$. While we would usually interpret this statement as *true* iff one of the state sub-statements holds, propositional logic allows both to hold at the same time and the connective still being *true*.

The syntactic structure as defined in Definition 2.1 does not yet have any meaning, e.g. we don't know when $sun \wedge warm$ of Example 2.1 evaluates to *true*. Therefore we need to define the semantics of propositional logic. To do so, we start by defining a way of evaluating the truth value of a propositional variable, which is given by a so-called *interpretation* I. An interpretation I assigns to each propositional variable a truth value. If not all variables are assigned, we call I a partial interpretation and otherwise a full assignment. Given an interpretation I, the evaluation of propositional atoms is simple. The unary operator $\neg$ defines the negation of the evaluation of its argument, i.e. $\neg true = false$ and $\neg false = true$. To determine the evaluation of the remaining propositional connectives, we can use the following *truth table* [Bradley and Manna, 2007b]:

φ	ψ	$\varphi \wedge \psi$	$\varphi \vee \psi$	$\varphi \Rightarrow \psi$	$\varphi \Leftrightarrow \psi$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Given an interpretation I we can use this truth table to recursively determine the evaluation of any formula by the evaluation of its respective subformulas. While the use of truth tables can be convenient, it is not always suitable, which is why we introduce an inductive definition of the semantics of propositional logic using the *model* operator $\models$. Again, we start our definition with the semantics of propositional variables:

$$\begin{aligned} I &\models x && \text{iff } I[x] = \textit{true} \\ I &\not\models x && \text{iff } I[x] = \textit{false} \end{aligned}$$

This means that the variable x has value *true* iff the interpretation I assigns *true* to x. Based on the propositional atoms, we can now inductively define connectives as:

$$\begin{aligned} I &\models \neg\varphi && \text{iff } I \not\models \varphi \\ I &\models \varphi_1 \wedge \varphi_2 && \text{iff } I \models \varphi_1 \text{ and } I \models \varphi_2 \\ I &\models \varphi_1 \vee \varphi_2 && \text{iff } I \models \varphi_1 \text{ or } I \models \varphi_2 \\ I &\models \varphi_1 \Rightarrow \varphi_2 && \text{iff, } I \not\models \varphi_1 \text{ or } (I \models \varphi_1 \text{ and } I \models \varphi_2) \\ I &\models \varphi_1 \Leftrightarrow \varphi_2 && \text{iff } I \models \varphi_1 \text{ and } I \models \varphi_2 \text{, or } I \not\models \varphi_1 \text{ and } I \not\models \varphi_2 \end{aligned}$$

Example 2.2. Let us assume some given interpretation $I = \{sun \mapsto true, warm \mapsto true, rain \mapsto false\}$, then

$$\begin{aligned}
I &\models sun \wedge warm \\
I &\not\models rain \\
I &\models sun \Rightarrow \neg rain \\
I &\models sun \vee rain
\end{aligned}$$

As we can see from the semantics of the implication operator, not all connectives are necessary for the full expressiveness of propositional logic. Accordingly, there are a number of rewrite rules that preserve the semantics of the formula and can be applied to the operators in propositional logic, some of which shall be explained in the following [Bradley and Manna, 2007b]. We start with the rewrite rules for the negation operator.

$$\begin{aligned}
\neg\neg\psi &\iff \psi \\
\psi_1 \Rightarrow \psi_2 &\iff \neg\psi_1 \vee \psi_2 \\
\psi_1 \Leftrightarrow \psi_2 &\iff (\psi_1 \Rightarrow \psi_2) \wedge (\psi_2 \Rightarrow \psi_1)
\end{aligned}$$

A second set of equivalences that rewrites conjunction and disjunction under negation is known as *De Morgan's Law*:

$$\begin{aligned}
\neg(\psi_1 \wedge \psi_2) &\iff \neg\psi_1 \vee \neg\psi_2 \\
\neg(\psi_1 \vee \psi_2) &\iff \neg\psi_1 \wedge \neg\psi_2
\end{aligned}$$

The distributive law of propositional logic is defined as:

$$\begin{aligned}
\varphi \wedge (\psi \vee \vartheta) &\iff (\varphi \wedge \psi) \vee (\varphi \wedge \vartheta) \\
\varphi \vee (\psi \wedge \vartheta) &\iff (\varphi \vee \psi) \wedge (\varphi \vee \vartheta)
\end{aligned}$$

Another important equivalence, that will be used extensively in later chapters is the equivalence of implication:

$$\begin{aligned}
&\psi \Rightarrow \varphi \\
\iff &\neg\psi \vee \varphi \\
\iff &\neg(\psi \wedge \neg\varphi) \\
\iff &\neg\varphi \Rightarrow \neg\psi
\end{aligned}$$

Example 2.3. Consider some abstract formula:

$$\neg(x \wedge z) \Rightarrow \neg(y \Leftrightarrow \neg z)$$

Equivalent formulas are for example:

$$\begin{aligned}
&\Longleftrightarrow (\neg x \vee \neg z) \Rightarrow \neg(y \Leftrightarrow \neg z) \\
&\Longleftrightarrow (\neg x \vee \neg z) \Rightarrow \neg((y \Rightarrow \neg z) \wedge (\neg z \Rightarrow y))
\end{aligned}$$

In the standard notion of propositional logic, connectives such as conjunction and disjunction are binary, i.e. we can only apply it to two subformulas. In order to allow a conjunction of three atoms, e.g. *sun* and *warm* and *rain*, we have to nest those binary connectives, such as $(\mathit{sun} \wedge (\mathit{warm} \wedge \mathit{rain}))$. In order to improve readability and simplify subsequent definitions, we lift those binary connectives to n-ary connectives, which is enabled by the associativity of the operators.

Definition 2.2 (n-ary con-/disjunction)**.** Given a set of literals P with $|P| = n$, we define the n-ary conjunction as

$$\bigwedge P := \bigwedge_{l_i \in P} l_i := l_1 \wedge l_2 \wedge \cdots \wedge l_n$$

and analogously, we define the n-ary disjunction as

$$\bigvee P := \bigvee_{l_i \in P} l_i := l_1 \vee l_2 \vee \cdots \vee l_n.$$

Corollary 2.1. *The semantics of* $\bigwedge$ *and* $\bigvee$ *is given as:*

$$\begin{aligned}
\bigwedge_{l_i \in P} l_i &\models \mathit{true} \ \mathit{iff}\ \forall l_i \in P.\, I \models l_i \\
\bigvee_{l_i \in P} l_i &\models \mathit{true} \ \mathit{iff}\ \exists l_i \in P.\, I \models l_i
\end{aligned}$$

The semantics of these n-ary operators are defined analogous to their binary counterparts. Using this n-ary conjunctions and disjunctions, we can define constructs and normal forms on the syntax of the formula.

Definition 2.3 (Cube [Bradley, 2011])**.** A *cube* is defined as a conjunction of literals.

Definition 2.4 (Clause [Bradley, 2011])**.** A *clause* is defined as a disjunction of literals.

Since a clause is defined as disjunction over literals and a cube is a conjunction over literals, there exists a duality between both, in the form that one is the negation of the other.

In order to reason about formulas, there exists a variety of normal forms. The most popular three, *negation normal form*, *disjunctive normal form* and *conjunctive normal form* shall be defined in the following.

Definition 2.5 (Negation normal form [Bradley and Manna, 2007b])**.** A formula φ in propositional logic is defined to be in *negation normal form* (NNF) if it contains only the connectives $\neg, \wedge, \vee$ and negations only appear in literals.

An arbitrary formula can be transformed into NNF using the equivalence transformations on page 12.

Example 2.4. Consider the formula that we rewrote in Example 2.3. We can transform it into the following NNF formula using the given equivalences:

$$\begin{aligned}
& (\neg x \vee \neg z) \Rightarrow \neg\left((y \Rightarrow \neg z) \wedge (\neg z \Rightarrow y)\right) \\
\Longleftrightarrow\ & (\neg x \vee \neg z) \Rightarrow \neg\left((\neg y \vee \neg z) \wedge (z \vee y)\right) \\
\Longleftrightarrow\ & (\neg x \vee \neg z) \Rightarrow \left(\neg(\neg y \vee \neg z) \vee \neg(z \vee y)\right) \\
\Longleftrightarrow\ & (\neg x \vee \neg z) \Rightarrow \left((y \wedge z) \vee (\neg z \wedge \neg y)\right) \\
\Longleftrightarrow\ & \neg(\neg x \vee \neg z) \vee \left((y \wedge z) \vee (\neg z \wedge \neg y)\right) \\
\Longleftrightarrow\ & (x \wedge z) \vee (y \wedge z) \vee (\neg z \wedge \neg y)
\end{aligned}$$

While negation normal form only forbids negations to occur in non-atomic levels of the formula structure, other normal forms pose much harder restrictions on the structure of the formula.

Definition 2.6 (Disjunctive Normal Form [Bradley and Manna, 2007b])**.** A formula φ is defined to be in *disjunctive normal form* (DNF) if it is a disjunction of cubes, i.e.

$$\bigvee_i \bigwedge_j l_{i,j} \text{ for literals } l_{i,j}$$

Similar to cubes and clauses, we can define an analogy to the disjunctive normal form that has a top-level conjunction instead of disjunction.

Definition 2.7 (Conjunctive Normal Form [Bradley and Manna, 2007b])**.** A formula φ is defined to be in *conjunctive normal form* (CNF) if it is a conjunction of clauses, i.e.

$$\bigwedge_i \bigvee_j l_{i,j} \text{ for literals } l_{i,j}$$

While the last formula considered in Example 2.4 happens to also be in DNF, this is not necessarily the case for all formulas. However, there do exists equivalent CNF/DNF formulas for arbitrary formulas in propositional logic.

Example 2.5. Considering the formula of Example 2.4, we can transform it into CNF using the equivalence transformations as shown on page 12, especially the distributive law.

$$\begin{aligned}
& (x \wedge z) \vee (y \wedge z) \vee (\neg z \wedge \neg y) \\
\iff & (((x \wedge z) \vee y) \wedge ((x \wedge z) \vee z)) \vee (\neg z \wedge \neg y) \\
\iff & ((x \vee y) \wedge (z \vee y) \wedge (x \vee z) \wedge (z \vee z)) \vee (\neg z \wedge \neg y) \\
\iff & \ldots \\
\iff & (x \vee \neg z) \wedge (y \vee \neg z) \wedge (x \vee \neg y) \wedge (y \vee \neg y) \wedge \\
& (z \vee \neg z) \wedge (y \vee \neg z) \wedge (z \vee \neg y) \wedge (y \vee y) \wedge \\
& (x \vee \neg z) \wedge (z \vee \neg z) \wedge (x \vee \neg y) \wedge (z \vee y) \wedge \\
& (z \vee \neg z) \wedge (z \vee \neg z) \wedge (z \vee \neg y) \wedge (z \vee y)
\end{aligned}$$

As we can see from Example 2.5, while there exists an equivalent CNF and DNF for every formula, some of these transformations may lead to formulas exponential in the size of the original formula. To tackle this problem, there exist other techniques that do not create equivalent formulas, but formulas that are equisatisfiable, i.e. formulas that are equivalent with respect to their satisfiability, such as Tseitin transformation [Tseitin, 1968].

2.1.2 First-order Logic

Let us consider the basic logic over the propositions *true* and *false* that was established in the previous section. This Boolean logic is perfectly suited for reasoning about Boolean systems, such as finite-state machines, a very common modelling formalism for real-world applications like hardware circuits. However, if we want to model a system that reasons about computations, the expressive power of propositional logic does not always suffice. We therefore extend the concepts introduced in the last section with functions, quantifiers and predicates, which gave name to the *predicate logic* or *first-order logic* (FO) [Bradley and Manna, 2007b]. Like in the last section, we will start by introducing the syntax of first-order logic and afterwards define the semantics of the syntactic elements.

Definition 2.8 (Syntax of first-order logic [Bradley and Manna, 2007b]**).** The syntax of first-order logic is defined by the following grammar:

$$\begin{aligned} t &:= a \mid x \mid f(t_1, \ldots, t_n) \\ \mathit{fo} &:= \exists x.\, \mathit{fo} \mid \forall x.\, \mathit{fo} \mid \neg \mathit{fo} \mid \mathit{fo}_1 \vee \mathit{fo}_2 \mid \mathit{fo}_1 \wedge \mathit{fo}_2 \mid \\ &\quad\ \mathit{fo}_1 \Rightarrow \mathit{fo}_2 \mid \mathit{fo}_1 \Leftrightarrow \mathit{fo}_2 \mid p(t_1, \ldots, t_n) \end{aligned}$$

As we can see from Definition 2.8, the syntax of first-order logic splits up in two main aspects, namely *terms* t and *formulas* fo. While formulas, just like for the propositional case evaluate to a Boolean, terms will evaluate to a value in their domain. A more detailed explanation on instantiations of concrete domains will be given in Section 2.1.2. Apart from the known concepts of PL, such as negation, conjunction, and so on, FO-formulas can contain three new syntactic elements: First of all, predicate symbols $p(t_1, \ldots, t_n)$ take n terms as input and map them to a Boolean. Furthermore FO adds the concept of *existential quantifiers* $\exists$ and *universal quantifiers* $\forall$. In $\forall x.\, \mathit{fo}$ and $\exists x.\, \mathit{fo}$, we call x the *quantified variable* and fo the *scope* of the quantifier $\forall x$. The variable x is called *bound* in fo by the quantifier. All variables that occur in fo and are not bound by some quantifier are called *free variables*. Formulas that do not contain free variables are called *closed* formulas. Sometimes a predicate is also considered as a generalized propositional variable [Bradley and Manna, 2007b].

Terms on the other hand can consist of a first-order variable x over some abstract domain D, a constant value a, or a function $f(t_1, \ldots, t_n)$ that evaluates to a value in the domain under n terms. The constant value a can be considered a 0-ary function.

Analogously to PL, we call a truth value or an n-ary predicate an *atom.* An atom or its negation is considered a *literal.*

Given the formal syntax of first-order logic, we define *FO* as the set of all possible words that can be derived from *fo* in the given grammar with a formula $\varphi \in \mathit{FO}$ being a word of *FO*.

After defining the syntactic elements of FO, we still need to define the semantics, giving a meaning to each concept. For the Boolean fragment of FO we can use the semantics of PL. However, this does not cover terms that evaluate to values other than truth values. In order to do so, we need to extend the known definition of *interpretation* I, whose domain D_I is now a nonempty set of values or objects. Given an interpretation I, an *assignment* α_I maps constants and variables to elements of D_I, functions to functions over D_I and predicate symbols to predicates over D_I. Together, the interpretation $I = (D_I, \alpha_I)$ is a pair, consisting of domain D_I and assignment α_I.

To determine whether a formula φ evaluates to *true* or *false* under a certain interpretation I, we define the semantics inductively:

$$\begin{aligned} &\alpha_I[f\,(t_1, \ldots, t_n)] = \alpha_I[f]\,(\alpha_I[t_1], \ldots, \alpha_I[t_n]) \qquad\qquad \text{and} \\ &\alpha_I[p\,(t_1, \ldots, t_n)] = \alpha_I[p]\,(\alpha_I[t_1], \ldots, \alpha_I[t_n]) \end{aligned}$$

for functions f, arbitrary predicates p and terms $t_1, \ldots, t_n$. Using the assignment α_I we define

$$\begin{aligned} &I \models f\,(t_1, \ldots, t_n) && \text{iff } \alpha_I[f\,(t_1, \ldots, t_n)] \text{ and} \\ &I \models p\,(t_1, \ldots, t_n) && \text{iff } \alpha_I[p\,(t_1, \ldots, t_n)]. \end{aligned}$$

For the semantics of quantifiers, we need to modify interpretation I slightly. We say that for a bound variable x an interpretation $J : (D_J, \alpha_J)$ is an *x-variant* of $I : (D_I, \alpha_I)$ if $D_I = D_J$ and $\alpha_I[y] = \alpha_J[y]$ for all symbols y except x. In other words, J and I are identical, except for the value of variable x. For some $v \in D_I$ we call $J : I \lhd \{x \mapsto v\}$ the x-variant of I where $\alpha_J[x] = v$. Using x-variants, we define the semantics of quantifiers as:

$$\begin{aligned} &I \models \forall x.\, \varphi && \text{iff for all } v \in D_I, I \lhd \{x \mapsto v\} \models \varphi \\ &I \models \exists x.\, \varphi && \text{iff there exists a } v \in D_I, I \lhd \{x \mapsto v\} \models \varphi \end{aligned}$$

Example 2.6. Consider the interpretation $I = (D_I, \alpha_I)$ over the infite domain of integers $D_I = \mathbb{Z}$ and the assignment $\alpha_I = \{+ \mapsto +_{\mathbb{Z}},\ \cdot \mapsto \cdot_{\mathbb{Z}}, = \mapsto =_{\mathbb{Z}},$
$x \mapsto 1, y \mapsto 2, z \mapsto 3\}$, then the formula

$$x \cdot z = x + y$$

evaluates to *true* under interpretation I, while

$$x + y + z = x \cdot y$$

evaluates to *false* under I.

In order to allow transformations of FO-formulas into the known normal forms of propositional logic, we need to define a few missing equivalence transformations. In particular, we can transform an arbitrary FO-formula into NNF using the following equivalences:

$$\neg\forall x.\, \varphi[x] \iff \exists x.\, \neg\varphi[x]$$
$$\neg\exists x.\, \varphi[x] \iff \forall x.\, \neg\varphi[x]$$

With the advent of quantifiers in FO-formulas, we can also define a normal form that considers the quantifiers in a FO-formula, which is called *prenex normal form*.

Definition 2.9 (Prenex normal form [Bradley and Manna, 2007b])**.** A FO-formula ψ is in *prenex normal form* (PNF, short) if all quantifiers in ψ appear at the beginning:

$$Q_1 x_1 \dots Q_n x_n .\, \varphi[x_1, \dots, x_n]$$

with quantifiers $Q_i \in \{\forall, \exists\}$ and quantifier-free φ.

Any arbitrary FO-formula ψ can be transformed into PNF by first converting ψ into NNF-formula ψ', renaming all quantified variable symbols in ψ' to prevent conflicts with other quantified or free variable symbols, resulting in formula ψ'' and shifting all quantifiers in ψ'' to the beginning of the formula.

Example 2.7. Consider the FO-formula

$$\varphi = \forall x.\,(\,x + 1 > 0 \vee \exists y.\,(\,-1 \cdot y = x \wedge \neg \forall z.\,(\,z = y \Rightarrow z > 0)))\,.$$

We start by transforming φ into NNF, resulting in

$$\begin{aligned} \psi &= \forall x.\, x + 1 > 0 \vee \exists y.\, -1 \cdot y = x \wedge \exists z.\, \neg\,(z = y \Rightarrow z > 0) \\ \iff & \ \forall x.\, x + 1 > 0 \vee \exists y.\, -1 \cdot y = x \wedge \exists z.\, z = y \wedge z > 0. \end{aligned}$$

Given ψ, we can now shift quantifiers, resulting in the PNF formula

$$\forall x, \exists y, \exists z.\, x + 1 > 0 \vee -1 \cdot y = x \wedge z = y \wedge z > 0.$$

First-order Theories

In the last section, we defined FO with interpretations over arbitrary domains. In other words, we built an abstract framework that we now want to instantiate. To do so, we will define so-called *first-order theories* that allow us to reason about application domains, such as programs.

Definition 2.10 (Theories [Kroening and Strichman, 2008])**.** A *first-order theory* $T : (\Sigma, \mathcal{A})$ is defined as a pair consisting of a set $\Sigma = C \cup F \cup P$ of constant, function and predicate symbols, called *signature* and a set $\mathcal{A}$ of closed FO-formulas in which only symbols of Σ appear, called *axioms*.

In the following chapters, we will use a number of concrete theories, that we will shortly sketch in the following. We will give the signature of the used theories, but the axiomatization, while mostly intuitive, can become very complex. For this reason we only give an informal description of the axiomatization or omit it completely, where obvious. For more details about the concrete axiomatization of the used theories, the reader is referred to the literature [Biere, Heule, et al., 2009; Bradley and Manna, 2007b; Kroening and Strichman, 2008].

Bitvector Theory (BV)

The Bitvector Theory (BV) is a theory that is highly relevant when reasoning about any sort of program. Regardless of the size and the type of a program variable, it will always have a finite representation consisting of n bits. A bitvector, as the name suggests, consists of a finite vector of Boolean entries, or bits, and is therefore ideal to model program variables. Furthermore, the used bitvector arithmetic allows to model not only the finite value of a program variable, but also enables us to determine its precise behaviour, such as underflows and overflows. The signature of the bitvector theory is given as follows :

$$\begin{aligned}\Sigma = \{&+, -, \cdot, /,\\ &\ll, \gg, \&, |, \oplus, \sim, t[c_1 : c_2], \circ\\ &<, =\\ &\wedge, \vee, \neg\}\end{aligned}$$

[Kroening and Strichman, 2008]

We can identify the following classes and elements of signature Σ:

- Arithmetic operators, such as addition $+$, subtraction $-$, multiplication $\circ$ and division $/$ with respect to bitvector logic, such as overflow and underflow,
- bit-level operators, such as shift $\ll$ and $\gg$, bit-level AND & and OR $|$, XOR $\oplus$ and bit-level negation $\sim$, extraction of a subvector from position c_1 to c_2 from some term t and concatenation $\circ$,
- relational operators, such as $<$ and $=$, and
- Boolean connectives, such as conjunction, disjunction and negation.

While there exist certain variable types that we cannot directly model using bitvectors, such as arrays or pointers, bitvectors offer a convenient way to model basic variable types.

Example 2.8. Using the bitvector theory we can reason about formulas such as

$$(((x \ll 0010) \,|\sim (y \oplus z)) + x) = w \wedge w[0100 : 0011] \circ w[0100 : 0011] = y$$

which evaluates to *true* under e.g. $\alpha_I = \{w \mapsto 0100, x \mapsto 1101, y \mapsto$

$0101, z \mapsto 1001, \dots\}$:

$$\begin{aligned}
&(((1101 \ll 0010) \,|\, \sim (0101 \oplus 1001)) + 1101) = 0100 \wedge \\
&0100[0100 : 0011] \circ 0100[0100 : 0011] = 0101 \\
\Longleftrightarrow\ &((0100 \,|\, \sim 1100) + 1101) = 0100 \wedge \\
&01 \circ 01 = 0101 \\
\Longleftrightarrow\ &((0100 \,|\, 0011) + 1101) = 0100 \wedge \\
&0101 = 0101 \\
\Longleftrightarrow\ &(0111 + 1101) = 0100 \wedge \\
&0101 = 0101 \\
\Longleftrightarrow\ &0100 = 0100 \wedge \\
&0101 = 0101
\end{aligned}$$

Undefined Functions (UF)

When reasoning about programs, input variables or nondeterministically determined values are usually of special interest. Without those, any program would only consist of computations about constants defined somewhere in the program, such that the model-checking problem for those programs could be solved by simulating the program or using a simple static program analysis. In real-world applications however, one is interested in how a program behaves under a certain set, interval or even arbitrary input values. To do so, we can employ so-called *undefined functions*. For undefined function symbols an interpretation need not satisfy any axioms for that function apart from it being consistent, i.e. given the same input, it produces the same output. The signature of the theory of undefined functions or, more precisely, the theory of equality and undefined functions (EUF) is simply given by the standard Boolean connectives, equality operator $=$ and the function symbols, usually denoted by capital letters [Kroening and Strichman, 2008].

Arrays

The concept of Arrays is widely used in various programming languages and allows iteration over consecutive memory arrays. The previously defined theories however do not allow this, since they abstract from concrete memory. In other words, a variable in BV does not occupy a specific memory region and thus there is no knowledge about the order in which they are allocated. To model arrays we thus use the dedicated *array theory*. Since arrays can also be considered as a mapping from an index to an element, the array theory combines an index theory and an element theory [Kroening and Strichman, 2008]. The signature of the array theory contains, apart from the Boolean fragment and quantifiers, just two operators: One for writing an element to an index, denoted by $term_A\{term_I \leftarrow term_E\}$ and reading from an index, denoted by $term_A[term_I]$. Those operations distinguish between terms of array theory $term_A$, terms of index theory $term_I$ and terms of element theory $term_E$ [Kroening and Strichman, 2008].

Linear Arithmetic

While BV theory allows us to verify bit-precise behaviour of software, such as overflows and bit-level operations like shifts, XOR and so on, we may add a lot of computational overhead for cases where this is not necessary. For example, we might either encounter a system in a language that does not allow bit-level operations or we may be able to statically detect intervals for all relevant variables that do not exceed the variables' sizes. In those cases we might as well reason about natural, unbounded integers which allows the solver to work more efficiently. For reasoning about programs two suitable theories are the theory of *Linear Integer Arithmetic* (LIA) and *Linear Real Arithmetic* (LRA) for integer and rational variables. As the name suggests, both theories only allow *Linear* equations and inequations. This means that LIA and LRA formulas can only be derived from the following grammar:

$$\begin{aligned}
\mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &: \mathit{sum}\ \mathit{op}\ \mathit{sum} \\
\mathit{op} &: = \mid \leq \mid < \\
\mathit{sum} &: \mathit{term} \mid \mathit{sum} + \mathit{term} \\
\mathit{term} &: \mathit{identifier} \mid \mathit{constant} \mid \mathit{constant}\ \mathit{identifier}
\end{aligned}$$

[Kroening and Strichman, 2008]

In other words, terms can be variables, constants or the product of both; terms can be summed and be subject to a relational operator which forms an atom. Such atoms or the conjunction of them is an LIA/LRA formula. Both theories share the same grammar, only that in LIA constants and variable valuations can only be integers, while for LRA they can be any real number.

2.2 Satisfiability

In the previous section we always assumed a given interpretation I for the presented logics. However, if we want to use logics to model a given problem, we do not know whether such I exists and if so how exactly it looks. Given a formula φ, if such I exists, we say that I *satisfies* φ or simply that φ is *satisfiable*, written as $\models \varphi$.

The problem whether *a propositional formula* φ *has a satisfying interpretation* I, called the *Boolean satisfiability problem* (SAT) is decidable but its complexity has been proven to be *NP-complete* [Cook, 1971], since we can non-deterministically guess the correct interpretation and check whether it satisfies φ in polynomial time. Despite its complexity there exist multiple SAT solvers [Audemard et al., 2013; Biere, 2014; Eén and Sörensson, 2003] that try to solve the satisfiability problem using various heuristics. During the last decade these tools have improved significantly and for many real-world problems they show impressive performance.

In certain situations however we might also be interested in whether φ evaluates to true not only under one I but rather under all possible interpretations. This is specifically the case if we consider an implication: Every interpretation that does violate the premise will automatically satisfy φ. However in those cases we are actually more interested whether φ evaluates to true under all interpretations I. If this is the case, we say that φ is *valid*. Due to the duality between validity and satisfiability we can reduce any validity problem to a satisfiability problem, such that we can use a SAT solver to check validity. In particular a formula φ is valid iff its negation $\neg\varphi$ is unsatisfiable.

For the remainder of this section we will introduce the basic concepts of the satisfiability problem, first for propositional logic and afterwards for first-order theories.

2.2.1 Boolean Satisfiability

Over the last decade the theoretical, as well as practical importance of the Boolean satisfiability problem has grown significantly and thus led to a large amount of research in the field. This in turn improved the performance and therefore the visibility of SAT solving.

While the problem itself is decidable, but NP-complete, two major heuristics for solving SAT problems exist. The first of those heuristics is *stochastic search* and, as the name suggests, it searches the space of all possible interpretations for a formula φ in a stochastic way, by guessing full interpretations I. If I does not satisfy φ, stochastic search starts flipping variable valuations with some greedy heuristic [Kroening and Strichman, 2008]. While stochastic search clearly has its advantages for randomly generated SAT instances, formulas modelling real-world problems usually have some degree of structure. Therefore heuristics that are able to exploit and learn from that structure seem to have a general advantage for practical applications. The most prominent heuristics in that regard is the *Davis-Putnam-Loveland-Logemann* (DPLL) framework. In contrast to stochastic search, DPLL traverses and backtracks over a binary tree [Biere, Heule, et al., 2009; Kroening and Strichman, 2008] and thus solves the problem in a more structured way. Due to this advantage for practical application, we will focus the remainder of this section on DPLL solvers exclusively.

From a very high-level point of view, the DPLL framework can be considered as follows: It makes a *decision* about a variable valuation, *propagates* the effects of this change to all applicable positions and in case this leads to a conflict it backtracks the decision. If we consider each of the resulting partial interpretations as a node, deciding to give a variable a positive or negative value creates two separate branches with transitions to new partial interpretations until ultimately reaching a full interpretation or a conflict. These interpretations terminate the search on this branch and can therefore be considered leaves in a binary tree, while all other partial interpretations are inner nodes with the empty interpretation being the root node [Kroening and Strichman, 2008].

In analogy to the decision tree where each node can be associated with a level, each decision can be associated with a *decision level* that is its depth in the binary search tree, starting at level 0 for the root node. Given an inner node n of the binary tree, we can reconstruct the partial interpretation I describing this node from the path leading from n to the root node. By partially evaluating the formula φ under I we can simplify φ to a smaller formula ψ. This partial evaluation is sometimes also referred to as conditioning. For formula φ and decisions $x_0, \neg x_1, \ldots, x_n$ we write $\varphi | x_0, \neg x_1, \ldots, x_n$ [Kroening and Strichman, 2008]. In order to simplify the DPLL algorithm from an algorithmic point of view it assumes an input formula in CNF. Since there exists an equivalent CNF for every propositional formula, as shown before, we can simply preprocess an input formula and run DPLL on the resulting CNF. Rather than always writing the full CNF, we will represent it as a set of sets of clauses in the following, where $(x_1 \vee x_2) \wedge (x_3 \vee x_4 \vee x_5) \wedge \ldots$ is represented by $\{\{x_1, x_2\}, \{x_3, x_4, x_5\}, \ldots\}$. Consequently the empty CNF $\{\}$ corresponds to *true* and the CNF containing the empty clause corresponds to *false*. Exploiting the CNF structure, we can say that a formula has a conflict if there exists at least one clause where all literals are assigned, but the clause is not satisfied.

For some arbitrary decision level i, DPLL can find itself in one of three different situations: First, the conditioning of φ with the decisions made so far results in a conflict, in which case we can stop exploring the current node and backtrack, since all subsequent decisions will also not satisfy φ. Second, we are at decision level i where i is the number of variables in φ, which means we have a full interpretation I such that φ conditioned under all decisions is empty. Or third, we are at some inner node with the conditioned φ neither containing the empty clause, nor being empty, in which case DPLL has to make more decisions.

However, if the conditioning of φ does not result in the empty CNF or contain the empty clause, we might still be able to terminate early due to so-called *unit clauses*. A unit clause is a clause containing only a single literal, such that the decision on this literal is easy: For unit clause $\{x\}$ we have to map x to *true* and for unit clause $\{\neg x\}$ we map x to *false*. The *unit resolution technique*, also referred to as *unit propagation* allows to simplify CNF φ by collecting the set Γ of unit clauses in φ, assuming their implied decisions and conditioning φ under these decisions. Since the result may again contain unit clauses, DPLL repeats unit propagation until it reaches a fixpoint, i.e. there are no more unit clauses. For φ with strong connection between the individual clauses this can obviously result in a significant reduction of the number of decisions needed. In particular, the result may be the empty CNF, such that, using unit propagation, DPLL can terminate early, avoiding many, possibly bad decisions and subsequent backtracking. Note that using unit propagation implies that variables are not examined in the same order on all branches of the decision tree, since variables in unit clauses are preferred over other variables.

Example 2.9. Consider some example formula that has been converted to the following

$$\begin{aligned}\varphi = &(\neg a \vee b \vee \neg c) \wedge (\neg c \vee d) \wedge (\neg d \vee \neg e) \wedge (\neg a \vee \neg b) \text{ or}\\ &\{\{\neg a, b, \neg c\}, \{\neg c, d\}, \{\neg d, \neg e\}, \{\neg a, \neg b\}\}\end{aligned}$$

When we invoke the DPLL algorithm (cf. Algorithm 1) with φ, the initial unit propagation fails, since there are no unit clauses in φ. Since φ is also neither empty nor contains the empty clause, we choose a literal, in this case c. We recursively call DPLL with the conditioning of φ under c which yields

$$\{\{\neg a, b\}, \{d\}, \{\neg d, \neg e\}, \{\neg a, \neg b\}\}.$$

Algorithm 1 DPLL framework for SAT solving [Biere, Heule, et al., 2009]

```
function DPLL($\varphi$)
Input: PL Formula $\varphi$ in CNF
Output: SAT iff $\varphi$ is satisfiable
$(\Gamma, \varphi) \leftarrow$ UNIT-PROPAGATION($\varphi$)
if $\varphi = \{\}$ then
    return $\Gamma$
else if $\{\} \in \varphi$ then
    return UNSAT
else
    choose variable $l$ in $\varphi$
    if $\Theta$ = DPLL($\varphi|l$) $\neq$ UNSAT then
        return $\Theta \cup \Gamma \cup \{l\}$
    else if $\Theta$ = DPLL($\varphi|\neg l$) $\neq$ UNSAT then
        return $\Theta \cup \Gamma \cup \{\neg l\}$
    else
        return UNSAT
```

Unit propagation will collect the unit clause $\{d\}$. Conditioning φ under d yields the unit clause $\{\neg e\}$, such that unit propagation ultimately returns $\Gamma = \{d, \neg e\}$ and $\varphi = \{\{\neg a, b\}, \{\neg a, \neg b\}\}$. Next, we pick variable a and condition φ under a, yielding $\{\{b\}, \{\neg b\}\}$. Unit clause propagation will pick up the unit clause $\{b\}$, returning $\Gamma = \{b\}$ and $\varphi = \{\{\}\}$. Since φ contains the empty clause, DPLL returns UNSAT. Since DPLL recursively calls itself, we backtrack to the last call context, which was the call of DPLL $(\varphi|a)$ that returned UNSAT. Therefore the condition of the **if** is violated, so we try conditioning with $\neg a$. This conditioning yields $\varphi = \{\}$, such that the call of DPLL returns an empty Γ. Since we have found a satisfying assignment, we ascend back up the call stack, collecting all decisions and unit propagations along the way. In our case, the top-level call of DPLL returns the set $\{c, d, \neg e, \neg a\}$ which represents the (partial) interpretation $I = \{a \mapsto \mathit{false}, c \mapsto \mathit{true}, d \mapsto \mathit{true}, e \mapsto \mathit{false}\}$. We notice that I is missing an assignment for variable b which means that I satisfies φ regardless of the value of b. In our example this is the case because I assigns a to *false*, which satisfies all clauses where b occurs. A decision tree for the presented formula is given in Figure 2.1.

While Algorithm 1 sketches the main idea of the DPLL framework we left

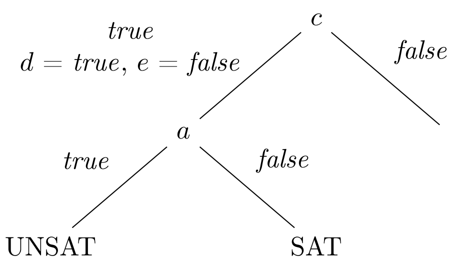


Figure 2.1: Decision tree from Example 2.9

some topics untouched. For example why did we choose variable c in Example 2.9. The strategy for this is called *decision heuristics* and determines a variable and its value for the decision. From the whole research field of SAT solving, there are various, more advanced techniques that improve the DPLL algorithm, but which are out of the scope of this thesis, some of which are covered in [Biere, Heule, et al., 2009].

2.2.2 Satisfiability Modulo Theories

While the presented techniques for solving the SAT problem allow us to reason about systems modeled in propositional logic, more complex systems require more complex logics, such as first-order theories. In the remainder of this section we will present the basic constructs for solving the satisfiability problem for first-order theories. We will however omit detailed descriptions of specific decision procedures, since each theory more or less has its own decision procedure. For a more detailed view into this matter the reader is referred to the literature [Bradley and Manna, 2007b; Kroening and Strichman, 2008].

To distinguish between the Boolean satisfiability problem (SAT) and the satisfiability problem for first-order logic, we adhere to the literature and call the latter problem *satisfiability modulo theories* (SMT) [Biere, Heule, et al., 2009; Bradley and Manna, 2007b; Kroening and Strichman, 2008]. Just like for propositional logic, SMT tries to solve the question whether for some formula φ there exists an assignment α_I, such that φ evaluates to *true* under α_I. By the definition of first-order formulas, the top-level connective, i.e. the root node of the syntax tree, must be a Boolean connective or a predicate. More precisely the predicate always maps from domain values to Boolean, but there exist no relevant operators that map from Boolean to theory. Therefore for some arbitrary first-order theory formula φ we know that either φ contains only Boolean connectives, variables and constants or that it contains theory operators, variables or constants and at least one predicate. If we look at the tree structure of φ in the latter case, we will find on each path through the tree a predicate that connects the underlying theory formula with the above Boolean formula. In other words, if we replace each predicate symbol with a Boolean variable, we obtain a Boolean formula, which we call the *Boolean skeleton* of φ. Since we already have a decision procedure for this Boolean skeleton, the high-level idea of lifting to SMT becomes very obvious: We replace each predicate symbol of φ by a propositional variable and start the SAT solving using the DPLL algorithm. Now we have to distinguish between two approaches called *fully lazy SMT solving* and *less lazy SMT solving*.

Algorithm 2 Fully lazy SMT solving [Kroening and Strichman, 2008]

```
function T-DPLL($\varphi$)
Input: Formula $\varphi$
Output: SAT iff $\varphi$ is satisfiable
$\psi \leftarrow$ Boolean skeleton of $\varphi$
while $\psi$ not UNSAT do
    $I \leftarrow$ DPLL($\psi$)
    if T-Solver($I$) = SAT then
        return SAT
    else
        $\psi \leftarrow \psi \wedge \neg I$
```

For *fully lazy SMT solving*, as shown in Algorithm 2 we first execute the SAT solving to achieve a full interpretation I. As we saw in Example 2.9, the resulting interpretation must not always be full, in which case we can just choose valuations for the missing variables, as long as we don't pick the same full interpretation twice during the SMT solving process. Given that full I we only now start the theory solver and check whether this assignment to the predicate symbols is actually feasible in the theory. If this succeeds, we have successfully found an assignment α_I for formula φ and if it fails, we have to exclude this specific I by adding the negation of its corresponding cube to φ and restarting the SAT solving. This process converges either in an assignment from the theory solver or in the SAT solver returning UNSAT because all satisfying assignments have been shown to be infeasible in the theory and thus have been excluded from φ.

Algorithm 3 Less lazy SMT solving [Kroening and Strichman, 2008]

```
function T-DPLL($\varphi, \Theta$)
Input: FO-formula $\varphi$ in CNF, initially empty set of decisions $\Theta$
Output: SAT iff $\varphi$ is satisfiable
$(\Gamma, \varphi) \leftarrow$ UNIT-PROPAGATION($\varphi$)
if T-SOLVER($\Gamma \cup \Theta$) = UNSAT then
    return UNSAT
else if $\varphi = \{\}$ then
    return $\Gamma$
else if $\{\} \in \varphi$ then
    return UNSAT
else
    choose variable $l$ in $\varphi$
    if DPLL($\varphi|l, \Theta \cup \Gamma \cup \{l\}$) $\neq$ UNSAT then
        return $\Theta \cup \Gamma \cup \{l\}$
    else if DPLL($\varphi|\neg l, \Theta \cup \Gamma \cup \{\neg l\}$) $\neq$ UNSAT then
        return $\Theta \cup \Gamma \cup \{\neg l\}$
    else
        return UNSAT
```

The alternative approach called *less lazy SMT solving* [Sebastiani, 2007], depicted in Algorithm 3, breaks up the strict top-down approach of fully lazy SMT solving and changes it to a more integrated interplay between the SAT solver and the theory solver. Whenever the SAT solver makes a decision, the partial I including this decision and the result of the subsequent unit propagation is given to the theory solver, which checks whether the partial assignment is feasible in the theory. If this is the case, the SAT solver can proceed its search. However, if the partial assignment is not feasible, we know that all subsequent decisions on that subtree are irrelevant, such that the SAT solver will backtrack. In order to avoid running into the same partial assignment again, the SAT solver learns the negation of the conflicting decision by adding it to φ.

Because SMT solving combines the DPLL algorithm with a theory solver, the general algorithm is often referred to as DPLL(T) or T-DPLL.

Example 2.10. Given a formula $\varphi = (x > 0 \vee x < 0) \wedge (2x = 4 \vee 2x = 0)$ in the first-order theory of Linear Integer Arithmetic, we want to check whether φ is satisfiable, using the less lazy SMT solving approach. The corresponding Boolean skeleton is $(a \vee b) \wedge (c \vee d)$

Since unit propagation fails and we cannot decide SAT/UNSAT yet, we start by picking a variable in the Boolean skeleton of φ, in this case we pick a and decide that its value should be *false*. The subsequent unit propagation will detect b to be a unit clause and thus assigns it to *true*. We now invoke the theory solver with the partial assignment $\{a \mapsto \mathit{false}, b \mapsto \mathit{true}\}$ which corresponds to the theory formulas $x \leq 0$ and $x < 0$ which is satisfiable in T. We therefore proceed by picking the next variable, say d to be *true*. Again, we call the theory solver and check whether the formulas $x \leq 0$, $x < 0$ and $2x = 0$ are feasible, which is not the case. As a result we notice that d is never satisfiable and we therefore learn the unit clause $\neg d$, i.e. $\varphi \leftarrow \varphi \wedge (\neg 2x = 0)$. As a result the DPLL algorithm will backtrack to the previous decision level and unit propagation will result in the assignment $\{a \mapsto \mathit{false}, b \mapsto \mathit{true}, c \mapsto \mathit{true}, d \mapsto \mathit{false}\}$, which is feasible in T for $x = 2$.

Having gotten an intuition about the mechanics of SAT and SMT solvers, for the remainder of this thesis we will treat them as black-boxes that take a propositional or first-order formula and decide whether it is satisfiable or unsatisfiable. In case of satisfiability it also gives an assignment to the variables which might be either a full assignment or a partial assignment. This assignment can help to understand why a formula is satisfiable. For unsatisfiability however, understanding the cause is a bit harder, since there exists no concrete example where unsatisfiability appears.

There does however exist a concept that can simplify the process of understanding why a formula φ is not satisfiable. The so-called *unsatisfiable core* or *unsatisfiable subset* of φ is a subset $U \subseteq X$ of all literals that appear in φ, such that for any valuation of the remaining literals, φ will still be unsatisfiable. This way, using unsat cores, we can pin down the cause for unsatisfiability to a possibly much smaller subformula. Especially for propositional formulas which can contain up to tens of thousands of variables, using unsat cores can have a strong impact on understanding the cause for unsatisfiability.

2.3 Model-based Verification

Over the last decades information and communication technology has become more and more important. At the end of the last millenium those systems mainly occured in the form of large and expensive personal computers or were only capable of simple tasks, such as calculators and early mobile phones. Since then advances in production and design have lead to increasingly powerful integrated circuits which at the same time have become smaller and smaller, down to several nanometers between individual signal lines. These improvements in performance and size have opened up a vast variety of new application scenarios for ubiquitous computing, such as smartphones, automated factories, smart homes or autonomously driving vehicles, just to name a few. While the decreasing size of computing systems has physically allowed adoption in more application scenarios, especially the growth in computing power has led to more and more complex applications and thus more complex software. But despite the rapid development of hardware, software development processes have not changed much in the same time: A requirements document defines the expected behaviour of the system. This document is handed over to the software developer that will write the software. Finally a number of test cases is generated that cover a non-representative number of program executions that resemble a specific class of executions paths. However, since the software is still designed by humans, in more complex systems errors become more likely. But in order to keep development costs low, software is often shipped to the customer without thoroughly eradicating all errors. While this approach may only cause minor problems such as a damaged customer satisfaction, e.g. in multimedia and entertainment systems, in other domains such behaviour might cause worse effects. A prominent example was a vulnerability called *Heartbleed* that was discovered in the OpenSSL cryptography library used for secure website communication. This security breach allowed attackers to access encrypted data, such as login credentials, transfered to up to 50% of all websites [Durumeric et al., 2014]. Such vulnerabilities can easily cause severe economical damage, as well as violations of privacy. However, there is still one category of software defects that is more critical than security vulnerabilities. When software systems operate in safety-critical environments, defects can easily cause injuries and death to those people that get in contact with the system. A tragic example of such a safety violation was the *Therac-25* radiation therapy machine, produced in 1982 [Baier and Katoen, 2008]. The machine was built to treat cancer patients with ionizing radiation. In the Therac-25 however, a bug in the concurrent program-

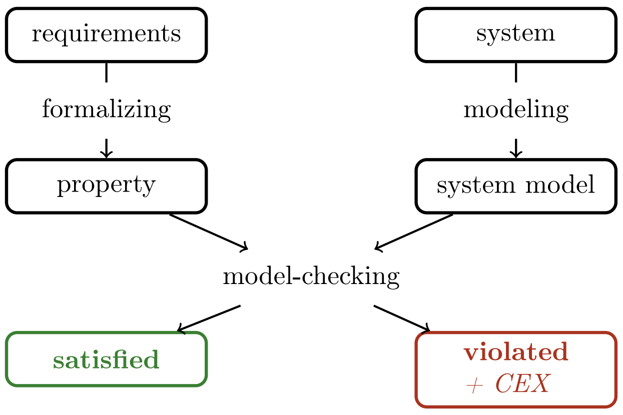


Figure 2.2: Model-checking process [Baier and Katoen, 2008]

ming caused the machine to output massive radiation overdoses that exceeded the normal doses by more than a hundred times. This radiation overdose killed three people and caused injuries for three more.

These experiences underline that the growing likelihood of software defects and the increasing number of applications for software systems pose a large problem for software developers. A promising approach to not only find bugs, but rather guarantee their absence is the *verification* of the system. In the following, we will describe a prominent verification method called *model-checking* that utilizes a model of the system in order to verify the absence of malfunction according to a given specification.

2.3.1 Model-Checking

For formal verification we require two inputs: First, a *system* in some arbitrary description format, such as a programming language like e.g. C++ or Java, a hardware description format such as e.g. Verilog or VHDL, or some abstract formalism like deterministic finite automata (DFA) or Kripke structures. Second, a description of the *requirements* of the system. Since the latter one results from the requirement engineers' description of *what* the system is expected to do, it is often given in some natural language that we cannot directly use for formal verification. We therefore need to formalize the human-readable requirements into a machine-readable *property* specification. While this formalization sometimes needs to be done by hand by a verification engineer, there has been some research in how to automate this formalization step, such as *structured english grammar* [Autili et al., 2015], which can be seen as a collection of fill-in-the-blank sentences that can be filled with actual names and values and allow parsing the requirements to transform them into a property specification. While the requirements specify *what* the system is expected to do, the system description itself addresses *how* the system behaves. While the system description is usually more precise than the requirements, we still need to transform it into a common *system model* that allows efficient verification of the system's behaviour against the property.

Given such a model of a system, a verification technique that aims to explore all possible states of the system in a systematic way is called *model-checking*. Using this systematic approach, model-checking can ensure that a model satisfies the property under all conditions. This contrasts techniques such as emulation, simulation and testing, which can only cover some pre-defined runs through the program and are therefore unable to reveal subtle errors, that can only be found using the systematic approach of model-checking. Just like the model, the specification must give precise and unambiguous statements about the properties of the system.

The result of the model-checking process can be either of the following: *satisfied* indicates that model-checking was able to prove that all possible states in the system model satisfy the property and therefore the system behaves exactly as demanded by the requirements. The second possible output of model-checking is *violated*, which indicates that there is some behaviour of the system model that violates the property specification. To help identify which behaviour exactly led to the violation, model-checking will give a counterexample (CEX) which helps to identify how the violation happened.

Apart from those two explicit outcomes, the model checker may also be incapable of producing a concrete result, because the model may be too large to fit into the memory of the computer, which we call *Out of Memory* or just *MemOut*. One reason for a model that is too large to fit into the memory is a combinatorial blow-up of the state-space, called the *state-space explosion problem*. This blow-up happens in particular with binary encodings.

Example 2.11. Consider a simple program with 3 16bit integer variables. The state space generated by those variables contains

$$\left(2^{16}\right)^3 = 281,474,976,710,656 \text{ states.}$$

In addition, each line in the program, also called *location*, can have individual variable valuations. Assume the program has 5 lines of code, the entire state space generated by this program can have

$$5 \cdot \left(2^{16}\right)^3 = 1,407,374,883,553,280 \text{ states.}$$

One way to prevent this exponential blowup will be considered in more detail in Section 2.3.3.

While there exist various types and definitions of models that are suitable for model-checking, we will start with *labeled transition systems* that also have a nice graphical representation.

Definition 2.11 (Labeled transition system [Baier and Katoen, 2008])**.** A *labeled transition system* (LTS) $\mathcal{M} = (S, \rightarrow, I, AP, \mathcal{L})$ is a tuple consisting of

- a set of states S
- a transition relation $\rightarrow \subseteq S \times S$
- a set of initial states $I \subseteq S$
- a set of atomic propositions AP
- a labeling function $\mathcal{L} : S \mapsto 2^{AP}$.

In the following we will only consider finite labeled transition systems, i.e. where S and AP are finite.

Given such an LTS $\mathcal{M}$, the behaviour of $\mathcal{M}$ will only be revealed over a certain, possibly infinite, number of steps. Note that even though $\mathcal{M}$ is finite, we can observe infinite progress on $\mathcal{M}$, due to cycles in $\mathcal{M}$. To reason about the behaviour of an LTS $\mathcal{M}$ over time, we define paths through $\mathcal{M}$.

Definition 2.12 (LTS path [Baier and Katoen, 2008])**.** A finite path in LTS $\mathcal{M}$ is a sequence of states $\pi = s_0, \ldots, s_n$, such that π is initial, i.e. $s_{i+1} \in Post(s_i)$, $0 \leq i < n$, $s_0 \in I$.

An infinite path in $\mathcal{M}$ is an initial, infinite sequence of states $s_0, s_1, \ldots$.

Furthermore, let $Paths(\mathcal{M})$ denote the set of all paths, finite and infinite, in $\mathcal{M}$ and $Paths^{fin}(\mathcal{M})$ the set of all finite paths in $\mathcal{M}$. Also, let $Paths(s)$ be the set of all maximal paths π starting in state s. Analogously $Paths^{fin}(s)$ is the set of all finite paths starting in s. Note that the definition of path only considers the states of $\mathcal{M}$ and not atomic propositions with which those states are labeled. However, states s in the transition system $\mathcal{M}$ are not observable, but only their atomic propositions. In order to incorporate those atomic propositions, we define so-called *traces* in $\mathcal{M}$.

Definition 2.13 (LTS trace [Baier and Katoen, 2008])**.** Given LTS $\mathcal{M}$, the *trace* of a finite path $\pi = s_0, \dots, s_n \in \mathit{Paths}^{fin}(\mathcal{M})$ is defined as $\mathit{trace}(\pi) = \mathcal{L}(s_0)\dots\mathcal{L}(s_n)$. Analogously the trace of an infinite path $\pi = s_0, s_1, \dots \in \mathit{Paths}(\mathcal{M})$ is defined as $\mathit{trace}(\pi) = \mathcal{L}(s_0)\mathcal{L}(s_1)\dots$.

Let the traces of a set of paths be defined as $\mathit{trace}(\Pi) = \{\mathit{trace}(\pi) \mid \pi \in \Pi\}$ and the set of traces starting in state s be $\mathit{Traces}(s) = \mathit{trace}(\mathit{Paths}(s))$. Then $\mathit{Traces}(\mathcal{M}) = \bigcup_{s \in I} \mathit{trace}\left(\mathit{Paths}\left(s\right)\right)$ denotes the set of all traces in $\mathcal{M}$.

In other words, the trace of a path π is the word over alphabet 2^{AP} that is induced by the atomic propositions along π.

Example 2.12. Consider a simplified snack machine that provides cookies or chocolate bars after inserting money and selecting the product. For simplicity we assume that both snacks have the same price and we can only throw a single type of coin in the machine, which matches the price of the products. An LTS $\mathcal{M}$ modeling this snack machine is depicted in Figure 2.3. We identify four states of the snack machine: (0) we need to insert a coin, (1) we need to select a product and (2/3) the snack machine dispenses cookies or chocolate bar, depending on our choice in (1). To observe these four states, we define the set of atomic propositions $AP = \{\mathit{pay}, \mathit{select}, \mathit{cookies}, \mathit{chocolate}\}$. The corresponding labeling function $\mathcal{L}$ for states $s \in \{0, 1, 2, 3\}$ is according to our description. Examples of path fragments for $\mathcal{M}$ are

$$\pi_1 = 0\ 1\ 2\ 0\ 1\ 2\ \dots$$
$$\pi_2 = 0\ 1\ 3\ 0\ 1\ 3\ \dots$$
$$\pi_3 = 0\ 1\ 2\ 0\ 1\ 3\ \dots$$

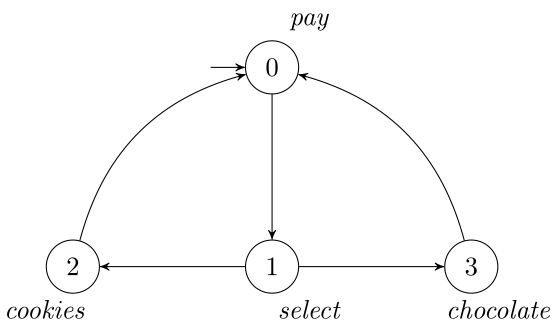


Figure 2.3: Example transition system for simplified snack machine

The corresponding traces for those path fragments are

$$\begin{aligned}
\mathit{traces}\,(\pi_1) &= \mathit{pay}\ \mathit{select}\ \mathit{cookies}\ \mathit{pay}\ \mathit{select}\ \mathit{cookies}\ \ldots\\
\mathit{traces}\,(\pi_2) &= \mathit{pay}\ \mathit{select}\ \mathit{chocolote}\ \mathit{pay}\ \mathit{select}\ \mathit{chocolote}\ \ldots\\
\mathit{traces}\,(\pi_3) &= \mathit{pay}\ \mathit{select}\ \mathit{cookies}\ \mathit{pay}\ \mathit{select}\ \mathit{chocolote}\ \ldots
\end{aligned}$$

Having defined a modelling formalism as well as a way to formally reason about the observable behaviour of the model, we still need to formalize a way to express properties, in order to check whether the traces of an LTS $\mathcal{M}$ satisfy the desired property.

2.3.2 Properties

A common formalism for properties are so-called *linear-time (LT) properties* that specify the admissible behaviour of the system as a set of infinite words over AP.

Definition 2.14 (Linear-time properties [Baier and Katoen, 2008]). A *linear-time property* P over the set of atomic propositions AP is a subset of $\left(2^{AP}\right)^{\omega}$.

If the property P is given as a set of infinite words that describe accepted system behaviour and the system behaviour is given in terms of possible execution traces $\mathit{Traces}(\mathcal{M})$ of the system model $\mathcal{M}$, then the property is *satisfied*

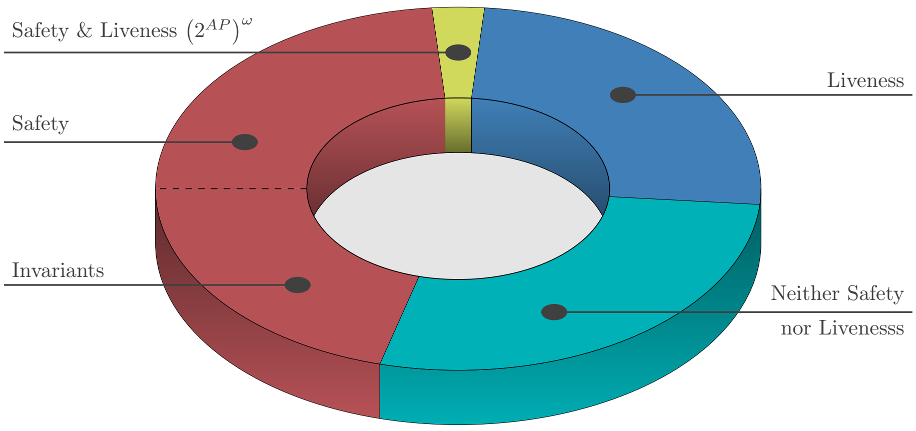

Safety & Liveness $(2^{AP})^{\omega}$
Liveness
Safety
Invariants
Neither Safety nor Livenesss

Figure 2.4: Dividing the space of all LT properties [Baier and Katoen, 2008]

by $\mathcal{M}$ iff $Traces(\mathcal{M}) \subseteq P$. Note that this definition also means that traces that run into a deadlock will never satisfy a property.

For verification of systems, we are interested in two categories of properties. The first category of properties is the one that states that *nothing bad* will happen, which is called *safety properties*. This category can be characterized by a *bad prefix*, i.e. a finite path that leads to a state where something *bad* has happened. The most liberal properties of this category is the property that allows every possible behaviour, i.e. $P = (2^{AP})^{\omega}$. This property is the only intersection to the otherwise disjunct category of *liveness properties* that specify that eventually *something good* will happen in the system, i.e. an AP occurs infinitely often on any infinite path. While *safety* and *liveness* properties are disjunct with the exception of $P = (2^{AP})^{\omega}$, they do not fully partition the whole space of all possible LT properties, but rather leave out some properties that are neither safety nor liveness properties.

For many industrial applications, progress is a desirable property, i.e. that the production does not come to a halt at some point in time. However, avoiding injuries and death is a much more crucial property that needs to be ensured under all circumstances. Especially for those application domains with a close interaction between heavy machinery and the workforce, such as for example car manufacturing or mining, safety is an important aspect in developing and maintaining a system. In those fields, malfunctions of machines can easily cause injuries and deaths. In some areas, such as transportation there even exist strict laws that require a certain, standardized level of safety. While there exist various mentioned techniques such as testing, simulation or emulation that might be lucky and find some errors, they can never fully ensure safety due to their lack of completeness, i.e. they can never cover all possible executions of the system. However, with model-checking we can ensure that each possible execution trace satisfies the given safety property and thus guarantee safety. Due to this strong need for safety verification in the industrial context, *we will focus the remainder of this thesis on safety properties.*

As mentioned before, a safety property P is violated by a finite path π that leads to a bad state s. Since every possible continuation of π violates P, we call π a *bad prefix.*

Definition 2.15 (Safety property [Baier and Katoen, 2008])**.** An LT property P is called *safety property* if for all $\sigma \in \left(2^{AP}\right)^{\omega} \setminus P$ there exists a finite prefix $\hat{\sigma}$ of σ such that

$$P \cap \{\sigma' \in \left(2^{AP}\right)^{\omega} \mid \hat{\sigma} \textit{ is a finite prefix of } \sigma'\} = \emptyset.$$

Each such $\hat{\sigma}$ is a *bad prefix* and the set of bad prefixes is denoted by $\mathit{BadPref}(P)$. We can check whether an LTS $\mathcal{M}$ satisfies such a safety property P by checking whether $\mathit{Traces}(\mathcal{M}) \cap \mathit{BadPref}(P) = \emptyset$.

A subclass of safety properties are *invariants*, which are properties that must hold in *every reachable state* of an LTS $\mathcal{M}$.

Definition 2.16 (Invariants [Baier and Katoen, 2008])**.** An LT property P over AP is called *invariant* if there exists a Boolean formula φ over AP such that

$$P = \{A_0 A_1 \dots \in \left(2^{AP}\right)^{\omega} \mid \forall j \geq 0. A_j \models \varphi\}.$$

The Boolean formula φ is called the *invariant condition* or *state condition* of P. The satisfaction for invariants can be checked in different ways:

$$\begin{aligned}
\mathcal{M} \models P &\iff \mathit{trace}(\pi) \in P, \forall \pi \in \mathit{Paths}(\mathcal{M}) \\
&\iff \mathcal{L}(s) \models \varphi, \forall s, p.\ s \in p, p \in \mathit{Paths}(\mathcal{M}) \\
&\iff \mathcal{L}(s) \models \varphi, \forall s \in \mathit{Reach}(\mathcal{M})
\end{aligned}$$

where $Reach(\mathcal{M})$ denotes the set of all reachable states in $\mathcal{M}$.

Hence we can also say that P is violated by $\mathcal{M}$, iff there exists a reachable state that does not satisfy φ, i.e.

$$\mathcal{M} \not\models P \iff \exists s \in Reach(\mathcal{M}).\mathcal{L}(s) \not\models \varphi$$

Since we can determine all states that do not satisfy φ upfront, checking whether P is an invariant can be reduced to a reachability problem of $\neg\varphi$-states.

Example 2.13. Let us reconsider the snack machine from Example 2.12. Assume we run this snack machine and want to make sure that we are not giving out free snacks. We can do so by checking the benign safety property that *the number of snacks dispensed is always less or equal to the number of coins inserted.* We can formalize this property as the set of all infinite words $A_0A_1A_2\ldots$ such that for all $n \geq 0$

$$P = |\{0 \leq i \leq n \mid pay \in A_i\}| \geq |\{0 \leq i \leq n \mid \{cookies, chocolate\} \cap A_i \neq \emptyset\}|$$

Examples of bad prefixes of P are

$$\emptyset\ \{pay\}\ \{cookies\}\ \{chocolate\} \quad \text{or}$$
$$\emptyset\ \{pay\}\ \{cookies\}\ \{pay\}\ \{cookies\}\ \{chocolate\}$$

From the labeled transition system $\mathcal{M}$ in Figure 2.3 on page 36 we see that $Traces(\mathcal{M}) \cap BadPref(P) = \emptyset$ because every state $s \in \{cookies, chocolate\}$ *must* be pre- and succeeded immediately by a *pay*-state. Therefore $\mathcal{M}$ satisfies the safety property $\mathcal{M}$ and we will never give out free snacks.

Since most of the aforementioned industrial use cases aim at avoiding potential safety risks at any time, the large majority of safety properties in those fields can be formulated as invariants that have to hold in every state of the system. For the remainder of this thesis we will hence sharpen our focus on invariants and *consider verification of invariant properties whenever we talk about verification and model-checking.* Furthermore we will use the reduction from invariant properties to reachability of violating states since this more general problem allows us to use a wider variety of approaches for verification of invariants.

2.3.3 Symbolic Model-Checking

As mentioned before the presented model-checking approach requires a gigantic amount of memory in the worst case due to the fact that every possible system state is considered as a distinct state in memory and has to be distinguishable from other states. For this explicit handling of states, the model-checking approach presented in the previous section is called *explicit (state) model-checking*. As seen in Example 2.11, explicit model-checking is almost infeasible for any meaningful real-world program due to its double-exponential growth in the size and number of variables. To avoid this exponential blowup, there exist a number of approaches that share the common feature of a symbolic representation of the state space in form of sets of states, thus giving it the name *symbolic model-checking*. In the following, we will briefly sketch two very popular symbolic model-checking approaches: *binary decision diagram* (BDD) and *SAT-based model-checking*.

As the name indicates, a binary decision diagram uses binary encodings of states and state sets. In particular, every state $s \in S$ is assigned an *encoding* $enc(s) = \{0,1\}^n$ in form of a bit-vector of length n. In order to characterize sets of states, we define a function that takes an arbitrary state encoding and decides its membership in the set. We call this the *characteristic function* $\chi : \{0,1\}^n \rightarrow \{0,1\}$. Since we reason about binary encodings rather than states, we need to modify the definition of the transition relation $\rightarrow \subseteq S \times S$ and initial set I to reflect this change. We therefore define a Boolean function $\Delta : \{0,1\}^{2n} \rightarrow \{0,1\}$ that takes two binary encodings and decides whether they are in transition relation $\rightarrow$ or not. In order to simplify BDDs we can remove decisions that are not essential for the BDD, e.g. those decisions where both results lead to isomorphic subtrees [Baier and Katoen, 2008]. Using these BDDs for initial states I, transition relation Δ and arbitrary other state sets, we can use BDD operations such as *AND* $\wedge$ to determine e.g. the set of one-step reachable states as $I \wedge \Delta$.

A more recent alternative to BDD-based model-checking is the somewhat similar *SAT-based model-checking*. Both approaches encode single states in terms of a set of Boolean variables $x_i \in X$ and operate on collections of these variables that encode sets of states. The main difference here is that one uses the switching function implicitly by representing it as a BDD with the outcome in the leaves, while switching functions in SAT-based model-checking are explicit propositional formulas. This way, the transition relation becomes a propositional formula, e.g. a DNF where each cube characterizes a state set. In order to check the satisfiability for BDDs we must determine it ”manually” on the BDD, while for SAT-based model-checking we can hand the formula over to a SAT-solver that determines satisfiability for us. While SAT-based model-checking in the purely SAT-based approach allows us to work with propositional formulas, we can also lift this to SMT which enables us to encode even infinite state sets, such as unbounded integers, as well as more expressive domains like floating points and arrays, in finite memory.

Example 2.14. Reconsider the snack machine from Example 2.12 with LTS $\mathcal{M}$ as depicted in Figure 2.3. We can encode states $0 - 3$ by their boolean encoding 00 to 11 and define characteristic variables x_1 and x_2, e.g. the interpretation $I = \{x_1 \mapsto \mathit{true}, x_2 \mapsto \mathit{false}\}$ represents the state $10 = 2$. To encode the symbolic transition relation Δ for $\mathcal{M}$ we use propositional formulas over $X = \{x_1, x_2\}$ for the source state and a copy of the target state encoding that we create by *priming* the variables in X. The transition relation for $\mathcal{M}$ would look as follows:

$$\begin{aligned}\Delta = &\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2' && \vee\\ &\neg x_1 \wedge x_2 \wedge x_1' \wedge \neg x_2' && \vee\\ &\neg x_1 \wedge x_2 \wedge x_1' \wedge x_2' && \vee\\ &x_1 \wedge \neg x_1' \wedge \neg x_2'\end{aligned}$$

Instead of $\neg x_1 \wedge x_2$ we also write $\bar{x}_1 \wedge x_2$ or simply $\bar{x}_1 x_2$. In Δ we can see two important aspects of symbolic encoding using propositional formulas. First, we model the nondeterminism in state 2 by simply giving both transitions. Due to the disjunction, both interpretations are allowed. Second, we modelled the transitions from states 2 and 3 back to 0 by the symbolic encoding x_1 for the state set $\{10, 11\}$.

While both approaches have some pros and cons, SAT-based model-checking seems to be the more promising approach at the time of writing this thesis: SAT- and SMT-solvers solve more general problems and therefore allow higher expressive powers than BDDs. Furthermore, even though there exist parallelization approaches for BDDs [Dijk and Pol, 2017], parallelization of SAT-solvers is straightforward, thus simplifying the use of modern multi-core hardware architectures.

2.4 Programs

For all software development projects reusability and modularity are highly important aspects of the development cycle. Their goal is to avoid code duplication, such that source code will be written a single time and called whenever its functionality is needed [Witt et al., 1993]. This in turn has many positive effects, like lower development costs and better maintainability, because changes in the functionality need to be implemented only once. Another important quality attribute is extensibility which anticipates future changes of the software's functionality and takes them into account in the software architecture.

As described in Section 2.3, the system under consideration is not suitable for verification without transforming it into a system model first. To improve maintainability and extensibility of the model-checking code we require succinctness of the system model, while still offering enough expressive power to match all possible input systems.

2.4.1 Guarded Command Language

For loop-free code, i.e. code without loops or jumps, a guarded command language (GCL) as first described by Dijkstra [Dijkstra, 1975, 1976] is a canonical candidate. It is a language skeleton that matches the previous listed requirements and contains only four basic derivation rules for high-level commands.

Definition 2.17 (GCL commands [Dijkstra, 1976]).

$$\mathsf{cmd} ::= x := e \mid \mathsf{assume}\ b \mid \mathsf{cmd}_1 ; \mathsf{cmd}_2 \mid \mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2,$$

where x is a left-hand side expression, e is a right-hand side expression and b is a boolean expression.

Commands can be one of the four follwing types: An assignment $x := e$ changes the value of x to the evaluation of the expression e. The assume statement $\mathsf{assume}\ b$, that is equipped with a boolean guard b, acts like a skip (or: noop) if b evaluates to *true*, but blocks further execution in case the guard evaluates to *false*. While this concept is hard to grasp when thinking about executing a program it is of great help in model-checking as an unsatisfiable guard indicates that this program path need not be further explored and the path cannot be a prefix of any counterexample path. To compose a program out of multiple commands, the sequence command $\mathsf{cmd}_1 ; \mathsf{cmd}_2$ can be equipped with any two commands including recursively nested sequences of commands. In order to allow branching as well as nondeterminism, GCL contains the choice command $\mathsf{cmd}_1 \,\Box\, \mathsf{cmd}_2$ that takes two arbitrary commands and nondeterministically executes one of them. However we can also model deterministic branching as in standard if-then-else constructs with the choice command by adding an assume command in the beginning of every branch. If the conjunction of both guards is unsatisfiable while their disjunction is a tautology, i.e. their predicates cover the whole state space but they are disjunct, then for every variable assignment there is exactly one branch that can be executed, while the other branch is blocked by the assume statement. A formal description is given by the following operational semantics. Note that we keep the GCL as succinct as possible in order to model only high-level program behaviour such as data-flow by allowing assignments and control-flow with assume, sequence and choice statements. In order to model actual program behaviour we can instantiate this abstract framework with any applicable first-order theory later. This gives us the freedom to dynamically configure our verification with first-order theories on demand. For example for programs with integers only, we use bit-vector theory for bit-precise verification or linear integer arithmetic if bit-precision is not necessary.

Definition 2.18 (GCL Semantics [Dijkstra, 1976]).
Let σ, σ' be two concrete (program) states. The semantics of a GCL command cmd are specified by the execution relation $\rightarrow$, where $\langle \mathsf{cmd}, \sigma \rangle \rightarrow \sigma'$ is a notation for the state σ *evolves to* σ' *under command* cmd. The execution relation $\rightarrow$ is inductively defined by:

(assign) $$\frac{}{\langle x := a, \sigma \rangle \rightarrow \sigma[x \mapsto a(\sigma)]}$$

(assume) $$\frac{b(\sigma) = \mathit{true}}{\langle \mathsf{assume}\ b, \sigma \rangle \rightarrow \sigma}$$

(seq) $$\frac{\langle \mathsf{cmd}_1, \sigma \rangle \rightarrow \sigma' \wedge \langle \mathsf{cmd}_2, \sigma' \rangle \rightarrow \sigma''}{\langle \mathsf{cmd}_1;\ \mathsf{cmd}_2, \sigma \rangle \rightarrow \sigma''}$$

(choice1) $$\frac{\langle \mathsf{cmd}_1, \sigma \rangle \rightarrow \sigma'}{\langle \mathsf{cmd}_1 \,\Box\, \mathsf{cmd}_2, \sigma \rangle \rightarrow \sigma'}$$

(choice2) $$\frac{\langle \mathsf{cmd}_2, \sigma \rangle \rightarrow \sigma'}{\langle \mathsf{cmd}_1 \,\Box\, \mathsf{cmd}_2, \sigma \rangle \rightarrow \sigma'}$$

2.4.2 Predicate Transformers

Given the operational semantics of the GCL, we can now also define the effect of GCL commands on arbitrary FO predicates [Dijkstra, 1976]. To start with, we will define the strongest postcondition (SP) which takes a GCL command cmd and a predicate φ representing a symbolic state set, called precondition, and gives us the resulting symbolic state set after applying cmd, called a postcondition.

Definition 2.19 (Strongest Postcondition [Bradley and Manna, 2007b])**.** Let cmd be a GCL command and φ an FO predicate, the strongest postcondition $sp(\mathsf{cmd}, \varphi)$ is defined as the smallest FO predicate satisfying:

$$\exists \sigma . \sigma' \models sp(\mathsf{cmd}, \varphi) \implies \sigma \models \varphi \qquad \text{for some } \langle \mathsf{cmd}, \sigma \rangle \to \sigma'$$

While the strongest postcondition takes a predicate φ and results in its successor predicate after execution of cmd, we might also be interested in the reverse direction: Which states can end in φ after execution of cmd?

Definition 2.20 (Weakest Precondition [Bradley and Manna, 2007b])**.** Let cmd be a GCL command and φ a FO predicate, the weakest precondition $wp(\mathsf{cmd}, \varphi)$ is defined as the largest predicate satisfying:

$$\forall \sigma . \sigma \models wp(\mathsf{cmd}, \varphi) \implies \sigma' \models \varphi \qquad \text{for all } \langle \mathsf{cmd}, \sigma \rangle \to \sigma'$$

At first sight, one might think that weakest precondition and strongest postcondition are dual and thus their relation would be symmetric. This however is not the case as

$$sp(\mathsf{cmd}, wp(\mathsf{cmd}, \varphi)) \implies \varphi \implies wp(\mathsf{cmd}, sp(\mathsf{cmd}, \varphi)). \tag{2.1}$$

[Bradley and Manna, 2007b]

The definition of weakest preconditions as in Definition 2.20 that can be found in the literature [Dijkstra, 1976] has a very strong constraint. It contains every state that reaches φ on *all possible executions* of command cmd. This also means that a state σ that can reach φ only via one branch will not be considered in the WP. While this might be helpful in other scenarios, it has a dramatic drawback for verification because states that might lead to a violation in certain scenarios will not be discovered. As this is not acceptable, we define a variant of the weakest precondition, called weakest *existential* precondition (WEP) that takes every state that might lead to φ under any possible execution into account.

Definition 2.21 (Weakest Existential Precondition)**.** Let cmd be a GCL command and φ a FO-formula, the weakest existential precondition of φ along cmd, denoted $wep(\mathsf{cmd}, \varphi)$ is defined as the largest predicate satisfying:

$$\forall \sigma . \sigma \models wep(\mathsf{cmd}, \varphi) \implies \sigma' \models \varphi \qquad \text{for some } \langle \mathsf{cmd}, \sigma \rangle \to \sigma'$$

Given the semantics of GCL, we can formulate partial mappings of the WEP function for some arbitrary FO postcondition φ. Considering the GCL command $\mathsf{assume}\ b$, we know that in order to terminate in φ, the precondition must have also satisfied b, i.e. the precondition must have been $\varphi \wedge b$. For an assignment $x := a$, any previous evaluation of x becomes meaningless as it is overwritten by the assignment. In fact, x might even be undefined previous to the assignment. Thus we can remove x from φ and replace its free occurrences by a to maintain the correctness of the remaining parts of φ.

Given those applications of WEP to elementary GCL commands, we can define the wep semantics of non-elementary commands. For the GCL sequence command $\mathsf{cmd}_1;\ \mathsf{cmd}_2$, we have to apply the WEP twice, but as WEP is computed backwards, we first have to apply WEP to cmd_2 before applying it to cmd_1. While the previous three results can be applied for WP analogously, the choice command $\mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2$ makes the difference between WP and WEP. For the WP every execution has to end in φ and thus the wp consists only of the intersection of the two individual WPs. For the WEP however, every state that ends in φ on any possible execution is part of the WEP, so we have to consider the disjunction of the individual WEPs for choice. Therefore the result of the weakest existential precondition can be derived as follows:

Definition 2.22 (Weakest Existential Precondition transformers)**.** The weakest existential precondition of a predicate φ is inductively defined as

$$\begin{aligned} wep(\mathsf{assume}\ b, \varphi) &= \varphi \wedge b \\ wep(x := a, \varphi) &= \varphi[x \mapsto a] \\ wep(\mathsf{cmd}_1;\ \mathsf{cmd}_2, \varphi) &= wep(\mathsf{cmd}_1, wep(\mathsf{cmd}_2, \varphi)) \\ wep(\mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2, \varphi) &= wep(\mathsf{cmd}_1, \varphi) \vee wep(\mathsf{cmd}_2, \varphi) \end{aligned}$$

In general, handling nondeterminism in the context of predicate transformers introduces additional problems, e.g. assigning a nondeterministic value to a program variable in the command of a WP application yields a formula with an all-quantifier over possible values. Such additional quantifiers can cause serious problems with the subsequent SAT solvers. However, in our context we don't have any nondeterminism on the abstraction level of the GCL, since we replace every expression which yields a nondeterministic value by two SMT terms: First, the nondeterministic value is "generated" by reading an array at an uninitialized index and second, incrementing the used index variable. This way, we allow nondeterministic/arbitrary values without explicit nondeterminism on the GCL level, thus not introducing problematic additional quantifiers.

While predicate transformers offer a simple and elegant way to map a pre-/post-condition to its successor/predecessor state according to the semantics of an instruction, it does include a large drawback. Under certain worst-case conditions the size of the resulting predicate can be exponential in the size of the original predicate. As the size of a query can influence the runtime of the SMT solver, this blow-up has theoretically the potential to foil all efforts spent in the actual verification algorithm. Thankfully this phenomenon occurs very rarely in practice.

Example 2.15. Consider the example program

$$x := x \cdot x;\; x := x \cdot x;\; x := x \cdot x$$

that calculates x^8. Given a postcondition $\varphi_0 = (x = 256)$, we get the following sequence of WEPs:

$$\begin{aligned}
\varphi_0 &= (x = 256)\\
\varphi_1 &= wep(x := x \cdot x, \varphi_0) = (x \cdot x = 256)\\
\varphi_2 &= wep(x := x \cdot x, \varphi_1) = (x \cdot x \cdot x \cdot x = 256)\\
\varphi_3 &= wep(x := x \cdot x, \varphi_2) = (x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x = 256)
\end{aligned}$$

We can see that in certain worst cases the size of the resulting wep can be exponential in the size of the input formula.

2.4.3 Control Flow

After establishing a simple, yet powerful guarded command language to express loop-free code, we are still unable to represent most basic programs. In particular, our modelling formalism lacks a way to represent full control flow, e.g. loops. To fix this, we leverage the expressiveness by wrapping our GCL with a so-called control-flow automaton (CFA) [Clarke, Grumberg, and Peled, 2001].

Definition 2.23 (Control-flow automaton)**.** A control-flow automaton (CFA) is a tuple $A = (L, G, \ell_0, \ell_E)$ consisting of a finite set of locations $L = \{\ell_0, \ldots, \ell_n\}$ and edges $G \subseteq L \times GCL \times L$ labeled with GCL commands, an initial location $\ell_0 \in L$ and an error location $\ell_E \in L$, for which $(\ell_E, \mathsf{cmd}, \ell) \notin G, \forall \mathsf{cmd}, \ell$.

Like a regular nondeterministic finite automaton (NFA), a CFA consists of a finite set of states (locations), as well as a number of transitions, which are represented as a relation: $(\ell, \mathsf{cmd}, \ell') \in G$ iff the location ℓ has an outgoing edge labeled with cmd that has ℓ' as target. Furthermore, a CFA contains an initial location ℓ_0 and an error location ℓ_E that correspond to initial and accepting states of a NFA. In contrast to the standard definition of an NFA, we ignore the infinite set of commands as input alphabet and we define a CFA to contain only a single error location rather than a set of accepting states.

Consider the alternative definition of a CFA $\bar{A} = (L, G, \ell_0, L_E)$ with a set of error locations. As an error location has by definition no successors, we can translate a CFA $\bar{A}$ with multiple error locations into a CFA $\bar{A}'$ with a singleton set of error locations by simply merging all error locations of $\bar{A}$ into a single error location:

Definition 2.24 (Simple path equivalence)**.** The CFAs $A = (L, G, \ell_0, L_E)$ and $B = (L, \overline{G}, \ell_0, \overline{L_E})$ are path equivalent iff

$$\begin{aligned}&\exists \ell_E \in L_E.(\ell, \mathsf{cmd}, \ell_E) \in G \iff \exists \overline{\ell_E} \in \overline{L_E}.(\ell, \mathsf{cmd}, \overline{\ell_E}) \in \overline{G} \qquad \text{and}\\ &\forall \ell, \ell' \in L \setminus \left(L_E \cup \overline{L_E}\right).(\ell, \mathsf{cmd}, \ell') \in G \iff (\ell, \mathsf{cmd}, \ell') \in \overline{G}\end{aligned}$$

We define the transition system semantics of a control-flow automaton in analogy to Definition 2.15 in [Baier and Katoen, 2008] and omit the details here. We define two CFAs to be *trace-equivalent* iff the underlying transition systems are trace-equivalent.

Corollary 2.2. *Given CFAs $\bar{A} = (L, G, \ell_0, L_E)$ and $\bar{A}' = (L, G', \ell_0, \{\ell'_E\})$ that are path equivalent, $\bar{A}$ and $\bar{A}'$ satisfy trace equivalence.*

To allow operations on the structure of the CFA, we define a few standard auxiliary functions for locations and edges.

Definition 2.25 (Edges/Locations)**.** Let $out : L \to 2^G$ be the function of outgoing edges of a location $\ell \in L$, such that

$$out(\ell) = \{g \mid g = (\ell, \mathsf{cmd}, \ell') \in G\}$$

Let $in : L \to 2^G$ be the function of incoming edges of a location $\ell \in L$, such that

$$in(\ell) = \{g \mid g = (\ell', \mathsf{cmd}, \ell) \in G\}$$

Let $succ : L \to 2^L$ be the function of successor locations of a location $\ell \in L$, such that

$$succ(\ell) = \{\ell' \mid (\ell, \mathsf{cmd}, \ell') \in G\}$$

Let $pred : L \to 2^L$ be the function of predecessor locations of a location $\ell \in L$, such that

$$pred(\ell) = \{\ell' \mid (\ell', \mathsf{cmd}, \ell) \in G\}$$

Given a CFA representation of a program, we might be interested in the set of variables that is used throughout the CFA, e.g. some operation might be bounded in the number of variables. To determine the set of variables in the whole CFA, lets first define the set of variables that are used in a single GCL command.

Definition 2.26 (GCL variables)**.** Given a GCL command cmd, we define the set Var of variables of cmd as

$$\begin{aligned}
Var(x := a) &:= \{x\} \cup Var(a) \\
Var(\mathsf{assume}\ b) &:= Var(b) \\
Var(\mathsf{cmd}_1;\ \mathsf{cmd}_2) &:= Var(\mathsf{cmd}_1) \cup Var(\mathsf{cmd}_2) \\
Var(\mathsf{cmd}_1 \,\Box\, \mathsf{cmd}_2) &:= Var(\mathsf{cmd}_1) \cup Var(\mathsf{cmd}_2)
\end{aligned}$$

and Var of expressions, for variables x and values v of the considered

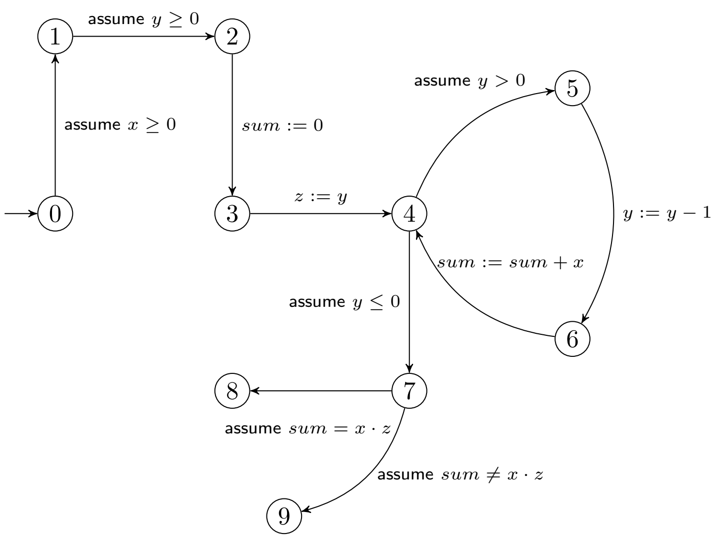


Figure 2.5: Example control flow automaton for multiplication

domain, as:

$$\begin{aligned} Var(v) = Var(true) = Var(false) &:= \emptyset \\ Var(x) &:= \{x\} \\ Var(a_1 \diamond a_2) &:= Var(a_1) \cup Var(a_2) \\ Var(a_1 \circ a_2) &:= Var(a_1) \cup Var(a_2) \\ Var(b_1 \wedge b_2) &:= Var(b_1) \cup Var(b_2) \end{aligned}$$

Using Definition 2.26, we can now define the set of variables used in the entire CFA.

Definition 2.27 (CFA Variables)**.** We define the set Var_A of CFA A as

$$Var_A = \{x \mid \exists\, (\ell, \mathsf{cmd}, \ell') \in G.\ x \in Var(\mathsf{cmd})\}$$

With the use a control-flow automata, the guarded command language and first-order theory (SMT) formulas, we have established three different layers of separation of concerns to abstract different aspects of the concrete input program. On the top-most level, the CFA resembles the control-flow structure of the input program, i.e. which abstract program paths can be chosen during execution. The guarded commands with which the CFA edges are labeled form the central tier and distinguishes whether the concrete instruction along the edges changes a variable assignment or may conditionally manipulate the control-flow based on the current variable assignment, in particular assume statements. This way the guarded command language connects the top-level control-flow automaton with the low-level SMT formula that gives the semantics of the expressions used in the guarded commands.

Example 2.16. Consider a simple program that takes two input variables and multiplies them using addition. We can model this using the following control flow automaton $A = (\{0, \ldots, 9\}, G, \{0\}, \{9\})$ with G as depicted in Figure 2.5. We label the edges $g \in G$ by GCL commands with terms in Linear Integer Arithmetic. For simplicity of the figure we use multiplication in out-going edges of location 7 which is not part of the signature of LIA. Our program first ensures that the input variables x and y are positive, non-zero integers. Afterwards it initializes the result variable sum to 0 and copies the value of y to z. Afterwards we enter a loop that decrements y and adds x to the sum as long as y is larger than 0. After leaving the loop, we verify the correctness of our computation by branching on whether sum is the product of x and y. Since we modified y in the run of our program, we use the copy z of y in order to verify our computation of the product. If the value in sum is not the product of x and z, we enter the error location 9.

Chapter 3

Inductive Hardware Verification

After its first publication in 2011, the inductive, incremental verification algorithm called IC3 [Bradley, 2011] had a major impact on the community due to its success for bit-level circuit verification. The key factor for its success has been the large performance advantage compared to other existing verification algorithms. Only months after its initial publication, [Eén, Mishchenko, et al., 2011] showed that with a small number of improvements, the new algorithm was able to beat state-of-the-art bit-level verification frameworks whose performance had been optimized for up-to a decade. This success generated a lot of interest and inspired many subsequent improvements to the original IC3 algorithm [Birgmeier et al., 2014; Cimatti and Griggio, 2012; Cimatti, Griggio, Mover, et al., 2014; Eén, Mishchenko, et al., 2011; Gurfinkel and Ivrii, 2015; Hassan et al., 2013; Hoder and Bjørner, 2012; Itzhaky et al., 2014; Lee and Sakallah, 2014; Suda, 2013; Vizel and Gurfinkel, 2014]. Nowadays variants of the IC3 algorithm are employed in many competitive bit-level verification frameworks [Griggio and Roveri, 2016].

Despite its recent success, the key idea of IC3 is based on the well-known concept of inductive invariants:

Definition 3.1 (Inductive invariant [Manna and Pnueli, 1995])**.** A safety property P for a Kripke structure $M = (S, I, R, L)$ is called inductive, if both:

$$s \in I \Rightarrow s \models P \qquad \text{(Initiation)}$$
$$s \models P \wedge R(s, s') \Rightarrow s' \models P \qquad \text{(Consecution)}$$

Given an arbitrary safety property P, if P is inductive, it is also an invariant. This can be easily proven: By initiation, all initial states are covered by P. By consecution, the set of states covered by P is closed under the transition relation R and thus all reachable states are covered by P, i.e. P is an invariant. However the reverse implication does not hold: Not every invariant is inductive, as an invariant may contain unreachable states which have non-invariant successor states and thus violate consecution. But if P is an invariant and it is not inductive, there exists a so-called *inductive strengthening* F of P, such that $P \wedge F$ is inductive. If M is finite, the trivial strengthening F is the enumeration of all reachable states, which however is not desirable due to its size. This way given any arbitrary safety property P, proving P to be an M-invariant can be reduced to finding an inductive strengthening F for P. Finding such a strengthening for a safety property on finite-state bit-level systems is the goal of *Finite-State Inductive Strengthening* (FSIS) [Bradley and Manna, 2007a].

Example 3.1. Consider the example Kripke structure in Figure 3.1. In this Kripke structure we encode states by their variable valuations and order them from most- to least-significant bit, i.e. $1100 \equiv x_1 x_2 \bar{x}_3 \bar{x}_4$. In addition, we keep the conjunctions implicit for the sake of readability.

Initial states are marked with blue boxes, while bad states, i.e. non-P states are marked with a red box.

Figure 3.1 shows that the safety property P that a state is '*not red*' is invariant, as all reachable states satisfy '*not red*'. However, while P is invariant, it is not inductive, as for example state 1100 is a '*not red*'-state that has a red successor. However, for a strengthening F such as $\bar{x}_2 \bar{x}_3 \vee \bar{x}_1 \bar{x}_4$, we can show that $P \wedge F$ is inductive and thus it follows that P is an M-invariant.

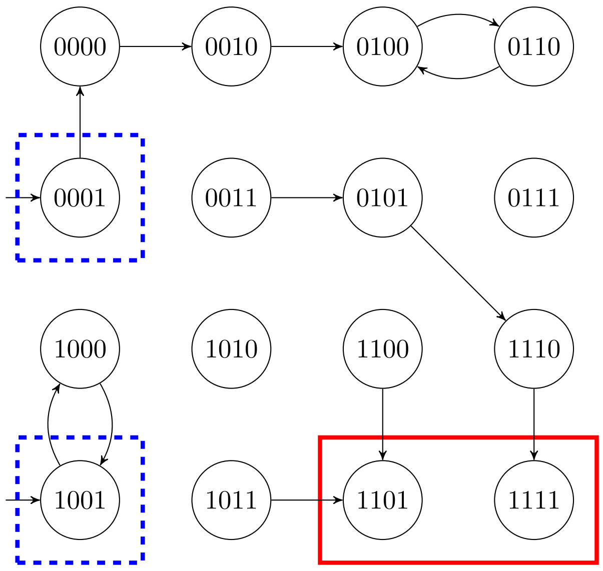


Figure 3.1: Example transition system

3.1 Finite-State Inductive Strengthening

While the concept of inductive invariants holds for general Kripke structures, the remainder of this chapter will consider a specific instantiation that is of particular interest when analyzing hardware circuits on a bit level. For this setting, Boolean transition systems are a common modeling formalism. A Boolean transition system $\mathcal{S} = (X, I, T)$ consists of a finite set X of variables, a propositional formula I over X describing the initial states and a propositional formula T over $X \cup X'$ characterizing the transition relation.

The algorithm *Finite-State Inductive Strengthening* (FSIS) [Bradley and Manna, 2007a] attempts to solve the model-checking problem for a safety property P on $\mathcal{S}$ by either showing that there exists a counterexample path leading from a state in I to a state in $\neg P$, or by proving P to be $\mathcal{S}$-invariant by constructing an inductive strengthening F for P. As FSIS changes P during the execution, we use a copy $\hat{P}$ which is initially equal to P. In order to prove or disprove P on $\mathcal{S}$, FSIS analyses the cause for a violated consecution, i.e. a pair of states (s, s') satisfying the query $\hat{P} \wedge F \wedge T \wedge \neg\hat{P}'$. Such a state s is called *Counterexample to Induction* (CTI). As s is an assignment of truth values to variables, it can be represented as a cube c over the assigned variables. Assume for now that s is unreachable, then excluding it will be a step in strengthening $\hat{P}$ to become inductive. In order to exclude s from $\hat{P}$, FSIS could simply conjoin $\neg c$ to the strengthening F, but this would in the worst case result in enumerating all CTIs, which is exponential in the size of X.

Therefore FSIS tries to deduce a so-called *minimal inductive subclause* $\bar{d}$ of $\neg c$, which blocks many more states apart from s, using the function MIC which will be considered in detail later. Given such a $\bar{d}$, FSIS will update the strengthening F to include the partial strengthening $\bar{d}$ by updating it: $F := F \wedge \bar{d}$. If on the other hand a predecessor of s exists such that $\neg c$ is not inductive, the search for a minimal inductive subclause of $\neg c$ relative to the property $\hat{P}$ fails and MIC returns $\top$.

In contrast to other model-checking algorithms, such as Bounded Model-Checking (BMC) [Biere, Cimatti, et al., 1999; Copty et al., 2001; Strichman, 2000] and Counterexample-Guided Abstraction Refinement (CEGAR) [Clarke, Grumberg, and Long, 1994], FSIS will not unroll the transition relation until it finds some state to be inductive or initial, but rather proceeds in a step-wise manner: By updating $\hat{P} := \hat{P} \wedge \bar{d}$ to exclude state s, proving the unreachability of s becomes a subgoal of proving the invariance of the original property P, as all predecessors of s will be a CTI in the next iteration. FSIS will proceed in this way until either $\hat{P} \wedge F$ is shown to be inductive and hence P is proven to be an invariant or no initial state is part of the refined property $\hat{P}$ any more. In this case there must exist a finite path $s_0, \ldots, s_n, n \geq 0$ such that $s_0 \models I$ and $s_n \models \neg P$.

Minimal Inductive Subclause

As outlined above, the function MIC is FSIS' main instrument to analyze CTIs and possibly deduces a subclause $\bar{d} \sqsubseteq \neg c$ for which the following must hold: $\bar{d}$ must be minimal w.r.t. subset inclusion on clauses $\sqsubseteq$ and the implications $I \Rightarrow \bar{d}$ and $\hat{P} \wedge \bar{d} \wedge T \Rightarrow \bar{d}'$, called $\bar{d}$ is inductive relative to $\hat{P}$. Note that $\bar{d}$ may not be of minimal size, i.e. there may exist other clauses f that are inductive relative to $\hat{P}$, such that $|\bar{d}| \geq |f|$ and neither is a subclause of the other. In order to do so, MIC uses auxiliary functions that operate on the subclause lattice $L_c : (2^c, \sqcap, \sqcup, \top, \bot, \sqsubseteq)$ that is induced by clause c and consists of

- 2^c : the subclauses of c
- $\sqcap$: the join operator, defined as disjunction
- $\sqcup$: the meet operator, defined as disjunction of all common literals
- $\top$: c
- $\bot$: *false*
- $\sqsubseteq$: the subclause relation with $c_1 \sqsubseteq c_2$ iff $c_1 \subseteq c_2$.

The first of these functions is DOWN which computes the unique largest (w.r.t. $\sqsubseteq$) subclause d of $\neg c$ which satisfies $\hat{P} \wedge d \wedge T \Rightarrow \neg c'$, if there exists one. While iteratively computing weakest preconditions would yield the greatest fixpoint of DOWN (see Algorithm 4), its complexity $\Omega(|c|^2)$ prohibits scaling of FSIS to large problems, due to the large amount of calls to MIC. Therefore, [Bradley and Manna, 2007a] presents a linear approach to compute underapproximations of the weakest precondition, leaving some predecessors to be subject of subsequent iterations: DOWN checks whether d satisfies consecution relative to $\hat{P}$. If this is the case and d also satisfies initiation, then it has reached a fixpoint and returns d. If however d does not satisfy consecution relative to $\hat{P}$, then there must exist some pair (s, s') that violates $\hat{P} \wedge d \wedge T \Rightarrow d'$. For the cube c corresponding to s, it constructs an overapproximation $\neg t$ of $\neg c$, updates $d := d \sqcap \neg t$ and recurses on the new d. In other words, DOWN iteratively removes literals from d that cause a violated consecution. If it succeeds to do so without violating initiation, then it returns the largest inductive subclause of $\neg c$. In the case that it discovers a clause that satisfies consecution, but violates initiation, it indicates a failure to find a largest inductive subclause. Algorithm 4 resembles a pseudo code sketch of the function DOWN.

Algorithm 4 Largest inductive subclause

```
function DOWN($\mathcal{S}$, $\hat{P}$, $L_c$, $d$)
Input: transition system $\mathcal{S} = (X, I, T)$, safety property $\hat{P}$, sub-clause
       lattice $L_c$, clause $d$
Output: largest $\hat{d} \sqsubseteq d$ such that $\hat{P} \wedge \hat{d} \wedge T \Rightarrow \neg c'$, or $\top$ if no such $\hat{d}$ exists
if $unsat\,(I \wedge \neg d)$? then
    return $\top$
else
    if $unsat\left(\hat{P} \wedge d \wedge T \wedge \neg d'\right)$? then
        return $d$
    else
        extract model $(s, s')$ from solver
        $\hat{s} \leftarrow \bigwedge s$                          ▷ Cube $\hat{s}$ representing model $s$
        $\neg\hat{t} \leftarrow$ overapproximate $\neg\hat{s}$ in $L_c$
        $\hat{d} \leftarrow d \sqcap \neg\hat{t}$
        DOWN($\mathcal{S}$, $\hat{P}$, $L_c$, $\hat{d}$)
```

Besides the function DOWN, FSIS computes a so-called *prime implicate* using the function IMPLICATE that, given a formula ρ and a clause d returns a minimal subclause $\hat{d} \sqsubseteq d$ such that $\rho \Rightarrow \hat{d}$ if such a $\hat{d}$ exists and $\top$ otherwise. FSIS suggests two possible implementations of IMPLICATE: One is by a linear search over the literals in clause d whether their elimination in d preserves the validity of the implication. A recursive pseudocode for IMPLICATE using linear search is given in Algorithm 5.

The alternative to this linear search is as follows: If d contains only a single literal, it returns this literal, otherwise the clause d is split into two disjunct subclauses d_l, d_r. Function IMPLICATE then checks whether $\rho \Rightarrow d_l$. If this is valid, then all literals in d_r can be dropped and IMPLICATE recurses on d_l. The same holds for d_r. If however both implications are not valid, then there exist literals in both subclauses that are necessary for the validity of the implication. In this case IMPLICATE first takes d_r as is and recurses on d_l with d_r as support, i.e. it splits d_l into d_{ll}, d_{lr} and checks $\rho \Rightarrow d_{ll} \vee sup$ and $\rho \Rightarrow d_{lr} \vee sup$ where $sup = d_r$. If IMPLICATE succeeds in finding a minimal subclause $\hat{d}_l$ that validates the implication relative to the support clause sup, it

Algorithm 5 Linear search for prime implicates

```
function IMPLICATE($\rho$, $d$, $res$)
Input: formula $\rho$, clause $d$, clause $res$ (empty in initial call)
Output: prime implicate $res$
if $d = \emptyset$ $(= \bot)$ then
    return $res$
else
    $hd \leftarrow$ HEAD($d$); $d \leftarrow$ TAIL($d$)
    if $unsat\ (\rho \wedge \neg d)$? then          ▷ Is $d$ prime implicate?
        IMPLICATE($\rho$, $d$, $res$)                ▷ drop $hd$
    else
        IMPLICATE($\rho$, $d$, $res \cup hd$)        ▷ keep $hd$
```

uses $\hat{d}_l$ as support clause and recurses on d_r to return the final $\hat{d}$. A recursive pseudocode for this binary search is given in Algorithm 6.

Function MIC computes a largest inductive subclause using DOWN if such exists and based on this computes the smallest inductive subclause using IMPLICATE as shown in Algorithm 7.

FSIS' way of proving a property differs a lot from other model-checking algorithms such as CEGAR or BMC. Most noticeable is the renunciation of an unrolling of the transition relation and rather only computing step-wise overapproximations to bad-state predecessors. This approach in particular saves FSIS from computing long paths represented by large formulas that are hard to check for modern SAT-solvers. In addition, the way FSIS generalizes from a single CTI to a set of bad or unreachable states allows for a faster convergence in practice. However, the downside of FSIS is that the search process is not guided in any way because it may find CTIs that are one step away from violating the original property or it may find CTIs to states that have been considered before and were excluded in the process. Thus the search for CTIs is neither a breadth-first search nor a depth-first search, but completely ad-hoc instead.

Algorithm 6 Binary search for prime implicates

```
function IMPLICATE($\rho$, $d$, $sup$)
Input: formula $\rho$, clause $d$, support clause $sup$ (empty in initial call)
Output: prime implicate of $d$
if $|d| = 1$ then
    return $d$
else
    $d_l, d_r \leftarrow$ SPLIT($d$);
    if $unsat\,(\rho \wedge \neg\,(d_l' \vee sup'))$? then
        IMPLICATE($\rho$, $d_l$, $sup$)
    else if $unsat\,(\rho \wedge \neg\,(d_r' \vee sup'))$? then
        IMPLICATE($\rho$, $d_r$, $sup$)
    else
        $l \leftarrow$ IMPLICATE($\rho, d_l, sup \cup d_r$)
        $r \leftarrow$ IMPLICATE($\rho, d_r, sup \cup l$)
        return $l \cup r$
```

Algorithm 7 Pseudocode for Minimal Inductive Subclause

```
function MIC($\mathcal{S}$, $\hat{P}$, $d$)
Input: transition system $\mathcal{S} = (X, I, T)$, property $\hat{P}$, clause $d$
Output: minimal $\hat{d} \sqsubseteq d$ such that $\hat{P} \wedge \hat{d} \wedge T \Rightarrow \neg c'$
$\bar{d} \leftarrow$ DOWN($\mathcal{S}$, $\hat{P}$, $d$)
if $\bar{d} = \top$ then
    return $\top$
else
    $\bar{d} \leftarrow$ IMPLICATE($\hat{P} \wedge d \wedge T, \bar{d}, \emptyset$)
    return $\bar{d}$
```

3.2 Incremental Inductive Strengthening

As seen in the previous section, the FSIS algorithm published in [Bradley and Manna, 2007a] in 2007, suffers from an unguided search and, in worst case, may do a full breadth first search of the state space before finding a counterexample. However, to efficiently find a counterexample in as few iterations as possible, a depth-first-search is generally preferable. In addition, FSIS might not even be able to find a suitable strengthening in reasonable time, as finding a single strengthening is generally hard: While a conjunction of assertions may be inductive, they may not be inductive on their own [Bradley, 2012]. In order to tackle those problems, the 2011 successor of FSIS aims at finding an *incremental strengthening* using previously computed invariants by constructing step-wise reachability information. The algorithm, called *Incremental Construction of Inductive Clauses for Indubitable Correctness* (IC3) [Bradley, 2011], breaks down the global strengthening F into a sequence of smaller strengthenings $F_0, \ldots, F_k, k \geq 1$, where each so-called *frame* F_i is an overapproximation of the states reachable in at most i steps from some initial state in I.

In order to include this reachability information, we modify the inductivity to a so-called *relative inductivity*: A clause $\neg c$ is inductive relative to the states reachable in $i-1$ steps, iff

$$I \Rightarrow \neg c, \; and \tag{3.1}$$

$$F_{i-1} \wedge \neg c \wedge T \Rightarrow \neg c' \tag{3.2}$$

are valid implications. However, since we want to use a SAT-solver to check whether $\neg c$ is inductive relative to F_{i-1}, we reformulate these implications to satisfiability queries: The implication (3.2) is valid, iff

$$F_{i-1} \wedge \neg c \wedge T \wedge c' \tag{3.3}$$

is unsatisfiable and (3.1) analogously. In order to break the global strengthening F up into step-wise strengthenings, we also need to maintain a number of invariants.

In particular, breaking up the F into $F_0, \ldots, F_k$ implies that every initial state must be in frame F_0 as every initial state is 0-step reachable. Furthermore, as frame F_i covers all states that are reachable in *at most* i steps, the states in any frame F_i are a superset of the previous frame F_{i-1}. While a strengthening F could potentially represent arbitrary states in $\neg P$, those states do not add any relevant information, so IC3 starts each new frame by strengthening it to

P. Lastly, each frame F_i represents i-step reachable states. This means that the successor of an F_i-state must be an F_{i+1}-state. Those invariants for the IC3 algorithm are formally given as:

$$
\begin{aligned}
I &\Rightarrow F_0 && && (3.4)\\
F_i &\Rightarrow F_{i+1}, && \forall 0 \leq i < k && (3.5)\\
F_i &\Rightarrow P, && \forall 0 \leq i \leq k && (3.6)\\
F_i \wedge T &\Rightarrow F'_{i+1}, && \forall 0 \leq i < k && (3.7)
\end{aligned}
$$

By constructing this sequence of frames, IC3 gains control over the search process, which was not possible for FSIS. By limiting the search depth using the "frontier" F_k, IC3 executes a breadth-first search to find the shortest counterexample path. If no such counterexample is found, k is incremented until either a counterexample of length k is found or the learned clauses form a sufficient strengthening to the property P.

Equations (3.6) and (3.7) imply that Counterexamples to Induction (CTI) can only occur in the last frame F_k, i.e.

$$F_k \wedge T \Rightarrow P' \tag{3.8}$$

is not a valid implication by construction. Like in FSIS, the implication of Equation (3.8) can be reformulated to a satisfiability query as follows: Equation (3.8) is valid iff

$$F_k \wedge T \wedge \neg P' \tag{3.9}$$

is unsatisfiable.

If Equation (3.9) is unsatisfiable, then there does not exist any state in F_k that is one step away from violating P. More specifically, there cannot exist any counterexample of length up to k. After finalizing iteration k, IC3 will check whether the computed frames $F_0, \ldots, F_k$ form a sufficient strengthening to prove the invariance of property P. To do so, IC3 searches for an inductive frame, i.e. a frame F_i, such that $F_i \wedge T \Rightarrow F_i$'. By Equation (3.7) this is the case exactly if $F_i \equiv F_{i+1}$ for some $0 \leq i < k$. If such F_i exists, IC3 has found a sufficient strengthening for P and can return *SAFE*. If IC3 fails to find such F_i, then the learned information is not a sufficient strengthening yet. But there does not exist a k-step counterexample, so there must either exist a longer counterexample or a more complex strengthening. Thus IC3 proceeds to the next iteration by incrementing k, similar to k-induction [Sheeran et al., 2000]. Following Equation (3.6), the new frame at level $k+1$ will be initiated by IC3 to P, i.e. $F_{k+1} := P$. While IC3 could start the new iteration immediately, it first executes a search whether some learned information can be propagated to a subsequent frame, i.e. it looks for some $\neg c \in F_i$ that also holds for F_{i+1}. This search is sometimes referred to as *propagation phase* and will be subject of Section 3.4.

Considering the case where the query Equation (3.9) is satisfiable. This means that there exists a CTI state, a state in F_k that has a transition to a $\neg P$ successor state. While there exist various approaches to compute predecessors, such as Weakest Preconditions [Dijkstra, 1976; Flanagan and Saxe, 2001; Leino, 2005], Quantifier Elimination [Biere, Heule, et al., 2009; Kroening and Strichman, 2008] or Interpolation [McMillan, 2003; Vizel, Weissenbacher, et al., 2015], IC3 refrains from specific predecessor computations and rather uses information that is provided by the SAT-solver at almost no cost: Given a satisfiable query, the solver will provide a satisfying model for all variables together with the result. By projecting this model on the non-primed state variables, IC3, like FSIS, can extract a set of state variables, that conjoined together form a cube over an underapproximation of the $\neg P$-predecessors in F_k. Depending on whether the solver returns a full model or an incomplete model, the resulting cube symbolically covers either a single state or a set of states. For more information on model generation see Section 2.2. A pseudocode of the top-level structure of IC3 is shown in Algorithm 8.

Given such a CTI cube c, IC3 will start a depth-first search for a counterexample path ending in a state described by c, bounded by length k. In order to keep track of the current search path, IC3 maintains a priority queue of tuples consisting of an index i and a cube c. As a tuple (i, c) represents the open proof, whether c is in F_i, this tuple is called a proof obligation[1]. In the following, we will use the inductivity relative to F_{i-1} for the obligation (i, c). Given a CTI cube c, IC3 initializes a new obligation queue Q with the obligation (k, c) to see whether c is inductive relative to F_{k-1}. The obligation queue is implemented as a priority queue, such that it returns obligations with the lowest index first, thus forming a depth-first search. Note that the obligation queue was not part of the initial presentation of IC3 in [Bradley, 2011], but was published in [Eén, Mishchenko, et al., 2011] shortly after and has a vital part of IC3 ever since. While the obligation queue is non-empty, IC3 will pop the obligation (i, c) with

[1]Note that the index used in a proof obligation is shifted in the literature [Bradley, 2011; Eén, Mishchenko, et al., 2011] with one approach using the notion whether c is inductive relative to F_i while another approach is to check inductivity relative to F_{i-1}.

Algorithm 8 IC3 main function [Bradley, 2011; Eén, Mishchenko, et al., 2011]

```
function MAIN($\mathcal{S}$, $P$)
Input: transition system $\mathcal{S} = (X, I, T)$, property $P$
Output: SAFE if $P$ is $\mathcal{S}$-invariant, UNSAFE otherwise
if $sat\,(I \wedge \neg P)$? or $sat\,(I \wedge T \wedge \neg P')$? then
    return UNSAFE
$F_0 \leftarrow I$
$F_1 \leftarrow P$
for $k := 1$ to ... do
    while $sat\,(F_k \wedge T \wedge \neg P')$? do
        $c \leftarrow$ SAT model
        $F_0, \ldots, F_k \leftarrow$ RECBLOCKCUBE($c$, $F_0, \ldots, F_k$)
    if $F_i = F_{i+1}$ for some $i < k$ then
        return SAFE
    $F_{k+1} \leftarrow P$
    $F_0, \ldots, F_{k+1} \leftarrow$ PROPAGATE($F_0, \ldots, F_{k+1}$)
```

the lowest index i to check whether the negation of c is inductive relative to F_{i-1}. In other words, if the states in c are excluded from F_i, one stays outside of c-states after one transition. This implies that there is no state which is in F_{i-1} and $\neg c$ that can reach a c-state under the given transition relation, i.e. c-states are unreachable from F_{i-1}-states and can be excluded from all frames up to F_i.

If this is not the case, i.e. c is not inductive relative to F_{i-1}, then there exists some state in F_{i-1} but not in c that can reach a c-state. Therefore the reachability of c at level i depends on whether these predecessors are reachable in at most $i-1$ steps. To check this, we have to create a new proof obligation $(i-1, c_{pre})$ where c_{pre} is deduced from the solver model of the inductivity query. In addition, we have to put the current obligation (i, c) back in to the queue, to reconsider it after investigating the reachability of c_{pre} in order to check for other predecessors not covered by c_{pre}.

If on the other hand, the cube $\neg c$ is inductive relative to F_{i-1} and does not intersect with the initial states I, we can exclude c from F_j, $\forall j \leq i-1$ by Equations (3.1), (3.2) and (3.5), but we can also exclude it from F_i, as we have shown that c is unreachable from F_{i-1}, such that by Equation (3.7) it is not reachable in F_i. To exclude c, we simply conjoin $\neg c$ to all frames F_j, $j \leq i : F_j \leftarrow F_j \wedge \neg c$.

However, there might be other unreachable states apart from those in c that can be excluded as well. More specifically, we are interested in a superset of the states in c, i.e. some cube g, such that $\forall s.s \models c \implies s \models g$. Such a larger cube g is called *Generalization* and will be subject of closer consideration in Section 3.3. In the remainder of this chapter we will see different approaches that deduce such a generalization from c either syntactically, by searching for a subcube, i.e. a cube g with a subset of the literals of c, or also semantically. While IC3 is sound and complete without generalization by enumerating models violating Equation (3.8) and Equation (3.2), the use of generalizations instead of concrete solver models obviously improves the performance of the algorithm, since it speeds up the convergence towards inductive frames.

Algorithm 9 IC3 blocking function [Bradley, 2011; Eén, Mishchenko, et al., 2011]

```
function RECBLOCKCUBE($c$, $F_0, \dots, F_k$)
Input: transition system $\mathcal{S} = (X, I, T)$, cube $c$, frames $F_0, \dots, F_k$
Output: frames $F_0, \dots, F_k$
Queue Q ← ($c$, k)
while Q is non-empty do
    s ← Q.pop                                  ▷ Obligation with smallest index
    if s.frame = 0 then
        return UNSAFE                          ▷ Counterexample found
    if $sat\,(F_{s.frame-1} \wedge \neg s.cube \wedge T \wedge s.cube')$? then
        p ← SAT model
        Q.add (p, s.frame−1)
        Q.add s
    else
        $g$ ← GENERALIZE(s.cube)
        $F_{s.frame} \leftarrow F_{s.frame} \wedge \neg g$
```

After blocking such a generalization g of some c at level i, IC3 can drop the current proof obligation (i, c) because it successfully proved that c is unreachable at level i, i.e. within i steps. It will therefore continue with the next proof obligation in the obligation queue Q, which, in the original version of the IC3 algorithm [Bradley, 2011], will mean continuing with the proof obligation $(i + 1, \bar{c})$ that was responsible for creating the obligation (i, c). In other words, IC3 will now backtrack and reconsider obligation $(i + 1, \bar{c})$ again, to check whether there exist other predecessors to $\bar{c}$ at level $i + 1$, i.e. other F_i-states apart from

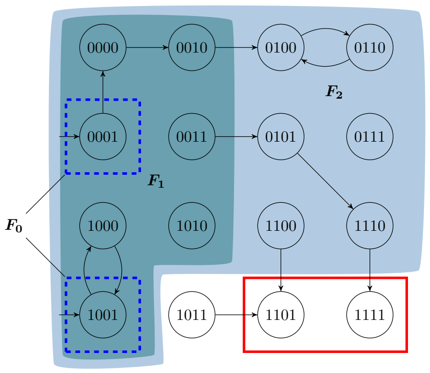
c that have not been blocked by g.

IC3 will continue this backtracking and descending until it meets one of the following conditions:

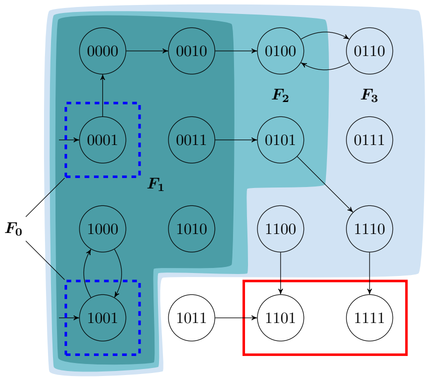
1. The IC3 algorithm has descended down to an obligation at level 0 and not visited any initial states on the way, i.e. in any previous obligation at level $i > 0$, thus if there exists a counterexample path, it exceeds the current bound k;

2. it descended down to some level j where the current cube c intersects the initial states, therefore finding a feasible counterexample path, or

3. it backtracked until the obligation queue Q is empty, i.e. until it has shown that the considered CTI is unreachable on any path, at which point it will start looking for another CTI at level k. If no more CTIs exist at level k, the IC3 algorithm will check whether it has created some frame that is inductive, i.e. $F_i = F_{i+1}$ and otherwise continue with the next major iteration, i.e. increment the bound k.



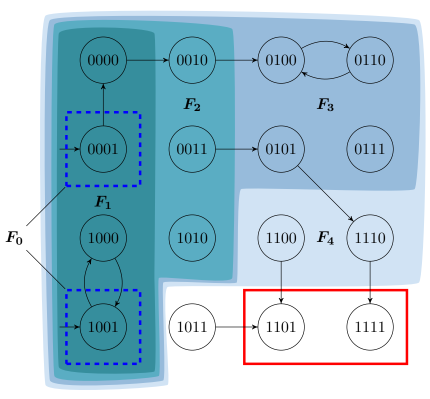
(a) IC3 frames at start of iteration $k = 2$



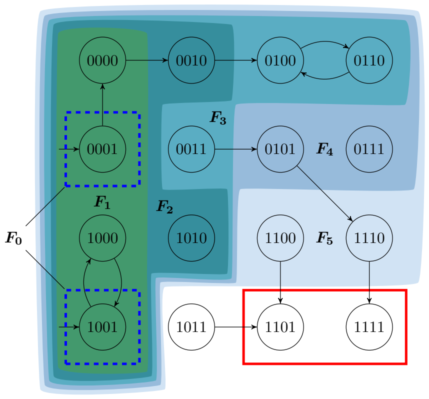
(b) IC3 frames at start of iteration $k = 3$

Figure 3.2: Frames of Example 3.2

Example 3.2. Consider the example transition system $\mathcal{S}$ of Figure 3.1 on



(a) IC3 frames at start of iteration $k = 4$



(b) IC3 frames at start of iteration $k = 5$

Figure 3.3: Frames of Example 3.2

page 53. First, let us construct the symbolic encoding of $\mathcal{S}$:

$$
\begin{aligned}
I &:= \bar{x}_2\bar{x}_3x_4\\
P &:= \neg\, x_1x_2x_4\\
T &:= \bar{x}_2\bar{x}_3x_4 \wedge \bar{x}'_2\bar{x}'_3\bar{x}'_4\ \vee\\
&\quad\ \bar{x}_1\bar{x}_3\bar{x}_4 \wedge \bar{x}'_1x'_3\bar{x}'_4\ \vee\\
&\quad\ \bar{x}_1x_3\bar{x}_4 \wedge \bar{x}'_1x'_2\bar{x}'_3\bar{x}'_4\ \vee\\
&\quad\ \bar{x}_1\bar{x}_2x_3x_4 \wedge \bar{x}'_1x'_2\bar{x}'_3x'_4\ \vee\\
&\quad\ \bar{x}_1x_2\bar{x}_3x_4 \wedge x'_1x'_2x'_3\bar{x}'_4\ \vee\\
&\quad\ x_1\bar{x}_2\bar{x}_3\bar{x}_4 \wedge x'_1\bar{x}'_2\bar{x}'_3x'_4\ \vee\\
&\quad\ x_1x_2\bar{x}_4 \wedge x'_1x'_2x'_4
\end{aligned}
$$

Following Algorithm 8, we will start IC3 by looking for no- and single-step violations of property P. As no state in $\mathcal{S}$ is an initial and bad state at the same time, *sat* $(I \wedge \neg P)$? is not satisfied. In addition, no initial state has a bad-state successor, thus *sat* $(I \wedge T \wedge \neg P')$? is also not satisfied.

After initializing frame F_0 to I we start the main loop with $k = 1$ and initialize F_1 to P. At the beginning of every main loop, we check, whether there exists a CTI state in F_k. As this is the case, the query $sat\,(P \wedge T \wedge \neg P')$? is satisfied and we assume the solver to identify state 1110 as the conflicting predecessor. Cube c will thus be assigned $x_1x_2x_3\bar{x}_4$. With c we will enter the blocking function, checking whether we can block c or whether there exists a path leading to state 1110. Inside the blocking function we will initialize the priority queue Q with the obligation $(c, 1)$. As this is the only obligation for now, we will immediately pop it, to check whether c is inductive relative to F_0. The queries $sat\,(I \wedge c)$? and $sat\,(F_0 \wedge \neg c \wedge T \wedge c')$? are both unsatisfied, as 1110 is not an initial state and the only predecessor to 1110 is 0101, which is not in I. Since we have determined that $\neg c$ is inductive relative to F_0, we are able to block c in F_1, i.e. we know that 1110 is not 1-step reachable from I. However, using generalization we can block many more states than just 1110. For the moment we treat generalization as a blackbox and assume that the result of the generalization is the cube $g = x_2$. A more detailed look at generalization will follow in the next section. Given the cube x_2, we block it by updating $F_1 := F_1 \wedge \neg x_2$. As cube c of obligation $(c, 1)$ was shown to be inductive relative to F_0, $(c, 1)$ can be dropped after blocking g. Since Q is now empty, we will return from the blocking function to IC3's main where we check whether there exists another CTI by querying $sat\,(F_1 \wedge T \wedge \neg P')$?

This query is satisfiable with the model $x_1\bar{x}_2x_3x_4$ for state 1011. We will again enter the blocking function, where we block state 1011, because it is obviously inductive due to missing predecessors. Thus it will return the modified $F_1 = \neg x_2 \wedge \neg x_1\bar{x}_2x_3x_4$ and we will check for additional CTIs.

This query is now unsatisfiable, because the third CTI 1100 was already blocked by the generalization x_2. Since there is no CTI left, we can check termination, i.e. whether there exists a frame F_i that is equal to its successor frame F_{i+1}. Since we have $F_0 = I$ and $F_1 = P \wedge x_2$, this condition is not met yet and we will continue initializing frame F_2 and propagating lemmas. Like generalization, we treat propagation as a black box, which returns frame $F_2 = P \wedge \neg x_1\bar{x}_2x_3x_4$. A more detailed look at why we were able to block state 1011 in frame F_2 will be taken later in this chapter.

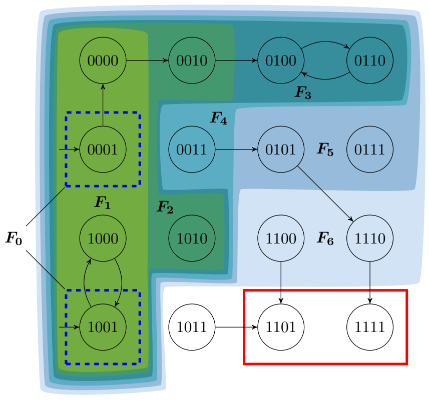
We then advance to the next major iteration $k = 2$ with frames F_0 - F_2 as depicted in Figure 3.2a. Assume the SAT-solver chooses state 1110 as CTI again, we will determine that 1110 is inductive relative to F_1. Assume our generalization will now be $g = x_1x_2$, we update $F_2 := F_2 \wedge \neg x_1x_2$. Since state 1011 was already blocked by pushing the clause $\neg x_1\bar{x}_2x_3x_4$

from frame F_1 forward, there is no CTI left, so we initialize frame F_3 and check whether we can propagate clauses, which again results in pushing $\neg x_1\bar{x}_2x_3x_4$ to frame F_3.

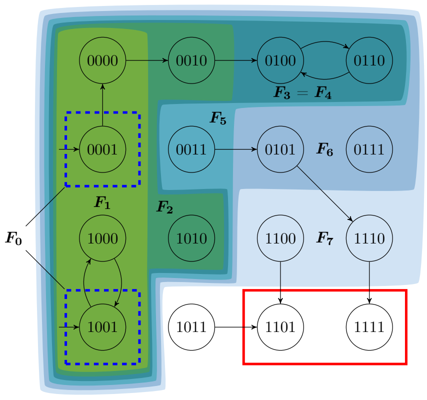
In iteration $k = 3$ we start with frames as depicted in Figure 3.2b. We again find the state 1110 as a CTI, such that we enter the blocking phase with obligation $(3, x_1x_2x_3\bar{x}_4)$. But this time, the SMT inductivity query $sat\,(P \wedge \neg x_1x_2 \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2x_3\bar{x}_4 \wedge T \wedge x_1'x_2'x_3'\bar{x}_4')$? is satisfied by state 0101. We will therefore add obligation $(2, \bar{x}_1x_2\bar{x}_3x_4)$, as well as the 'old' obligation $(3, x_1x_2x_3\bar{x}_4)$ to Q. In the next iteration of the **while**-loop, we will pop $(2, \bar{x}_1x_2\bar{x}_3x_4)$ from Q, since it has the smallest index. We will now check, whether state 0101 is inductive relative to frame F_1, i.e. check if $sat\,(P \wedge \neg x_2 \wedge \neg x_1x_2 \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg\bar{x}_1x_2\bar{x}_3x_4 \wedge T \wedge \bar{x}_1'x_2'\bar{x}_3'x_4')$?. The formula is satisfied with state 0011, which means that we add the obligations $(\bar{x}_1\bar{x}_2x_3x_4, 1)$ and $(\bar{x}_1x_2\bar{x}_3x_4, 2)$ to Q. Finally checking inductivity of $\bar{x}_1x_2\bar{x}_3x_4$ relative to F_0 fails. The corresponding generalization of cube $\bar{x}_1x_2\bar{x}_3x_4$ may return $g = x_3$, thus we update $F_1 = F_1 \wedge \neg x_3$. Next, the obligation with the smallest index is $(\bar{x}_1x_2\bar{x}_3x_4, 2)$ is popped, i.e. we reconsider state 0101 at level 2. But since we excluded state 0011 from frame F_1, 0101 now has no more F_1-predecessor and thus is inductive relative to F_1 which means we can generalize 0101, which may result in the cube x_2 that we block in frame F_2. Afterwards we reconsider obligation $(x_1x_2x_3\bar{x}_4, 3)$ and determine that 1110 is inductive relative to F_2, blocking the generalization x_1x_2. The resulting frames F_0 - F_4 after finishing iteration $k = 3$ and going over to $k = 4$ are shown in Figure 3.3a.

As we have seen all basic concepts of the search and blocking phase of the IC3 algorithm, we will skip iterations $k \in \{4, 5, 6\}$ and rather have a look at how the abstractions of reachable states in the frame sequence evolves in those iterations. From Figures 3.3b and 3.6b we can see that IC3 was not able to generalize state 0011 to cube x_3, as it has been in iteration $k = 3$, but rather has been generalized to the cube x_3x_4, which in turn triggered that state 0101 could not be generalized to x_2, but only to x_2x_4. This blocking repeats in iteration $k = 5$ (Figure 3.4a) and $k = 6$ (Figure 3.4b) with iteration $k = 6$ satisfying the termination criterion, that two consecutive frames are identical. Remember that the idea of the frame sequence was to break up the complex task of finding an inductive strengthening into a simpler, stepwise approach. Finding two identical, consecutive frames in iteration $k = 6$ as depicted in Figure 3.4a, means, that all states in F_3 are inductive, as their successors are still in the set of states of F_3. This implies that F_3 is an inductive strengthening to the property P. The final strengthening is depicted in Figure 3.5.

Looking at Figure 3.5 we can see that even though F_3 describes states that are reachable in at most three steps, IC3 does not compute exact reachability sets, but overapproximations to those reachability sets. We can see the effect of this in two aspects: First, state 0110 in the top right corner is a reachable state, but it is not reachable in 3, but at least in 4 steps. Nevertheless it is part of F_3 due to the overapproximation. Secondly, the state 1010 is completely isolated and is still in F_3 due to the overapproximation.



(a) IC3 frames at start of iteration $k = 6$



(b) IC3 frames at end of iteration $k = 6$

Figure 3.4: Frames of Example 3.2

As we have seen in Example 3.2, the presented IC3 algorithm will need n steps, for a backward reachable path of length n. To improve this, [Bradley, 2011] proposes to add a new obligation $(c, i + 1)$, whenever a proof obligation (c, i) is shown to be inductive relative to F_{i-1} and c is being blocked at level i. This allows IC3 to explore deep, backward reachable states with fewer iterations.

One weakness of the implementation of the original IC3 was in the way it handles frames. [Bradley, 2011] proposes to implement a frame F_i as a set of clauses *clauses*(F_i), such that the formula for F_i is the conjunction $\bigwedge clauses(F_i)$. However, to satisfy invariant (3.5) IC3 will add a cube that has to be blocked at frame F_i to *all* frames F_j for $j \leq i$. This obviously creates a huge amount of redundant data. To avoid this redundancy, PDR [Eén,

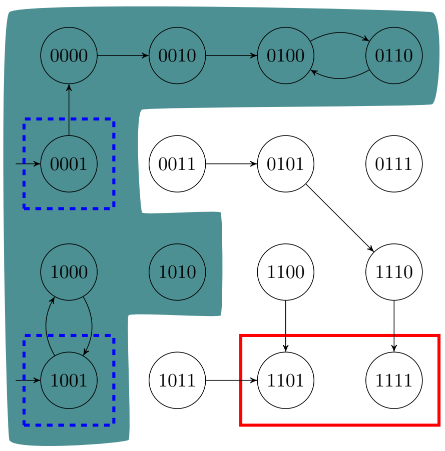


Figure 3.5: IC3 inductive strengthening for transition system of Figure 3.1

Mishchenko, et al., 2011] implements a so-called *delta trace* that just stores the clauses that are valid up to F_i, i.e. $clauses(F_i) \setminus clauses(F_{i+1})$. This way a clause is added only once, while the formula for frame F_i is the conjunction $\bigwedge F_j \ , \ i \leq j \leq k$.

$$\begin{aligned}
F_0 &:= I \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2 \wedge \neg x_3\\
F_1 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2 \wedge \neg x_3\\
F_2 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2\\
F_3 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2\\
F_4 &:= P \wedge \neg x_1\bar{x}_2x_3x_4
\end{aligned}$$

(a) Frames after iteration $k = 3$

$$\begin{aligned}
F_0 &:= I \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_2 \wedge \neg x_3x_4 \wedge \neg x_3\\
F_1 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_2 \wedge \neg x_3x_4 \wedge \neg x_3\\
F_2 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_2 \wedge \neg x_3x_4\\
F_3 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4\\
F_4 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2\\
F_5 &:= P \wedge \neg x_1\bar{x}_2x_3x_4
\end{aligned}$$

(b) Frames after iteration $k = 4$

$$\begin{aligned}
F_0 &:= I \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4 \wedge \neg x_2 \wedge \neg x_3\\
F_1 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4 \wedge \neg x_2 \wedge \neg x_3\\
F_2 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4 \wedge \neg x_2\\
F_3 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4\\
F_4 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4\\
F_5 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2\\
F_6 &:= P \wedge \neg x_1\bar{x}_2x_3x_4
\end{aligned}$$

(c) Frames after iteration $k = 5$

$$\begin{aligned}
F_0 &:= I \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4 \wedge \neg x_2 \wedge \neg x_3\\
F_1 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4 \wedge \neg x_2 \wedge \neg x_3\\
F_2 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4 \wedge \neg x_2\\
F_3 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4\\
F_4 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4 \wedge \neg x_3x_4\\
F_5 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2 \wedge \neg x_2x_4\\
F_6 &:= P \wedge \neg x_1\bar{x}_2x_3x_4 \wedge \neg x_1x_2\\
F_7 &:= P \wedge \neg x_1\bar{x}_2x_3x_4
\end{aligned}$$

(d) Frames after iteration $k = 6$

Figure 3.6: Iterations 3 to 6 of IC3 example

3.3 Generalization

As seen in Section 3.2, generalizing a cube c to some larger cube g can allow the IC3 algorithm to exclude more states in one iteration, resulting in a faster convergence of the algorithm. For a cube g to be a suitable generalization of cube c, it must satisfy two important properties: (1) blocking g must block at least all states that would be blocked by c and (2) as c is inductive relative to F_{i-1}, g must also be inductive relative to F_{i-1}. In other words, g can only add new states to c but only unreachable states can be part of the generalization g. These properties maintain the correctness of the overall algorithm, by ensuring that a generalized cube g neither leaves unreachable states unblocked and to be reconsidered again, nor considers any reachable states due to incorrect blocking, thus violating completeness of the algorithm.

As long as it satisfies these two requirements, the generalization procedure can be chosen without any further restrictions. This naturally means that there exists a broad set of possible methods to find the generalization procedure that suits the desired setting best. In this section, we will sketch a handful of approaches to apply generalization for a bit-level IC3 algorithm.

Algorithm 10 Pseudocode for Syntactic Generalization

```
function DROP($\mathcal{S}$, $F_{i-1}$, $c$)
Input: transition system $\mathcal{S} = (X, I, T)$, frame $F_{i-1}$, cube $c$
Output: minimal cube $\bar{c}$ that is inductive relative to $F_{i-1}$
$\bar{c} \leftarrow c$
for all $lit \in literals\,(c)$ do
    $\bar{c} \leftarrow (\bar{c} \setminus lit)$
    if $sat\,(I \wedge \bar{c})$? or $sat\,(F_{i-1} \wedge \neg\bar{c} \wedge T \wedge \bar{c}')$? then
        $\bar{c} \leftarrow (\bar{c} \cup lit)$
return $\bar{c}$
```

Syntactic Generalization

The most prominent generalization procedure is the one presented in [Bradley, 2011], a modified version of the search for minimal inductive subclauses of FSIS. The procedure aims to find a subcube $g \subseteq c$ such that $\neg g$ is still inductive relative to F_{i-1}. As g is a subcube of c, i.e. it contains a subset of the literals, blocking g will block at least as many states as blocking c would do. While checking all exponentially many subcubes of c is obviously highly inefficient, we can find a smallest subcube by iterating over the literals of c and trying to exclude them one by one, resulting in a linear algorithm to find a smallest subcube g. A pseudo-code for this *linear search* can be found in Algorithm 10.

The presented syntactical generalization tries to find a suitable generalization of the cube by changing the cube c based on its syntactical structure by probing which literal is necessary in order to stay inductive relative to F_{i-1}. This procedure has the advantage that it can compute a generalization in linear time that is considerably smaller than the original cube c.

Example 3.3. Recall Example 3.2 on page 64. In iteration $k = 2$, we determined that the negation of state 1110 of $\mathcal{S}$ is inductive relative to $F_1 = P \wedge \neg x_2$, which means that $sat\,(P \wedge \neg x_2 \wedge \neg x_1 x_2 x_3 \bar{x}_4 \wedge T \wedge x_1' x_2' x_3' \bar{x}_4')$? is unsatisfiable. In order to generalize the clause $\neg x_1 x_2 x_3 \bar{x}_4$, we pick literal x_4, drop it and see whether the clause $\neg x_1 x_2 x_3$ is still inductive relative to F_1. Since $x_1 x_2 x_3$ is the cube that covers states 1110 and 1111, none of these has a $\neg x_2$-state predecessor, which means that it is inductive relative to F_1 and we do not have to readopt x_4. Next, we try to drop x_3, excluding states 1100, 1101, 1110 and 1111 with clause $\neg x_1 x_2$. Since these states have only two predecessors, namely 1011 and 0101, but both are blocked in F_1, $x_1 x_2$ is inductive relative to F_1 as well. Next, we try to block x_2, which fails, since $\neg x_1$ excludes the initial state 1001 and thus is not inductive relative to any frame, i.e. we have to keep x_2. Last, we try to drop x_1 and check whether $\neg x_2$ is inductive relative to F_1. This fails too, since F_1-state 0011 has x_2-successor 0101. We thus have to keep x_2 as well and therefore return $\neg x_1 x_2$ as the final generalization.

However, this procedure can have several pitfalls. In particular the linear search may traverse the subclause lattice in a worst possible way: If there exists any generalization that is a strict subclause of c the linear search may return a cube that contains only one literal less than c. In addition, the search may interfere with efficient implementations of cube data structures: In order to find the smallest possible subcube, a randomized order of literals is preferable. This contradicts efficient data structures that use canonical representations of e.g. set data structures in order to offer efficient comparison operators. Especially for cases where a majority of the literals can be dropped, the syntactic generalization can be improved even further by replacing the linear by a *binary search*. In this approach, the cube is partitioned into two subcubes. By definition at most one of these can be inductive relative to F_{i-1}. If a cube is relative induc-

Algorithm 11 Pseudocode for syntactic generalization with binary search

```
function BINARY($\mathcal{S}$, $F_{i-1}$, $c$)
BINSEARCH($\mathcal{S}$, $F_{i-1}$, $c$, $\emptyset$)
Input: transition system $\mathcal{S} = (X, I, T)$, frame $F_{i-1}$, cube $c$
Output: minimal cube $\bar{c}$ that is inductive relative to $F_{i-1}$
function BINSEARCH($\mathcal{S}$, $F_{i-1}$, $c$, $sup$)
if $\|c\|$ then
    return $c$
$left, right \leftarrow$ SPLIT($c$)
if $unsat\ (F_{i-1} \wedge \neg left \wedge T \wedge left')$? then
    BINSEARCH($\mathcal{S}$, $F_{i-1}$, $left$, $sup$)
else if $unsat\ (F_{i-1} \wedge \neg right \wedge T \wedge right')$? then
    BINSEARCH($\mathcal{S}$, $F_{i-1}$, $right$, $sup$)
else
    $left \leftarrow$ BINSEARCH($\mathcal{S}$, $F_{i-1}$, $left$, $(sup \cup right)$)
    $right \leftarrow$ BINSEARCH($\mathcal{S}$, $F_{i-1}$, $right$, $(sup \cup left)$)
    return $left \cup right$
```

tive we can proceed by recursing on this cube and splitting it further. However in certain cases none of the two subcubes may be inductive relative to F_{i-1} because both subcubes contain literals that are necessary to maintain inductivity. In this case we can split the first subcube, while keeping the second as a support. Once we have computed the smallest inductive subcube of the first split subcube, we can take this result as a support for finding the smallest inductive subcube of the second half. A pseudocode for this binary search process can be found in Algorithm 11 accompanied by an example in Example 3.4.

Example 3.4. Recall Example 3.2 (page 64), in $k = 1$, we encountered CTI 1110 which is inductive relative to F_0 and generalized it to x_2. We can generalize the clause $\neg x_1 x_2 x_3 \bar{x}_4$ representing state 1110 using binary generalization by splitting up $\neg x_1 x_2 x_3 \bar{x}_4$ into left subcube $\neg x_1 x_2$ and right subcube $\neg x_3 \bar{x}_4$. We start by checking whether $\neg x_1 x_2$ is inductive relative to $F_0 = I$. This query is unsatisfiable, since no initial state has a $x_1 x_2$-state successor, thus $\neg x_1 x_2$ is inductive relative to F_0. As a consequence, we drop the right subcube and continue splitting up $\neg x_1 x_2$. $\neg x_1$ is not inductive relative to F_0 since it intersects with the initial states, but $\neg x_2$ is inductive relative to F_0. We thus generalize to $\neg x_2$ after three queries.

As we can see in Example 3.4, the binary search for the generalization can be up to exponentially better, compared to the linear search. On the other hand the worst-case complexity of the binary search can be exponentially worse: For a cube of size n where no literal can be dropped at all, the binary search will do 2^n SAT-queries, recursively splitting up smaller and smaller portions of the cube.

Another approach to syntactic generalization of cubes was presented in [Eén, Mishchenko, et al., 2011] and uses *ternary simulation*, which extends the domain from Boolean values to a domain with three values: *true*, *false* and X, where X is referred to as *don't-care*. As the name suggests, ternary simulation uses a term, which is the minterm extracted from the satisfiable model of the SAT-solver, and simulates it over one step in the transition relation T. In particular, it successively replaces each literal's evaluation in the minterm by X, simulates this term over one step of the transition relation T and checks whether X appears in the result. If X appears, the replaced value will be reverted. But if not, the corresponding position does not influence the result and thus can be removed.

Semantic Generalization

While the syntactic generalization presented in the last section is clearly one of the most prominent generalization methods, it does have some downsides, some of them illustrated above. Another drawback that has not been discussed is the fact that the presented method only works on the syntactic structure. It merely probes whether a smaller cube would still be inductive relative to F_{i-1} and constructs the generalizations based on the SAT-solver as an oracle for relativity. In other words, it is more or less blind in the sense that it does not construct a generalization based on the actual structure of the cube c and its interaction with the transition relation T. We therefore emphasize another category of generalization methods that we call *semantic generalization* as they are based on other information apart from only the structure of the cube.

A prominent example for a semantic generalization is *Craig interpolation*, based on Craig's Interpolation theorem from 1957 [Clarke, Grumberg, and Peled, 2001].

Theorem 3.1 (Craig's Interpolation Theorem [Clarke, Grumberg, and Peled, 2001])**.** *If* $\models \Phi \Rightarrow \Psi$*, then there exists a* φ *such that* $\models \Phi \Rightarrow \varphi$ *and* $\models \varphi \Rightarrow \Psi$ *with* $var(\varphi) \subseteq var(\Phi) \cap var(\Psi)$.

Theorem 3.1 implies that for a pair of formulas (A, B) with *unsat* $(A \wedge B)$?, an interpolant φ is a formula, with $A \Rightarrow \varphi$, *unsat* $(\varphi \wedge B)$? and φ only refers to common variables of A and B [McMillan, 2003].

Theorem 3.2 (Generalization via Craig Interpolation)**.** *Given a frame F_{i-1}, a cube c and a transition relation T, with unsat $(F_{i-1} \wedge \neg c \wedge T \wedge c')$?, an interpolant φ' of pair $(A = c', B = F_{i-1} \wedge \neg c \wedge T)$ provides a valid generalization of c' iff it does not intersect with the initial states, i.e. $I \Rightarrow \neg\varphi$ is preserved.*

Note that by construction φ' will reason about primed state variables, so in order to block the generalization, one has to use the unprimed φ.

Proof. Since $\neg c$ is inductive relative to F_{i-1} by construction, it holds that *unsat* $(F_{i-1} \wedge \neg c \wedge T \wedge c')$?. For the pair $(A = c', B = F_{i-1} \wedge \neg c \wedge T)$ it follows that

1. $c' \Rightarrow \varphi'$, thus blocking an unprimed φ will exclude more states than blocking c, and

2. *unsat* $(\varphi' \wedge F_{i-1} \wedge \neg c \wedge T)$? $\equiv$ *unsat* $(F_{i-1} \wedge \neg c \wedge T \wedge \varphi')$?, thus preserving inductivity relative to F_{i-1} if $I \Rightarrow \neg\varphi$.

□

As shown in [Huang, 1995; Krajícek, 1997; McMillan, 2003; Pudlák, 1997], such a Craig interpolant φ can be derived by the SAT-solver from a proof of unsatisfiability. Therefore φ will represent a more complete set of states that caused unsatisfiability of the query, i.e. more states that are inductive relative to F_{i-1}. But apart from reasoning about the same variables, there will be no syntactic relation between φ and $\neg c$. We call φ a semantic generalization, as it generalizes $\neg c$ in a semantic way, without any constraints on the syntactic structure of the resulting generalization.

There is however one aspect that we have to work around. As mentioned, the derived interpolant φ has no syntactic relation to $\neg c$. However, the IC3 algorithm requires a generalization to be a cube, but φ will be a FO predicate with arbitrary structure. In order to fix this, we need to transform φ into a cube. While conversion to a single cube is not possible in general, we can always convert φ into *Disjunctive Normal Form* (DNF), i.e. a disjunction of cubes.

Hybrid Generalization

So far we identified two generalization approaches: syntactic generalization that operates solely on the syntactic structure of the cube and tries to drop literals, and semantic generalization that derives interpolants based on the proof of unsatisfiability from the SAT-solver. There is however a third approach that incorporates aspects from both approaches: As presented in [Bradley, 2011], the *unsatisfiable core* [Biere, Heule, et al., 2009; Kroening and Strichman, 2008] obtained from the SAT-solver indicates which literals in the cube are causing the unsatisfiability of the SAT-query and thus cause the cube to be inductive relative to F_{i-1}. This generalization obviously takes the form of a semantic generalization as it is guided by the SAT-solver's resolution refutations. On the other hand, it can only result in a subcube of the original cube c, giving it some characteristics of a syntactic generalization. We therefore identify unsatisfiable cores for generalization by a hybrid mixture of syntactic generalization and semantic generalization.

Example 3.5. Recall Example 3.2 on page 64. In iteration $k = 1$, we find CTI state 1110 and the corresponding clause $\neg x_1 x_2 x_3 \bar{x}_4$ is inductive relative to frame F_0, since the query $sat\,(F_0 \wedge \neg x_1 x_2 x_3 \bar{x}_4 \wedge T \wedge x_1' x_2' x_3' \bar{x}_4')?$ is unsatisfiable, To ease readability, we will strip down the transition relation T to only those transitions that are actually useful in this example: Since F_0 only contains initial states, we just need to consider those transitions that start in any of the two initial states. The SAT-query for inductivity of $\neg x_1 x_2 x_3 \bar{x}_4$ relative to F_0 therefore looks like:

$$\begin{aligned} sat(&\bar{x}_2 \bar{x}_3 x_4 \wedge \\ &\neg x_1 x_2 x_3 \bar{x}_4 \wedge \\ &(\bar{x}_1 \bar{x}_2 \bar{x}_3 x_4 \bar{x}_1' \bar{x}_2' \bar{x}_3' \bar{x}_4' \vee x_1 \bar{x}_2 \bar{x}_3 x_4 x_1' \bar{x}_2' \bar{x}_3' \bar{x}_4') \wedge \\ &x_1' x_2' x_3' \bar{x}_4')? \end{aligned}$$

Here we can see that the query is unsatisfied as neither transition satisfies the postcondition $x_1' x_2' x_3' \bar{x}_4'$. Considering the second transition, the SAT-solver might return an unsat core whose projection on our cube $x_1 x_2 x_3 \bar{x}_4$ indicates a conflict in the literals x_2 and x_3. We could therefore generalize $x_1 x_2 x_3 \bar{x}_4$ to $x_2 x_3$.

As we see from Example 3.5, an unsatisfiable core can significantly reduce the size of the cube with almost no additional costs. How to obtain a minimal unsatisfiable core has been studied extensively [Davydov et al., 1998; Fleischner et al., 2002; Kleine Büning, 2000; Papadimitriou and Wolfe, 1988].

Improving Generalization

As presented in [Hassan et al., 2013], the IC3 algorithm can be improved by considering causes for a failed generalization. Following the notion of CTIs, such states are referred to as a *Counterexample to Generalization* (CTG). Whenever dropping a literal *lit* fails, $F_{i-1} \wedge \neg c \wedge T \wedge c'$ becomes satisfiable, meaning that some s exists that corresponds to the minterm of the model of the SAT-query. This s will reach $\neg c$ with one step in T and thus causes c to be generalized. However, s may not be reachable and is very likely to be farther away, but still backwards reachable from a bad state. As such, pausing the generalization of c and investigating the reachability of s may allow IC3 to drop *lit* from c. In addition, blocking s may preempt the backwards search of some other CTI. However, an unbounded inspection of a CTG will most likely diverge. To avoid this, [Hassan et al., 2013] proposes two bounds for the CTG procedure: First, the search-depth is limited by some *maxDepth*, usually very low. Experimental results in [Hassan et al., 2013] have shown best performance with $maxDepth = 1$, i.e. whenever a CTG is not inductive, the search is aborted immediately. Apart from *maxDepth*, a limitation is in the number of CTGs that get inspected. If this number exceeds the bound *maxCTG*, it will also abort to avoid discovering too many CTGs. For *maxCTG*, the experiments of [Hassan et al., 2013] have shown that 3-5 works best.

Example 3.6. Again, consider the transition system $\mathcal{S}$ from Figure 3.1 on page 53. Suppose that 1100 is the first CTI to be found. Its negation is inductive since it has no predecessors. Assume that for generalization we try to drop the third literal yielding the cube $c = x_1 x_2 \bar{x}_4$. The negation of c however is not inductive, due to 1110's predecessor 0101. Thus any generalization containing 1100 and 1110 must contain 0101 and 0101's predecessor 0011 must also be in this generalization. But the smallest cube containing 1100 and 0011 is *true*, which does also include all initial states and is thus not a valid generalization. Similar situations apply to all other literals. Thus IC3 would only be able to block the CTI. However, [Hassan et al., 2013] proposes to drop x_3 and investigate why it fails, i.e. due to the CTG 0101. Given a *maxDepth* of 2, we would determine that 0101 is not inductive, too, and descent into CTG 0011. After blocking 0011, CTG 0101 becomes inductive, such that x_3 can be dropped.

Example 3.6 illustrates that Counterexamples to Generalization can help to discover deep, backward reachable states early and thus allow for better generalizations that speeds up the convergence. On the other hand choosing suitable bounds for the maximal depth and the maximal number of CTGs is a hard task, since choosing too large bounds will lead to additional, possibly worthless overhead, whereas choosing too small bounds will prevent noticeable impact of the proposed improvements.

3.4 Propagation

The presented generalization procedure for reducing cubes in their size and thus excluding larger state sets is a key factor for the scalability and the impact of IC3. However, as mentioned in Section 3.2, another key aspect of IC3 is the so-called *propagation phase*, sometimes also referred to as pushing. In this phase, that takes place after extending the frontier k, IC3 tries to extend the learned lemmas to subsequent frames. In order to do so, it iterates over all frames and all clauses, and checks whether for frame F_i and clause $d \in F_i$

$$F_i \wedge T \Rightarrow d' \tag{3.10}$$

is a valid implication. In other words, it checks whether d would also hold after one step. If this is the case, then by invariant (3.7) of the IC3 algorithm $d \in F_{i+1}$ and thus IC3 will push d forward to F_{i+1}. This procedure has proven to be especially useful in two aspects. First, IC3 can learn information about the new frame at $k+1$ without searching CTIs and executing the backward-search. Second, by pushing clauses forward, IC3 accelerates the convergence of frames and thus potentially terminates faster.

Example 3.7. Reconsider Example 3.2 on page 64. After iteration $k = 1$, IC3 maintains the following frames:

$$\begin{aligned} F_0 &:= I \wedge \neg x_2 \wedge \neg x_1 \bar{x}_2 x_3 x_4 \\ F_1 &:= P \wedge \neg x_2 \wedge \neg x_1 \bar{x}_2 x_3 x_4 \\ F_2 &:= P \end{aligned}$$

To push a clause forward, we will check for each clause d in the last frame they appear in, whether it holds in the next frame, too. In our case we consider the clauses $\neg x_2$ and $\neg x_1 \bar{x}_2 x_3 x_4$ in frame F_1:

$$\begin{aligned} &\mathit{sat}\,(F_1 \wedge T \wedge x_2')? && \text{is satisfiable, e.g. } 0010 \\ &\mathit{sat}\,(F_1 \wedge T \wedge x_1' \bar{x}_2' x_3' x_4')? && \text{is unsatisfiable} \end{aligned}$$

This means, that pushing $\neg x_2$ failed, because of the F_1-state 0010 and its x_2-successor state 0100, but $\neg x_1 \bar{x}_2 x_3 x_4$ can successfully be pushed forward to F_2.

However, a big downside of this approach to pushing clauses is that after *every* iteration IC3 will check *every* clause d in *every* frame F_i for pushing. This naturally creates a large overhead, as many clauses cannot be pushed to F_{i+1}. To solve this problem, [Suda, 2013] proposes a technique called *triggered clause pushing*. From an abstract point of view, triggered clause pushing has some similarities to Counterexamples to Generalization: it takes a closer look at the reason for a pushing attempt to fail.

If pushing fails, the query $F_i \wedge T \wedge \neg d'$ is satisfied by some model. [Suda, 2013] classifies the state s extracted from the SAT-solver model as a *witness* to why (3.10) is not valid. As long as this witness is contained in F_i, (3.10) will be satisfied for clause d. In order to detect when s will get blocked in F_i, [Suda, 2013] uses subsumption on every newly added clause $\hat{d}$ to check whether the cube c representing state s is blocked by $\hat{d}$. If $\hat{d} \subseteq \neg c$, i.e. $\hat{d}$ is subsumed by $\neg c$, then s is blocked from F_i by $\hat{d}$. This means that the witness for pushing clause d does not prevent pushing of d anymore. While checking subsumption after every blocking also adds some overhead to the IC3 algorithm, [Suda, 2013] argues that subsumption checks already happen in IC3 routinely in order to remove subsumed clauses to keep the memory footprint low: [Bradley, 2011] does so during pushing, while [Eén, Mishchenko, et al., 2011] removes subsumed clauses during each blocking. In addition, syntactic subsumption checks are typically much faster than semantic checks of the form (3.10) that involve a SAT-solver.

Apart from saving SAT-queries, the authors of [Suda, 2013] also propose not to postpone pushing until the end of the iteration, but to check pushing of d early. They implement this early check by adding additional objects to the obligation queue, so-called *push requests*. Such a push request indicates that clause d can be pushed from frame F_i to F_{i+1}. It will be enqueued in the priority queue at level i and in case proof obligations and push requests occur at the same level, proof obligations will always be preferred. With this partial order, push requests will always be handled during backtracking the search from level i to $i+1$: Every proof obligation (i, c) at level i will be popped from the queue Q before some push request. If however, a proof obligation shows its respective cube not to be inductive, the search procedure continues with the obligation $(i-1, \hat{c})$ and push requests at level i will only be popped when the search returns back to level i.

An interesting approach that is also proposed in [Suda, 2013] is to use subsumption checks in order to delay proof obligations: Given an obligation (i, c) in queue Q and blocking some clause d at level i, such that $\neg c \subseteq d$. As $\neg c$ is subsumed by d, this means that c is not part of F_i any more and thus checking inductivity becomes redundant. Thus, any such proof obligation will be shifted to level $i+1$.

Example 3.8. Let us reconsider the failed pushing attempt in Example 3.7 on page 79. We tried to check whether the clause $\neg x_2$ can be pushed from F_1 to F_2, but this failed due to the F_1-state 0010 and its x_2-successor 0100. Using triggered clause pushing, we detect 0010 as a witness to why the clause $\neg x_2$ cannot be pushed. We therefore store 0010 as witness for $\neg x_2$ and whenever we add a new clause to F_1, we check via subsumption, whether 0010 is still contained. The first time this subsumption fails is in iteration $k = 3$, where we added the clause $\neg x_3$ to F_1, such that 0010 is removed from F_1. Since $\neg x_3$ also blocks state 0011, we can now successfully push $\neg x_2$ forward to frame F_2. However, if we would stick to the original IC3, we would push $\neg x_2$ only after successfully blocking all CTIs. This however makes pushing $\neg x_2$ redundant, since we would have already blocked x_2 as the generalization of 0101 at frame F_2 in the next iteration of the blocking function. Thus we can see that the introduction of push requests to the obligation queue Q is essential for the impact of triggered clause pushing, since it prepones the pushing of $\neg x_2$ to F_2 to happen directly after blocking $\neg x_3$ in F_1. In turn, pushing $\neg x_2$ to F_2, the obligation $(\bar{x}_1 x_2 \bar{x}_3 x_4, 2)$ to prove inductivity of 0101 at level 2 becomes obsolete, since 0101 has been blocked by $\neg x_2$, which means that it will be shifted to a later iteration.

Example 3.8 shows that IC3 can largely benefit from preferring pushing already learned lemmas over learning new ones and more tightly integrating the interactions between blocking and pushing into the main search phase. Another recent improvement that emphasizes the important role of pushing for IC3 is called *Quip*, short for *Quest for Inductive Proofs* [Gurfinkel and Ivrii, 2015]. Quip offers a number of improvements to different parts of IC3, many of which are motivated by the fact that propagation of already learned lemmas is cheaper than learning new lemmas. However, propagating too many lemmas in an unguided way will just flood the frame sequence with often unnecessary clauses, causing another performance drop. Therefore Quip aims at finding lemmas that are advantageous to propagate while ignoring those that are not advantageous.

The first modification introduced by Quip is the addition of a second type of proof obligation. Standard proof obligations as used in IC3 are called *must*-obligations in Quip, as they must be proven in order to prove the property P safe. The respective counterpart are *may*-obligations, that could be helpful, but P might also be safe without them. While *must*-obligations originate from failed inductivity queries, *may*-obligations have a strong analogy to push requests:

Given a clause $d \in F_i$, Quip produces a *may*-proof obligation of c $(= \neg d)$ at level $i + 1$ that will be checked for inductivity relative to F_i. If this check succeeds, then c will be blocked in F_{i+1}, which means that clause d will be added to F_{i+1}, i.e. d is pushed from F_i to F_{i+1}. If on the other hand the inductivity check fails, then IC3 finds a witness trace showing why d cannot be pushed to F_{i+1}. Such a witness trace will indicate a concrete state s that is forward reachable, but excluded by d. While s stems from a *may*-proof obligation, it must not necessarily lead to a violation of the property P. But it does explain why d is not inductive and thus that d will not be part of the inductive invariant. Therefore Quip classifies d as a *bad* lemma. Apart from classifying d as a *bad* lemma, s can also be used to invalidate other *may*-proof obligations or to discover a real counterexample with less transitions. Therefore s will be stored in a set of reachable states. Like push requests, *may*-proof obligations will be preferred over *must*-proof obligations of the same level.

In contrast to *bad* lemmas, Quip periodically determines the maximal inductive subset of all lemmas that have been learned and stores this subset in a separate frame called F_∞. All lemmas in F_∞ are considered *good* lemmas that will always remain in the system. The idea of a separate frame F_∞ storing the maximal inductive subset was presented in [Eén, Mishchenko, et al., 2011], but the authors concluded that determining the maximal inductive subset was too expensive and thus outbalanced possible, but small improvements.

However, by combining *good* and *bad* lemmas, Quip is able to guide the pushing process in order to only push clauses that are advantageous and prune clauses that are not, such that determining maximal inductive subsets offers more potential for optimizations.

Following the spirit of preferring pushing over learning new lemmas, Quip also lifts the restriction of other IC3 algorithms that push only up to the current bound k.

Chapter 4

Software Verification with IC3

In the previous section, we have seen the IC3 algorithm for verifying safety properties on Boolean transition systems. The publication of IC3 generated a lot of attention, not only in hardware verification, but also in other, related domains, and many subsequent publications took up the ideas of IC3, improved and refined them and also adapted them to many other problems. The main reason for this huge success of IC3 is most likely its striking performance compared to other tools. In its original version, as published in [Bradley, 2011], IC3 scored *third* in UNSAT category, i.e. only cases where no bad state is reachable, of the Hardware Model-Checking Competition (HWMCC) 2010. However, within less than a year after IC3's publication, [Eén, Mishchenko, et al., 2011] presented a small number of improvements to the IC3 algorithm and showed that their improved IC3 variant, called *Property-Directed Reachability*, would have allowed IC3 to win HWMCC'10. With this amazing result and competitors that fine-tuned their tools for many years, questions arose why this new algorithm was so much faster. In [Bradley, 2012], the authors identify three main aspects for the success of IC3. First, it tries to prove inductivity, rather than reachability, such as an early attempt on linear inequality invariants in [Colón et al., 2003; Sankaranarayanan et al., 2005]. This approach however suffered from enumerating instantiations of the parameters of the inequalities and thus generated the strongest possible over-approximation. Second, to avoid enumeration, [Bradley and Manna, 2006, 2007a] take a property-guided approach to guide the search for inductivity towards CTIs. The third important aspect of IC3 is given by its incrementality. Rather than taking a monolithical approach and trying to come up with a strengthening at once, IC3 creates its incremental frame sequence.

4.1 Previous approaches

After its impressive performance for hardware model-checking, the question was how to apply IC3 to software model-checking. But even though the domains of hardware and software model-checking share some common ground, IC3 cannot be applied directly, due to a number of differences in both settings. Maybe the biggest difference between both settings is the state space that they operate on. Boolean transition systems consist of a number of Boolean variables $x \in X$ that span the state space, i.e. for $|X| = n$ the state space contains 2^n states. For software systems however, variable valuations map to much more expressive domains than Boolean, such as integers of various or infinite size and floating point numbers, which also contain values such as $\pm\infty$ or *not-a-number* NaN. These values can change from one program line to the other, such that for a variable space of size n and j program lines, the state space contains $n \cdot j$ states.

Furthermore we might also encounter nondeterminism, depending on our input model, which is not possible in standard hardware model-checking [Griggio and Roveri, 2016]. Possible forms of nondeterminism would be either data nondeterminsm, i.e. the programs reads input values from the user or the current timestamp, or control-flow nondeterminsm, i.e. a program can nondeterministically branch or loop. While constructs for non-deterministic control-flow are not present in most programming languages, from a program-analysis point-of-view, we can easily construct such nondeterminism by comparison of values with nondeterministic data, such as user input, in the guards of *if*- or *while*-statemens. In addition, simple *probabilistic programs* that can branch based on the result of throwing a fair coin, can be translated into non-deterministic branching, if one is only interested in reachability of a certain state.

IC3-SMT The problem of more expressive variables can easily be solved by replacing propositional logic with first-order theories, replacing a SAT-solver with an SMT-solver. With this modification, an intuitive lifting [Cimatti and Griggio, 2012] is to model all variables in first-order theory and add an additional variable *pc* that models the *program counter*, i.e. the respective position in the program.

However, even this simple approach, called *IC3-SMT*, reveals a problem inherent to all IC3 liftings: The termination of IC3 relies on the finiteness of the state space and a progress assumption, i.e. in each step at least one new state has to be discovered. More concretely, IC3 uses the solver's model for satisfied formulas to extract the violating predecessor and later generalizes this state to a set of states by the procedures explained in Section 3.3. This however is not straightforward in such a lifting since the generalizations of Section 3.3 can only yield finite sets, while the state space for the SMT case can be infinite for theories like LRA/LIA (see Section 2.1.2). The consequence is that only finite subparts of the infinite state space are excluded, thus termination cannot be guaranteed.

To circumvent this, [Cimatti and Griggio, 2012] recommends to use exact preimages of bad states. To do so, the authors use weakest preconditions (WP) using quantified formulas, i.e. a formula which existentially quantifies the variables after the transition, and solve the formula using quantifier elimination. This however is not always possible, but only for those SMT theories that admit quantifier elimination. This is not the case for all theories, but for the remainder we assume only theories containing quantifier elimination algorithms. For more information on quantifier elimination see [Kroening and Strichman, 2008]. Note that while quantifier elimination can solve the problem of diverging into blocking single-state predecessors, the elimination is usually computationally expensive.

Example 4.1. We will sketch the idea of IC3-SMT on the SV-COMP benchmark *gcd_2_true-unreach-call* from the *bitvector* category. For simplicity, Listing 1 shows the inlined C code. We will not exercise a full run of IC3-SMT, but rather show a small excerpt that illustrates the way IC3-SMT behaves.

We start by translating the C program from Listing 1 into the global transition relation T:

$$
\begin{aligned}
T := \ldots & \\
& ((pc = 4)) \implies ((pc' = 5) \wedge (y' = b)) \;\wedge \\
& ((pc = 5) \wedge (a < 0)) \implies (pc' = 6) \;\wedge \\
& ((pc = 5) \wedge \neg(a < 0)) \implies (pc' = 7) \;\wedge \\
& \ldots \\
& ((pc = 9) \wedge (b \neq 0)) \implies (pc' = 10) \;\wedge \\
& ((pc = 9) \wedge \neg(b \neq 0)) \implies (pc' = 13) \;\wedge \\
& \ldots \\
& ((pc = 12)) \implies (pc' = 9) \;\wedge a' = t \\
& \ldots
\end{aligned}
$$

```
 1  #include "assert.h"
 2
 3  int main() {
 4      signed char a = __VERIFIER_nondet_char();
 5      signed char b = __VERIFIER_nondet_char();
 6      signed char y, t;
 7      y = b;
 8      if (a < (signed char)0) {
 9          a = -a; }
10      if (b < (signed char)0) {
11          b = -b; }
12      while (b != (signed char)0) {
13          t = b;
14          b = a % b;
15          a = t; }
16      if (y > (signed char)0) {
17          assert(y >= a); }
18      return a; }
```

Listing 1: C program computing the GCD (from SV-COMP bit-vector set)

Note that we start the program counter value in the first line of the main, such that line numbers in Listing 1 and pc values are offset by three. Such a T looks very similar to what we have seen from IC3 in Chapter 3, except that we now take the control flow of the program into account by the program counter pc. We can see that T matches the *if* in line 8 of Listing 1 by two distinct transitions from $(pc = 5)$ with different premises and implications. Furthermore, we represent the loop condition of the *while* loop in line 12 by an increment of the pc if the guard is satisfied and a jump to line 16 when the guard is not valid. At the end of the loop, we jump back to evaluate the guard again.

Given T, we identify $P = \neg(pc = 14 \wedge y < a)$. We now start IC3 as presented in Section 3.2, by checking for zero-step counterexamples, i.e. $sat\,(pc = 1 \wedge pc = 14 \wedge y < a)$?, and one-step counterexamples, i.e. $sat\,(pc = 1 \wedge T \wedge pc' = 14 \wedge y' < a')$?, which are both unsatisfiable. After these initial checks, we initialize the frame sequence to $F_0 = pc = 1$ and $F_1 = P$. As a first CTI we will find a partial assignment φ like

$pc = 13, y = 1, a = 0$ with $c = (pc = 13) \wedge (y = 1) \wedge (a = 0)$. Given this CTI cube c, we check whether c is inductive relative to F_0. Therefore we check $(pc = 1) \wedge \neg((pc = 13) \wedge (y = 1) \wedge (a = 0)) \wedge T \wedge ((pc' = 13) \wedge (y' = 1) \wedge (a' = 0))$. This query is not satisfied, since T does not contain a transition from $(pc = 1)$ to $(pc' = 13)$. We therefore want to block c and thus generalize it. To do so, we start by dropping the literal $pc = 13$ from c and check whether the remaining cube is still inductive relative to F_0. The query $(pc = 1) \wedge \neg((y = 1) \wedge (a = 0)) \wedge T \wedge ((y' = 1) \wedge (a' = 0))$ is satisfied by the assignment $pc = 1, y = 1, a = 1, y' = 1, a' = 0$ and thus dropping $(pc = 13)$ fails. This is a very characteristic outcome of IC3-SMT when trying to drop the literal containing the program counter variable pc from c. Due to the linear behaviour of programs, dropping this clause will fail in almost all cases, leading to a large amount of unnecessary SMT queries. We omit further details on the flow of IC3 and refer the reader to Example 3.2 for a more detailed example on IC3.

While IC3-SMT offers a very intuitive and simple way to adapt IC3 to software model-checking, the evaluation of [Cimatti and Griggio, 2012] shows that in practice, the performance of IC3-SMT is very limited. While [Cimatti and Griggio, 2012] does not offer any explanation why IC3-SMT performs bad in practice, we believe that one of the reasons is that IC3-SMT attempts to mix control- and dataflow by modelling control-flow as just another program variable. While this approach is sound, it seems that control- and dataflow are just too different to be mixed this way. In particular modelling the program counter as a simple program variable will lead IC3 in many failing attempts to generalize parts of the binary representation of the program counter. By trying to drop some of the Boolean variables representing the program counter, IC3 will create a cube that distributes over multiple, non-adjacent program counter evaluations, such that inductivity of such cube fails most likely.

Tree-IC3 Therefore [Cimatti and Griggio, 2012] propose another, more advanced approach to software model-checking via IC3. The method called *Tree-IC3* tries to compensate the shortcomings of IC3-SMT by exploiting the control-flow structure of the program. To do so, Tree-IC3 takes the control-flow graph of a program and, just like CEGAR, unrolls it into an *abstract reachability tree* (ART). This way, control-flow is represented in an explicit way, while data is still kept symbolic. [Cimatti and Griggio, 2012] describe their method as "an *explicit-symbolic* approach, similar to the lazy abstraction approach of [Henzinger et al., 2002]".

The ART, that is unwound using a DFS strategy, associates each node n with a tuple (ℓ, D) consisting of a program location ℓ and a set of clauses $D = \{d_0, \ldots, d_n\}$, where location ℓ_i characterizes the program counter value $pc = i$ control-flow representation. Like [Henzinger et al., 2002], Tree-IC3 unrolls the ART until it unrolls to a node n_e that is associated with an error location ℓ_E. Given such a node n_e, the path $\pi = n_i, \ldots, n_e$ from n_i to n_e corresponds to a sequence of program locations $\ell_0, \ell_1, \ldots, \ell_E$ that lead from an initial location ℓ_0 to an error location ℓ_E. As such π could be a counterexample path indicating a violation of the property. However, π is only based on the control-flow of the program and we don't know yet, whether the path is feasible, i.e. there exists an execution of the program along π. We therefore call π an *abstract counterexample path.* While both procedures, Tree-IC3 and the lazy abstraction from [Henzinger et al., 2002], are identical up to this point, Tree-IC3 diverges from [Henzinger et al., 2002] by applying a procedure that mimics the blocking procedure of IC3 to π. While [Henzinger et al., 2002] analyzes the abstract counterexample in a monolithic way by considering all transitions at once, Tree-IC3 uses the characteristic single-step approach of IC3. Since the analysis is limited to π only, Tree-IC3 can ignore the control-flow aspect and more or less apply the standard IC3 approach, except for a few differences:

1. since clauses of a node are implicitly conditioned to a control-flow location a clause such as $\neg(pc = pc_i) \vee d$ in IC3-SMT becomes just d in node $n = (pc_i, \ldots)$. As a consequence the clause expressing the unreachability of the error location $\neg(pc = pc_e)$ becomes the empty clause $\bigvee \emptyset = \mathit{false}$.

2. given a transition formula T_i from node n_i to node n_{i+1}, Tree-IC3 encode only the dataflow of the transition into T_i since the control-flow part is expressed explicitly in the ART.

3. consider an abstract counterexample path $\pi = n_i, n_1, \ldots, n_k, n_e$ on which an IC3-style blocking phase is to be applied. Whenever a CTI state in D_e associated to node n_e is found to not be inductive, its predecessors have to be determined. However, due to the explicit-symbolic setting, the control-flow is to be respected and transitioning is only allowed to the location ℓ_k associated to node n_k. This implies that in every step in π different transitions T_i are to be considered.

[Cimatti and Griggio, 2012] also note that the original inductivity check (3.2) cannot be applied to Tree-IC3, but rather needs to be changed to

$$F_{i-1} \wedge T_{i-1} \Rightarrow \neg c' \tag{4.1}$$

in order to stay sound. While [Cimatti and Griggio, 2012] explain this with the different transition formulas T_i, we will have a more in-depth look at the reasons for this modification and when to relax (4.1) in Section 4.2.

Due to the explicit handling of control-flow it is no longer possible to use IC3's termination criterion directly, but rather we have to reformulate it in the spirit of the explicit-symbolic state space representation. On Boolean transition systems, IC3 will terminate when it encounters an inductive frame, i.e. a frame F_i for which it holds that $F_i = F_{i+1}$. For IC3-SMT we were able to apply this criterion as well, since the program counter became part of the symbolic state space. However, for explicit control-flow, we must reconsider what inductivity means in this context. On a very abstract level, inductivity means that given a set of states S and a transition relation T, S is closed under T, i.e. $\forall s \in S.(s, s') \in T \Rightarrow s' \in S$. When we lift this interpretation to ARTs, this means that all reachable states must have been visited. Such an ART is called *complete*. Formally, no node exists that has successor nodes with associated tuples (ℓ, D), such that there exists another node associated with (ℓ', D') where $\ell = \ell'$ and $D' \Rightarrow D$. In other words, we must forbid visiting nodes that are associated to a location that we have visited before and this previously visited node is associated with a larger state set. Thankfully this criterion, called *node coverage*, has been studied extensively in the context of ART-based algorithms such as CEGAR, e.g. [Henzinger et al., 2002]. Tree-IC3 therefore adopts the standard coverage check and terminates when all nodes are covered.

While Tree-IC3 noticeably differs from IC3-SMT, three key aspects of IC3 are recognizeable in Tree-IC3:

1. By unrolling the ART and refining clause sets according to IC3's invariants (3.4) to (3.7)[1] until all nodes are covered, it aims at finding a sufficient strengthening for all nodes that results in an *inductive* ART.

2. Furthermore, Tree-IC3 constructs its strengthening for ART nodes in an *incremental* way by checking inductivity on abstract counterexample paths in a step-wise fashion, which contrasts the monolithic approaches usually found in ART-based CEGAR algorithms.

3. Finally, by using the IC3-style blocking procedure on abstract counterexample paths, it can be considered as property guided. More specifically, an error node corresponds to a $\neg P$-state in standard IC3 with the last non-error node being the CTI state.

[1]Note that due to the extraction of the *pc* variable, (3.5) is satisfied by construction and thus not considered any further.

$$\begin{tikzcd}
{[a = ?]^1} \arrow[d] & & & {[b \neq 0]^9} \arrow[d, "\top"] \arrow[dddd, bend left, "\perp"] & \\
{[b = ?]^2} \arrow[d] & & & {[t = b]^{10}} \arrow[d] & \\
{[y = 0, t = 0]^3} \arrow[d] & & & {[b = a\%b]^{11}} \arrow[d] & \\
{[y = b]^4} \arrow[d] & & & {[a = t]^{12}} \arrow[uuu, bend left] & \\
{[a < 0]^5} \arrow[r, "\perp"] \arrow[d, "\top"] & {[b < 0]^7} \arrow[d, "\top"] \arrow[uuuurr, bend left, "\perp"] & & {[y > 0]^{13}} \arrow[r, "\perp"] \arrow[d, "\top"] & {} \\
{[a = -a]^6} \arrow[ur] & {[b = -b]^8} \arrow[uuuuurr, out=0, in=180] & Error & {[y \geq a]^{14}} \arrow[l, "\perp"] \arrow[r, "\top"] & {}
\end{tikzcd}$$

Figure 4.1: Control-flow graph corresponding to Listing 1

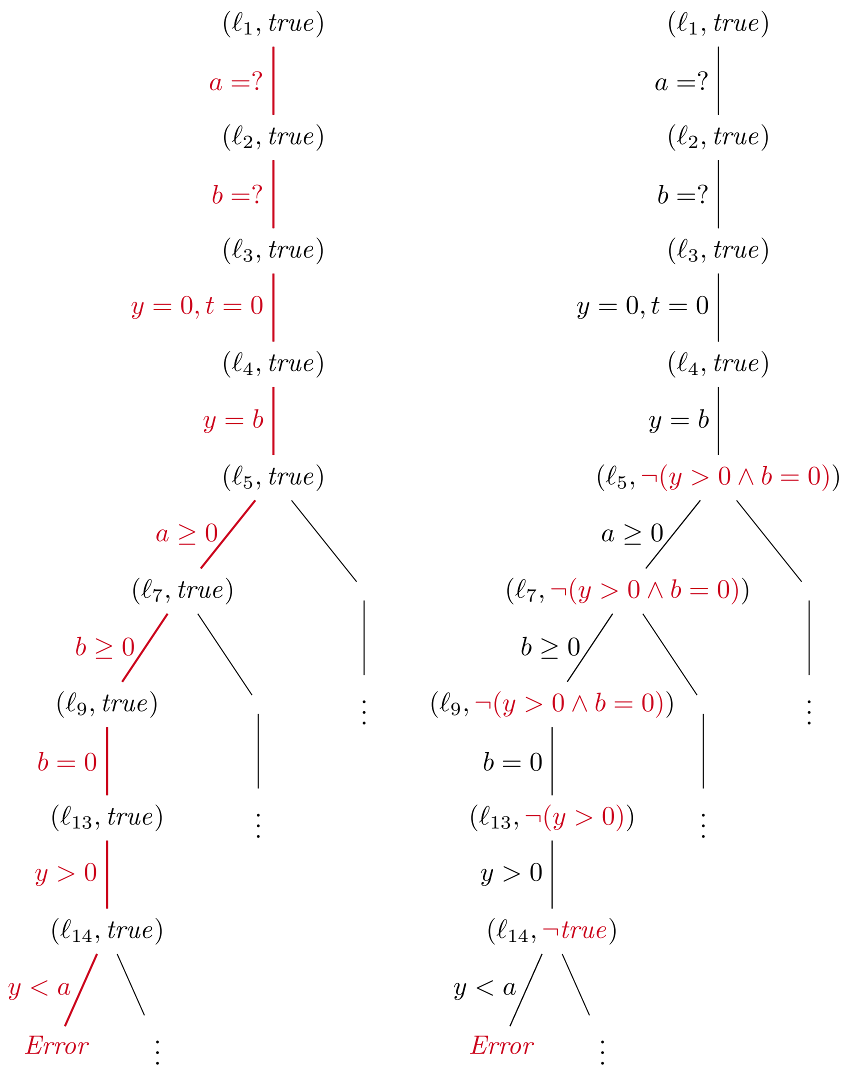
Example 4.2. Recall the example C program from Listing 1 on page 86. We start Tree-IC3 by building the control-flow graph $\mathcal{G}$ of the program as depicted in Figure 4.1. We unroll $\mathcal{G}$ into an ART like any other standard ART-based model checker until we find an error location in the ART. A sketch of such an ART is shown in Figure 4.2a with the abstract counterexample path marked in red. For this abstract counterexample, Tree-IC3 has to check whether it is spurious or not. It does so by starting the IC3 search phase as usual: We determine the CTI cube $c = (y \geq a)$ and check whether this is inductive relative to F_{j-1} where j is the depth of the counterexample trace, in this case 10. Using the known IC3 search phase, we find that the abstract counterexample path is backwards feasible until we reach location ℓ_5 where the cube c of proof obligation $(5, c = (y < a \wedge y > 0 \wedge b = 0 \wedge b \geq 0 \wedge a \geq 0)$ is inductive relative to F_4. This is due to the transition $y = b$ that conflicts with c. We can use the unsatisfiable core of the query to generalize c to $y > 0 \wedge b = 0$.

We omit the remainder of the blocking phase and instead give the resulting ART (depicted in Figure 4.2b) after blocking all CTIs. Thus the abstract counterexample path was spurious. Since there exist uncovered nodes, we continue the ART unrolling.

[Cimatti and Griggio, 2012] compare their implementations of IC3-SMT and Tree-IC3, together with another variant using Craig interpolation, that will be considered later, and other model-checking tools that use lazy abstraction. The experimental results reveal that Tree-IC3 cannot only solve a significantly larger portion of the chosen benchmark set, but also does so much faster. For some benchmarks, the performance differs up to three orders of magnitude.

The results from [Cimatti and Griggio, 2012] motivate the extraction of control-flow in the form of explicit-symbolic representations that simplify the handling of the different aspects of control- and dataflow. However, we found that unrolling the control-flow into the tree-structured ART, as well as the necessary coverage checks that come associated with this, introduce an additional overhead while at the same time diverge from the original spirit of the IC3 algorithm.

To solve this discrepancy we aimed to design an algorithm that more closely resembles IC3 while at the same time exploiting the advantages offered by extracting the control-flow from the symbolic representation. In the remainder of this chapter, we start with a detailed description of our algorithm called *IC3CFA* for IC3 software model-checking. In Section 4.2 we adapt the search/blocking



(a) Abstract reachability tree unrolled from CFG of Figure 4.1

(b) Abstract reachability tree after IC3 search/blocking phase

Figure 4.2: Example ART unrolling of CFG from Figure 4.1

phase of IC3 to the new control-flow structure. Section 4.3 presents how generalization can be applied to our algorithm and which consequences this has. In Section 4.4 we will take a closer look at the pushing/propagation phase of IC3CFA and discuss issues and solutions. The chapter concludes with a comparison to other methods for IC3-style software model-checking in Section 4.5.

4.2 IC3CFA

In this section, we present the search phase of our algorithm for IC3-style software model-checking that is applied to a control-flow automaton (CFA) without unrolling the transition relation, called *IC3CFA*. We will start by introducing the main idea underlying IC3CFA and later give a pseudocode sketch of IC3CFA with annotated pre- and postconditions. We will present the algorithm in a way that is similar to the presentation of IC3 in [Bradley, 2011] and show that the proof of correctness is mostly analogously. Afterwards we will review the relative inductivity query (4.1) presented in [Cimatti and Griggio, 2012] and present a relaxation. We will conclude this section by a detailed evaluation of the IC3CFA algorithm with comparison to other control-flow oriented IC3 algorithms, such as Tree-IC3.

4.2.1 Preliminaries

In the previous section, we saw the Tree-IC3 algorithm of [Cimatti and Griggio, 2012], one of the first liftings of IC3 to software model-checking. In the comparison to IC3-SMT [Cimatti and Griggio, 2012], we saw that the extraction of control-flow has a large beneficial effect on the overall algorithm. However, unrolling the control-flow graph (CFG) into an ART and doing all the coverage checks for all nodes adds some tedious overhead to the algorithm. So in order to avoid this overhead our aim is to apply IC3 in a more direct way. In particular, our objective is to avoid the unrolling, since monolithic unrolling also diverges from the incremental nature of IC3.

For our algorithm, we use a slightly different form of control-flow representation than [Cimatti and Griggio, 2012]. The most common definition of control-flow graphs (CFG) in the sense of [Nielson et al., 1999] considers control-flow graphs to be graphs where nodes are labeled with program instructions and edges indicate successor instructions. However, for our setting of lifting IC3 to control-flow, we found the notion of program instructions as transitions between variable valuations (states), used in control-flow automata (CFA) as defined in other software model-checking environments [Beyer, Cimatti, et al., 2009], more appropriate. A more detailed explanation and a formal definition of a CFA is given on page 47. In such CFA, the nodes are simply abstract states before execution of an instruction and edges are labeled with program instructions. From a semantic point of view, an edge transforms a pre-state to a post-state by the predicate transformer that is defined by the edge label.

In order to apply IC3 to a CFA, let us reconsider IC3-SMT. While not the best performing, its simple design makes it easy to reason about and its correctness is obvious from the correctness of IC3. Given the state space of IC3-SMT, extracting the control-flow variable pc into a CFA with n locations yields a partitioning of n equally large, disjunct state spaces S_i that are induced by the program variables $v \in Var$, where each such S_i is implicitly conditioned by $pc = i$. If we define the program counter value to start with 0 before the first program instruction and to be incremented after each instruction, we associate each $pc = i$ with location ℓ_i and thus also S_i is associated with ℓ_i.

In order to formalize this splitting, we define *data regions* and *regions* along the lines of [Henzinger et al., 2002].

Definition 4.1 (Data region)**.** A data region is represented by a quantifier-free FO-formula φ over Var and consists of all variable assignments σ satisfying φ, i.e., $\{\sigma \mid \sigma \models \varphi\}$.

Following Definition 4.1, we can now reason about sets of states in the state space that are defined only by program variables. In order to incorporate the program location in the spirit of one location implicitly being conditioned to a data region, we define what we call *regions*, in [Henzinger et al., 2002] also referred to as *atomic region*.

Definition 4.2 (Region)**.** We define a region $r = (\ell, \varphi)$ as a pair consisting of location $\ell \in L$ and data region φ. The corresponding formula of region $r = (\ell_i, \varphi)$ is defined as $(pc = pc_i \wedge \varphi)$. Analogously, the corresponding formula for $\neg r$ is defined as $\neg\,(pc = pc_i \wedge \varphi)$.

Assume an IC3-SMT-transition $pc = i \wedge \psi \wedge pc' = j$ with some predicate transformer $\psi(Var, Var')$ transforming variable assignment σ to primed successor assignment σ'. The transformer ψ will be the label of CFA $\mathcal{A} = (L, G, \ell_0, \ell_E)$ along the edge from location ℓ_i to ℓ_j, i.e. $(\ell_i, \psi, \ell_j) \in G$ [2]. For convenience, we define local transition formulas that map to the corresponding transition formula whenever such an edge exists.

[2]Note that, while CFA edges are defined over $L \times GCL \times L$, the edge label cmd and its corresponding formula representation $\psi(Var, Var')$ will be used where appropriate, i.e. for analogy to IC3-SMT, $\psi(Var, Var')$ is more appropriate, whereas for WP computations we have to use cmd.

Definition 4.3 (Local transition formula)**.** Given two locations $\ell_1, \ell_2 \in L$, we define the *transition formula* between ℓ_1 and ℓ_2 as

$$T_{\ell_1 \to \ell_2} = \bigvee \psi_i, \text{ for all } (\ell_1, \psi_i, \ell_2) \in G \tag{4.2}$$

$$\tag{4.3}$$

We define the global transition formula as follows.

Definition 4.4 (Global transition formula)**.** Given a CFA $\mathcal{A}$, we define the global transition relation as

$$T = \bigvee_{(\ell_1, t, \ell_2) \in G} T_{\ell_1 \to \ell_2}. \tag{4.4}$$

Example 4.3. Reconsider our example C program from Listing 1 on page 86. A corresponding CFA $\mathcal{A}$ according to Definition 2.23 is depicted in Figure 4.3. This CFA $\mathcal{A}$ induces local transition formulas like

$$T_{\ell_3 \to \ell_4} = (y' = b),$$
$$T_{\ell_5 \to \ell_6} = (a < 0), \; or$$
$$T_{\ell_3 \to \ell_5} = \mathit{false}.$$

The extraction of control-flow has some noticeable effect on two important state sets of IC3, namely the initial states I and the property states P:

For the verification of safety properties, we can add a dedicated error state, as suggested by [Cimatti and Griggio, 2012] with pc_E. This error state indicates that the property P of the system is violated and as such the pure reachability of pc_E suffices for the verification engine to return *UNSAFE* as a result. Using pc_E, the property formulation is as simple as $P = (pc \neq pc_E)$. In analogy our CFA $\mathcal{A}$, as defined in Section 2.4, consists of a dedicated error location ℓ_E. When extracting the control flow, P becomes the empty set of clauses $\bigwedge \emptyset = \mathit{true}$ for all regions with locations $\ell \neq \ell_E$. In other words, every state is a P-state, if it is not associated with ℓ_E.

A similar behaviour occurs for the initial states: In software model-checking, an initial state can only occur at $pc = 0$, but depending on the semantics of the programming language we encounter different assumptions about uninitialized

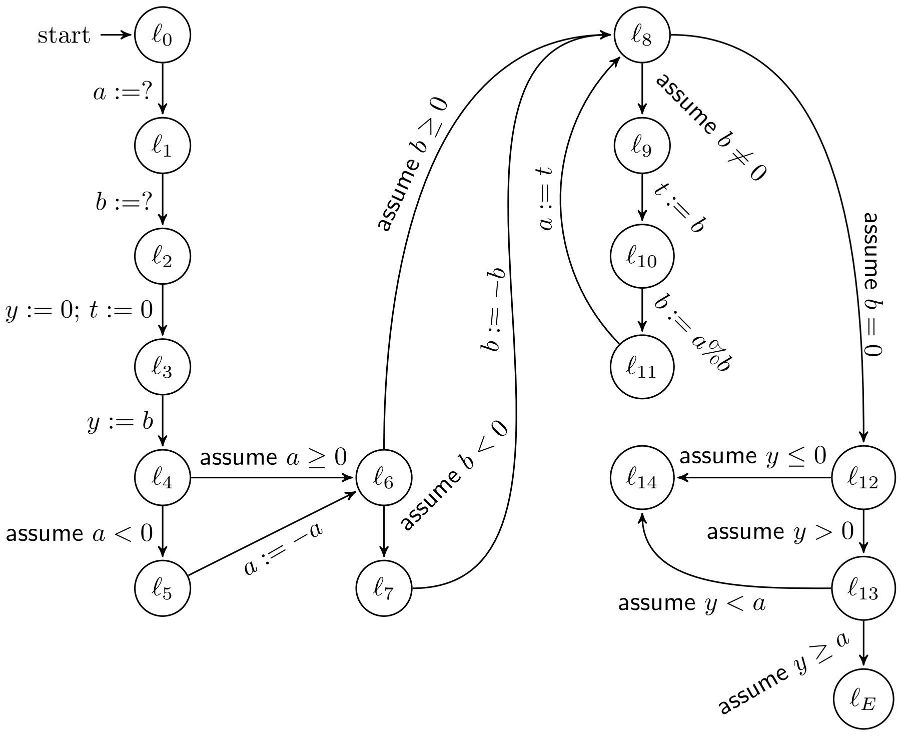


Figure 4.3: Control-flow automaton corresponding to Listing 1

variables[3]. For the remainder of this thesis, we assume the most general setting which is the one where an uninitialized variable can have any arbitrary value from their respective domain. We therefore formulate an IC3-SMT initial state representation as $I = (pc = 0)$. Just like for the error location, the extraction of control flow yields the symbolic representation $\bigwedge \emptyset = \mathit{true}$ for the initial location ℓ_0 of CFA $\mathcal{A}$ and *false* for all other locations $\ell \in L \setminus \{\ell_0\}$.

Establishing the partitioning of the state space, we also have to split the frame sequence $F_0, \ldots, F_k$ into a family of frame sequences $\{F_{(0,\ell_0)}, \ldots, F_{(k,\ell_0)}; \ldots; F_{(0,\ell_n)}, \ldots, F_{(k,\ell_n)}\}$, one for each CFA location $\ell_i \in L$. According to this partitioning, for a clause $d \in F_i$ it holds that $d|_{pc} \in F_{(i,\ell_j)}$ iff $d[pc] \models \mathit{binary}(j)$, i.e. the binary encoding of j is one possible model of the pc-variables of d. This means that d reasons at least about regions in program location ℓ_j, but it might also reason about regions in many more program locations. With such a splitting into a family of frame sequences, each entry $F_{(i,\ell)}$ characterizes the states in the variable's state space that are reachable in location ℓ in at most i steps. Note that we follow IC3 by not creating frames for the error location ℓ_E, just like IC3 does not include $\neg P$-states in any F_i according to (3.6).

As a consequence of *location-local* frames, we have to adapt the invariants (3.4) to (3.7) slightly: Given a CFA $\mathcal{A}$, for all $\ell \in L \setminus \{\ell_E\}$ it holds that:

$$\mathit{true} \Rightarrow F_{(0,\ell_0)} \tag{4.5}$$

$$F_{(i,\ell)} \Rightarrow F_{(i+1,\ell)}, \qquad \forall\, 0 \leq i < k \tag{4.6}$$

$$F_{(i,\ell)} \Rightarrow \mathit{true}, \qquad \forall\, 0 \leq i \leq k \tag{4.7}$$

$$F_{(i,\ell)} \wedge T_{\ell \to \ell'} \Rightarrow F'_{(i+1,\ell')}, \qquad \forall\, 0 \leq i < k. \tag{4.8}$$

As we can see, the invariants (4.5) and (4.7) become fairly simple for IC3CFA, due to the explicit handling of control-flow locations. For the construction of our algorithm, we will thus make sure that $F_{(0,\ell_0)}$ is initialized to *true* and all other $F_{(0,\ell)}$ are initialized to *false* to cover (4.5). Similarly, we will initialize every frame $F_{(i,\ell)}$ to *true* for $i \geq 1$ to satisfy (4.7).

One of the main differences in the construction of Tree-IC3 and IC3CFA is IC3CFA's ability to represent loops as such. Tree-IC3's approach of unrolling control-flow into an ART enables it to reason about simple, linear paths. However, the advantage of simplicity also brings the disadvantage of making it hard to detect loops in the program as such. In addition, as far as [Cimatti and

[3]For $pc = 0$ we have not executed any program instruction and thus every variable is uninitialized.

Griggio, [2012] goes, Tree-IC3 is not able to use already learnt lemmas about the state space of a control-flow location. For IC3CFA however, we abandon the monolithic unrolling and determine reachability information from single steps in the CFA's transition relation. Furthermore the explicit presence of loops, allows us to establish (4.6) in order to detect inductivity almost like IC3. More specifically, we can detect that IC3CFA has reached a fixpoint iff there exists some i, such that $F_{(i,\ell)} = F_{(i+1,\ell)}$ for every $\ell \in L \setminus \{\ell_E\}$.

Lemma 4.1. *If for some $0 \leq i < k$ it holds that $F_{(i,\ell)} = F_{(i+1,\ell)}$ for all $\ell \in L \setminus \{\ell_E\}$ then the conjunction $\bigwedge_{\ell \in L \setminus \{\ell_E\}} F_{(i,\ell)}$ forms an inductive strengthening.*

Proof. In order to prove Lemma 4.1, we show the equivalence of termination of IC3CFA and IC3-SMT. The termination of IC3-SMT has a trivial correspondence to termination in IC3, such that we omit this extra step here.

Assume that for some i in IC3-SMT, $F_i = F_{i+1}$. Let C be the set of all possible states of the input system. We can partition C into j disjoint sets C_j, such that $c \in C_j \Leftrightarrow (pc = j \wedge c \in C)$. The termination condition is symbolically encoded by:

$$
\begin{aligned}
& \exists F_i, F_{i+1} \subseteq C.\ F_i = F_{i+1} \\
\overset{(3.7)}{\iff}\ & \exists F_i, F_{i+1} \subseteq C.\ F_i = F_{i+1} \wedge (\forall f \in F_i.\ succ(f) \in F_{i+1}) \\
\iff\ & \forall pc_j \in PC\ \exists F_{(i,pc_j)}, F_{(i+1,pc_j)} \subseteq C_j. \\
& F_{(i,pc_j)} = F_{(i+1,pc_j)} \wedge \left(\forall f \in F_{(i,pc_j)}.\ succ(f) \in \bigcup_{pc_s \in succ(pc_j)} F_{(i+1,pc_s)} \right) \\
\iff\ & \forall \ell \in L \setminus \{\ell_E\}\ \exists F_{(i,\ell)}, F_{(i+1,\ell)} \subseteq C_j. \\
& F_{(i,\ell)} = F_{(i+1,\ell)} \wedge \left(\forall f \in F_{(i,\ell)}.\ succ(f) \in \bigcup_{\ell_s \in succ(\ell)} F_{(i+1,\ell_s)} \right) \\
\overset{(4.8)}{\iff}\ & \forall \ell \in L \setminus \{\ell_E\}\ \exists F_{(i,\ell)}, F_{(i+1,\ell)} \subseteq C_j. F_{(i,\ell)} = F_{(i+1,\ell)}.
\end{aligned}
$$

In order to show the equivalence of both termination methods, we start by considering the termination of IC3-SMT, where we terminate iff there exists two consecutive frames F_i and F_{i+1} that are identical. In order to prove the equivalence we also need to take into account the invariant (3.7), since both conditions are required for inductivity. Given the IC3-SMT formulation of inductivity, we partition the state space as explained above into a set of disjoint state sets, each indicating a different pc value. The partitioning of the state space obviously implies that frames, which can reason about many different pc values, have to be split, too. By definition, a pc location can only have a fixed set of successor locations, such that we don't have to consider $F_{(i+1,pc_h)}$ for all $pc_h \in PC$, but only for the subset $\{F_{(i+1,pc_s)} \mid pc_s \in succ(pc_j)\}$ of successor locations of pc_j. Using the analogy between the program counter value and the location $\ell \in L \setminus \{\ell_E\}$ of CFA $\mathcal{A}$ we can replace every occurence of pc_i by ℓ_i. In the last step, we can remove the successor condition, since it is ensured by invariant (4.8). □

Having considered in Lemma 4.1 what inductivity in the presence of a CFA means, the next step is to define a notion of *inductivity relative* to some state set. Just like for inductivity, the extraction of control-flow allows us to exploit the control structure of the program in order to consider only small parts of arbitrarily large and complex programs for a single step of the transition relation. We therefore refer to inductivity relative to some state set along a specific edge $T_{\ell_1 \to \ell_2}$ as *edge-relative inductivity.*

Definition 4.5 (Edge-relative inductivity)**.** Given a CFA $\mathcal{A}$ and locations $\ell_1, \ell_2 \in L$, a formula φ is edge-relative inductive to another formula ρ if

$$\rho \land \varphi \land T_{\ell_1 \to \ell_2} \implies \varphi' \tag{4.9}$$

is valid.

Note that edge-relative inductivity does also hold if $(\ell_1, \varphi, \ell_2) \notin G$ for any φ. In this case, $T_{\ell_1 \to \ell_2} = \mathit{false}$, which makes (4.9) hold trivially, i.e. if $\mathcal{A}$ is in a state satisfying φ and cannot leave it via the considered edge, it remains in a state satisfying φ. However, as already hinted at earlier and also mentioned in [Cimatti and Griggio, 2012], we have to modify relative inductivity somewhat. In order to do so, we only consider the case that is at the heart of the IC3 algorithm: Inductivity of a negated formula $\neg\varphi$ relative to a non-negated formula ρ. In analogy, we will reason about inductivity of a negated region r_1 relative to a non-negated region r_2.

Lemma 4.2 (Relative inductive regions)*.* *Assuming two regions $r_1 = (\ell_1, \varphi_1)$, $\neg r_2 = \neg(\ell_2, \varphi_2)$, the edge-relative inductivity of $\neg r_2$ to r_1 equals*

$$\varphi_1 \land T_{\ell_1 \to \ell_2} \Rightarrow \neg\varphi_2' \qquad , \textit{if } \ell_2 \neq \ell_1 \tag{4.10}$$

$$\varphi_1 \land \neg\varphi_2 \land T_{\ell_1 \to \ell_2} \Rightarrow \neg\varphi_2' \qquad , \textit{if } \ell_2 = \ell_1 \tag{4.11}$$

Proof. To prove Lemma 4.2, we first define an extended transition formula

$$\hat{T}_{\ell_1 \to \ell_2} := pc = \ell_1 \wedge T_{\ell_1 \to \ell_2} \wedge pc' = \ell_2$$

and replace all occurrences of $T_{\ell_1 \to \ell_2}$ in Lemma 4.2 with $\hat{T}_{\ell_1 \to \ell_2}$. This is a valid substitution, since it preserves validity of Lemma 4.2 and only adds additional, explicit assignments of pc and pc'. Given two regions $r_1 = (\ell_1, \varphi_1)$ and $r_2 = (\ell_2, \varphi_2)$ with corresponding formulas ψ_1 and ψ_2, we have:

$$\psi_1 \equiv (pc = \ell_1 \wedge \varphi_1) \qquad\qquad \neg\psi_2 \equiv \neg(pc = \ell_2 \wedge \varphi_2).$$

Definition 4.5 yields:

$$\begin{aligned} &(pc = \ell_1 \wedge \varphi_1) \wedge \neg(pc = \ell_2 \wedge \varphi_2) \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow \neg(pc' = \ell_2 \wedge \varphi_2') \\ \equiv\ &(pc = \ell_1 \wedge \varphi_1) \wedge (pc \neq \ell_2 \vee \neg\varphi_2) \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow (pc' \neq \ell_2 \vee \neg\varphi_2'). \end{aligned}$$

If $\ell_1 \neq \ell_2$, this is equisatisfiable to

$$\begin{aligned} &(\mathit{true} \wedge \varphi_1) \wedge (\mathit{true} \vee \neg\varphi_2) \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow (\mathit{false} \vee \neg\varphi_2') \\ \equiv\ &\varphi_1 \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow \neg\varphi_2' \\ \equiv\ &\varphi_1 \wedge T_{\ell_1 \to \ell_2} \Rightarrow \neg\varphi_2'. \end{aligned}$$

Otherwise, we obtain

$$\begin{aligned} &(\mathit{true} \wedge \varphi_1) \wedge (\mathit{false} \vee \neg\varphi_2) \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow (\mathit{false} \vee \neg\varphi_2') \\ \equiv\ &\varphi_1 \wedge \neg\varphi_2 \wedge \hat{T}_{\ell_1 \to \ell_2} \Rightarrow \neg\varphi_2' \\ \equiv\ &\varphi_1 \wedge \neg\varphi_2 \wedge T_{\ell_1 \to \ell_2} \Rightarrow \neg\varphi_2'. \end{aligned}$$

□

Applying Lemma 4.2 to edge-reglative inductivity for frames yields the refined definition of edge-relative inductivity formalized in Definition 4.6.

Definition 4.6 (Edge-relative inductivity)**.** Given a CFA $\mathcal{A}$ and two locations $\ell_1, \ell_2 \in L$, a clause $\neg c$ is inductive relative to frame $F_{(i,\ell_1)}$ if

$$F_{(i,\ell_1)} \wedge T_{\ell_1 \to \ell_2} \Rightarrow \neg c' \qquad , \text{ if } \ell_2 \neq \ell_1 \tag{4.12}$$

$$F_{(i,\ell_1)} \wedge \neg c \wedge T_{\ell_1 \to \ell_2} \Rightarrow \neg c' \qquad , \text{ if } \ell_2 = \ell_1. \tag{4.13}$$

Example 4.4. Reconsider the CFA $\mathcal{A}$ from Figure 4.3 on page 97. We will use this example CFA to illustrate how relative inductivity works for IC3CFA. Consider the cube $c = (y \geq a)$, the transition from ℓ_{12} to ℓ_{13} and frame $F_{(5,\ell_{12})} = \neg (y > 0)$. Then $\neg c$ is inductive relative to $F_{(5,\ell_{12})}$ along $e = (\ell_{12}, \textsf{assume } y > 0, \ell_{13}) \in G$ iff

$$\mathit{unsat}\,(\neg (y > 0) \wedge (y > 0) \wedge (y \geq a))?$$

This query is unsatisfiable due to the conflict between $\neg (y > 0)$ and $(y > 0)$ and therefore c is inductive relative to F_5 along e. Note that neither y nor a appear primed in c', since $T_{\ell_{12} \to \ell_{13}}$ does not assign these variables.

Having presented the previous liftings of IC3's main concepts of the search phase to the presence of a CFA $\mathcal{A}$ for software verification, we are still missing one important link: Given a clause $\neg c$ at some level i that is not inductive relative to F_{i-1}, IC3 needs a predecessor s to continue the search by checking inductivity of s relative to F_{i-2}. Determining a predecessor in IC3 is fairly simple and comes at almost no cost, since it takes the model of the SAT-solver which gives a satisfying assignment for a $\neg c$-state s_{pre} that has a primed c-successor s_{post}, which is the reason for the violation of inductivity relative to F_{i-1}. Projecting the satisfying model on non-primed variables only yields a cube that is an underapproximation of the predecessors of c. For IC3 picking such s is a very convenient choice, since s will be generalized afterwards, in order to speed up convergence, and with finitely many iterations IC3 can check all predecessor states in the finite Boolean transition system.

But, as already mentioned earlier in the context of Tree-IC3 [Cimatti and Griggio, 2012], picking a solver model is not suitable for IC3-style software verification, since there may be *infinitely many predecessor states.* A possible solution is to use theory-aware generalizations. We try to avoid this in order to allow a modular design of the underlying structure, i.e. exchange theories as necessary depending on the input model, and to avoid implementing a generalization procedure for every theory (combination). The alternative, which is also used by [Cimatti and Griggio, 2012], is to use weakest preconditions (WP) in order to compute an *exact* preimage of c. While WP allows us to remove repeated computations of underapproximations, it does come at a high price: While on average the costs for WP computations are mostly linear, for some instances, also in practical applications, it exhibits its worst-case exponential behaviour. A somewhat more involved method is to lazily compute underapproximations using model-based projection [Bjørner and Janota, 2015; Komuravelli, Bjørner, et al., 2015; Komuravelli, Gurfinkel, et al., 2014], but this again requires dedicated theory-aware projections for each theory. We therefore omit model-based projection in the remainder and use the mentioned WP construction. Later in this chapter we will present and evaluate a method that can help to reduce the impact of the WP computation by efficiently caching parts of the WP construction that lead to the exponential blow-up.

As a last step, we have to modify proof obligations: In order to correctly create an inductivity query from a proof obligation, we need to store not only the level i and the cube c, but also the CFA location ℓ to associate it with a frame $F_{(i,\ell)}$ to check inductivity of c relative to $F_{(i,\ell)}$.

4.2.2 The IC3CFA algorithm

With the presented adaptations of all important aspects of IC3's search phase to CFAs, we will now present the basic IC3CFA algorithm. We will focus the remainder of this section on the search phase of IC3CFA, present the IC3CFA algorithm, and prove its correctness. In Sections 4.3 and 4.4, we will have a closer look at how we can apply generalization and propagation to IC3CFA and which peculiarities we have to consider in that case.

In order to stress the similarities between IC3 and IC3CFA and to simplify the presentation, we will follow the presentation of [Bradley, 2011] and highlight our modifications if they are more than just notational changes, e.g. we don't highlight all changes from F_i to $F_{(i,\ell)}$, but we will mark how ℓ is determined. Considering the proof of correctness of the algorithm, we will use annotated code with pre-/postconditions like [Bradley, 2011] which allows us to construct the proof more easily. We begin with the top-level function PROVE which includes the top-level loop over iterations $1, \ldots, k$.

Except for a few small modifications, the PROVE function of Algorithm 12 works more or less like the original of [Bradley, 2011]. Apart from the modified termination that was subject of Lemma 4.1, we only have to adapt the initial checks for zero- and one-step counterexamples that cannot be detected in the main loop of IC3. Zero-step counterexamples in IC3 are considered those where an initial state already violates the property. For our CFA-based approach this is exactly the case when the initial location ℓ_0 is also the error location ℓ_E. For one-step counterexamples, i.e. $I \wedge T \wedge \neg P'$ in [Bradley, 2011], we have to detect whether a) there exists a transition from ℓ_0 and ℓ_E and b) whether this transition is actually feasible. Since the initial region as well as the error region is unconstrained in the data region, we just have to check whether there

Algorithm 12 Outer loop [Lange et al., 2015]

```
Ensure: return $false$ iff $\ell_E$ is reachable
  function PROVE($\mathcal{A}$)
  Input: CFA $\mathcal{A}$
  Output: $true \iff$ SAFE
  if $\ell_0 = \ell_E$ or $((\ell_0, \psi, \ell_E) \in G$ and $sat(\psi)$?) then
      return $false$
  initialize frames
  for $k = 1$ to $\ldots$ do
      if not STRENGTHEN($k$) then
          return $false$
      PROPAGATECLAUSES($k$)
      if clauses($F_{(i,\ell)}$)=clauses($F_{(i+1,\ell)}$) for some $0 \leq i < k$, all $\ell \in L$ then
          return true
```

exists any assignment that satisfies the transition formula ψ. If these checks succeed, we proceed as in [Bradley, 2011] by initializing frames at level $i = 0$ and $i = 1$. We do so as described above by initializing $F_{(0,\ell)}$ to *false* for all $\ell \in L$ except for $F_{(0,\ell_0)}$ which we set to *true*. This corresponds to $F_0 = I$. All frames $F_{(1,\ell)}$ are being initialized to *true* for all $\ell \in L$ which conforms to $F_1 = P$. In the remainder we iterate over the "frontier" k, starting with $k = 1$. After entering this loop, we try to strengthen the frames at level k. If the function STRENGTHEN fails, we have found a counterexample trace and we can return *false* which corresponds to UNSAFE. Note that in Algorithm 12 we do not consider counterexamples, but we will later present a simple way to determine counterexample traces that can be enabled with only minor modifications. On the other hand a successful strengthening means that no CTI is reachable any more and we can propagate clauses. Before advancing to the next iteration $k+1$ we check whether we have constructed a sufficient strengthening according to Lemma 4.1. If this is the case, then we can return *true* which corresponds to SAFE; otherwise we have to increment k.

Whether a strengthening for some k exists is determined by the function STRENGTHEN presented in Algorithm 13. Just like in the design of the IC3 algorithm in [Bradley, 2011], the function PROVE loops as long as there exist CTI states. For IC3 this means that as long as there exists a satisfying model for the formula $F_k \wedge T \wedge \neg P'$ it will extract the predecessor s from the solver model and explore the path leading to s. Due to the fact that we use WP to

Algorithm 13 Strengthening [Lange et al., 2015]

```
Require: (a) $k \geq 1$
Require: (b) $\forall 0 \leq i < k, \ell \in L, F_{(i,\ell)} \Rightarrow F_{(i+1,\ell)}$
Require: (c) $\forall 0 \leq i < k, \ell, \ell' \in L$, s.t. $(\ell, \psi, \ell') \in G$, $F_{(i,\ell)} \wedge T_{\ell \to \ell'} \Rightarrow F'_{(i+1,\ell')}$
Ensure: $\forall 0 \leq i < k, \ell \in L, F_{(i,\ell)} \Rightarrow F_{(i+1,\ell)}$
Ensure: if ret. value then $\forall 0 \leq i < k, \ell, \ell' \in L$, s.t.$(\ell, \psi, \ell') \in G$, $F_{(i,\ell)} \wedge T_{\ell \to \ell'} \Rightarrow$
    $F'_{(i+1,\ell')}$
Ensure: if $\neg$ret. value, there exists a counterexample path
  function STRENGTHEN($k$)
  Input: Max level $k$
  Output: $true \iff$ strengthening at $k$ exists
  while $\exists \ell$, s.t. $sat\left(F_{(k,\ell)} \wedge T_{\ell \to \ell_E}\right)$? do
      @assert (b),(c)
      $\varphi :=$ predecessor data region
      if not BACKWARDBLOCK($k$, ($\ell$, $\varphi$)) then
          return $false$
      @assert $\varphi \not\models F_{(k,\ell)}$
  return true
```

extract an exact preimage of $\neg P$ we only have to consider a single data region per location, such that this loop degrades to looping as long as there exists a location ℓ for which a state in $F_{(k,\ell)}$ can take a transition to the error location ℓ_E. If such ℓ with $e = (\ell, \mathsf{cmd}, \ell_E) \in G$ exists, we compute the CTI data region φ as $wep\,(\mathsf{cmd}, true)$ and call BACKWARDBLOCK with k and the region (ℓ, φ). If this returns *false*, it means that the backward search hit the initial location ℓ_0, i.e. we found a feasible counterexample path, such that we update *false* upwards the call stack to PROVE. On the other hand, if BACKWARDBLOCK succeeds to block the path up to φ, we can continue to the next iteration of the loop and see whether there exists another CTI region to be considered. If the loop terminates and there are no more CTI regions available we have constructed a strengthening for level k and can return to PROVE. Note that in the search for CTI regions, according to Definition 4.4, we don't have to specifically handle locations ℓ which do not have a transition to ℓ_E, since for those locations $T_{\ell \to \ell_E} = false$ by definition, such that $F_{(k,\ell)} \wedge T_{\ell \to \ell_E}$ is not satisfiable.

While Algorithms 12 and 13 follow [Bradley, 2011], we found their presentation of the backward search for paths leading to CTI (ℓ, φ) to be unneces-

Algorithm 14 Inner loop [Lange et al., 2015]

```
Require: (b),(c)
Require: $sat\left(F_{(\hat{\imath},\hat{\ell}')} \wedge \hat{\varphi} \wedge T_{\hat{\ell}' \to \ell_E}\right)$?
Ensure: if ret. value, then $\neg\hat{\varphi}$ is inductive relative to $F_{(\hat{\imath}-1,\ell)}$, $\forall \ell$, $(\ell, t, \hat{\ell}') \in G$
Ensure: if ret. value, then (b),(c)
Ensure: if ¬ret. value, there exists a feasible path $\ell_0 \rightsquigarrow \hat{\ell}'$
  function BACKWARDBLOCK($\hat{\imath}$, ($\hat{\ell}'$, $\hat{\varphi}$))
  Input: CTI level $\hat{\imath}$, CTI location $\hat{\ell}'$, CTI data region $\hat{\varphi}$
  Output: $true \iff$ CTI is unreachable
  $Q.add(\hat{\imath}, \hat{\ell}', \hat{\varphi})$
  while $|Q| > 0$ do
      @assert $\forall (i, \ell', \varphi) \in Q.\ 0 \leq i \leq k$
      @assert $\forall (i, \ell', \varphi) \in Q.\ \exists$ path $(\ell', \varphi) \rightsquigarrow (\ell_E, true)$
      $(i, \ell', \varphi) = Q.pop$
      if $i = 0$ then
          return $false$
      else
          @assert $(\ell', \neg\varphi)$ is inductive rel. to $F_{(j,\ell)}$, $\forall 0 \leq j < i$, $\ell \in L \setminus \{\ell_E\}$
          for each $\ell$, s.t. $(\ell, t, \ell') \in G$ do
              if $\ell = \ell'$ and $sat\left(F_{(i-1,\ell)} \wedge \neg\varphi \wedge T_{\ell \to \ell'} \wedge \varphi'\right)$? OR $\ell \neq \ell'$ and $sat\left(F_{(i-1,\ell)} \wedge T_{\ell \to \ell'} \wedge \varphi'\right)$? then
                  generate predecessor $\psi$ of $\varphi$
                  @assert $\forall (i, \ell', \varphi) \in Q, \psi \neq \varphi$
                  add $(i-1, \ell, \psi)$ and $(i, \ell', \varphi)$ to $Q$
              else
                  GENERALIZECLAUSE($\varphi$)
                  block $\varphi$ in frames $F_{(j,\ell')}$ for $0 \leq j \leq i$
      return $true$
```

sarily complex and instead we chose the more appealing presentation of [Eén, Mishchenko, et al., 2011] for the function BACKWARDBLOCK in Algorithm 14. Calling BACKWARDBLOCK with the CTI's level $\hat{\imath}$, location $\hat{\ell}$ and data region $\hat{\varphi}$, we start by pushing this triplet as the initial proof obligation into the obligation queue Q. Like [Eén, Mishchenko, et al., 2011] the remainder of BACKWARDBLOCK consists of a loop that lasts as long as the obligation queue is non-empty. Having entered the loop, we take the first proof obligation out of

Q. If this obligation has level 0 we can return *false* without further investigation since the obligation *must* intersect the initial states; we will later see why. Assume that the obligation has level $i \neq 0$, then we have to check inductivity, like IC3 [Bradley, 2011] and PDR [Eén, Mishchenko, et al., 2011]. But due to our explicit control-flow structure, we have to add another loop to Algorithm 14 which is not present in [Bradley, 2011; Eén, Mishchenko, et al., 2011]: IC3 checks whether $\neg\varphi$ is inductive relative to F_{i-1}, but since we split F_{i-1} into $\{F_{(i-1,\ell_1)}, \ldots, F_{(i-1,\ell_n)}\}$ we have to check inductivity of φ relative to $F_{(i-1,\ell)}$ for every ℓ of incoming edge $(\ell, \varphi, \ell') \in G$.

However, the indegree of CFAs derived from real programs is commonly very small, often less or equal three, such that the additional loop does not add much overhead, given an efficient data structure for storing predecessor locations. Since we know from Lemma 4.2 that we can use different inductivity formulations depending on pre- and post-locations of the edge that we consider, we find a three-fold branching inside the loop in Algorithm 14: Given that pre- and post-location are identical and φ is not inductive relative to $F_{(i-1,\ell)}$, we determine ψ, the predecessor of φ, using WP and add the new proof obligation $(i-1, \ell, \psi)$ to the obligation queue Q and put the current proof obligation back into Q. For the case where $\ell \neq \ell'$, we do the same thing, but consider the simpler relative inductivity that resembles reachability of φ from $F_{(i-1,\ell)}$. If both conditions fail, i.e. if φ is inductive relative to $F_{(i-1,\ell)}$, we block φ in all $F_{(j,\ell')}$ for $0 \leq j \leq i$.

Example 4.5. Let us illustrate the way IC3CFA proves a property by an example run, using the GCD computation of Listing 1.

We start IC3CFA by checking the CFA $\mathcal{A}$ (see Figure 4.3 on page 97) for 0-step counterexamples, but, since $\mathcal{A}$ has disjunct *initial* and *error* states, none exists. For 1-step counterexamples, we have a similar situation, since no direct edge between initial and error location exists. We therefore proceed to the initialization of frames. Here we set the frame at level 0 for the initial location ℓ_0 to *true* and for all other locations to *false*, i.e. $F_{(0,\ell_0)} = \mathit{true}$ and $F_{(0,\ell)} = \mathit{false}, \forall \ell \in L_P \backslash \{\ell_0\}$. Note that for convenience we define an auxiliary location set $L_P = L \backslash \{\ell_E\}$ which contains all locations, except the error location. Furthermore, we initialize all frames at level 1 to *true*, i.e. $F_{(1,\ell)} = \mathit{true}, \forall \ell \in L_P$. Afterwards, we enter the strengthening procedure (see Algorithm 13) with $k = 1$. Since location ℓ_{13} is a predecessor to ℓ_E and the query $\mathit{sat}\left(F_{(1,\ell_{13})} \wedge T_{\ell_{13} \to \ell_E}\right)$?, i.e. $\mathit{sat}\,(\mathit{true} \wedge y \geq a)$?, is satisfied, we compute the CTI data region

$\varphi = (y \geq a)$ using the WEP according to Definition 2.21. We enter BACKWARDBLOCK with the parameters $(1, (\ell_{13}, y \geq a))$ and add the corresponding proof obligation $(1, (\ell_{13}, y \geq a))$ to the queue Q. After entering the **while** loop, we immediately pop this obligation back off the queue. Since i is not equal to 0, we check for each predecessor location, in our case just ℓ_{12}, whether φ is inductive relative to $F_{(0,\ell_{12})}$ along edge $e = (\ell_{12}, \textsf{assume } y > 0, \ell_{13})$, i.e. whether $sat\left(F_{(0,\ell_{12})} \wedge T_{\ell_{12}\to\ell_{13}} \wedge y' \geq a'\right)$? is satisfied or in our case $sat\left(false \wedge y > 0 \wedge y \geq a\right)$?, which is obviously unsatisfiable. We therefore block $\varphi = (y \geq a)$ in $F_{(1,\ell_{13})}$, i.e. $F_{(1,\ell_{13})} \leftarrow \neg(y \geq a)$. Note that we effectively block φ in $F_{(i,\ell)}$ due to the use of delta encoding, as presented in the context of IC3 on page 69. Since Q is empty, we return from BACKWARDBLOCK to STRENGTHEN and find no remaining CTI region, since for location ℓ_{13}, the query $sat\left(F_{(1,\ell_{13})} \wedge T_{\ell_{13}\to\ell_E}\right)$? is unsatisfiable and no other predecessor location to ℓ_E exists. We have thus found a strengthening for level $k = 1$ and can check whether for some $0 \leq i \leq k$ and all $\ell \in L_P$ the clauses of $F_{(i,\ell)}$ and $F_{(i+1,\ell)}$ are identical. In our case, $i = 0$ is the only applicable level to check and the equivalence check already fails for ℓ_1, where $F_{(0,\ell_2)} = false \neq true = F_{(1,\ell_2)}$. We therefore proceed to the next iteration $k = 2$. There we again, find the CTI data region $\varphi = (y \geq a)$ and enter BACKWARDBLOCK with $2, (\ell_{13}, y \geq a)$, which is not inductive relative to $F_{(1,\ell_{12})}$ along e, in this case. We thus determine the predecessor data region $\psi = (y \geq a) \wedge (y > 0)$ via WEP, add the new obligation $o_2 = (1, (\ell_{12}, \psi))$ to the queue and also put the original obligation $o_1 = (2, (\ell_{13}, \varphi))$ back into the queue. In the next iteration of the **while**-loop we will pop o_2, since it has the lowest index. However, o_2 is not inductive relative to $F_{(0,\ell_8)}$ along $e_2 = (\ell_8, \textsf{assume } b = 0, \ell_{12})$ because of $F_{(0,\ell_8)}$ being *false*. We therefore block ψ in $F_{(1,\ell_{12})}$ and continue with o_1, which has become inductive by blocking ψ in $F_{(1,\ell_{12})}$ and can therefore be blocked in $F_{(2,\ell_{13})}$, such that no CTI exists, any more. In the termination check, we now have two indices to check: For $i = 0$, it fails, due to $F_{(0,\ell_2)} = false \neq true = F_{(0,\ell_2)}$, while for $i = 1$ it fails because of $F_{(1,\ell_{12})} = \neg\psi \neq true = F_{(2,\ell_{12})}$. We omit further iterations here. Our implementation of the IC3CFA algorithm according to Algorithms 12 to 14 is able to prove the correctness of the C program as shown in Listing 1 (without any minimizations and optimizations on the C code) after 101 iterations, which takes a total verification time of 177.8 seconds, of which 151.6 seconds are spent for 15602 solver calls.

4.2.3 Correctness

Having established the IC3CFA algorithm composed of Algorithms 12 to 14 in a similar fashion as the original IC3 algorithm presented in [Bradley, 2011] with some borrowing of [Eén, Mishchenko, et al., 2011], we still have to show the correctness of our algorithm. In order to simplify the proof, we have already annotated Algorithms 12 to 14 with a set of pre- and postconditions and assertions.

In the remainder we will show the correctness of our IC3CFA algorithm by proving the correctness of the postconditions given the respective preconditions of Algorithms 12 to 14. Since the correctness of a function relies on the correctness of each function it calls, we build our proof in a bottom-up fashion, starting with the correctness of Algorithm 14, followed by Algorithm 13 and finally the correctness of the main function PROVE in Algorithm 12.

Lemma 4.3. *Function BACKWARDBLOCK of Algorithm 14 returns true iff* $\neg\widehat{\varphi}$ *is inductive relative to* $F_{(\widehat{i}-1,\ell)}$ *for all locations* $\ell \in pred(\widehat{\ell'})$*.*

Proof. Function BACKWARDBLOCK starts the **while** loop by examining the proof obligation in the queue that has the lowest frame index i. If $i = 0$, then $\hat{\ell}'$ must be ℓ_0, because $F_{(0,\ell_0)}$ is the only region with index $i = 0$ to which any other region may be inductive relative to, by construction. This way, there must exist a feasible path from ℓ_0 to $\hat{\ell}'$.

If there exists no such feasible path from ℓ_0 to $\hat{\ell}'$ given $F_{(0,\ell)}, ..., F_{(k,\ell)}$ for all $\ell \in L \setminus \{\ell_E\}$, then every path of length j ending in $\hat{\ell}'$ starts in a location ℓ_s, s.t. the region $(\ell_s, \neg\varphi)$ is inductive relative to $F_{(k-j-1,\ell^x)}$ for all ℓ^x, s.t. $(\ell^x, \rho, \ell_s) \in G$. This means that every proof obligation added to the obligation queue Q is ultimately inductive relative to its predecessors and thus also $\neg\hat{\varphi}$ is inductive relative to $F_{(\hat{i}-1,\ell^x)}$. □

With the correctness of BACKWARDBLOCK, we can continue with Algorithm 13 which calls BACKWARDBLOCK. We can guarantee termination of Algorithm 13 for the same reason as for BACKWARDBLOCK: Given the exact predecessor computation using WP, we only have to search for at most n_E CTIs, where $n_E = |pred(\ell_E)|$.

Lemma 4.4. *Function STRENGTHEN of Algorithm 13 terminates with true iff there exists an inductive strengthening for* $F_{(k,\ell)}$ *in all* $\ell \in L \setminus \{\ell_E\}$*.*

Proof. Assume a call of function STRENGTHEN returns *false*, then there must have been a call to BACKWARDBLOCK with some (k, ℓ, φ), such that BACKWARDBLOCK returned *false*. From Lemma 4.3 we know that in this case, there

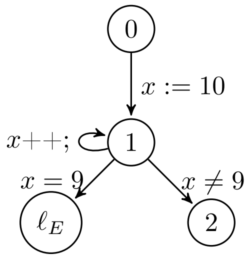


Figure 4.4: Example for non-termination of Algorithm 12 [Lange et al., 2015]

exists a feasible path of length k from ℓ_0 to ℓ that ends up in data region φ. Because φ is a predecessor of ℓ_E and φ is a precondition under $T_{\ell \to \ell_E}$, there exists a counterexample path of length $k+1$. Otherwise every call of BACKWARDBLOCK returned *true*, which means that every predecessor location (and data region) of ℓ_E is unreachable in the current frame sequence. Thus every predecessor of ℓ_E was excluded from their frames at level k which yields an inductive strengthening for $F_{(k,\ell_x)}$ for all $\ell_x \in L \setminus \{\ell_E\}$. □

Given the correctness of Algorithms 13 and 14 as stated in Lemmas 4.3 and 4.4 it would be desirable to show the same for Algorithm 12. However, we cannot guarantee correctness, since the search phase might diverge over some domains. We will disprove termination by a simple counterexample.

Lemma 4.5. *Algorithm 12 does not terminate under all possible inputs.*

Proof by Contradiction. Assume that Algorithm 12 would terminate under all possible inputs. Then it should also terminate given the CFA $\mathcal{A}$ from Figure 4.4 with unbounded integers in the LIA theory. Given $\mathcal{A}$, IC3CFA will compute the CTI region $(1, x = 9)$ in every iteration and loop in location 1, computing ever smaller valuations for x in every step of the search phase. For theories such as BV, that are finite, we would eventually terminate, but for unbounded theories such as LIA, IC3CFA will enumerate predecessors down to $x = (9 - k)$ for all bounds k, i.e. it will enumerate all paths leading to $(1, x = 9)$ for any length k of which there exist infinitely many. Therefore IC3CFA will not terminate on $\mathcal{A}$. $\lightning$

Since we have shown that IC3CFA may not terminate under all inputs, we can still prove partial correctness of Algorithm 12, i.e. answering the question whether, if it terminates, PROVE outputs the correct result.

Lemma 4.6 [Lange et al., 2015]. *In case function PROVE of Algorithm 12 terminates, it returns true iff there exists an inductive strengthening F for P, s.t. $F \wedge P$ is inductive.*

Proof. Assume PROVE terminates and returns *true*, then every call to function STRENGTHEN for every $j < k$ must have returned *true* and there must exist a frame with index $i < k$, s.t. $F_{(i,\ell)} = F_{(i+1,\ell)}$ for all $\ell \in L \setminus \{\ell_E\}$, i.e. alls frames $F_{(i,\ell)}$ are inductive, because $F_{(i,\ell)} \wedge T_{\ell \to \ell'} \implies F_{(i,\ell')}$ for all $\ell, \ell' \in L \setminus \{\ell_E\}$. Therefore there cannot exist a counterexample path of length k (more precise of length i) or less and there cannot exist one of length greater than k, because $F_{(i,\ell)}$ is inductive for all $\ell \in L \setminus \{\ell_E\}$.

Now assume that PROVE returns *false*: Then there must exist a k, s.t. for no $i < k$, $F_{(i,\ell)}$ is inductive for all $\ell \in L \setminus \{\ell_E\}$ and STRENGTHEN for k returns *false*, i.e. there exists a path of length k from the initial to the error state. □

As mentioned earlier, one of the, if not *the*, biggest problem in lifting IC3 to software model-checking is the question how predecessors are computed. Accordingly our choice of using weakest preconditions has noticeable impact in almost every aspect of our IC3CFA algorithm. While we already saw that we can simplify the **while**-loop of STRENGTHEN in Algorithm 13 to consider every predecessor location only once, we can reduce the CTI computation even further. If we take a closer look at Algorithm 13, we observe that the predecessor data region φ of every $\ell \in \mathit{pred}(\ell_E)$ is determined as $\varphi := \mathit{wp}\,(\mathsf{cmd}, \mathit{true})$ for $(\ell, \mathsf{cmd}, \ell_E)$. Since by definition every data region of ℓ_E is a $\neg P$-state, the respective predecessor data region φ does not depend on any information learnt by IC3CFA. While this dependency is not explicit in IC3, the decisions made in the SAT-solver may lead to different models in different iterations and therefore lead to different predecessor states depending on information learnt in the SAT-solver. But since in IC3CFA the set of CTI regions $\{(\ell, \varphi) \mid \ell \in \mathit{pred}(\ell_E), \varphi = \mathit{wp}\,(\mathsf{cmd}, \mathit{true})\}$ can be statically determined up front, we can compute this set once, before starting the main **for**-loop in PROVE of Algorithm 12 and in STRENGTHEN of Algorithm 13 call BACKWARDBLOCK with the items of the set of CTI regions $\{(\ell, \varphi) \mid \ell \in \mathit{pred}(\ell_E), \varphi = \mathit{wp}\,(\mathsf{cmd}, \mathit{true})\}$ and varying k.

4.2.4 Discussion

Using this approach we can statically determine all CTI regions upfront, rather than recomputing the same region in every iteration. However, we can go even further: In practice the WP computation leads to a situation where IC3CFA will compute the same set of obligations in every iteration, which becomes very obvious for IC3CFA without generalization as presented in Algorithms 12 to 14. How this situation changes in the presence of generalization will be considered in Section 4.3. Assume a CFA $\mathcal{A}$ and some iteration k where a path of length k exists in $\mathcal{A}$, but there exists no counterexample path of length at most k. Then the set of all proof obligations created in the k^{th} iteration

$$O_{\overrightarrow{k}} = O_k \uplus O_{\overrightarrow{k-1}} = \{(\ell, \varphi) \mid (i, \ell, \varphi) \in Q \text{ at some point in iteration } k\}$$

is the disjoint union of the set of newly explored obligations of the k^{th} iteration

$$O_k = \{(\ell, \varphi) \mid (0, \ell, \varphi) \in Q \text{ at some point in iteration } k\}$$

and the set of obligations created in the $k - 1^{st}$ iteration

$$O_{\overrightarrow{k-1}} = \{(\ell, \varphi) \mid (i, \ell, \varphi) \in Q \text{ at some point in iteration } k-1\}.$$

The reason for this observation in our deterministic IC3CFA algorithm is easy to understand: The search phase of IC3CFA will explore all maximal, reachable path fragments of length up to k that can reach a CTI state. In other words, it implicitly creates a tree of regions that are backwards reachable from CTI regions and implicitly encodes the nodes of this tree as proof obligations. After proceeding to iteration $k + 1$, IC3CFA may be able to explore direct predecessors of all those regions that are leafs in the implicit backward reachability tree. However, in order to explore these states, we force IC3CFA to explore the whole backward reachability tree of iteration k.

After having observed this connection between obligations of different iterations, we aim to reuse obligations of iteration k in iteration $k + 1$, just like Tree-IC3 continues to unroll the ART rather than deleting and re-unrolling it after every iteration [Mertens, 2016]. To do so we only have to modify the internal behaviour of our obligation queue Q: Rather than taking obligation $(i, (\ell, c))$ out of the queue in the standard way, we propose to keep $(i, (\ell, c))$ in Q and just mask the obligation such that it will not be reconsidered in the next iteration. In case we attempt to put a previously popped obligation $(i, (\ell, c))$ back into the queue, i.e. when IC3CFA detected a non-inductive region and needs to create a predecessor obligation, we remove the masking of $(i, (\ell, c))$ in Q. If Q only contains masked obligations, we consider Q empty. After checking inductivity on the frame sequences we can then recover the obligations computed during this iteration by removing all maskings from obligations in Q. But in order to reuse these obligations in the next iteration, we first have to modify them: Due to the backward search approach of IC3 that starts at the iteration count k, new obligations in $k+1$ are created one level after the ones created in iteration k. Therefore we have to increment the level entry i of every obligation $(i,(\ell,c))$ in Q. The resulting obligation queue allows us to start our backward search at those regions that we were not able to explore any further due to the iteration's bounded search length k.

Another aspect that arises with the use of exact preimages due to WP predecessor computation can be observed when a region has been shown to be inductive relative to all predecessor locations' frames. In the case that a proof obligation $(i,(\ell,c))$ succeeds, the cube c is blocked in $F_{(i,\ell)}$ and we can distinguish two possible situations afterwards: Either there exists another obligation $(i,(\ell',c'))$ at level i or $(i,(\ell,c))$ was the last obligation at level i and the next obligation appears at level $i+1$. In the first case there exist other locations ℓ' that are predecessor to some $\bar{\ell}$ that appears in a proof obligation at level $i+1$ and we have to check whether these obligations $(i,(\ell',c'))$ are inductive relative to their respective predecessor frames. But in the latter case, that no other obligation at level i exists, we will return back to obligation $(i+1,(\bar{\ell},\bar{c}))$ after exploring all paths leading up to $\bar{\ell}$ of length up to i. While for standard IC3 we might encounter new predecessor states, the use of WP guarantees that no predecessor region can reach the region $(\bar{\ell},\bar{c})$ which implies that $(\bar{\ell},\bar{c})$ is inductive and the proof obligation $(i+1,(\bar{\ell},\bar{c}))$ succeeds. As a result of this observation we can immediately block each obligation that has already been considered before and which has been put back into the obligation queue Q.

The last adjustment that we have to make to our obligation queue Q concerns an aspect that is often not considered in detail, but has immense value for any verification tool. While most algorithmic descriptions such as the ones given in [Bradley, 2011; Eén, Mishchenko, et al., 2011] only consider the main algorithm for determining whether an input system satisfies the given property or violates it, a concrete counterexample is of large value for practical applications. For the representation of such a counterexample two common approaches exist: We can either just provide an initial variable assignment that can then be simulated to provide a concrete run of the program that violates the property. Depending on the structure of the program this approach can be sufficient and offers the advantage that simulation is very efficient and on the other hand allows the user to see the violating run directly on the input program. For IC3CFA such an initial assignment can be efficiently extracted from the computed information: If we encounter a violation in BACKWARDBLOCK of Algorithm 14 we have popped an obligation of the form $(0,(\ell,c))$ from Q. Furthermore, by construction, $\ell=\ell_0$ must hold and the cube c represents the data region that has successors leading to the error location ℓ_E. So in order to extract a violating initial variable assignment, we just have to take one state from the violating set of states represented by c, i.e. we query the SMT-solver with $sat(c)$?, which returns SAT by construction, and extract the model which represents one possible variable assignment of the data region c. However, simulating an initial assignment may fail if there occur nondeterministic assignments or return values of unknown, external functions that cannot be covered by simulation. In this case the simulator has to decide on a value which can lead to simulating program runs that do not violate the property.

The second, more involved approach for counterexample generation is to construct a full counterexample trace directly in the model checker. In order to achieve this with IC3CFA, or in fact even with the algorithms presented in [Bradley, 2011; Eén, Mishchenko, et al., 2011], we have to store some sort of relationship between obligations in Q, e.g. parent pointers that indicate based on which other obligation an obligation has been created. Such relation between obligations allows us to trace paths through the, previously mentioned, implicit backward reachability tree that is given by the proof obligations in Q. Tracing the dependencies of obligations allows us to extract a set of obligations $(0, (\ell_0, c_0)), \ldots, (k, (\ell_k, c_k))$. After ordering the obligations based on their level i in ascending order, the list of obligations implies a full path $\pi := \ell_0 = \ell_0, \ell_1, \ldots, \ell_k, \ell_E$ through the CFA $\mathcal{A}$ with annotated data regions $c_0, \ldots, c_k$ except for ℓ_E that can be used to extract concrete variable assignments for each location in $\mathcal{A}$. This approach does compensate for much of the shortcomings of simulating initial variable assignments by not relying on a simulation and being able to handle nondeterministic assignments through the data regions learnt during exploration. But it does also have a large disadvantage, especially in practical application: Without additional overhead, the counterexample trace can only be mapped back to the input CFA $\mathcal{A}$. However, many verification tools, such as our implementation presented in Section 5.1, heavily preprocess the input program in order to reduce the complexity of the model which allows to scale verification to much larger inputs. This on the other hand breaks a clear mapping between CFA and input program, such that the counterexample path on the CFA cannot easily be traced back to a concrete error path on the program, which on the other hand is crucial for the user who does not know about the internals of the model checker.

4.3 Generalization

In the previous section we introduced the basic search phase of the IC3CFA algorithm in analogy to the IC3 [Bradley, 2011] and PDR [Eén, Mishchenko, et al., 2011] algorithms. However, in our presentation of IC3CFA in Algorithms 12 to 14 we did omit a crucial part of the IC3 algorithm, which is generalization. As explained in Sections 3.2 and 3.3, IC3 is sound and complete without the presence of generalization, but is hardly scalable. As such, generalization is not strictly necessary but the success of the IC3 algorithm is mainly due to its ability to efficiently prune the state space using generalization. For IC3CFA however, we need to answer three main questions:

1. how can we generalize a cube of some first-order theory, rather than pure Boolean logic;
2. how do we handle generalization in the presence of multiple incoming edges and multiple predecessor frames;
3. what are the effects of weakest predecessor computation on the generalization?

We will consider and answer these three questions in the next three subsections.

4.3.1 Generalization of a cube

When considering the generalization of a first-order cube, we can distinguish between two approaches: theory-specific generalization and theory-unaware generalizations. For the first category, there exist many different approaches, learning linear invariants for LIA using learning frameworks such as [Garg et al., 2014; Löding et al., 2016] or bitvector generalizations for BV [Welp and Kuehlmann, 2013]. However, all those generalizations can only be applied to their respective theory and thus contradict our previously established goals for a modular verification algorithm with interchangeable backend theories. For this reason we take a closer look at *theory-unaware* generalizations of first-order cubes in the remainder of this section.

In order to find a theory-unaware generalization we draw inspiration from SMT solvers. Just like an SMT solver starts by abstracting away all theory terms, we abstract all theory aspects from our cube, resulting in the Boolean skeleton of the first-order cube. Due to the strict definition of the first-order cube, the abstraction of predicate symbols results in a Boolean skeleton which has the form of a propositional cube. Based on this skeleton, we can execute the standard IC3 generalization, which in our case results in dropping predicate symbols from the first-order cube. This lifting of the syntactic generalization of IC3 offers a simple way to generalize first-order cubes in IC3CFA using linear or binary search as presented in Section 3.3.

However, apart from this lifting of syntactic dropping generalization, we can also apply other generalization approaches to the Boolean skeleton of the first-order cube, such as the methods that were presented in Section 3.3. In particular, unsatisfiable cores can be applied to the Boolean skeleton just like in IC3.

Apart from syntactic generalization, we can also utilize the semantic generalization given by interpolation. While interpolation in general must be considered as a theory-aware procedure, many SMT solvers offer interpolation engines that completely encapsulate the theory-specific aspects and just offer an interface to call the interpolation engine. As such, we can try to extract an interpolation from the solver. If it succeeds, we obtain a valid generalization and if the solver indicates that interpolation is not supported for the theory (combination) we must fall back to syntactic generalization. However, trying to find an interpolant should always be preferred, since for some theories, such as LRA, there exist very efficient interpolation algorithms which yield semantic generalizations of high quality.

Up to now we only considered the set of presented generalization techniques in the context of a single edge, which we call a *edge-local generalization*. However, in order to apply generalization to the IC3CFA algorithm, we must apply generalization in the context of explicit control-flow: For IC3CFA as shown in the previous section, we do not consider a single, global transition relation, but rather small parts of the global transition relation that represent the isolated effect of a single edge between two locations in the CFA. The difference between IC3 and IC3CFA in this aspect becomes most obvious when a location ℓ has more than one predecessor, i.e. $|pred(\ell)| > 1$. A way to determine a generalization that is valid with respect to inductivity relative to all predecessor edges and frames will be presented in the following.

4.3.2 Generalization on multiple edges

In order to generalize a cube c at some location ℓ' at index i, we need to consider a generalization of c that is inductive relative to $F_{(i-1,\ell')}$ along all incoming edges $T_{\ell\rightarrow\ell'}$ for $\ell \in pred(\ell')$. To represent this n-dimensional inductivity for $n = |pred(\ell')|$, we must construct an SMT query of the form

$$\big((F_{(i-1,\ell_1)} \wedge T_{\ell_1 \to \ell'}) \vee \cdots \vee (F_{(i-1,\ell_n)} \wedge T_{\ell_n \to \ell'})\big) \wedge c'. \tag{4.14}$$

Note that we only use query (4.10) from Lemma 4.2 on page 100. A more detailed explanation why weakening the inductivity query is necessary will be given later in this section. This query allows us to check inductivity of c relative to all predecessor frames along all incoming edges and thus to determine a valid generalization of c for ℓ.

Lemma 4.7. *Cube c satisfies (4.14) iff c satisfies (4.9) for all $T_{\ell_j \to \ell'}$, $1 \leq j \leq n$.*

Proof. We prove Lemma 4.7 using the satisfiability of queries (4.14) and (4.9). We can reformulate (4.14) to

$$(F_{(i-1,\ell_1)} \wedge T_{\ell_1 \to \ell'} \wedge c') \vee \cdots \vee (F_{(i-1,\ell_n)} \wedge T_{\ell_n \to \ell'} \wedge c') \tag{4.15}$$

using the distributivity law. Let us now define $q_j = F_{(i-1,\ell_j)} \wedge \neg c \wedge T_{\ell_j \to \ell'} \wedge c'$ for each ℓ_j instance of (4.9), then (4.15) is equivalent to

$$q_1 \vee \cdots \vee q_n. \tag{4.16}$$

This means that (4.16) and thus also (4.14) is unsatisfiable iff every q_j is unsatisfiable. □

However, query (4.14) combines the inductivity of all incoming edges into a single, monolithic query and thus contradicts IC3's principle of avoiding monolithic approaches. We therefore strive to find a method that determines generalizations over multiple incoming edges in a more incremental fashion. To do so, we will start by breaking the monolithic approach above down into isolated generalizations of each incoming edge of ℓ and presenting a way to combine these into a valid generalization for c at ℓ.

Given a location ℓ' and a set of predecessor locations $\mathit{pred}(\ell') = \{\ell_1, \ldots, \ell_n\}$, the set of incoming edges into ℓ' is $\mathit{in}(\ell') = \{e_j \mid (\ell_j, \mathsf{cmd}_j, \ell') \in G\}$. For an index i and cube c, we can determine a generalization g_j of c that is inductive relative to $F_{(i-1,\ell_j)}$ for each incoming edge e_j and its respective transition relation T_{e_j}. Given such a set of edge-local generalizations $\{g_1, \ldots, g_n\}$, a generalization for c along all incoming edges can only drop literal $\mathit{lit} \in c$ iff it can be dropped along each edge e_j individually, a consequence of Lemma 4.7. Using the set notion of cubes, this means that the union of literals of all edge-local generalizations is a valid generalization for location ℓ'.

Corollary 4.1. *Given a set Gen of generalizations $g_j \in Gen$ of cube c along edge $e_j \in in(\ell')$, $g = \bigcup_{j=1}^{n} g_j$ is a valid generalization for c at ℓ' along all e_j.*

Corollary 4.1 offers a way to determine a safe and valid generalization based on a number of edge-local generalization and as such breaks down the complexity of a large SMT query into a number of small queries, an approach that has shown positive effects on performance [Bradley, 2011] and will prove to improve performance in our evaluation (see Chapter 5). However, our current interpretation of first determining the set of edge-local generalizations and then combining those is still a rather monolithic approach that determines edge-local generalizations in an isolated fashion without using information of other edge-local generalizations.

Example 4.6. Reconsider our example program from Listing 1 (page 86) and corresponding CFA $\mathcal{A}$ from Figure 4.3 (page 97) and assume frames $F_{(2,\ell_4)} = \neg(y > 0 \wedge b = 0)$ and $F_{(2,\ell_5)} = \neg true$. We check the proof obligation $(3, (\ell_6, y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0))$ and find it to be inductive relative to $F_{(2,\ell_4)}$ along $e_4 = (\ell_4, \textsf{assume}\ a \geq 0, \ell_6)$, as well as to $F_{(2,\ell_5)}$ along $e_5 = (\ell_5, a := -a, \ell_6)$. We can thus generalize the cube with respect to both edges e_4 and e_5. We start by generalizing $c = y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0$ along e_4, which results in $g_4 = y > 0 \wedge b = 0$. For edge e_5 we can generalize c relative to $F_{(2,\ell_5)}$ to $g_5 = true$. As a result, we join the two results to $g = g_4 \cup g_5 = (y > 0 \wedge b = 0)$ and block this g in $F_{(3,\ell_6)}$.

Using the standard IC3 generalization with the lifting to CFAs, we are able to prove the property in the example benchmark, shown in Listing 1, after 56 iterations in 9.3 seconds, of which 8.6 seconds are spent for 14583 SMT calls.

To improve on this aspect, we take another close look at Corollary 4.1: Consider an edge-local generalization $g_h = c \setminus \delta$ with δ containing all literals that g_h dropped from c. Since $g = \bigcup_{j=1}^{n} g_j$, we can deduce that $g_h \subseteq g$ and hence for $g = c \setminus \Delta$ it follows that $\Delta \subseteq \delta$. In other words, if we know that on some edge e_h we can drop the literals $lit_j \in \delta$ and while obtaining the valid edge-local generalization g_h then the final generalization will drop *at most* the literals in δ. This allows us to execute the full syntactic generalization only on the first incoming edge and then check whether g_h is inductive along the other edges as well. If it is not inductive on some other edge e_m, we have to backtrack

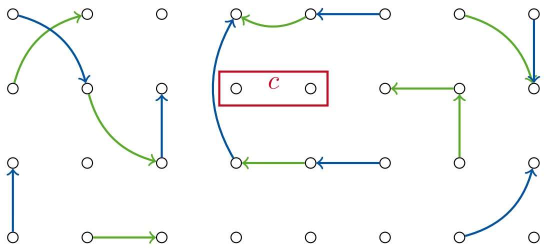
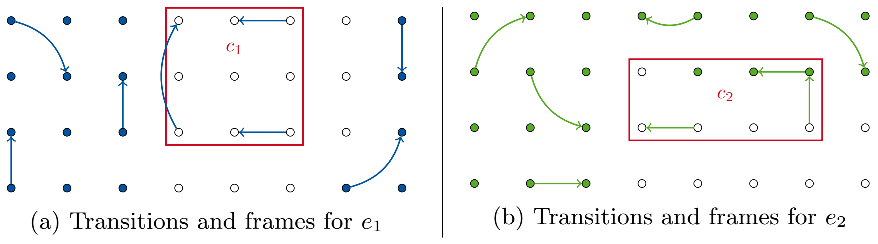


Figure 4.5: Example transition system

based on δ. Note that here it does not suffice to iteratively shift literals back from δ into g_m (which is initially identical to g_j) and probe inductivity on the new g_m since we might introduce many unnecessary literals this way. A more formal perception can be given in terms of the induced subclause lattice introduced in [Bradley and Manna, 2007a]. When dropping literals from clause $d = \neg c$, we traverse this lattice downwards towards the empty clause $\bot$. In case that $d_j = \neg g_j$ is not inductive along e_m we find ourselves at some place inside the subclause lattice that is not inductive. From [Bradley, 2011; Bradley and Manna, 2007a] and the restriction given in Corollary 4.1, we know that there must exist at least one clause d_m that is a valid generlization along e_m and for which it holds that $d_j \subset d_m$, i.e. d_m is reachable from d_j by traversing the subclause lattice upwards. However, since there exist exponentially many subclauses d_m for which $d_j \subset d_m$ holds, precisely 2^n with $n = |c \backslash g_j|$, of which not many are inductive due to dependencies in the theory valuations of the literals. As such it is very likely that we will end up with a subclause that is not minimal. In order to obtain a minimal subclause we could traverse the subclause lattice back down from d_m, similar to the procedure given in [Bradley and Manna, 2007a], but we might also just give up minimality as done in other places and content with d_m. A third approach would be to refrain from traversing the subclause lattice upwards in the first place and rather restart generalization of c but only check the literals of δ for dropping. In practice the question which of these three methods performs best will depend heavily on the input model and vary from case to case. For the remainder we will choose the third approach for its simplicity.

Having obtained an insight into the generalization of IC3CFA, we can take a closer look at why the stronger inductivity query of Lemma 4.2 can no longer be applied in the presence of generalization. We start by motivating the problem with a graphical example and afterwards give a formal proof that Lemma 4.2



(a) Transitions and frames for e_1 (b) Transitions and frames for e_2

Figure 4.6: Isolated generalizations for Figure 4.5

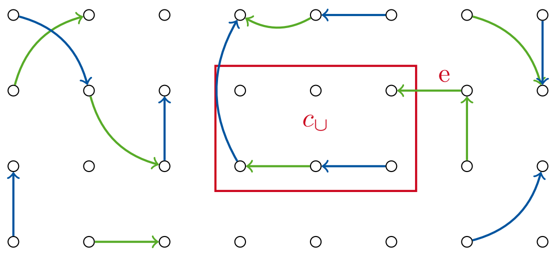


Figure 4.7: Result of merging generalizations of Figure 4.6

does in fact not hold for generalizations. Let $(i, (\ell, c))$ be an obligation consisting of cube c and location ℓ at index i, with ℓ having two incoming edges e_1 and e_2, where e_2 is a self-loop.

Example 4.7. To simplify things we sketch a simplified state space for unprimed state variables without extracting control-flow, like in IC3-SMT [Cimatti and Griggio, 2012]. The example transition system that we use here is depicted in Figure 4.5. For simplicity, arrows depict a transition from an unprimed source state towards a primed target state in direction of the arrow. Blue arrows represent transitions according to edge e_1 and green arrows stand for the self-loop edge e_2. Red boxes give a graphical representation of the states symbolically encoded by the annotated cube. As proven in [Lange et al., 2015] we can use the original inductivity query

$$F_{(i,\ell)} \wedge \neg c \wedge T_e \Rightarrow \neg c' \tag{4.17}$$

for IC3CFA without generalization as it is presented in Section 4.2.

However, by the use of $\neg c$ in the premise of the implication, we exclude all states in c from considering their successors. We now isolate edge e_1 and e_2 as depicted in Figure 4.6a for e_1 and in Figure 4.6b for e_2. We construct generalizations $g_1 = \neg c_1$ and $g_2 = \neg c_2$ that are inductive relative to the corresponding frames, where a state, depicted by a dot, is in $F_{(i-1,\ell_{pre})}$ iff the dot is filled in Figure 4.6 for source location ℓ_{pre} of the respective edge. Both generalizations g_1 and g_2 represent an overapproximation of c and g_i is still inductive relative to the predecessor frame along edge e_i. To block a safe superset of states of c, we must block only those states that are not reachable on both edges, i.e. the intersection of both state sets which is $g = g_1 \cup g_2$. However as depicted in Figure 4.7 we might have not considered reachability of g_2 via transitions that originate from states in g_2, in Figure 4.6a depicted by arrows inside of g_2. By uniting both generalizations g_1 and g_2, i.e. taking the intersection of the respective state sets, the result may cut exactly those transitions that were not considered in the inductivity query of g_2, due to the use of $\neg c$ in the premise of (4.17), leading to a violation of the inductivity query for g.

Formally, the problem that was motivated in Example 4.7 is given as follows:

Theorem 4.1. *Generalization does not preserve* (4.17)*:*

$$\begin{aligned} & (F_1 \wedge T_1 \Rightarrow \neg g_1') \wedge (F_2 \wedge \neg g_2 \wedge T_2 \Rightarrow \neg g_2') \\ \not\Rightarrow \quad & ((F_1 \wedge T_1) \vee (F_2 \wedge \neg (g_1 \wedge g_2) \wedge T_2) \Rightarrow \neg (g_1' \wedge g_2')) \end{aligned}$$

Proof. Given the premises

$$\begin{aligned} &a)\ F_1 \wedge T_1 \wedge g_1' \text{ unsatisfiable} \\ &b)\ F_2 \wedge \neg g_2 \wedge T_2 \wedge g_2' \text{ unsatisfiable} \end{aligned}$$

the conclusion must be invalid, i.e. there must be a satisfying assignment to

$$\begin{aligned} & (F_1 \wedge T_1 \wedge g_1' \wedge g_2') \vee (F_2 \wedge \neg (g_1 \wedge g_2) \wedge T_2 \wedge g_1' \wedge g_2') \\ \overset{a)}{\Longleftrightarrow}\ & F_2 \wedge \neg (g_1 \wedge g_2) \wedge T_2 \wedge g_1' \wedge g_2' \\ \Longleftrightarrow\ & F_2 \wedge (\neg g_1 \vee \neg g_2) \wedge T_2 \wedge g_1' \wedge g_2' \\ \Longleftrightarrow\ & (F_2 \wedge \neg g_1 \wedge T_2 \wedge g_1' \wedge g_2') \vee (F_2 \wedge \neg g_2 \wedge T_2 \wedge g_1' \wedge g_2') \\ \overset{b)}{\Longleftrightarrow}\ & F_2 \wedge \neg g_1 \wedge T_2 \wedge g_1' \wedge g_2'. \end{aligned}$$

This formula obviously has a satisfying assignment, thus generalization does not preserve inductivity as given in (4.17). □

To fix the problem that we formalized in Theorem 4.1, we must permanently strenghten the inductivity query by weakening the premise and removing $\neg c$ from (4.17).

Theorem 4.2. *Generalization preserves* (4.12)*:*

$$\begin{aligned}&(F_1 \wedge T_1 \Rightarrow \neg g_1') \wedge (F_2 \wedge T_2 \Rightarrow \neg g_2')\\ \Longrightarrow\ &((F_1 \wedge T_1) \vee (F_2 \wedge T_2) \Rightarrow \neg (g_1' \wedge g_2'))\end{aligned}$$

Proof by Contradiction.

$$\begin{aligned}&\text{Assume } (((F_1 \wedge T_1) \vee (F_2 \wedge T_2)) \wedge g_1' \wedge g_2') \text{ is satisfiable.}\\ \Longleftrightarrow\ &\exists s.s \models (((F_1 \wedge T_1) \vee (F_2 \wedge T_2)) \wedge g_1' \wedge g_2')\\ \Longleftrightarrow\ &\exists s.s \models (F_1 \wedge T_1 \wedge g_1' \wedge g_2') \vee (F_2 \wedge T_2 \wedge g_1' \wedge g_2')\\ \Longrightarrow\ &\exists s.s \models (F_1 \wedge T_1 \wedge g_1') \vee (F_2 \wedge T_2 \wedge g_2')\end{aligned}$$

This contradicts validity of $(F_1 \wedge T_1 \Rightarrow \neg g_1')$ and $(F_2 \wedge T_2 \Rightarrow \neg g_2')$. $\lightning$

Definition 4.7 (Valid generalization)**.** Given a cube c at index i and location ℓ', an edge $e = (\ell, \mathsf{cmd}, \ell')$ and a frame $F_{(i-1,\ell)}$, then

$$\begin{aligned}&g \in gen(F_{(i-1,\ell)}, e, c)\\ &\quad\Longleftrightarrow g \text{ is a valid generalization of } c \text{ relative to } F_{(i-1,\ell)} \text{ and } e.\end{aligned}$$

A first, simple improvement that we can make to the generalization in order to save calls to the SMT solver is to statically check for duplicate literals in the formula, especially those that appear in the cube c and in the frame. This optimization is enabled by the structure of the inductivity query, as a large conjunction which contains the cube, which is a conjunction itself. As such, if the frame formula contains a clause $\neg lit$ which also appears as $\neg lit \in c$, then the satisfiability of the formula does not change when we remove $\neg lit$ from c. Therefore we can statically drop literals which saves additional solver calls. Since these literals have no importance for the satisfiability of the query, we call them *don't care* literals, and analogously the approach of removing those based on the static check of membership in the frame formula is called *don't care* generalization.

4.3.3 Interaction with weakest preconditions

As seen in Section 4.2, the use of weakest existential preconditions (WEP) allows us to efficiently compute the exact set of predecessor states for a cube that is not inductive relative to its predecessor frame along some edge. Due to the significance of the predecessor computation it does have various effects also for generalization. Thus we will take a closer look at various implications of WEP to generalization [Prinz, 2016]. For the remainder of this section we will assume all edges to contain straight-line code, i.e. no GCL choice command $\Box$. An edge with arbitrary GCL command according to Definition 2.17 can be translated into a set of edges with choice-free commands using the split function.

Definition 4.8. Let (L, G, ℓ_0, ℓ_E) be a CFA and cmd be a GCL command. We define a function $\mathit{split}(\mathsf{cmd})$, which translates cmd into a set of choice-free GCL commands as follows:

$$\mathit{split}(\mathsf{cmd}) = \begin{cases} \mathit{split}(\mathsf{cmd}_1) \cup \mathit{split}(\mathsf{cmd}_2) & \text{if } \mathsf{cmd} = \mathsf{cmd}_1 \,\Box\, \mathsf{cmd}_2 \\ \{c_1;\, c_2 \mid i \in \{1,2\},\, c_i \in \mathit{split}(\mathsf{cmd}_i)\} & \text{if } \mathsf{cmd} = \mathsf{cmd}_1;\, \mathsf{cmd}_2 \\ \{\mathsf{cmd}\} & \text{otherwise} \end{cases}$$

Using *split* as given in Definition 4.8, we transform edges $e = (\ell, \mathsf{cmd}, \ell') \in G$ to a set G' of choice-free edges $e' = (\ell, \mathsf{cmd}', \ell') \in G'$, for $\mathsf{cmd}' \in \mathit{split}(\mathsf{cmd})$. For the remainder of this section we assume edges to be choice-free. Furthermore we restrict all Boolean guards b of $\mathsf{assume}\ b$ and $\mathsf{assert}\ b$ commands to not contain the or connective $\vee$. The restriction on Boolean guards and choice-free edges simplifies the subsequent techniques, since all CTIs will be cubes and the WEP of a cube will again yield a cube.

Using WEP for inductivity queries Given Theorems 4.1 and 4.2 what we call *inductive* is in fact more precisely characterized as reachability given some pre-image. The query

$$\mathit{unsat}\left(F_{(i-1,\ell)} \wedge T_{\ell\to\ell'} \wedge c'\right)?$$

asks whether some state in a set of states characterized by $F_{(i-1,\ell)}$ can reach a state in c via $T_{\ell\to\ell'}$. A semantically equivalent formulation is whether there exists some state that is in the pre-image of c under $T_{\ell\to\ell'}$ *and* in $F_{(i-1,\ell)}$. Since the first part corresponds to the weakest existential precondition (WEP), as defined in Definition 2.21 on page 45, this corresponds to the query

$$\mathit{unsat}\left(F_{(i-1,\ell)} \wedge \mathit{wep}(T_{i-1\to\ell}, c)\right)?$$

Definition 4.9 (Alternative Relative Inductivity)**.** Let $i \in \mathbb{N}$ and CFA $A = (L, G, \ell_0, \ell_E)$ with edge $e = (\ell, \mathsf{cmd}, \ell') \in G$. Given the frame $F_{(i-1,\ell)}$ and a cube c, we define two functions:

$$\begin{aligned} \mathit{relInd}(F_{(i-1,\ell)}, e, c) &\iff \mathit{unsat}\left(F_{(i-1,\ell)} \wedge T_{\ell\to\ell'} \wedge c'\right)? \\ \mathit{relIndAlt}(F_{(i-1,\ell)}, e, c) &\iff \mathit{unsat}\left(F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}, c)\right)? \end{aligned}$$

The cube c is called relative inductive to the frame $F_{(i-1,\ell)}$ and the edge e, iff the function $\mathit{relInd}(F_{(i-1,\ell)}, e, c)$ evaluates to *true*; analogously for $\mathit{relIndAlt}(F_{(i-1,\ell)}, e, c)$.

Given the functions from Definition 4.9 we can show that both are equivalent.

Theorem 4.3. *Let $i \in \mathbb{N}$ and (L, G, ℓ_0, ℓ_E) be a CFA with edge $e = (\ell, \mathsf{cmd}, \ell') \in G$. Furthermore, let c be a cube and $F_{(i-1,\ell)}$ be a frame. It holds that*

$$\mathit{relIndAlt}(F_{(i-1,\ell)}, e, c) \iff \mathit{relInd}(F_{(i-1,\ell)}, e, c)$$

Proof.

$$\begin{aligned} & \mathit{relIndAlt}(F_{(i-1,\ell)}, e, c) \\ \iff & \mathit{unsat}(F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}, c)) \\ \iff & \mathit{unsat}(F_{(i-1,\ell)} \wedge T_{\ell\to\ell'} \wedge c') && \text{(Def. 2.21)} \\ \iff & \mathit{relInd}(F_{(i-1,\ell)}, e, c) \end{aligned}$$

□

While both formulations are equivalent, as shown in the above proof, and at first sight, both seem more or less identical, their execution time in the SMT-solver may vary heavily for two reasons. First, *relInd* reasons about primed and unprimed variables, while *relIndAlt* only reasons about unprimed variables.

This means that the SMT query of *relInd* may, in worst-case, contain twice as many variables as the one for *relIndAlt*. As we know [Biere, Heule, et al., 2009; Bradley and Manna, 2007b; Kroening and Strichman, 2008], the number of variables is not a precise metric for the complexity of a formula and the execution time of the SMT-solver, but in general behaves proportional to these. And second, using *relIndAlt* can be beneficial for SMT solvers with caching, since for different edges with different GCL commands cmd and cmd' that yield the same WEP, i.e. $\mathsf{cmd} \neq \mathsf{cmd}'$ and $\mathit{wep}(\mathsf{cmd}, c) = \mathit{wep}(\mathsf{cmd}', c)$, the queries *relInd* are different, but the ones for *relIndAlt* are identical and can thus benefit from caching. On the other hand, even though we store the computed WEP to create new obligations if c is not inductive relative to $F_{(i-1,\ell)}$, it does introduce some additional overhead: For *relIndAlt* we have to compute the WEP for every inductivity query whereas for *relInd* we only have to compute it if c is not inductive relative to $F_{(i-1,\ell)}$ and as we know from Section 2.4 the computation of weakest preconditions has exponential worst-case complexity. So in general it is uncertain whether the use of *relIndAlt* has benefits over *relInd* and this depends on the structure of the problem. We will give a detailed evaluation of the effects of *relIndAlt* on real-world benchmarks in Chapter 5.

Assumed literals As seen in Section 2.4, GCL contains a so-called assume command, which is a liberal version of the more strict assert command[4]. While an assert command should never evaluate to false in a correct program, assume can evaluate to *false* without any consequence apart from terminating control-flow on this execution branch. As mentioned in Section 2.4, combining the assume command and the choice command $\Box$ allows us to model deterministic branching in the control-flow. As such, we can consider assume statements as a guard for the edge, which means that for an edge $e = (\ell, \mathsf{assume}\ b, \ell') \in G$ *every* state in ℓ that terminates in ℓ' *must* satisfy the guard b and every $\neg b$-state in ℓ has no successor along e and thus especially no $\neg c$-successor. This means that for a literal $\mathit{lit} \in c$ of a cube c that is inductive relative to $F_{(i-1,\ell)}$ along e, then $c \setminus \{\mathit{lit}\}$ is still inductive relative to $F_{(i-1,\ell)}$ along e and as such, we can statically check whether such literals are contained in the cube in order to use it as generalization.

[4]Though being part of our implementation and the original GCL [Dijkstra, 1975, 1976], we omitted the assert command from Definition 2.17 since we transform all edges with assert commands into one edge that assumes the assert to be valid and one that assumes it to be false, where the former one maintains the original source and target location and the latter leading to the error location. This transformation is done as part of the pre-processing that is out of the scope of this thesis.

In order to formalize this generalization based on literals that are contained in assume , we first need to show that the WEP is distributive over conjunction of cubes.

Lemma 4.8. *Let* cmd *be a choice-free GCL command. Given two cubes* c_1, c_2*, it holds that*

$$\mathit{wep}(\mathsf{cmd}, c_1 \wedge c_2) \iff \mathit{wep}(\mathsf{cmd}, c_1) \wedge \mathit{wep}(\mathsf{cmd}, c_2)$$

Proof. We prove Lemma 4.8 by structural induction over GCL command cmd without choice.

- $\mathsf{cmd} = \mathsf{assume}\ b$:

$$\begin{array}{rl} & \mathit{wep}(\mathsf{assume}\ b, c_1 \wedge c_2) \\ = & (c_1 \wedge c_2) \wedge b \\ \iff & (c_1 \wedge b) \wedge (c_2 \wedge b) \\ = & \mathit{wep}(\mathsf{assume}\ b, c_1) \wedge \mathit{wep}(\mathsf{assume}\ b, c_2) \end{array}$$

- $\mathsf{cmd} = x := a$:

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

- $\mathsf{cmd} = \mathsf{cmd}_1;\ \mathsf{cmd}_2$:

$$\begin{array}{rll} & \mathit{wep}(\mathsf{cmd}_1;\ \mathsf{cmd}_2, c_1 \wedge c_2) & \\ = & \mathit{wep}(\mathsf{cmd}_1, \mathit{wep}(\mathsf{cmd}_2, c_1 \wedge c_2)) & \\ = & \mathit{wep}(\mathsf{cmd}_1, \mathit{wep}(\mathsf{cmd}_2, c_1) \wedge \mathit{wep}(\mathsf{cmd}_2, c_2)) & \text{(by hypothesis)} \\ = & \mathit{wep}(\mathsf{cmd}_1, \mathit{wep}(\mathsf{cmd}_2, c_1)) \wedge \mathit{wep}(\mathsf{cmd}_1, \mathit{wep}(\mathsf{cmd}_2, c_2)) & \\ & & \text{(by hypothesis)} \\ = & \mathit{wep}(\mathsf{cmd}_1;\ \mathsf{cmd}_2, c_1) \wedge \mathit{wep}(\mathsf{cmd}_1;\ \mathsf{cmd}_2, c_2) & \end{array}$$

□

Considering the definition of WEP from Definition 2.21, we can see that for a choice-free GCL command cmd, for all literals *lit* that are assumed in cmd, *lit* can be removed from c without any additional SMT query.

Theorem 4.4. *Let g be the generalization of a cube c at index $i \in \mathbb{N}$ and location $\ell' \in L$ with respect to the edge $e = (\ell, \mathsf{cmd}, \ell') \in G$. Given a literal $lit \in g$ it holds that for all $\mathsf{cmd}' \in split(\mathsf{cmd})$ and 'virtual' edges $e' = (\ell, \mathsf{cmd}', \ell')$*

$$wep(\mathsf{cmd}', lit) \subseteq wep(\mathsf{cmd}', true) \implies relInd(F_{(i-1,\ell)}, e', c \setminus \{lit\}).$$

Proof. Let $\widehat{c} = c \setminus \{lit\}$ be the reduced cube. We prove the relative inductivity of $\widehat{c}$ with respect to every 'virtual' edge $e' = (\ell, \mathsf{cmd}', \ell')$, that results from splitting cmd into $\mathsf{cmd}' \in split(\mathsf{cmd})$.

$$\begin{aligned}
& relInd(F_{(i-1,\ell)}, e', c) \\
\iff & relInd(F_{(i-1,\ell)}, e', \widehat{c} \wedge lit) \\
\iff & relIndAlt(F_{(i-1,\ell)}, e', \widehat{c} \wedge lit) && \text{(Theorem 4.3)} \\
\iff & unsat(F_{(i-1,\ell)} \wedge wep(\mathsf{cmd}', \widehat{c} \wedge lit)) \\
\iff & unsat(F_{(i-1,\ell)} \wedge wep(\mathsf{cmd}', \widehat{c}) \wedge wep(\mathsf{cmd}', lit)) && \text{(Lemma 4.8)} \\
& \textit{Since } wep(\mathsf{cmd}', lit) \subseteq (\mathsf{cmd}', true), \\
& \textit{we get } wep(\mathsf{cmd}', lit) \Rightarrow wep(\mathsf{cmd}', true). \\
\implies & unsat(F_{(i-1,\ell)} \wedge wep(\mathsf{cmd}', \widehat{c}) \wedge wep(\mathsf{cmd}', true)) \\
\iff & unsat(F_{(i-1,\ell)} \wedge wep(\mathsf{cmd}', \widehat{c} \wedge true)) && \text{(Lemma 4.8)} \\
\implies & unsat(F_{(i-1,\ell)} \wedge wep(\mathsf{cmd}', \widehat{c})) \\
\iff & relIndAlt(F_{(i-1,\ell)}, e', \widehat{c}) \\
\iff & relInd(F_{(i-1,\ell)}, e', \widehat{c}) && \text{(Theorem 4.3)} \\
\iff & \widehat{c} \in gen(F_{(i-1,\ell)}, e', c)
\end{aligned}$$

Since edge e has been split into a set of 'virtual' e' edges, we can treat the results of the set of virtual edges, as if they were regular edges, in order to obtain a generalization for e. □

Example 4.8. Consider the following situation: We want to check whether a cube $c = x > y \wedge y > 0$ along edge $e = (\ell, \mathsf{assume}\ y > 0, \ell')$ is relative to $F_{(i,\ell)}$. The above theorem allows us to generalize c to $g = (x > y)$ without any SMT checks.

Dropping a literal based on the GCL assumptions is a very simple, yet efficient enhancement of verification. Checking whether an edge contains assume statements is linear in the size of the GCL command. Furthermore since we only consider choice-free edges some of the exponential blowup is avoided, with only some blowup possible in the pathological case that was shown in Example 2.15 on page 46. For real-world applications the computation will be almost linear in the size of the GCL command.

Static inductivity checks and generalization using WEP As shown, we can use the WEP to replace the transition part of the inductivity query to the SMT solver thus reducing the size and the complexity of the solver query. However, we can also use the WEP to statically detect that a cube c *must* be inductive. As we will see, the static check is only able to show that a cube c is inductive, but if the check fails, we cannot derive that c is not inductive, so we have to fall back to a semantic check using the SMT solver. Nevertheless, the presented static check is favourable, since it is cheap to execute and can save expensive solver calls, and, in addition, allows us to statically deduce a generalization of good quality, without any additional solver queries. The key idea can be sketched as follows:

Given that c is inductive relative to $F_{(i-1,\ell)}$ along $e = (\ell, \mathsf{cmd}, \ell')$, then the formula $F_{(i-1,\ell)} \wedge \mathit{wep}(\mathsf{cmd}, c)$ is unsatisfiable by Theorem 4.3. In other words, no state in $\mathit{wep}(\mathsf{cmd}, c)$ is contained in $F_{(i-1,\ell)}$. Due to the exclusion of disjunctions in Boolean guards and the restriction to choice-free GCL commands, $\mathit{wep}(\mathsf{cmd}, c)$ will yield a cube c_{pre} and all states $s \in c_{pre}$ are blocked in $F_{(i-1,\ell)}$. Possibly all $s \in c_{pre}$ will be blocked by a single cube $c_F \in F_{(i-1,\ell)}$ with $c_F \subseteq c_{pre}$, i.e. c_F blocks a superset of the states in c_{pre}. We formalize this static check in Theorem 4.5.

Theorem 4.5. *Let $i \in \mathbb{N}$, (L, G, ℓ_0, ℓ_E) be a CFA and $e = (\ell, \mathsf{cmd}, \ell') \in G$. Consider cube c at location ℓ' and index i with corresponding frame $F_{(i-1,\ell)}$, then*

$$\exists c_F . \neg c_F \in F_{(i-1,\ell)} \wedge c_F \subseteq \mathit{wep}(\mathsf{cmd}, c) \implies \mathit{relInd}(F_{(i-1,\ell)}, e, c)$$

$$\begin{tikzcd}
c \arrow[r, "wep"] \arrow[d, phantom, "\cup|"] & wep(\mathsf{cmd}, c) \arrow[d, "\cup|"] \\
c_g & c_F \arrow[l, "wep^{-1}"]
\end{tikzcd}$$

Figure 4.8: Deducing a generalization from a static wep check

Proof.

$$\begin{aligned}
& \exists c_F.\, \neg c_F \in F_{(i-1,\ell)} \land c_F \subseteq wep(\mathsf{cmd}, c) \\
\Longrightarrow\; & \exists c_F.\, unsat(F_{(i-1,\ell)} \land c_F) \land c_F \subseteq wep(\mathsf{cmd}, c) \\
\Longrightarrow\; & unsat(F_{(i-1,\ell)} \land wep(\mathsf{cmd}, c)) \\
\Longrightarrow\; & relIndAlt(F_{(i-1,\ell)}, e, c) && \text{(Definition 4.9)} \\
\Longrightarrow\; & relInd(F_{(i-1,\ell)}, e, c) && \text{(Theorem 4.3)}
\end{aligned}$$

□

As we can see from Theorem 4.5, if we are able to find a cube c_F that appears negated in $F_{(i-1,\ell)}$ and whose literals are also a subset of the literals in $wep(\mathsf{cmd}, c)$, then c must be inductive relative to $F_{(i-1,\ell)}$ along edge e. However, if this check fails, we cannot deduce that c is not inductive relative $F_{(i-1,\ell)}$ along edge e, since c may still be blocked in $F_{(i-1,\ell)}$, but by a set of cubes each of which blocks a certain subset of the states of c. However, for IC3CFA frames don't grow as big as they use to for standard IC3 and thus the cost of statically checking whether such c exists in $F_{(i-1,\ell)}$ is negligible.

Example 4.9. Reconsider the CFA $\mathcal{A}$ from Figure 4.3 on page 97 and assume the obligation $(3, (\ell_6, (y \geq a \land y > 0 \land b = 0 \land b \geq 0))$ from Example 4.6. For cube $c = y \geq a \land y > 0 \land b = 0 \land b \geq 0$ along $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$, the WEP of c is $wep(\mathsf{assume}\ a \geq 0, c) = y \geq a \land y > 0 \land b = 0 \land b \geq 0 \land a \geq 0$. Since $F_{(2,\ell_4)} = \neg\,(y \geq 0 \land b = 0)$, the clause $\neg c_F = \neg(y \geq 0 \land b = 0)$ is contained in $F_{(2,\ell_4)}$ and $c_F \subseteq wep(\mathsf{assume}\ a \geq 0, c)$, such that c is inductive relative to $F_{(2,\ell_4)}$ along e_4. From another point of view, $wep(\mathsf{assume}\ a \geq 0, c)$ contains all states that can possibly go to c via e_4. These are only states where y is larger than 0 *and* b equals 0.

> But from previously learnt information, we know that reachable states in $F_{(2,\ell_4)}$ are those that satisfy $y \leq 0$ or $b \neq 0$ and thus conflict all states in $\mathit{wep}(\textsf{assume } a \geq 0, c)$.

Apart from checking for inductivity, the existence of $\neg c_F \in F_{(i-1,\ell)}$ has even more implications. In fact, it also implies that every cube c_{post} with $c_F \subseteq \mathit{wep}(\textsf{cmd}, c_{post})$ is inductive relative to $F_{(i-1,\ell)}$. Thus, if we able to find the cube c_g with $c_F = \mathit{wep}(\textsf{cmd}, c_g)$, then c_g is the best generalization that we can deduce from the presence of c_F in $F_{(i-1,\ell)}$. Starting from the cube c_F that we have identified earlier during the inductivity search, c_g is the strongest postcondition $\mathit{sp}(\textsf{cmd}, c_F)$ from a semantic point of view. However, syntactically $\mathit{sp}(\textsf{cmd}, c_F) \subseteq c$ does generally not hold and thus does not qualify as a valid generalization. We therefore have to search for a slightly different function that inverts the *wep* and preserves the syntactical subset relation as sketched in Figure 4.8.

In order to construct such a function wep^{-1}, we again exploit the fact that *wep* is distributive for conjunctions. When we partition the cube c into its literals $c = \{\mathit{lit}_1, \ldots, \mathit{lit}_n\}$, we can determine the *wep* of each literal lit_i individually. However we may take special care of assumed literals, since their behaviour has some interesting side effects, as shown (in another context) on page 125.

Lemma 4.9. *Given a choice-free GCL command* cmd *and cube c, then*

$$\mathit{wep}(\textsf{cmd}, c) = \bigwedge_{\mathit{lit} \in c} \mathit{wep}(\textsf{cmd}^{\textsf{assume}}, \mathit{lit}) \wedge \mathit{wep}(\textsf{cmd}, \mathit{true})$$

where $\textsf{cmd}^{\textsf{assume}}$ *is the command* cmd *with all* assume *statements removed.*

Proof. We prove this via structural induction over the GCL command cmd (ϵ for the empty command):

- assume b

$$\begin{aligned} & \mathit{wep}(\textsf{assume } b, c) \\ = {} & c \wedge b \\ = {} & \left(\bigwedge_{\mathit{lit} \in c} \mathit{lit} \right) \wedge b \\ = {} & \bigwedge_{\mathit{lit} \in c} \mathit{wep}(\epsilon, \mathit{lit}) \wedge \mathit{wep}(\textsf{assume } b, \mathit{true}) \end{aligned}$$

- $x := a$

$$\begin{aligned}
& wep(x := a, c) \\
= {} & c[x \mapsto a] \\
= {} & \left(\bigwedge_{lit \in c} lit[x \mapsto a]\right) \wedge true \\
= {} & \left(\bigwedge_{lit \in c} lit[x \mapsto a]\right) \wedge true[x \mapsto a] \\
= {} & \left(\bigwedge_{lit \in c} wep(x := a, lit)\right) \wedge wep(x := a, true)
\end{aligned}$$

- $\mathsf{cmd}_1;\ \mathsf{cmd}_2$

$$\begin{aligned}
& wep(\mathsf{cmd}, c) \\
= {} & wep(\mathsf{cmd}_1;\ \mathsf{cmd}_2, c) \\
= {} & wep(\mathsf{cmd}_1, wep(\mathsf{cmd}_2, c)) \\
= {} & wep(\mathsf{cmd}_1, \bigwedge_{lit \in c} wep(\mathsf{cmd}_2^{\mathsf{assume}}, lit) \wedge wep(\mathsf{cmd}_2, true)) && \text{(Ind. Hyp.)} \\
= {} & wep(\mathsf{cmd}_1, \bigwedge_{lit \in c} wep(\mathsf{cmd}_2^{\mathsf{assume}}, lit)) \wedge wep(\mathsf{cmd}_1, wep(\mathsf{cmd}_2, true)) \\
= {} & wep(\mathsf{cmd}_1^{\mathsf{assume}}, \bigwedge_{lit \in c} wep(\mathsf{cmd}_2^{\mathsf{assume}}, lit)) \wedge \\
& wep(\mathsf{cmd}_1, true) \wedge wep(\mathsf{cmd}_1, wep(\mathsf{cmd}_2, true)) && \text{(Ind. Hyp)} \\
= {} & \bigwedge_{lit \in c} wep(\mathsf{cmd}_1^{\mathsf{assume}}, wep(\mathsf{cmd}_2^{\mathsf{assume}}, lit)) \wedge \\
& wep(\mathsf{cmd}_1, true \wedge wep(\mathsf{cmd}_2, true)) \\
= {} & \bigwedge_{lit \in c} wep(\mathsf{cmd}^{\mathsf{assume}}, lit) \wedge wep(\mathsf{cmd}_1, wep(\mathsf{cmd}_2, true)) \\
= {} & \bigwedge_{lit \in c} wep(\mathsf{cmd}^{\mathsf{assume}}, lit) \wedge wep(\mathsf{cmd}, true)
\end{aligned}$$

□

This way we preserve the semantics as well as the syntactic structure of the *wep*. While this decomposition into n *wep* applications does not alter the complexity or semantics, it allows us to store a mapping $g : \mathit{Literal} \mapsto \mathit{Literal}$ from $\varphi = \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, \mathit{lit}_i)$ to lit_i of the assume-free GCL command.

Corollary 4.2. *If a cube $c_F \in F_{(i-1,\ell)}$ with $c_F \subseteq \mathit{wep}(\mathsf{cmd}, c)$ exists, then*

$$c_F \subseteq \bigwedge_{\mathit{lit} \in c} \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, \mathit{lit}) \wedge \mathit{wep}(\mathsf{cmd}, \mathit{true}).$$

Using Corollary 4.2, we apply the mapping g to the set of literals $\{\varphi_j \mid \varphi_j \in (c_F \setminus \mathit{wep}(\mathsf{cmd}, \mathit{true}))\}$, i.e.

$$g(\varphi_j) = \mathit{lit}_j \wedge \varphi_j \in c_F \implies \mathit{lit}_j \in c_g$$

Definition 4.10. Given cubes $c_1 = \{\mathit{lit}_1, \ldots, \mathit{lit}_n\}$ and $c_2 \subseteq c_1$, choice-free GCL command cmd and partial mapping $g : \mathit{Literal} \mapsto \mathit{Literal}$ with

$$g(\varphi_i) = \mathit{lit}_i \iff \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, \mathit{lit}_i) = \varphi_i$$

we define the function $\mathit{wep}_{c_1}^{-1} : \mathit{GCL} \times \mathit{Cube} \mapsto \mathit{Cube}$ as

$$\mathit{wep}_{c_1}^{-1}(\mathsf{cmd}, c_2) = \bigwedge \{\mathit{lit}_j \mid g(\varphi_j) = \mathit{lit}_j \wedge \varphi_j \in c_2 \cap \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, c_1)\}$$

Example 4.10. Reconsider Example 4.9. After identifying $c_F = (y \geq 0 \wedge b = 0) \in F_{(2,\ell_4)}$, we can deduce the generalization of c along edge $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$ from c_F. In this case, the mapping g is very simple: Since the GCL command of e_4 does not contain any assignments, it is the identity of the literals of c. Thus we first remove the literals in $\mathit{wep}(\mathsf{assume}\ a \geq 0, \mathit{true}) = (a \geq 0)$ from c_F, i.e.

$$\begin{aligned} &\{\varphi_j \mid \varphi \in (c_F \setminus \mathit{wep}(\mathsf{assume}\ a \geq 0, \mathit{true}))\} \\ =&\{y \geq 0, b = 0\}. \end{aligned}$$

Given that $g = \mathit{id}$, we obtain the result

$$\mathit{wep}_{c_1}^{-1}(\mathsf{cmd}, c_2) = \bigwedge \{y \geq 0, b = 0\}.$$

This result is identical to the result that we would have obtained using linear IC3-style generalization, but without the use of a solver.

Note that we define individual functions $wep_{c_1}^{-1}$ based on the cube c_1 that is used to construct the mapping g.

Corollary 4.3. *Let* cmd *be a choice-free GCL command. Given three cubes $c_1 \subseteq c, c_2 \subseteq c$, it holds that*

$$wep_c^{-1}(\mathsf{cmd}, c_1 \wedge c_2) \iff wep_c^{-1}(\mathsf{cmd}, c_1) \wedge wep_c^{-1}(\mathsf{cmd}, c_2)$$

Given Definition 4.10 we can see that Corollary 4.3 obviously holds. Apart from distributivity, we must ensure that our newly constructed wep_c^{-1} is a monotonic function, too.

Lemma 4.10. *For cubes c, $c_1 \subseteq c$ and $c_2 \subseteq c$ and choice-free GCL command* cmd*, wep_c^{-1} is a monotonic function, i.e.*

$$c_1 \subseteq c_2 \implies wep_c^{-1}(\mathsf{cmd}, c_1) \subseteq wep_c^{-1}(\mathsf{cmd}, c_2)$$

Proof. Given premise $c_1 \subseteq c_2$, we can partition c_2 as $c_2 = c_1 \cup c_\Delta$, such that

$$\begin{aligned}
& wep_c^{-1}(\mathsf{cmd}, c_2) = \bigwedge\{lit_j \mid g(lit_j) = \varphi_j \wedge \varphi_j \in c_2\} && \text{(Def. 4.10)}\\
\iff & wep_c^{-1}(\mathsf{cmd}, c_2) = \bigwedge\{lit_j \mid g(lit_j) = \varphi_j \wedge \varphi_j \in (c_1 \cup c_\Delta)\} && \text{(Premise)}\\
\iff & wep_c^{-1}(\mathsf{cmd}, c_2) = \bigwedge\{lit_j \mid g(lit_j) = \varphi_j \wedge (\varphi_j \in c_1 \vee \varphi_j \in c_\Delta)\}\\
\iff & wep_c^{-1}(\mathsf{cmd}, c_2) = \bigwedge\{lit_j \mid (g(lit_j) = \varphi_j \wedge \varphi_j \in c_1) \vee\\
& \qquad (g(lit_j) = \varphi_j \wedge \varphi_j \in c_\Delta)\}\\
\iff & wep_c^{-1}(\mathsf{cmd}, c_2) = \bigwedge\{lit_j \mid g(lit_j) = \varphi_j \wedge \varphi_j \in c_1\} \cup\\
& \qquad \{lit_j \mid g(lit_j) = \varphi_j \wedge \varphi_j \in c_\Delta\}\\
\iff & wep_c^{-1}(\mathsf{cmd}, c_2) = \bigwedge\{lit_j \mid g(lit_j) = \varphi_j \wedge \varphi_j \in c_1\} \wedge\\
& \qquad \bigwedge\{lit_j \mid g(lit_j) = \varphi_j \wedge \varphi_j \in c_\Delta\}\\
\iff & wep_c^{-1}(\mathsf{cmd}, c_2) = wep_c^{-1}(\mathsf{cmd}, c_1) \wedge\\
& \qquad \bigwedge\{lit_j \mid g(lit_j) = \varphi_j \wedge \varphi_j \in c_\Delta\} && \text{(Def. 4.10)}\\
\implies & wep_c^{-1}(\mathsf{cmd}, c_1) \subseteq wep_c^{-1}(\mathsf{cmd}, c_2)
\end{aligned}$$

□

Given that wep_c^{-1} is monotonic, we need to show that it is in fact an inverse of *wep*, as the name already suggests. To do so, we start with the more intuitive application, that wep_c^{-1} is a *left inverse* of *wep*, i.e. applying wep_c^{-1} to the result of *wep* yields the identity.

Lemma 4.11. *Given cube c and GCL command* cmd*, wep_c^{-1} is a left inverse of wep:*

$$wep_c^{-1}(\mathsf{cmd}, wep(\mathsf{cmd}, c)) = c$$

Proof.

$$\begin{aligned}
& wep_c^{-1}(\mathsf{cmd}, wep(\mathsf{cmd}, c)) \\
&= wep_c^{-1}(\mathsf{cmd}, \bigwedge_{lit \in c} wep(\mathsf{cmd}^{\mathsf{assume}}, lit) \wedge wep(\mathsf{cmd}, true)) && \text{(Lemma 4.9)} \\
&= \bigwedge_{lit \in c} wep_c^{-1}(\mathsf{cmd}, wep(\mathsf{cmd}^{\mathsf{assume}}, lit)) \wedge wep_c^{-1}(\mathsf{cmd}, wep(\mathsf{cmd}, true)) \\
& && \text{(Corollary 4.3)} \\
&= \bigwedge_{lit \in c} lit \wedge \bigwedge \emptyset && \text{(Definition 4.10)} \\
&= c
\end{aligned}$$

□

Applying Definition 4.10, we see that there exists no entry in mapping g for literals in $wep(\mathsf{cmd}, true)$ such that wep_c^{-1} returns the conjunction over the empty set, which is equivalent to *true*. By definition, g contains a mapping $\varphi \mapsto lit$ for $\varphi = wep(\mathsf{cmd}^{\mathsf{assume}}, lit)$ for each $lit \in c$. As such wep_c^{-1} will map φ back to *lit*.

Lemma 4.12. *Given GCL command* cmd*, cubes c and $\bar{c} = wep(\mathsf{cmd}, c)$ then*

$$wep(\mathsf{cmd}, wep_c^{-1}(\mathsf{cmd}, \bar{c})) = \bar{c}$$

Proof.

$$\begin{aligned}
&\mathit{wep}(\mathsf{cmd}, \mathit{wep}_c^{-1}(\mathsf{cmd}, \bar{c}))\\
&= \mathit{wep}(\mathsf{cmd}, \bigwedge\{\mathit{lit} \mid g(\varphi) = \mathit{lit} \land \varphi \in \bar{c} \cap \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, c)\}) \quad \text{(Definition 4.10)}\\
&= \mathit{wep}(\mathsf{cmd}, \bigwedge\{\mathit{lit} \mid g(\varphi) = \mathit{lit} \land \varphi \in \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, c)\})\\
&= \mathit{wep}(\mathsf{cmd}, c)\\
&= \bar{c} \quad \text{(Premise)}
\end{aligned}$$

□

In the last step of the proof we can use the premise that $\bar{c}$ is the $\mathit{wep}(\mathsf{cmd}, c)$. Together with Lemma 4.9 this means that $\bar{c} \supseteq \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, c)$ and therefore the intersection $\bar{c} \cap \mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, c)$ is exactly $\mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, c)$. As a consequence, each literal φ of the $\mathit{wep}(\mathsf{cmd}^{\mathsf{assume}}, c)$ is mapped back to the original literal $\mathit{lit} \in c$, such that the conjunction over the set of literals is exactly c.

Using the previous results we can show that if there exists a clause $\neg c_F$ in $F_{(i-1,\ell)}$, and the corresponding cube c_F is a subset of the $\mathit{wep}(\mathsf{cmd}, c)$, then appyling wep_c^{-1} to c_F yields a subset of the original cube c.

Lemma 4.13. *Given cube c is inductive relative to $F_{(i-1,\ell)}$ along $e = (\ell, \mathsf{cmd}, \ell')$ then*

$$\exists c_F .\, \neg c_F \in F_{(i-1,\ell)} \land c_F \subseteq \mathit{wep}(\mathsf{cmd}, c) \implies \mathit{wep}_c^{-1}(\mathsf{cmd}, c_F) \subseteq c$$

Proof.

$$\begin{aligned}
&\exists c_F .\, \neg c_F \in F_{(i-1,\ell)} \land c_F \subseteq \mathit{wep}(\mathsf{cmd}, c)\\
\implies &\exists c_F .\, \neg c_F \in F_{(i-1,\ell)} \land \mathit{wep}_c^{-1}(\mathsf{cmd}, c_F) \subseteq \mathit{wep}_c^{-1}(\mathsf{cmd}, \mathit{wep}(\mathsf{cmd}, c))\\
&\qquad \text{(Lemma 4.10)}\\
\implies &\mathit{wep}_c^{-1}(\mathsf{cmd}, c_F) \subseteq c \qquad \text{(Lemma 4.11)}
\end{aligned}$$

□

Having shown that $\mathit{wep}_c^{-1}(\mathsf{cmd}, c_F)$ is a subset of the original cube c, we now show that if c is inductive relative to $F_{(i-1,\ell)}$, then $\mathit{wep}_c^{-1}(\mathsf{cmd}, c_F)$ is also inductive relative to $F_{(i-1,\ell)}$.

Lemma 4.14. *Given cube c is inductive relative to $F_{(i-1,\ell)}$ along $e = (\ell, \mathsf{cmd}, \ell')$ then*

$$\exists c_F.\, \neg c_F \in F_{(i-1,\ell)} \wedge c_F \subseteq \mathit{wep}(\mathsf{cmd}, c) \implies \mathit{relInd}(F_{(i-1,\ell)}, e, \mathit{wep}_c^{-1}(\mathsf{cmd}, c_F))$$

Proof by Contraposition.

$$\begin{aligned}
& \neg\mathit{relInd}(F_{(i-1,\ell)}, e, \mathit{wep}_c^{-1}(\mathsf{cmd}, c_F)) \\
\implies & \neg\mathit{relIndAlt}(F_{(i-1,\ell)}, e, \mathit{wep}_c^{-1}(\mathsf{cmd}, c_F)) \\
\iff & \neg\mathit{unsat}(F_{(i-1,\ell)} \wedge \mathit{wep}_c^{-1}(\mathsf{cmd}, c_F)) \\
\iff & \mathit{sat}(F_{(i-1,\ell)} \wedge \mathit{wep}_c^{-1}(\mathsf{cmd}, c_F)) \\
\iff & \mathit{sat}(F_{(i-1,\ell)} \wedge c_F) \\
\implies & \neg\left(\exists c_F \in F_{(i-1,\ell)}\right) \\
\implies & \neg\left(\exists c_F \in F_{(i-1,\ell)}.\, c_F \subseteq \mathit{wep}(\mathsf{cmd}, c)\right)
\end{aligned}$$

□

Using the previously established results, we can now prove that under the given premises that c is inductive relative to $F_{(i-1,\ell)}$ and there exists a clause $\neg c_F$ in $F_{(i-1,\ell)}$, and the corresponding cube c_F is a subset of the $\mathit{wep}(\mathsf{cmd}, c)$, then appyling wep_c^{-1} to c_F in fact yields a valid generalization of c.

Theorem 4.6. *Given that cube c is inductive relative to $F_{(i-1,\ell)}$ along $e = (\ell, \mathsf{cmd}, \ell')$ then*

$$\begin{aligned}
&\exists c_F.\, \neg c_F \in F_{(i-1,\ell)} \wedge c_F \subseteq \mathit{wep}(\mathsf{cmd}, c) \\
&\qquad\qquad \implies \mathit{wep}^{-1}(\mathsf{cmd}, c_F) \in \mathit{gen}(F_{(i-1,\ell)}, e, c).
\end{aligned}$$

Proof. For $\mathit{wep}^{-1}(\mathsf{cmd}, c_F)$ to be a valid generalization of c, it must satisfy

a) $\mathit{wep}^{-1}(\mathsf{cmd}, c_F) \subseteq c$

b) $\mathit{relInd}(F_{(i-1,\ell)}, e, \mathit{wep}_c^{-1}(\mathsf{cmd}, c_F))$

Where a) has been shown in Lemma 4.13 and b) in Lemma 4.14. □

Using the result from Theorem 4.6, we can enhance our search for predecessor cubes from Theorem 4.5 to construct the wep_c^{-1} of such a c_F in order obtain a generalization for cube c without any solver queries.

4.3.4 Efficient handling of generalizations

Let us reconsider the *split* operation from Definition 4.8. Given an edge $e = (\ell, \mathsf{cmd}, \ell')$ where cmd contains at least one choice command $\Box$. The function *split* will return a set of GCL commands that can also be considered as parallel edges in the CFA. Whenever we encounter such an edge e and cube c is not inductive relative to $F_{(i-1,\ell)}$, then we obtain a set of n parallel GCL commands $\{\mathsf{cmd}_1, \ldots, \mathsf{cmd}_n\}$ for each of which we apply *wep* resulting in a set of predecessor cubes $\{c_1, \ldots, c_n\}$ of c which are all one-step predecessors of states in c. We therefore have to create proof obligations for each of these cubes in IC3CFA. This however immediately raises the question whether the order in which obligations for the same index are handled matters and if so, which order is optimal or at least best overall.

Since the created obligations only differ in the cube that they contain, we have to find some order based on a metric for the *"quality"* of the cube. Like in standard IC3, we aim to find generalizations that block large regions of states, i.e. those that contain the fewest literals. For that reason we propose to prefer cubes with less literals over cubes with many literals, since cubes with less literals are more likely to produce small generalizations than cubes with many literals. As such, we change the priority queue of our obligation queue to order cubes based on ascending index and for obligations with the same index sort them in ascending order of their cube's size. Other possible orderings include descending cube size or a random order, but neither of them are convincing theoretically, nor experimentally (see Chapter 5). Our result that ordering obligations according to their cube's size in ascending order performs well has also been confirmed in [Gurfinkel and Ivrii, 2015] in another context. In [Gurfinkel and Ivrii, 2015], proof obligations are not only created when cubes are not inductive relative to their respective predecessor frame, but also for cubes that may be helpful to be proven inductive in order to block larger regions of the state space and thus improving convergence. Since multiple of these so-called *may*-obligations can be created for the same index, the problem of ordering obligations also arose and was solved by ordering obligations based on the cube size in ascending order.

Caching generalizations As seen in this section, IC3CFA makes heavy use of the WEP for predecessor computation, which enables multiple optimizations to the generalization procedure of IC3CFA. Apart from exploiting the presence of exact predecessors as seen above, we will now focus on a different aspect: When looking at the process of how IC3CFA searches the state space for counterexamples and how it creates stepwise reachability information in the frame sequence, we can see that due to the deterministic search on the CFA, as well as the precise predecessor computation, we determine the same cubes over and over again, but at different indices, leading to repeated generalizations of the same cube. While arguably generalizing a cube again in a randomized procedure, resulting in different cubes will help IC3CFA to not get stuck with cubes of bad quality, it may also slow down the convergence since different cubes prevent the syntactic termination criterion that IC3CFA and other IC3 algorithms use [Bradley, 2011; Eén, Mishchenko, et al., 2011]. Here we assume that an inductive frame has been reached if two consecutive frames have identical clauses, which is harder to achieve if many different cubes are generalized and blocked. To achieve this we would need to make the generalization procedure even more deterministic.

Since we find ourselves in the situation that the same cubes have to be generalized multiple times and we strive to find a deterministic result in each of these generalizations, caching the results of the generalization procedure becomes desirable. However, it is not directly obvious how such a cache may look like, because the obvious mapping $\mathit{cache} : \mathit{Cube} \mapsto \mathit{Cube}$ from input cube c to generalized cube g is not valid, since the generalization also depends on other factors, such as the predecessor frame $F_{(i-1,\ell)}$ and the edge formula T_e representing the GCL command cmd of the edge $e = (\ell, \mathsf{cmd}, \ell')$ along which we generalize c.

To store the correct setting in which a cube was generalized and to map it to the generalized cube, we introduce so-called *generalization contexts* that store exactly this context of frame and edge of the generalization.

Definition 4.11 (Generalization Context).
The generalization context GC_i is defined as

$$GC_i \subseteq 2^{Cube \times G \times Frame \times Cube}$$

containing entries of the form (c, e, F, g) where $g \subseteq c$ is a generalization of input cube c.

Given such a generalization context, we can store the exact setting in which a cube c was generalized to cube g. Similar to frames, we make generalizations happening at index j available in each generalization contexts GC_i for $i \geq j$. As a first step, we can now use this cached information if we have to generalize cube c at index i and $(c, e, F, g) \in GC_i$ relative to the same frame $F = F_{(i-1,\ell)}$ and edge e to the generalized cube g.

This however will not produce many cache hits, since frames evolve and change over time. Here, however, we can exploit the fact that a frame F can only monotonically grow, i.e. we just conjoin clauses to F, but never remove them. This also means that the set of states represented by frame F only decreases. Therefore, if at some point we have a set of states represented by cube c and we find that no state in F has a successor state in c and that it does not even have a successor state in the generalized cube g, then for some other frame $F' \sqsubseteq F$ that contains less states than F, F' also does not have any successor state in g.

Theorem 4.7. *Given index i, edge $e = (\ell, \mathsf{cmd}, \ell')$ and let c be a cube to be generalized at i and ℓ' relative to $F_{(i-1,\ell)}$ and e, then*

$$(c, e, F, g) \in GC_i \wedge F \subseteq F_{(i-1,\ell)} \implies g \in gen(F_{(i-1,\ell)}, e, c)$$

Proof.

$$\begin{aligned}
& (c, e, F, g) \in GC_i \wedge F \subseteq F_{(i-1,\ell)} \\
\implies & \exists j \leq i.\ g \in gen(F_{(j-1,\ell)}, e, c) \wedge F = F_{(j-1,\ell)} \wedge F_{(j-1,\ell)} \subseteq F_{(i-1,\ell)} \\
\implies & relInd(F_{(j-1,\ell)}, e, g) \wedge F_{(j-1,\ell)} \subseteq F_{(i-1,\ell)} \\
\implies & unsat(F_{(j-1,\ell)} \wedge T_e \wedge g') \wedge F_{(j-1,\ell)} \subseteq F_{(i-1,\ell)} \\
\implies & unsat(F_{(j-1,\ell)} \wedge F_\Delta \wedge T_e \wedge g') \wedge F_{(j-1,\ell)} \subseteq F_{(i-1,\ell)} \\
\implies & unsat(F_{(i-1,\ell)} \wedge T_e \wedge g') \\
\implies & relInd(F_{(i-1,\ell)}, e, g) \\
\implies & g \in gen(F_{(i-1,\ell)}, e, c)
\end{aligned}$$

□

Using Theorem 4.7, we can now use the stored information (c, e, F, g) of the generalization context GC_i not only for situations where we have the exact same setting, but also for situations where we have to generalize the same cube c along edge e but relative to a stronger frame $F_{(i-1,\ell)} \supseteq F$, too. While this improves the number of cache hits dramatically, we should not immediately block g as the generalization of c. Even though g satisfies as a valid generalization, since it is a subset $g \subseteq c$ and is relative inductive to $F_{(i-1,\ell)}$, it may not be optimal and in practice it rarely is. The reason for this is that due to the stronger frame $F_{(i-1,\ell)} \supseteq F$, there are now many more states blocked in $F_{(i-1,\ell)}$ than there were in F, such that even larger cubes $\bar{g} \subseteq g$ are now one-step unreachable from $F_{(i-1,\ell)}$. In other words, all literals $lit \in c \backslash g$ can definitely be dropped without any solver checks, but there might also be others that we may be able to drop, so we have to check the remaining literals. We therefore call the cube g obtained from GC_i as a result of Theorem 4.7 an *upper bound* of the final generalization of c.

Example 4.11. Assume that the generalization from Example 4.10 has already happened and has been cached. Furthermore, assume that in the meantime $F_{(2,\ell_4)}$ has grown from $F_{(2,\ell_4)} = \neg(y \geq 0 \wedge b = 0)$ to $F_{(2,\ell_4)} = \neg(y \geq 0 \wedge b = 0) \wedge \neg(a\%b = 0 \wedge b \neq 0)$ (with the new lemma stemming from one iteration of the loop) and we want to generalize $c = y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0$ along $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$ relative to this new, larger $F_{(2,\ell_4)}$. Then we can automatically deduce that we only need to check whether we can drop the literals $(y \geq 0)$ or $(b = 0)$. This means that we can save two SMT queries. However, we can also see that it is in fact advisable to check the remaining literals, since we can drop the literal $(b = 0)$, too, after blocking $(a\%b = 0 \wedge b \neq 0)$.

We can however improve the *upper bound* for the generalization of cube c even further by using additional cache hits. So far, we have only used entries $(c, e, F, g) \in GC_i$ for identical matches of cube c. We can however also use $(\bar{c}, e, F, g) \in GC_i$ for cubes $g \subseteq c$ and frames $F \subseteq F_{(i-1,\ell)}$ to increase the level of information reuse even further.

Theorem 4.8. *Given index i, edge $e = (\ell, \mathsf{cmd}, \ell')$ and let c be a cube to be generalized at i and l' relative to $F_{(i-1,\ell)}$ and e, then*

$$\left((\bar{c}, e, F, g) \in GC_i \wedge F \subseteq F_{(i-1,\ell)} \wedge g \subseteq c\right) \implies g \in gen(F_{(i-1,\ell)}, e, c)$$

Proof.

$$\begin{aligned} & (\bar{c}, e, F, g) \in GC_i \wedge F \subseteq F_{(i-1,\ell)} \wedge g \subseteq c & \\ \implies & g \in gen(F_{(i-1,\ell)}, e, \bar{c}) \wedge g \subseteq c & \text{(Theorem 4.7)} \\ \implies & relInd(F_{(i-1,\ell)}, g, e) \wedge g \subseteq c & \text{(Definition 4.7)} \\ \implies & g \in gen(F_{(i-1,\ell)}, e, c) & \end{aligned}$$

□

Since the generalizations resulting from Theorem 4.8 are larger and thus of worse quality than the ones obtained using Theorem 4.7, we make the search in our cache staged and start by searching for entries that satisfy Theorem 4.7 and only if that does succeed, we search for entries satisfying Theorem 4.8 in a second step.

Example 4.12. Consider a similar situation as in Example 4.11, but now we want to generalize the cube $c = y \geq a \wedge y > 0 \wedge -b = 0 \wedge b < 0$ that originates from the backward trace $\ell_8 \rightarrow \ell_7 \rightarrow \ell_6$ instead of $\ell_8 \rightarrow \ell_6$ along edge $e_4 = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6)$. We also assume the current frame $F_{(2,\ell_4)} = \neg(y \geq 0 \wedge b = 0) \wedge \neg(a\%b = 0 \wedge b \neq 0)$ and the cache to contain the generalization from Example 4.10. This means, that

$$\begin{aligned}
&(\bar{c}, e, F, g) \in GC_i, \text{ with} \\
&\quad \bar{c} = y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0, \\
&\quad e = (\ell_4, \mathsf{assume}\ a \geq 0, \ell_6), \\
&\quad F = \neg(y \geq 0 \wedge b = 0), \text{ and} \\
&\quad g = y \geq 0 \wedge b = 0.
\end{aligned}$$

Since this entry satisfies $F \subseteq F_{(2,\ell_4)}$ and $g \subseteq c$, the previous generalization g gives an upper bound on the literals for the current generalization.

Apart from the *upper bounds* on the literals for the final generalization of cube c, we can however also use the information stored in the generalization context GC_i to derive a *lower bound* on the literals. Intuitively, these are the literals of which we know that they definitely cannot be dropped from c.

The intuition behind such *lower bounds* is that if we encounter a generalization context (c, e, F, g) with frame $F_{(i-1,\ell)} \subseteq F$, then we assume that each $\widehat{g} \subseteq g$ was not a valid generalization relative to F. Since $F_{(i-1,\ell)} \subseteq F$ such $\widehat{g}$ will then also not be a valid generalization relative to $F_{(i-1,\ell)}$ and thus all literals in g cannot be dropped from c.

Theorem 4.9. *Given index i, edge $e = (\ell, \mathsf{cmd}, \ell')$ and let c be a cube to be generalized at i and l' relative to $F_{(i-1,\ell)}$ and e, then*

$$\begin{aligned}
&(c, e, F, g) \in GC_i \wedge F_{(i-1,\ell)} \subseteq F \wedge \widehat{g} \subset g \\
\implies\ &\widehat{g} \notin gen(F_{(i-1,\ell,)}e, c) \\
\implies\ &g \subseteq g_{fin} \subseteq c,\ \forall g_{fin} \in gen(F_{(i-1,\ell)}, e, c)
\end{aligned}$$

Proof.

$$
\begin{aligned}
& (c, e, F, g) \in GC_i \wedge F_{(i-1,\ell)} \subseteq F \wedge \widehat{g} \subset g \\
\implies & \exists j \leq i.\ g \in gen(F_{(j-1,\ell)}, e, c) \wedge F = F_{(j-1,\ell)} \\
\implies & \exists j \leq i.\ \widehat{g} \notin gen(F_{(j-1,\ell)}, e, c) \wedge F = F_{(j-1,\ell)} \\
\implies & \neg relInd(F_{(j-1,\ell)}, e, \widehat{g}) \\
& \text{Since } F_{(i-1,\ell)} \subseteq F_{(j-1,\ell)} = F, \text{ let } F_{(j-1,\ell)} := F_{(i-1,\ell)} \cup F_{\Delta}. \\
\implies & \neg relInd(F_{(i-1,\ell)} \cup F_{\Delta}, e, \widehat{g}) \\
\implies & \neg relInd(F_{(i-1,\ell)}, e, \widehat{g}) \\
\implies & \widehat{g} \notin gen(F_{(i-1,\ell)}, e, c)
\end{aligned}
$$

□

Example 4.13. Consider an inverse situation as the one in Example 4.11. We approach ℓ_6 with $c = y \geq a \wedge y > 0 \wedge b = 0 \wedge b \geq 0$ that we want to generalize relative to $F_{(2,\ell_4)} = \neg(y \geq 0 \wedge b = 0)$ along $e_4 = (\ell_4, \textsf{assume } a \geq 0, \ell_6)$ and there exists some cached generalization for c such that

$$
\begin{aligned}
& (c,e, F, g) \in GC_i, \text{ with} \\
& \quad e = (\ell_4, \textsf{assume } a \geq 0, \ell_6), \\
& \quad F = \neg(y \geq 0 \wedge b = 0) \wedge \neg(a\%b = 0 \wedge b \neq 0), \text{ and} \\
& \quad g = y \geq 0.
\end{aligned}
$$

Since all premises from Theorem 4.9 are satisfied, we can deduce that $(y \geq 0)$ *must* be part of the final generalization of c and we don't have to check whether it can be dropped. If we are lucky and additionally an entry exists that gives us an upper bound like $(y \geq 0 \wedge b = 0)$, we effectively only have to check a single literal for dropping, reducing the number of SMT checks in this case by 75%.

Note that we assume the generalized cube g to be minimal. While minimality is obviously desirable, it may come at a very high price. Therefore many modern implementations of IC3 dismiss minimality in the generalization in order to speed up the generalization process. This however does not mean that the proposed method becomes invalid, but rather that we may keep literals that could have been dropped otherwise, i.e. the generalized cube will not be minimal again, which in this setting is acceptable.

When we combine the caching of generalization contexts and the presented methods to derive *lower bounds* and *upper bounds* on the literals to be generalized, we can save a significant amount of effort in generalization. Given a cube c to be generalized, for a set of literals $c_{\lfloor\rfloor}$ obtained as lower bound from Theorem 4.9 and a set of literals $c_{\lceil\rceil}$ obtained as upper bound from Theorems 4.7 and 4.8, we only have to check which literals from $c_{\lceil\rceil} \backslash c_{\lfloor\rfloor}$ can be dropped.

4.3.5 On other generalization techniques

Having presented a set of new and improved techniques for the generalization of cubes in IC3CFA, we will also have a short look at some other generalization methods that have been considered in the literature, in particular *unsatisfiable cores* and *nterpolation.*

Unsatisfiable Cores The first use of *unsatisfiable cores* (short *unsat cores*) for IC3 generalization was already published as a side note to the IC3 algorithm in [Bradley, 2011]. Here the author states that the unsatisfiable core of a successful query for inductivity of cube c relative to F_{i-1} ”can be used to reduce s, often significantly, before applying inductive generalization” [Bradley, 2011]. This in fact enables a significant improvement for bit-level IC3, as well as for IC3CFA. The problem however is, that unsatisfiable cores that can be extracted from the solver are not minimal in any sense, but can be extracted with almost no costs. Therefore unsatisfiable cores are a good starting point for inductive generalization as stated in [Bradley, 2011], but should not be considered final generalizations. From a SAT/SMT solving point of view, we may think about further minimizing unsatisfiable cores, which has been subject to various publications [Bruni, 2003; Gershman et al., 2008; Marques-Silva et al., 2013; Nadel, 2010; Nadel et al., 2013; Oh et al., 2004; Zhang et al., 2006]. However, we will see that the minimization techniques of unsatisfiable cores and IC3's generalization are very closely related to each other.

To demonstrate this, we use the minimization technique from [Guthmann et al., 2016]. The algorithm proceeds as follows: Given a formula φ in CNF, remove an unmarked clause $d \in \varphi$ from φ and check whether the resulting formula is still unsatisfiable. If it is, we keep φ without d and otherwise we mark d and put it back into φ. Afterwards, we check whether all clauses are marked and proceed by either picking another unmarked clause or return φ as minimal unsatisfiable core.

We can easily see that, if we start with a CNF $\varphi = F_{(i-1,\ell)} \wedge T_e \wedge c'$ and already mark all clauses in $F_{(i-1,\ell)}$ and T_e, then we are only left with the

”clauses”, i.e. literal, of c to check. As a result we will linearly iterate over the literals of c and check which can be removed. This exactly resembles the standard linear generalization of IC3.

As such, using simple minimization of unsatisfiable cores is not advisable, since it is not able to fully exploit the structure of our queries, as the methods presented in this section are.

Interpolation Another interesting approach to generalization is the use of *interpolation* for generalization. As mentioned on page 74 in the context of IC3, interpolation is able to deduce generalizations that block a superset of the states of cube c, but are not a subset of c. Thus, we are able to block states on a semantic level, which may speed up convergence considerably and in some cases can be the key to solving the problem at all. One popular example where interpolation can make the difference between running out of resources and terminating successfully is for programs that check loop invariants or loop nondeterministically and check a relation between certain variables after the loop has terminated. For such cases, standard IC3CFA will not converge, since it will enumerate possible runs through the program. With interpolation however, the generalization is able to establish e.g. linear inequalities that enable termination after very few iterations. Such an approach has been published in [Birgmeier et al., 2014] for a different IC3 software model-checking algorithm, but the experimental evaluation of the authors has revealed an overall negative effect of using interpolation on the used benchmark set. We implemented the same approach in our verification framework for the IC3CFA algorithm and were able to reproduce these results. After investigating the reason for the negative impact, we found out that interpolation in general can take a considerable amount of computation time and sometimes will not even terminate within a time frame of 30 minutes. However, this dramatic increase in interpolation time was mostly due to interpolating large formulas, while for small formulas the interpolation time was negligible, but was able to solve some instances that cannot be solved without interpolation. In addition we realised that after the input CFA has been minimized by our static preprocessing, many loops have been minimized into a self-loop in the CFA. We therefore added a check that disables interpolation on non-loop edges, such that we save interpolation calls where they are not as helpful. Using these restricted interpolations, we were able to solve more instances, while at the same time preserving the performance on instances that don’t benefit from interpolation. An experimental evaluation of this approach will be subject of Chapter 5.

4.4 Propagation

The propagation of learnt clauses into subsequent frames is an important aspect of the performance of IC3, as illustrated in Section 3.4. It allows IC3 to learn clause d for frame F_i, based on d's existence in frame F_{i-1} for all $1 \leq i \leq k$ and thus saves the backward-search that may have otherwise led to learn d after many search steps. In Section 3.4 we saw that this *pushing* of clause d into frame F_i is not necessary for the correctness of IC3, but largely benefits the performance since it accelerates the convergence of frames.

Because of this beneficial effect of propagation to IC3, we tried to apply it to IC3CFA as well. In the remainder of this section we will illustrate the challenges and problems that arise with propagation in the IC3CFA setting. To start with, let us shortly reconsider how pushing works in IC3: After checking termination and incrementing the step bound k, we check for each clause $d \in F_i$ at each index $0 \leq i \leq k-1$ whether the implication $F_i \wedge T \Rightarrow d'$ is valid ((3.10) on page 79). If this is the case, then we can *push* d forward into F_{i+1}, since it also holds in the next frame. Since we iterate over all frames starting with F_0, we will consider d in F_{i+1} again, therefore pushing d forward as far as possible. This in turn results in a faster convergence of frames and faster termination.

If we now apply this idea to IC3CFA, we first have to consider the explicit handling of control-flow locations, which splits our step-wise reachability frames F_i into a set of frames $F_{(i,\ell)}$ for each location ℓ in the CFA. Our implication (3.10) then corresponds to the implication

$$F_{(i,\ell)} \wedge T_{\ell\to\ell'} \wedge d' \qquad\qquad \text{for all } \ell' \in succ(\ell). \tag{4.18}$$

If the implication (4.18) is valid, then we can add d to $F_{(i+1,\ell')}$. However, this query in practice often fails, since the edge $e = (\ell, \mathsf{cmd}, \ell')$ often modifies the variable valuations, especially after model minimizations have taken place[5]. A more successful approach to propagating learnt knowledge into subsequent frames is the use of strongest postconditions, which give exact reachability information for $F_{(i+1,\ell')}$ based on the blocked cubes in $F_{(i,\ell)}$. Note that propagating clauses from $F_{(i,\ell)}$ into $F_{(i+1,\ell')}$ using strongest postconditions does not require additional SMT checks since it is always valid to add these. Nevertheless, this approach is also not useful in IC3CFA, since strongest postconditions will change the syntactic structure of the clause, such that our syntactic termination check does not benefit from these. On the contrary, it even slows down termination, since we now have to wait until we can push such a clause from

[5]More on model minimizations in Section 5.1.2.

$F_{(i+1,\ell)}$ into $F_{(i+2,\ell')}$ in order to satisfy the termination criterion again. Changing the termination check into a semantic check is also not advisable, since such a check requires a solver query for each $F_{(i,\ell)}$, for all $0 \leq i \leq k-1$ and $\ell \in G$.

Another problem with propagation in IC3CFA, is that pushing in general does not properly support termination, since they work somewhat orthogonal: For termination, we check whether for all ℓ there exists a frame $F_{(i,\ell)}$ that is equal to $F_{(i+1,\ell)}$. However, using (4.18) we propagate d from $F_{(i,\ell)}$ into $F_{(i+1,\ell')}$. Note the difference between termination checking on the same location and propagation pushing clauses into successor locations. As such, (4.18) does not really *push* d to F_{i+1} and (with delta encoding, see page 69) remove d from F_i, as in standard IC3, but it rather *duplicates* d into $F_{(i+1,\ell')}$ of the successor location.

Due to these reasons, we refrained from using propagation in IC3CFA. Even using more involved optimization strategies for the propagation of clauses, such as [Gurfinkel and Ivrii, 2015; Suda, 2013], we were not able to improve the performance of IC3CFA compared to a configuration without any form of propagation.

Changing the overall handling of frames in IC3CFA may lead to a better performance of propagation, however it is questionable whether this outweighs possible performance drawbacks in the search phase of the algorithm: For example, one might change the current representation of frames for each location and merge them for each index, such that again, we obtain a set of frames $F_0, \ldots, F_k$ and inside these frames we store clauses annotated with their reachability information in the CFA, e.g. $(\ell, d) \in F_i$, which could also be lifted to store clauses for arbitrary sets of control-flow locations, such as $(\{\ell_1, \ldots, \ell_n\}, d) \in F_i$. While this representation of frames would also closely resemble the information that the original IC3 stores and improve the application of propagation, it would largely complicate the handling of the search phase. Since the search phase is to be considered the most crucial phase of each IC3 algorithm, this change of the frame structure does not seem promising.

4.5 Comparison

We started this chapter by illustrating the process of lifting the IC3 algorithm to software model-checking on the example of IC3-SMT and Tree-IC3 [Cimatti and Griggio, 2012]. To complete the presentation of our IC3CFA algorithm, we will have a brief look at a selection of other algorithms for IC3-style software model-checking.

We start with the IC3+IA algorithm of [Cimatti, Griggio, Mover, et al., 2014] which builds upon the work of [Cimatti and Griggio, 2012]. The main motivation of IC3+IA is to integrate IC3 with (predicate) abstraction, in this case *Implicit Abstraction* (IA) as presented in [Tonetta, 2009]. In the interaction between IC3 and IA, IC3 operates purely on the Boolean level of the abstracted state space and discovers inductive clauses over the abstraction predicates. Like IC3CFA, IC3+IA is able to handle a variety of background theories because it does not rely on ad-hoc extensions like quantifier elimination or theory-aware generalization procedures. But in contrast to IC3CFA, the IC3+IA algorithm does not explicitly represent the transition relation but keeps this symbolic. In addition, the use of IA allows IC3+IA to abandon the explicit computation of the abstract system. The algorithm proceeds as follows: Given some set of predicates $\mathbb{P}$, we consider the Boolean state space that is spun by $\mathbb{P}$ and apply standard IC3 on it, with the exception that the transition relation is given in terms of IA in order to avoid quantifier elimination. To check inductivity of a $\mathbb{P}$-cube in this state space relative to a $\mathbb{P}$-frame, [Cimatti, Griggio, Mover, et al., 2014] introduces a modified check that includes the lifting to IA. Due to the use of predicate abstraction, a counterexample found in the search phase of IC3 may be spurious, i.e. it may not be a counterexample on the concrete state space. This situation common to predicate abstraction is resolved by simulating the given abstract counterexample path on the concrete system. If the counterexample also exists in the concrete system, IC3+IA detected a real counterexample. If however the counterexample does not exist on the concrete system, the set of predicates must be refined, just like in other predicate abstraction settings. Based on this new set of predicates the IC3 is executed again. A distinct feature of IC3+IA is that it only adds new predicates to $\mathbb{P}$, such that $\mathbb{P}$ monotonically increases, which allows IC3+IA to keep all learnt clauses and rather than restart IC3, it can just be continued. In [Cimatti, Griggio, Mover, et al., 2014] the authors also evaluated a combination of Tree-IC3 and IA, but while being more efficient than Tree-IC3 with interpolation (the best configuration from [Cimatti and Griggio, 2012]) it is outperformed by IC3+IA.

Another abstraction based approach for IC3 software model-checking called *Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)* is presented in [Birgmeier et al., 2014]. The CTIGAR algorithm also maintains a set of predicates $\mathbb{P}$ that abstract the concrete state space. But where IC3+IA only refines $\mathbb{P}$ whenever a spurious counterexample trace has been found in the abstract state space, CTIGAR triggers refinement of $\mathbb{P}$ over single-step queries in two situations: A so-called *lifting failure* occurs whenever an abstract cube c is not inductive, i.e. (3.2) (see page 59) fails. When this happens, IC3 extracts the state s from the solver model, as well as an assignment z to the primary inputs, and tries to lift the full assignment s to a partial assignment. The corresponding query $s \wedge z \wedge T \wedge \neg c'$ [Chockler et al., 2011] is satisfied by default in IC3, however, for the abstracted version $\widehat{s}$ of s, it may fail, due to states in $\widehat{s}$ that have $\neg\widehat{c}$-successor states. The second situation in which predicate refinement must take place in CTIGAR is when the abstract cube $\widehat{c}$ is not inductive relative to some frame F_i, i.e. the abstract consecution fails, but the corresponding concrete consecution succeeds. Such a situation is called *consecution failure* and is triggered when the concrete cube c does not have F_i-predecessors, but by abstracting it to $\widehat{c}$, the abstraction includes a successor to some F_i-state. Like other predicate abstraction algorithms, CTIGAR uses interpolants for refinement if they are available in the respective theory. However, when a *consecution* or *lifting failure* occurs, CTIGAR may not eagerly refine whenever an abstraction failure occurs, but may choose a lazy approach and only refine later. In fact, the experimental evaluation in [Birgmeier et al., 2014] revealed that a lazy refinement approach is in practice the best performing in terms of solved benchmarks, as well as in cumulative time (except for one configuration which solves 6 instances less).

There also exist other approaches to software model-checking with IC3 that are not theory-independent like IC3+IA or CTIGAR. Such an approach, called *Generalized Property Directed Reachability (GPDR)*, is presented in the eponymous paper [Hoder and Bjørner, 2012]. The main focus of GPDR is the lifting of IC3/PDR to *nonlinear fixed-points* that become important when considering procedure calls. Such nonlinear predicate transformers can be represented as general *Horn clauses*, which are clauses where *at most one* literal is *unnegated*. Additionally, in [Hoder and Bjørner, 2012] GPDR is lifted to use *Linear Real Arithmetic* (LRA, see Section 2.1.2). This restrictive representation is tailored towards *timed push-down systems* where it is shown to perform much better than competing approaches.

A second approach that is tailored towards a specific representation and class of programs is [Welp and Kuehlmann, 2013] which applies IC3/PDR to the domain of quantifier-free bit-vectors (QFBV, see Section 2.1.2). This allows the algorithm to reproduce the exact bitprecise program behaviour including over-/underflows of variables. In addition, since all variables are bit-vectors, they can also be expanded to a purely Boolean representation, which is a known technique called *bit-blasting*. As such, [Welp and Kuehlmann, 2013] are basically able to use the standard IC3 algorithm for all bit-blasted variables.

Chapter 5

Experimental Results

Following the presentation of the IC3CFA algorithm and the generalization, as well as our improvements to the generalization in Chapter 4, we have implemented all presented techniques on top of an existing, proprietary verification framework. The remainder of this chapter is structured into two parts: The implementation of IC3CFA in Section 5.1 and the evaluation in Section 5.2. We will start Section 5.1 with a more detailed description of the architecture of the framework in Section 5.1.1 to illustrate the way that our verification framework is constructed and which components interact in order to efficiently verify different input programs. Section 5.1.2 will illustrate some of the employed preprocessing methods that can have a large effect on the performance of IC3CFA, in particular static analyses and the corresponding model minimizations to reduce the size of the CFA that we use as input for the actual verification algorithm. In Section 5.1.3 we will give some more insight on specific implementation details of the individual techniques that were presented in Chapter 4. Section 5.2 starts with a presentation of the setup and configurations under which the benchmarks have been executed (in Section 5.2.1). In Section 5.2.2 we will discuss the results of executing our IC3CFA implementation and evaluate benefits and drawbacks of the respective techniques. We conclude this chapter with some industrial experiences in Section 5.2.3.

5.1 Implementation

In Chapter 4, we have introduced multiple new concepts for the verification of software using the IC3CFA algorithm, including a detailed look at how to lift the generalization of IC3 to software and how to improve it in the presence of explicit control-flow and exact pre-images. While this part focussed on the theoretical aspects of these methods, we will now consider the implementation aspects of IC3CFA and illustrate important details when implementing IC3CFA whenever they are not obvious from the descriptions in Chapter 4.

5.1.1 Architecture

As mentioned before, we implemented all contributions on top of an existing proprietary framework for software verification (in the remainder: *the verifier*). Since we cannot publish any source code of the verifier and its IC3CFA implementation, we will sketch the architecture and flow of the verifier and its respective parts. Figure 5.1 shows an architecture diagram for the verifier. As usual for model checkers (see Section 2.3.1), we take requirements in form of assertions and initial conditions as input and combine them into a formal specification. We also take an input program, in form of a C file and parse it to translate it into our custom *intermediate verification language* (short, IVL). Using a custom intermediate language enables us to implement a modular front-end for input languages, such that we can easily adapt the verifier to new input languages without the necessity to change any code in the back-end. Because our IVL is Turing-complete, we are able to translate arbitrary languages into IVL. Based on such IVL code and the given specification, we first apply a model minimization (see Section 5.1.2) and, based on the minimized model, construct the CFA of the resulting program. Afterwards, we minimize the CFA using *Large-Block Encoding* [Beyer, Cimatti, et al., 2009]. The resulting, minimized CFA will be the input to the verification algorithms, which can be chosen at run-time via command line arguments. Besides the IC3CFA algorithm, which was the topic of Chapter 4, the verifier also contains a bounded model-checking (BMC) and a Counterexample-guided abstraction refinement (CEGAR) engine. Independent of the choice, all verification engines will access the same solver interfaces, which decouples the different solvers, in our case we offer Z3 [Moura and Bjørner, 2008], CVC4 [Barrett et al., 2011] and MathSat5 [Cimatti, Griggio, Schaafsma, et al., 2013].

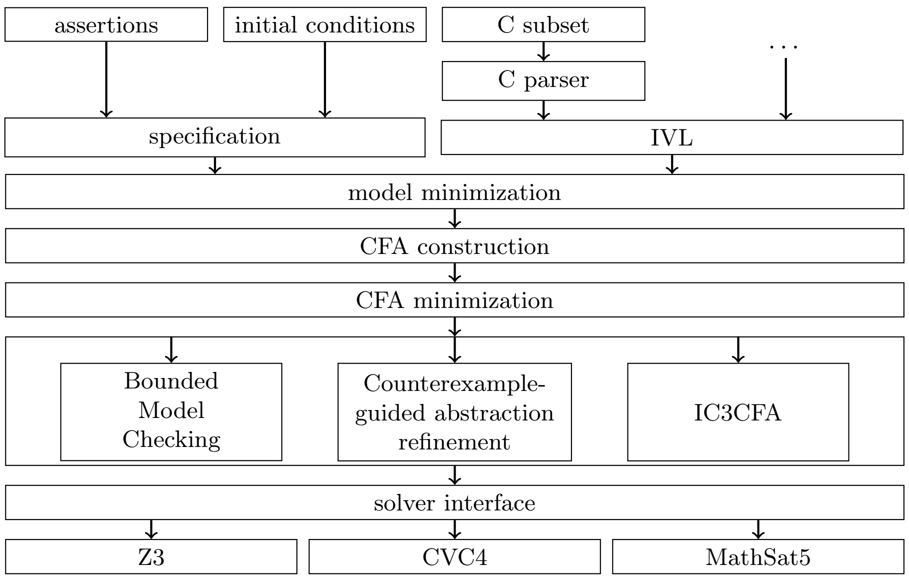


Figure 5.1: Architecture diagram of the information flow through our verification framework

5.1.2 Preprocessing

As seen in Figure 5.1, the input model given in IVL is subject to some preprocessing, in order to minimize the model, before it is given to the verification engine. This preprocessing consists of two different phases: The static minimizations on IVL code and the CFA minimization. The first minimizations, that are executed on the IVL code consist of a number of static program analyses, such as *initialized variables*, *needed variables* or *reaching definitions* analysis [Nielson et al., 1999], *Steensgaard's pointer* analysis or *partition refinement* (on the locations of the CFA), and the corresponding program transformations, like forward propagation [Lange et al., 2013], program slicing, pointer resolution and bisimulation minimization. These optimizations allow us to handle simple pointer programs, simplify expressions and slice away unnecessary parts of the program. However, these transformations are very likely to influence the results of some analyses, such that we have to execute the analyses and transformation again, until we reach a fixed-point.

For the CFA minimization we use the so-called *Large-Block Encoding* (LBE) [Beyer, Cimatti, et al., 2009]. The LBE algorithm applies three simple rules of which two are again applied in a fixed-point approach: (1) the *error sink* rule removes all outgoing edges of the error location ℓ_E. Since this rule can only be applied once, it can be considered as a pre-processing rule for the following two rules. (2) the *sequence* rule merges consecutive edges into a single edge, such that edges $e_1 = (\ell_1, \mathsf{cmd}_1, \ell_1')$ and $e_2 = (\ell_2, \mathsf{cmd}_2, \ell_2')$ with $\ell_1' = \ell_2$ are replaced by a new edge $e = (\ell_1, \mathsf{cmd}_1; \mathsf{cmd}_2, \ell_2')$. (3) the *choice* rule merges parallel edges, i.e. $e_1 = (\ell_1, \mathsf{cmd}_1, \ell_1')$ and $e_2 = (\ell_1, \mathsf{cmd}_2, \ell_1')$ are replaced by a single edge $e = (\ell_1, \mathsf{cmd}_1 \,\square\, \mathsf{cmd}_2, \ell_1')$ where the nondeterministic choice of the CFA between e_1 and e_2 is represented by the choice operator $\square$ on GCL level. Rules (2) and (3) are executed in a loop until the CFA reaches a fixed-point. However, our experiments have shown similar results as in [Beyer, Keremoglu, et al., 2010], i.e. minimizing the CFA until it reaches a fixed-point may sometimes not be optimal for the performance of subsequent verification engines. In our approach we implemented thresholds on the size of the formula representing the semantics of the command and stop modifying an edge whenever it cannot be modified using rules (2) and (3) or when the threshold is exceeded. We then terminate whenever no edge can be modified any more. This approach yields the best performance for IC3CFA, as well as for the CEGAR engine.

5.1.3 Implementation details

In the remainder we will consider some details that are not relevant for the theoretical considerations of Chapter 4, but become important when implementing IC3CFA.

All core parts of the verifier are written in the functional OCaml language. On the one hand writing the verifier in a functional language comes in handy in many cases, where the theory is naturally recursive, e.g. the fixed-point approaches described in Section 5.1.2. On the other hand, when we adhere to write tail-recursive functions, the OCaml compiler is able to translate the code into efficient machine code that can keep up with other native code implementations.

Cubes and frames As we already discussed in Section 4.2, the encoding of frames in the original IC3 algorithm [Bradley, 2011] is not very efficient and the delta-encoding of PDR [Eén, Mishchenko, et al., 2011] leads to a dramatic performance improvement. Since we store not a single frame sequence but $|L_P|$ sequences, with $L_P = L \setminus \{\ell_E\}$, the effect of delta-encoding is even more crucial.

One of the main differences between the theoretical aspect of IC3CFA and the implementation is in the main data structure: While on a theoretical level cubes and clauses are dual and one results from negating the other, an implementation representing this duality will struggle with circular dependencies, i.e. negation of a cube will return a clause and vice versa. However, we do not have to offer explicit implementations for both of them. Instead, we keep clauses implicit as long as possible by using their dual cube. As such, a frame is implemented as a set of cubes that implicitly represents the conjunctions of the clauses that arise from negating the cubes. Whenever we have to solve a query that involves the frame formula, we have to determine it anyway, due to delta-encoding. Assume we want to determine the frame formula for $F_{(i,\ell)}$, we have to iterate over all entries j with $i \leq j \leq k$ and collect the cubes in $R_{(j,\ell)}$. During these iterations, we can easily add the negation to all collected cubes and in the end conjoin all negated clauses to create the correct frame formula. Note the difference between the entries $R_{(i,\ell)}$ and the frame $F_{(i,\ell)}$, where $F_{(i,\ell)} = \bigwedge R_{(j,\ell)}, i \leq j \leq k$ according to delta-encoding [Eén, Mishchenko, et al., 2011]. In contrast to the standard IC3, which stores a vector of such F/R, the lifting to IC3CFA introduced a second dimension of frames with respect to the locations $\ell \in L_P$. As a consequence, we do not have to store a vector of size k, but rather a matrix of size $k \times |L_P|$. While an imperative programming language may offer the possibility to implement this matrix directly, allowing random access to all entries, we are not able to properly implement this in OCaml. The solution here is to use a nested map, which naturally leads to the question which dimension to use in the inner/outer map, because once we have mapped the outer dimension, we can easily iterate over entries in the inner map. However, for IC3CFA we iterate over the index, e.g. for collecting frame entries in delta-encoding, as well as over the location, e.g. the termination check where we check whether for some i whether $F_{(i,\ell)} = F_{(i+1,\ell)}$ for all $\ell \in L_P$. Because frame formulas will be created much more often than the termination check happens, we implemented frames as a nested map $map_{idx} \circ map_{loc}$ with $map_{idx} : \mathbb{N} \mapsto Map_{loc}$ and $map_{loc} : L_P \mapsto R$.

Apart from avoiding a tedious circular data structure, we can easily check for predecessor cubes as presented in Theorem 4.6 on page 136, as a set membership of the corresponding frame entries.

SMT solvers As seen in Figure 5.1, the verifier uses a modular approach that allows easy extension to other solvers by using a solver interface that encapsulates the actual solver. While this allows us to determine the used solver during runtime and also allows the verification algorithms to hotswap solvers in case one does not offer the specific functionality, e.g. interpolation, it also requires a custom formula representation to decouple from the different formula representations of each solver. Using efficient techniques, such as hashconsing, we are able to store formulas in our own formula implementation and translate them to the solver API without too much overhead.

In Chapter 4 we often discussed that the exact predecessor computation using WEP leads to many redundant cubes. While these do not introduce any overhead in the formula representation, we might execute many redundant SMT checks with these repeating cubes. In order to save computation time, our IC3CFA implementation adds another layer on top of the solver interface, which internally caches the formulas that we assert and stores the result from the solver. Since we allow the verifier to dynamically set the solver timeouts, depending on whether it is a critical or non-critical query, we have to consider the timeout in caching. A typical example for a critical query is an inductivity query in the search phase, i.e. if we do not know whether predecessors exist or not, we cannot continue the search phase, while inductivity queries in the generalization are generally considered non-critical and a timeout can be considered as failed dropping attempt, i.e. we just keep the literal, which may lead to a slightly less general result, but does not compromise the correctness of the result. An instance of such a so-called *CachedSolver* will offer all functionality of a standard solver, but internally cache the result for this formula and timeout. Whenever the same formula is checked with a timeout less or equal to the stored timeout, it will return the cached result and for longer timeouts, it will check the formula again. Apart from caching SMT queries, a *CachedSolver* will also cache interpolation calls.

Caching For caching, our implementation uses *Least-Recently-Used (LRU) caches* in almost all situations. For all presented caches, such as generalization contexts or CachedSolvers, the number of keys in the map is infinite, such that caching all results ever seen will quickly pollute the memory and will exceed the allocated resources, such that execution is terminated with a *Memout* error. To prevent the verifier from quickly wasting its resources, the LRU caches allow to limit the memory used for the cache and at the same time only keep the least recently used entries, which are the ones that are most likely to reappear anyway.

Counterexamples An important aspect that is not considered in [Bradley, 2011] or [Eén, Mishchenko, et al., 2011] is the extraction of counterexamples, whenever the property is proven to be violated. To do so, we extended proof obligations o with an *id* and a *parent id* that stores the id of the obligation o_p that caused the creation of o. Using these entries we can reconstruct the counterexample path from the CTI down to the obligation at level 0. To check that no id is assigned twice, we use the obligation queue as a singleton that stores and hands out new ids. This combination of id and parent is of course due to the limitations given by OCaml, where an imperative implementation would simply store pointers to the corresponding parent obligation.

Inductivity queries over multiple edges Section 4.3 illustrates how we can generalize a cube with respect to multiple edges. The situation is slightly easier for relative inductivity, as shown in Algorithm 13 on page 105: A cube c at location ℓ' is inductive relative to its predecessors, iff it is inductive relative to $F_{(i-1,\ell)}$ for all $\ell \in pred(\ell')$. In Algorithm 13, we omit this general check and rather check each edge individually. However, for our implementation we encapsulated inductivity queries in a separate module which offers a number of similar but slightly differing implementations for inductivity queries. (1) The most atomic function that is also used by many of the subsequent functions, called *check_single_edge* is the one that represents the check in Algorithm 13 and checks inductivity with respect to a single edge. (2) The function *check* represents the opposite extreme, i.e. it uses a single query to check inductivity with respect to all incoming edges at once. Whenever a cube is inductive with respect to all edges, this query is preferable. However, for the search phase of IC3CFA, the simple result *non-inductive* is not helpful since we need to know which edges violate the inductivity and have to be considered in more detail with new proof obligations. (3) The function *check_by_filter* solves this problem by encapsulating the approach of Algorithm 13: It checks each edge individually using function (1) and return just those edges that violate inductivity, i.e. it filters all edges to which cube c is inductive. (4) To find a balance between (2) and (3), the function *check_and_filter* tries to take the best from both worlds: It starts with the check of (2) and if that fails it uses (3) to find those edges that violate inductivity. All these functions serve slightly different purposes and are applied in different parts of the IC3CFA algorithm.

5.2 Evaluation

After presenting our IC3CFA algorithm in Chapter 4 and its implementation details in Section 5.1.1, we will evaluate the performance of IC3CFA and the proposed improvements in detail.

To ensure a fair comparison, we will use a tool called *benchexec* which is developed by the organizers of the *Competition of Software Verification* (SV-COMP) [Vojnar and Beyer, 2018]. The main reason why we use benchexec is that is enables truly reproducible results due to exhaustive resource management, such as allocation of processor cores and memory to prevent other applications, even operating system service from interfering with the allocated resources for benchmarking. Furthermore it monitors and logs the consumed memory and CPU time and can even compare the output of the verification tool with the expected result for the respective input file. All these information, as well as all output of the verification tool can be processed with a small tool called *table-generator* that is shipped with benchexec yielding an html file with a customizable table of all results and corresponding plots.

5.2.1 Setup

For executing our benchmarks, we use a host system with Intel® Xeon E5-2670 CPUs with a frequency of 2.3Ghz and a total memory of 64GB running Debian version 9.4 64bit with Linux kernel version 4.9.0 release 5 and using benchexec version 1.17-dev. All configurations that we executed were allowed to use one core per instance for a time of 3600 seconds with a maximum memory of 3000MB before being aborted with a time-out or memory-out (MemOut), respectively.

To properly evaluate the effect of all our proposed techniques, we have implemented a number of command line options that allow us to enable, disable or modify each technique individually. The following options are available:

Generalize: Possible values are {**None(N)**, **Old(O)**, **New(W)**}, where generalization is disabled by *None*; *Old* uses the standard generalization of [Bradley, 2011] lifted to SMT atoms as literals and *New* uses our improved generalization procedure with all techniques that are activated (and don't care generalization as presented on page 122 activated by default).

Edge approach: Possible values are {**Multi(M)**, **Single(S)**, **Incremental(I)**}, where *Multi* generalizes a cube with respect to all edges at once, *Single* breaks it down into separate generalizations with respect to each edge and unites the results and *Incremental* is similar to *Single*, but incrementally refines a set of necessary literals while computing generalization along all edges individually.

Obligation order: Possible values are {**Smallest(S)**, **Largest(L)**, **Random(R)**}, where *Smallest* sorts the cubes ascending based on their cardinality, *Largest* sorts descending and *Random* sorts them in an explicitly randomized order.

Reuse: This **bool** option enables the reuse of obligations after IC3CFA proceeds to the next iteration of k when set to *true* and disables it when set to *false*. For more information on obligation reuse see page 112.

RelIndAlt: This **bool** option enables the alternative relative inductivity approach as described in Theorem 4.3 when set to *true* and disables it when set to *false*.

PreCubes: This **bool** option enables the static generalization based on subset of the *wep* if a subcube is contained in the respective frame when set to *true* and disables it when set to *false*.

Generalization Caching: This **bool** option enables the caching of generalization in generalization contexts and uses them for upper bounds on the literals of the final generalization when set to *true* and disables it when set to *false*.

Lower Bounds: This **bool** option enables the additional use of generalization context for lower bounds on the literals of the final generalization when set to *true* and disables it when set to *false*.

Extended Context: This **bool** option enables the additional search for subcubes in the generalization context if no direct hit can be found when set to *true* and disables it when set to *false*.

Interpolation: This **bool** option enables the additional interpolation of the cube during generalization when set to *true* and disables it when set to *false*.

Semantic Blocking: This **bool** option enables an additional, semantic check whether the cube to be blocked is already subsumed by the frame it is blocked in when set to *true* and disables it when set to *false*.

Based on these configurations, we created a set of different configurations that aim to evaluate the effects of different techniques. An overview of all configurations is given in Table 5.1.

All listed configurations have been evaluated on a benchmark set consisting of a total of 406 C files. For this benchmark set, we chose the following subsets: We took 99 C files from the benchmark set used for evaluation in [Cimatti and Griggio, 2012]. We chose this set because the contained programs represent interesting properties that challenge the verification engine, rather than just the static minimizations or the SMT solver. We removed some programs from the set used in [Cimatti and Griggio, 2012] to avoid duplicates with other chosen benchmark sets. In addition, we used the ample set of benchmarks from SV-Comp [Vojnar and Beyer, 2018] that are freely available from the SV-Comp homepage. Since these sets aim for many different verification goals, such as safety, memory safety, concurrency or termination, we focus on the *ReachSafety* category, which contains only benchmarks that check reachability of bad states. From this category, we selected the sub-categories *ReachSafety-BitVectors*, *ReachSafety-ControlFlow*, *ReachSafety-Floats* and *ReachSafety-Loops*. This excludes the sub-categories ReachSafety-ECA, ReachSafety-ProductLines and ReachSafety-Sequentialized, because the structure of these programs can not easily be represented and processed by our front-end leading to timeouts that are not caused by the verification. Furthermore our selection excludes the category ReachSafety-Heap which aims at the verification of dynamic data-structures on the heap, which can not be expressed in our IVL. A similar situation leads to the exclusion of the programs from the ReachSafety-Recursive sub-category, since we do not support verification of recursive functions and inline all function calls, the inlining in the preprocessor will diverge without ever starting the verification engine. According to the specifications of SV-Comp, all programs are verified against reachability of the error location and all programs are verified against a 32-bit architecture.

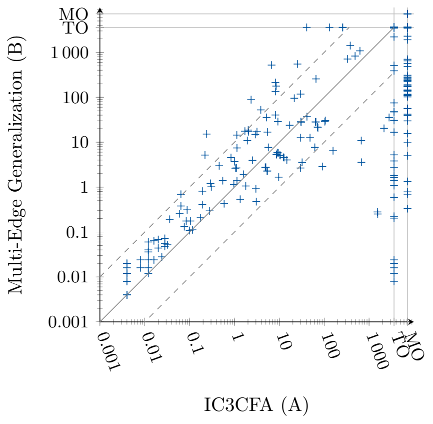
5.2.2 Results

In the remainder of this section, we will present a detailed evaluation of all techniques presented in Chapter 4, as well as implementation tweaks presented in Section 5.1. For this purpose we will mainly use two types of plots: Scatter plots, like the one in Figure 5.2b, give a detailed comparison of the distribution

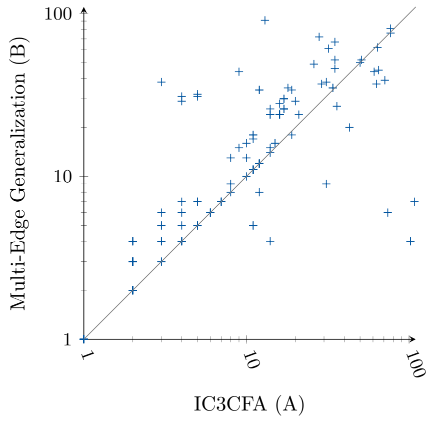
	Generalize	**Edge approach**	**Obligation order**	**Reuse**	**RelIndAlt**	**PreCubes**	**Generalization Caching**	**Lower Bounds**	**Extended Context**	**Interpolation**	**Semantic Blocking**
A	N		S	F	F	F	F	F	F	F	F
B	O	M	S	F	F	F	F	F	F	F	F
C	O	S	S	F	F	F	F	F	F	F	F
D	O	I	S	F	F	F	F	F	F	F	F
E	O	I	L	F	F	F	F	F	F	F	F
F	O	I	R	F	F	F	F	F	F	F	F
G	O	I	S	T	F	F	F	F	F	F	F
H	W		S	T	F	F	F	F	F	F	F
I	W		S	T	T	F	F	F	F	F	F
J	W		S	T	F	T	F	F	F	F	F
K	W		S	T	F	F	T	F	F	F	F
L	W		S	T	F	F	T	T	F	F	F
M	W		S	T	F	F	T	T	T	F	F
N	W		S	T	F	F	F	F	F	T	F
O	W		S	T	T	T	T	T	F	F	F
P	W		S	T	T	T	T	T	F	T	F
Q	W		S	T	T	T	T	T	F	F	T

A = IC3CFA | B = IC3-Gen Multi | C = IC3-Gen Single | D = IC3-Gen Incremental | E = IC3-Gen Order Largest | F = IC3-Gen Order Random | G = IC3-Gen Reuse | H = New-Gen Don't care | I = New-Gen RelIndAlt | J = New-Gen PreCubes | K = New-Gen Upper Bounds | L = New-Gen Lower Bounds | M = New-Gen Extended Context | N = New-Gen Interpolation | O = New-Gen All | P = New-Gen All Interpolation | Q = New-Gen All Semantic Blocking

Table 5.1: Benchmark configurations



(a) Verification time

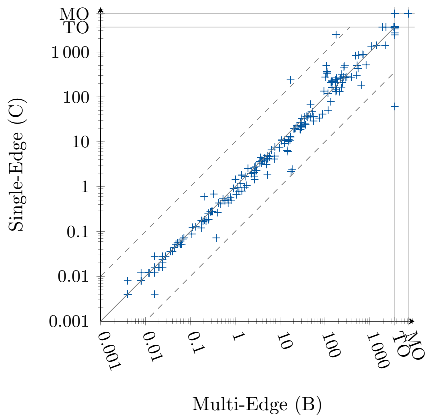


(b) Iterations

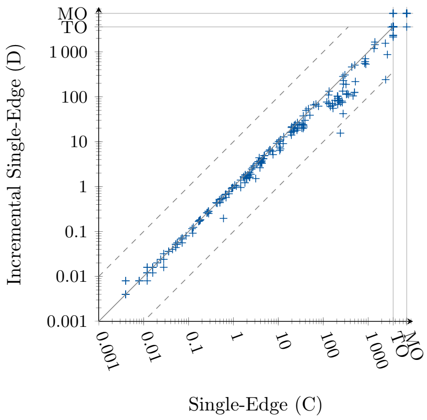
Figure 5.2: IC3CFA vs. Multi-Edge Generalization

of individual results from one configuration against another. Each mark in the plot resembles one input program and is positioned on the x-axis based on the result from the respective x-configuration and on the y-axis based on the result from the y-configuration. As a result, scatter plots can be read as follows: marks in the upper left half are in favor of the x-configuration, and vice versa for the lower right half. For scatter plots representing verification times, we introduce three additional components: dashed lines indicate a factor of 10 on the logarithmic axes and dedicated TO/MO lines reveal timeouts and memory-outs, respectively. The second type of plot that we use is called *quantile plot* or sometimes also referred to as *cactus plot* because of their similarity to the branches of large, treelike cacti. While less precise in the details, quantile plots can offer a good impression of the performance of multiple tool executions in one plot, in contrast to scatter plots which only allow comparing exactly two executions. As such, quantile plots are widely used to visualize the results of the SV-Comp [Vojnar and Beyer, 2018].

Generalization Since a detailed evaluation of the performance of IC3CFA against comparable implementations was given in [Lange et al., 2015], we start with the evaluation of the naive generalization approach over multiple edges, as presented in Lemma 4.7, against the IC3CFA approach of [Lange et al.,



(a) Single-Edge Generalization

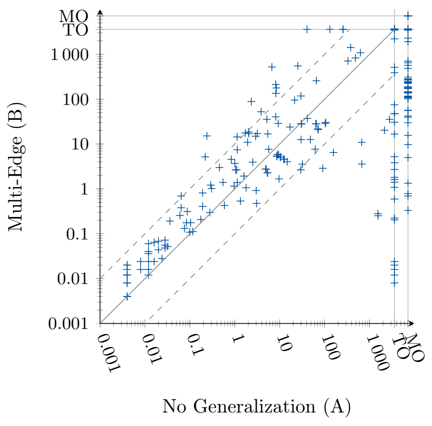


(b) Incr. Single-Edge Generalization

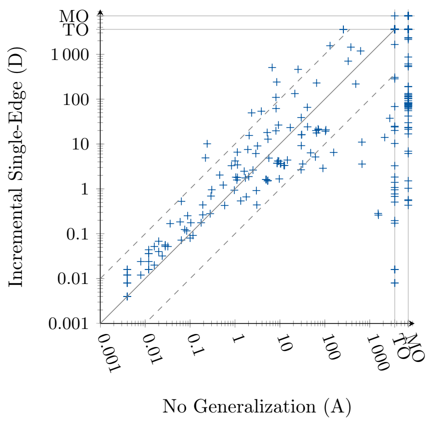
Figure 5.3: Verification times of different generalization approaches

2015] which does not contain any generalization. The comparison of verification times depicted in Figure 5.2a shows an indefinite situation. While there is a large number of instances for which IC3CFA runs into a timeout, for the majority of cases that can be solved by both configurations, IC3CFA is slower with generalization than without. To investigate the details for this drop in performance, we can have a look at Figure 5.2b, which depicts the number of iterations for instances that can be solved by both configurations. Here we see that the number of iterations rises for almost all instances, which is caused by the exact predecessor computation using WEP. Without generalization, IC3CFA constructs an exact, symbolic representation of the backwards reachable states. However, when using generalization, IC3CFA will start with a very coarse over-approximation and iteratively refines it until finally reaching the exact state set. This clearly contrasts standard IC3, which is able to include other reachable states in the over-approximation and therefore saves subsequent iterations. While Figure 5.2b gives the impression that generalization in IC3CFA is self-defeating, the combination with Figure 5.2a should rather show that generalization has large potential to solve additional benchmarks, but the used form of generalization does not fit well with IC3CFA.

To improve the generalization, we presented Corollary 4.1 on page 118, which breaks the large inductivity query down into smaller parts for single edges. The



(a) Multi-Edge Generalization

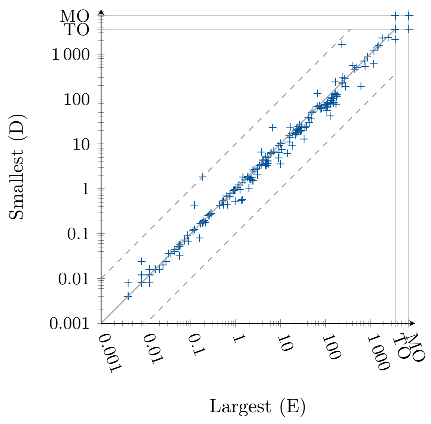


(b) Incr. Single-Edge Generalization

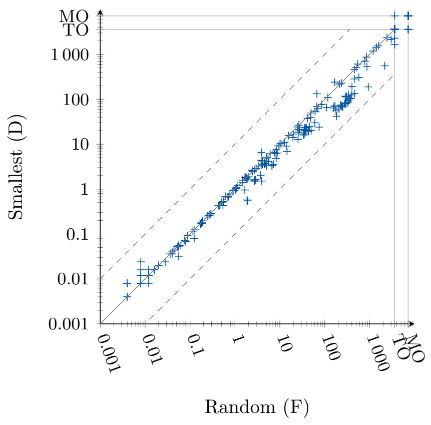
Figure 5.4: Comparison of generalization approaches

results depicted in the scatter plot in Figure 5.3a support our idea that breaking down large inductivity queries into a number of smaller queries improves the performance noticeably, with the exception of some outliers with very long verification time. However, we also presented an approach that implements IC3's principle of breaking large monolithic methods into smaller, incremental steps, by using the results of the previous generalizations along incoming edges of the same location to minimize the number of literals that have to be checked for dropping. This approach sketched on page 118 shows another performance improvement for the generalization of IC3CFA, as depicted in Figure 5.3b. The scatter plot shows that for all instances of reasonable size (verification time larger than 0.1 seconds), the verification time drops by a factor of up to one order of magnitude.

Note that all presented approaches just differ in the way that edges in the CFA are handled inside the generalization. The core generalization approach in all of these configurations is still just lifting of IC3-style generalization to theories. To re-evaluate the performance of generalization, we show the scatter plots of IC3CFA without generalization vs. multi-edge generalization (copy of Figure 5.2a) and IC3CFA without generalization vs. incremental single-edge generalization next to each other in Figure 5.4. While Figure 5.4a shows three timeouts introduced with generalization and a tendency towards worse verifica-



(a) Sorting cubes based on cardinality

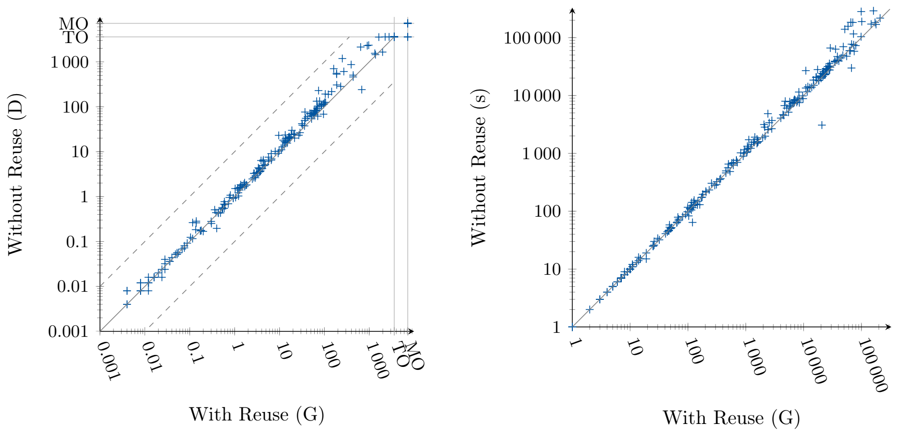


(b) Cardinality-based and random cube order

Figure 5.5: Comparison of different obligation orders

tion time for commonly solved instances, the number of new timeouts can be reduced to one with incremental single-edge generalization and the performance for commonly solved benchmarks is now more equally spread around the diagonal. Since incremental single-edge generalization has proven to be the most efficient approach for lifting generalization to CFAs, all subsequent results will use the incremental single-edge approach.

Obligation ordering and reuse Next, we want to evaluate another technique that became necessary when lifting IC3 to CFAs. As illustrated on page 137 the order in which multiple cubes at one location are considered arises with the introduction of DNF splitting of WEP formulas. To evaluate which order works best, we implemented three different strategies: *largest* orders cubes based on their cardinality in descending order, i.e. the largest cube is considered first, while *smallest* also sorts cubes based on cardinality, but in ascending order. For comparison, we also implemented the strategy *random*, which shuffles the cubes. The results are depicted in Figure 5.5 and confirm the results of [Gurfinkel and Ivrii, 2015] that ordering obligations in ascending cardinality of their cubes yields the overall best results. While the performance gain is obviously not big, the implementation effort is neither. Since for most data



(a) Verification time

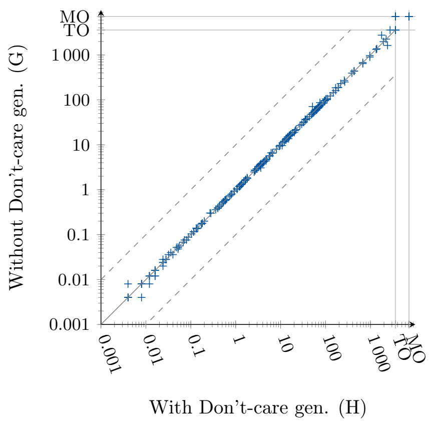
(b) #SMT queries

Figure 5.6: Evaluation of obligation reuse

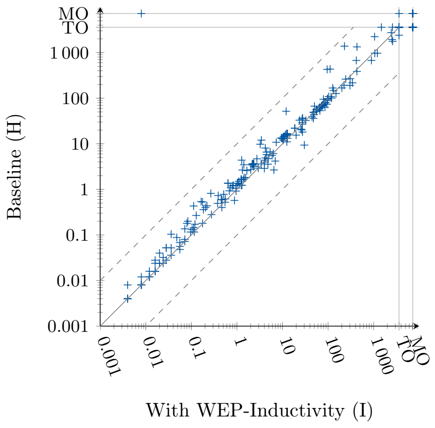
structures sorting is already implemented in the standard interface, this usually introduces at most a single line of new code.

Let us now evaluate the isolated performance effect of obligation reuse, which keeps obligations of previous iterations and reuses them to proceed the backward search where the previous iteration was capped due to the depth bound k. In Figure 5.6a, we can see that, except for three outliers, the verification time is not only improved throughout the whole spectrum, but the advantage of enabling obligation reuse grows proportional with the verification time, the reason being that with the size of the search depth k, the amount of obligations that are recomputed without obligation reuse grows, too. Thus for long counterexamples or hard properties, the amount of SMT calls, that can be saved due to obligation reuse is large, as we can see from Figure 5.6b, which depicts a scatter plot of the total number of SMT calls for each instance.

Don't care detection Just like the obligation ordering, the don't care detection as presented on page 122 does not cause a massive performance gain, but is also not much of an implementational and computational overhead. While hard to see in the scatter plot in Figure 5.7a, the use of don't care detection has a marginally positive effect on almost all benchmarks, except for a few negative outliers with execution time below 0.01 seconds, therefore negligible and one



(a) Don't care Generalization

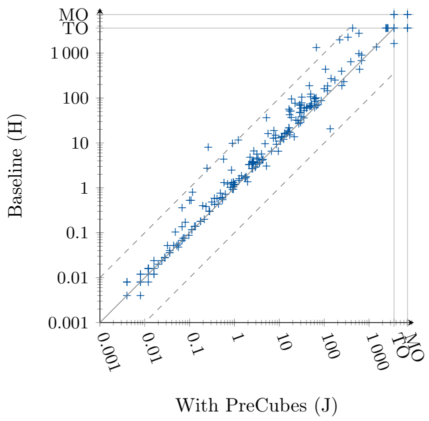


(b) WEP-based inductivity

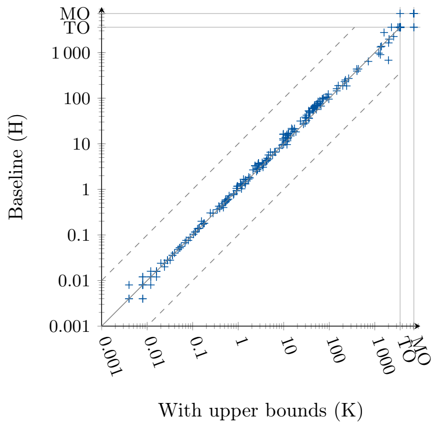
Figure 5.7: Evaluation of Don't-care detection and WEP-based inductivity

strong positive outlier with execution time above 100 seconds. For this reason, we consider don't-care generalization a small, but useful optimization. To ensure a fair evaluation of each of the presented improvements in Section 4.3, we will use this configuration, i.e. incremental single-step generalization, smallest cube obligations first, obligation reuse and don't-care generalization enabled. We will use this configuration (H in Table 5.1) as *Baseline* for the subsequent evaluation.

WEP-based inductivity Figure 5.7b depicts the evaluation of the WEP-based inductivity query as shown in Theorem 4.3 on page 124. As we can see, the results are not equally good for all inputs. In fact, Figure 5.7b shows that WEP-based inductivity is favorable for small and medium hard inputs with verification times up to around 100 seconds, where the results start to become indefinite with many instances performing slightly worse with WEP-based inductivity, but also some instances performing much better and one instance being almost one order of magnitude faster with WEP-based inductivity. In addition we can see from Figure 5.7b that WEP-based inductivity is able to solve one instance shortly before the timeout that ran into a timeout without WEP-based inductivity. Due to the large performance gains for many instances and the marginal performance deterioration for some, we consider WEP-based inductivity in total as a helpful



(a) Predecessor Cubes for Generalization



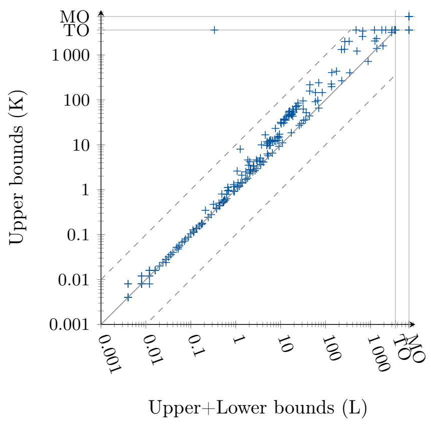
(b) Generalization Context (upper bounds)

Figure 5.8: Evaluation of Predecessor cubes and Generalization Contexts for upper bounds

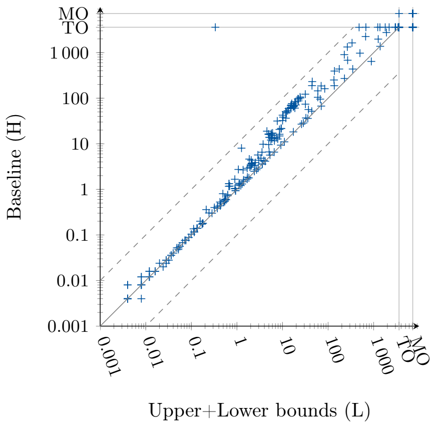
technique.

Predecessor cubes Next, we evaluate the isolated effect of predecessor cubes as presented on page 136, by comparing the *Baseline* configuration with one that is identical except that the static predecessor cube generalization is enabled. The resulting scatter plot is depicted in Figure 5.8a and nicely illustrates how much performance can be gained by static improvements. Except for one noticeable outlier, almost all other instances can be verified faster with two instances even more than one order of magnitude faster with predecessor than *Baseline*. Furthermore we can see that using predecessor cubes, IC3CFA is able to solve four instances that run into a timeout in *Baseline*.

Generalization context As presented in Section 4.3, the generalization context (Definition 4.11 on page 138) can be used in several ways. We start with the simple version as shown in Theorem 4.7 where we just check whether a subset of the current frame has been stored in the context and use the result as upper bound on the literals of the new generalization. As we can see from the corresponding scatter plot in Figure 5.8b, the effect is negligible for easy



(a) Isolated



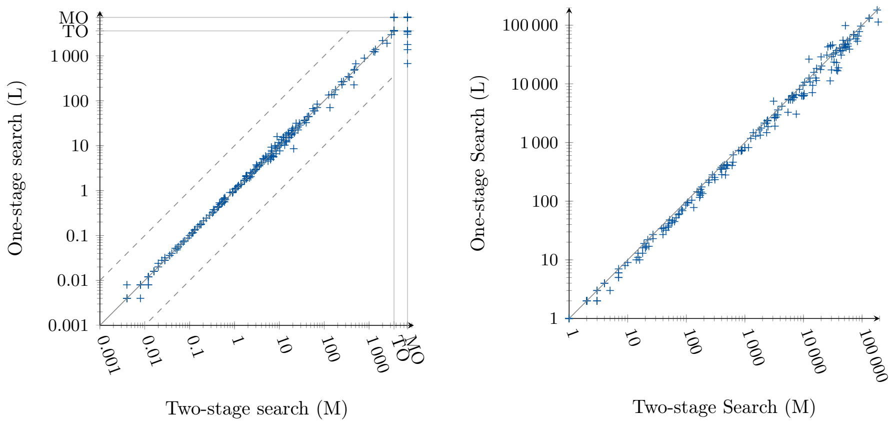
(b) Upper + Lower Bounds

Figure 5.9: Evaluation of generalization contexts with lower bounds isolated and in combination with upper bounds

instances and the curve gets a slight bump for harder benchmarks but converges to the diagonal near the timeout. This effect is justified by the overhead introduced by managing and filling the cache and searching all the cache entries. For small/easy instances, the overhead of filling and managing the cache compensates the benefits given by the few cache hits. The larger/harder the instances get, the more cache hits occur and outweigh the costs of managing the cache. However, when the cache gets full and especially when the generalization contexts contain very large frames, the subset search becomes more and more costly until it at some point again outweighs the positive effects.

Figure 5.9a depicts the scatter plot with the results of the comparison between the configurations K and L (see Table 5.1), which allows us to evaluate the isolated effect of lower bounds from the generalization context as they were presented in Theorem 4.9. We can see that the configuration with lower bounds is better in every single instance, which is due to the fact that lower bounds use the same caches as upper bounds, such that the overhead of managing the cache is not necessary a second time and therefore each cache hit improves the performance. In addition there are some instances that can be solved with lower bounds, that would have otherwise run into a timeout.

To set the isolated results of lower and upper bounds into a context, Fig-

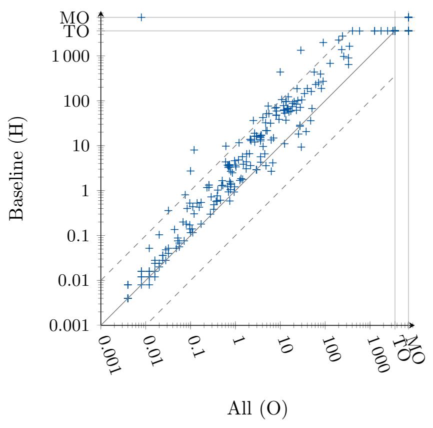


(a) Verification time (b) Number of cache hits

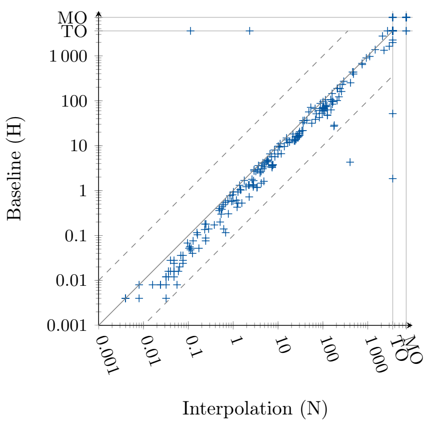
Figure 5.10: Evaluation of two-stage search in generalization contexts

ure 5.9b compares *Baseline* to a configuration with both, upper and lower bounds enabled. Here we can see that the marginal performance gain of upper bounds, which is mainly due to the overhead of managing the cache, is heavily boosted by adding lower bound extraction, such that all instances benefit from generalization caching with many instances almost reaching one order of magnitude improvement and additionally many new instances can be solved compared to *Baseline*.

In Section 4.3 we proposed an additional search stage for the generalization contexts that would search for generalizations that are a subset of the cube under consideration, which is evaluated in Figure 5.10. Figure 5.10a indicates that the second search stage does not change the verification time for almost all instances except for a few where the performance deteriorates noticeably. Interestingly, if we take Figure 5.10b into account, we can see that the number of cache hits grows noticeably, considering the logarithmic scale, for a very large number of instances. This indicates that the second search stage improves the number of cache hits and therefore the degree of information reuse considerably, the search is simply too costly and therefore outweighs the positive effects of the additional cache hits. We therefore suggest to either refrain from using this additional stage, or otherwise investigate how to improve the search considerably.



(a) All proposed improvements

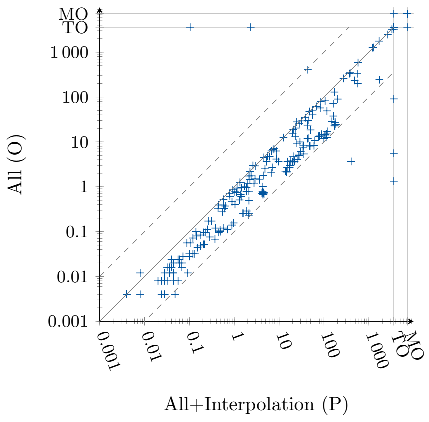


(b) Interpolation

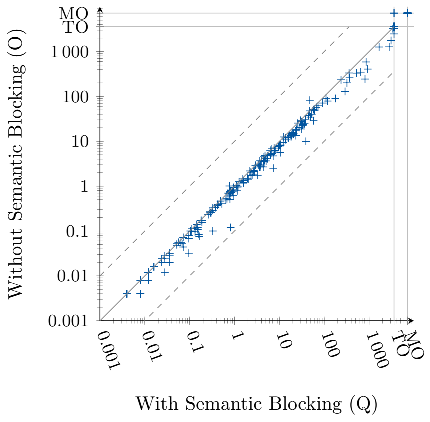
Figure 5.11: Evaluation of all new improvements and interpolation

Combining the optimizations After evaluating our optimizations to the generalization of IC3CFA, namely WEP-based inductivity, predecessor cubes and generalization caching for lower and upper bounds, in an isolated way, by comparing a *Baseline* configuration to configurations with each approach enabled individually, we want to see how all improvements perform together. To do so, we provide a configuration which differs from *Baseline* in that it enables all previously mentioned optimizations. The scatter plot comparing the results of this *All* configuration (O in Table 5.1) with *Baseline* is depicted in Figure 5.11a. As we can see, the performance of almost all instances is improved with many instances being solved up to or even more than one order of magnitude faster.

In contrast to this strong performance gain, the results of evaluating interpolation are not as convincing. Independently of whether we add interpolation to *Baseline* or to *All*, as shown in the scatter plots in Figures 5.11 and 5.12a enabling interpolation deteriorates the performance by up to one order of magnitude. In addition, while interpolation is able to solve two instances that run into a timeout otherwise, it is also not able to solve four instances that can be solved in *Baseline* and *All*, respectively. As such, while theoretically appealing for generalization, in a practical implementation the performance can not keep up with our other techniques.



(a) 'All' plus interpolation



(b) Semantic subsumption check during blocking of cubes

Figure 5.12: Evaluation of interpolation with all improvements and semantic subsumption check

Figure 5.12b shows the scatter plot comparing the results of *All* (O in Table 5.1) against a configuration that just enables the semantic check in addition (Q in Table 5.1). As we can see in Figure 5.12b, enabling the semantic check for subsumption in blocking performs better in exactly one instance and deteriorates the performance in all other instances. While theoretically the additional check promises a less polluted frame set and therefore more efficient search in frames, as well as smaller formulas to the SMT solver, the practical results are again not convincing.

Comparison with other software model checkers All results presented so far have been comparisons between different configurations of our IC3CFA algorithm. To set the overall performance into the context of other tools, we will evaluate the performance of our best configuration against other tools that implement IC3 algorithms for software model checking. However, since the number of freely available tools with IC3 software model checking is very low, the only tools used for comparison are the *Vienna Verification Tool* (VVT, [Günther et al., 2016]), which implements the CTIGAR approach and *SeaHorn* [Gurfinkel, Kahsai, et al., 2015] with Spacer/Z3PDR. For the evaluation with both tools, we used the last version that took part in the SV-Comp('16) with the same configuration as used in SV-Comp. All tools were executed on the same benchmark set, on the same machine, with the same timeout and memory limit as described above.

To graphically illustrate the comparison between the different tools, we use a score-based quantile plot. The score in the sense of SV-Comp is determined as follows: A correct result reporting a violation of the property (*correct FALSE*) gives a score of 1 point, while a correct result reporting no property violation (*correct TRUE*) scores 2 points. This is supposed to balance the hardness of the verification task, i.e. finding an error is generally considered easier than proving the absence of errors, which requires completeness. If however, the results given by the verification tool are incorrect, an *incorrect FALSE* result gives a penalty of -16 points, while for an *incorrect TRUE* result 32 points are deducted. Again, an incomplete analysis that finds an error where no error is will just cause some additional work of the developer to find out that it was a *false alarm*, where a missed bug due to an unsound analysis will give a dangerous confidence that everything is safe and thus has to be punished harder.

The score-based quantile plots is computed as follows: The graph itself depicts only correct results, which are ordered based on their runtime and for each of these results, the score is added up. In other words, each point (x, y) on the graph means that all verification runs up to y seconds achieve a total score of x points. To incorporate the penalties for incorrect results, it doesn't matter how fast they were at producing the incorrect result, such that we can ignore the time and just take the sum of all penalties collected. This summed penalties determine the value by which the graph is shifted along the x-axis.[1]

As we can see from Figure 5.13, the curve of SeaHorn is less steep than the one from VVT, which means that it can solve more instances in a short time. In addition, the graph itself is wider, which indicates that SeaHorn is able to solve more instances than VVT. But we can also see that the graph of SeaHorn is shifted much more to the left: SeaHorn collected 1024 penalty points for 48 incorrect *FALSE* and eight incorrect *TRUE*. This extreme amount of penalties impairs the general trust in the results that SeaHorn outputs.

In contrast, VVT only gives two incorrect *TRUE* and no incorrect *FALSE*, resulting in 64 penalty points. We can also see that in contrast to the other verification tools, the Verifier only outputs correct results, hence starts at a

[1]Note that since SV-Comp 2017 the rules of the competition require witnesses for both verification results to gain the mentioned score points, which was implemented to prevent guessing the result. However, since none of the tools that we compare with is able to give such a witness, we omit witnesses and assign the score just based on the output of the tool.

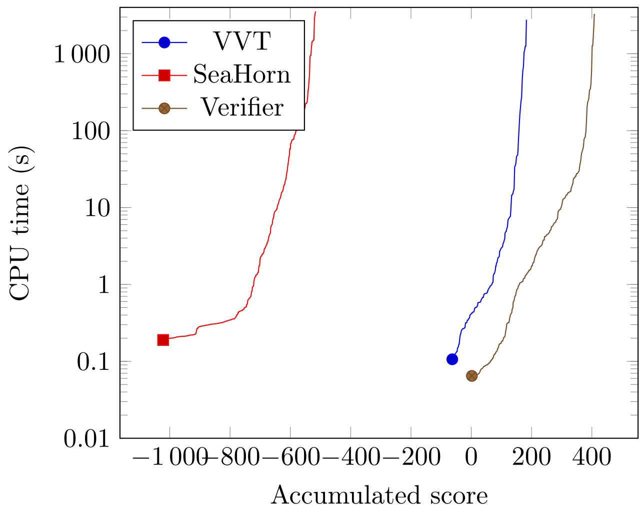


Figure 5.13: Score-based comparison of VVT, SeaHorn and our Verifier

score of 0 and achieves the highest score of all, ending at 410 points for 161 correct *TRUE* and 88 correct *FALSE*.

5.2.3 Industrial experience

In the previous section we evaluated the performance of all presented approaches to inductive incremental verification for software individually. However, while the SV-Comp offers large sets of benchmarks to evaluate the performance of tools on a broad variety of inputs, many of these benchmarks are either hand-crafted, auto-generated and/or tuned to work best with a specific technique. Therefore they barely reflect verification tasks that arise in real-world, industrial scenarios. On the other hand, these industrial settings are the ones, where safety verification is needed the most and can have the biggest impact. In the remainder of this section we will present some programs from our industrial partner *Siemens* to give an impression which sort of programs can be subject to safety verification. Furthermore, we will illustrate the peculiarities of the programs with respect to the programming language and the constructs used for these programs, as well as our results of verifying these programs.

At the heart of almost all industrial control systems, we will find *Pro-*

grammable Logic Controllers (PLC), which are ruggedized processing units that are adapted for control engineering. As special features, PLC are very modular and easy to install in a DIN rail rack and can be extended by different components such as interface modules, power supplies, communication processors and input/output (I/O) modules that can handle up to thousands of digital I/Os. However, maybe the most striking difference between PLCs and other controllers is in the way they execute their firmware: Because PLCs are mostly deployed in control engineering, their main task is to control sensor values and determine the appropriate state of the connected actuators to keep the system in the desired state. To support this task, PLCs have a cyclic execution model, which means that they start a cycle by reading the input values of the connected sensors, execute the firmware to determine the outputs and then only at the end of the cycle write the output values to the physical outputs synchronously. If timing is crucial, the PLC can then wait until it starts the next cycle to keep a consistent cycle interval, or otherwise it will directly start the next cycle. Due to the difference in the execution model between PLCs and regular CPUs for PCs, the code also differs structurally. While regular (e.g. C) programs without loops are hard to find at all, loops are rarely found in PLC code, due to the native cycling in the execution model. However, together with our industrial partner we analysed the code base of safety-critical applications from multiple large customers and found that data types such as *arrays* and *composite data types*, also called *record* or *struct* are much more common in real-world PLC code than they are in common C-code. Thus the requirements for our verification slightly differ, especially with respect to handling these data types, whose support enables us to verify these programs in the first place. To this aim the interchangeable support of multiple background theories, such as array theory for efficient handling of arrays, floating points to support trigonometric functions, common with motor control, or bitvector theory to enable bitprecise analysis of the machine code becomes even more important. This in turn proves theory-unaware verification algorithms, as we have presented in Chapter 4, to be of great significance. In the remainder of this section we will illustrate the application of our verification framework to a real-world industrial PLC application that is offered by our partner to customers for education purposes.

The program we consider is at the core of many control engineering applications. Whether we are considering scenarios like robot arms in an assembly line at a car manufacturer or cranes on a construction site or in a harbor, large and powerful electric motors are at the heart of many industrial applications and move heavy goods. But in most of these cases movement goes along with danger, especially when either a desired position can not be reached or a dan-

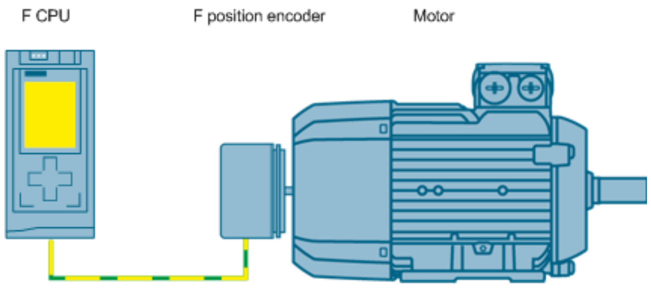


Figure 5.14: Failsafe position encoder (© Siemens AG, [Siemens AG, 2018a])

gerous position is reached. These dangers can ultimately range from financial damage to danger of life. Consider for example a harbor crane that discharges a shipment from a modern containership. These ships can carry up over 7000 standardized 40-foot containers on their deck in stacks of up to 9 containers. Now consider that due to a software error, the motor is not able to reach its desired height position and the crane that unloads such a ship topples over the whole stack and 8 other containers fall into the harbor basin, making the loaded electronic components worthless and creating a large financial damage. On the other hand, consider the same situation, but the container can be picked up without problems, but while placing the container on a truck, the motor control software reaches a forbidden horizontal position and the 40-foot container with a maximal payload of over 30 tons is lowered onto the driver's cab.

To avoid such dangers caused by the mis-positioning of motors, the manufacturers try to ensure that the control software satisfies all defined constraints to ensure safety, for example by the use of failsafe position encoders that communicate over a failsafe bus protocol to a PLC.

The program code for such a failsafe position encoder is a perfect example for the use of safety verification. Whenever the program computes the position, speed, direction or standstill incorrectly, the connected motor control may drive the motor into a dangerous position with the respective consequences. As such, safety verification allows us to verify that the encoder software behaves as specified and thus ensure that no dangerous situation may happen.

To detect a dangerous position of the motor, the function needs four parameters as input: To define the beginning of a danger range, the variable *beginMulti* defines the number of whole revolutions of the motor while *beginSingle* defines the number of steps inside one revolution, i.e. the danger range starts at the

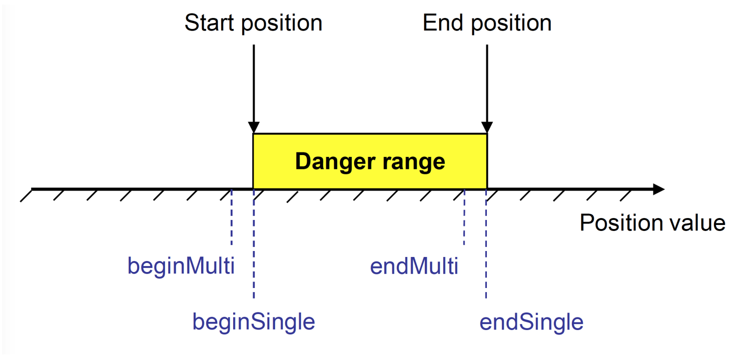


Figure 5.15: Definition of a danger range (© Siemens AG, [Siemens AG, 2018b])

beginSingleth steps after the *beginMulti*th revolution. Analogously, the danger range ends, after *endMulti* full revolutions and *endSingle* steps. Consequently, the current position is encoded in the two variables *posSingle* and *posMulti*. A graphical representation of the danger range is given in Figure 5.15. Whenever the current position is detected to be inside these boundaries, the specification requires a Boolean variable *dangerrange* to be set to *true*.

To simplify the presentation, we will not show the whole program, but rather just the formula that determines the *dangerrange* flag, which is

$$\begin{aligned} &((posMulti = endMulti \land posSingle \leq endSingle) \lor posMulti < endMulti) \\ \land\ & posMulti \geq beginMulti \land posSingle \geq beginSingle. \end{aligned} \tag{5.1}$$

The property that we want to prove is that whenever the program is inside the danger range, the flag is set to *true*. When verifying this property on the program with our verifier, we get a counterexample with the following states:

$endMulti =$	2289
$beginSingle =$	-32767
$posSingle =$	-32768
$beginMulti =$	-2289
$posMulti =$	2288

This counterexample contains two interesting pieces of information: (1) the counterexample is incomplete, i.e. for the given values, *every* value of *endSingle* will violate the property. (2) When closely inspecting the formula (5.1) we can see that the given counterexample is just *one example* of a whole class of values that violate the property. By hand, we can extract that each variable assignment that satisfies $beginMulti \leq posMulti < endMulti$ and $posSingle < beginSingle$ will violate the property. In other words, in each full revolution that is inside the danger range, the *dangerrange* flag will only be set when it reaches the *beginSingle*value. Therefore, if the position is before the danger range and motor drives quickly, then depending on the cycle duration, the motor might do a full revolution, now being inside the danger range, but because the number of steps in this revolution is less than the number of steps that defines the beginning of the danger range, the flag will not be set, thus violating the property that the position must never be inside the danger range and the *dangerrange* flag not being set.

Though just an example program for educational purposes, the example of the failsafe position encoding software nicely demonstrates the importance of formal verification for industrial control systems. Due to the high dangers to life and limb that come from industrial systems, such as smelters, chemical plants, cranes and trains, just to name a few, high safety standards, while not always enforced by law, should be implemented to ensure safety under all circumstances. A sufficient level of safety however can not be established by any form of testing, but only by formal verification, which is complete and therefore covers *all* possible execution scenarios.

Chapter 6

Conclusion

In this thesis, we presented *IC3CFA* (Section 4.2), an incremental, inductive verification algorithm for software model-checking that lifts the ideas of IC3 from hardware to software. In contrast to other existing IC3-style software verification algorithms it combines different features into a single algorithm to create a unique way of verifying software systems. The use of control-flow automata enables IC3CFA to observe the control-flow, an inherently important structure to all software systems, in a very simple way. This in turn allows IC3CFA to heavily prune the state space and improve the search performance drastically. On the other hand, IC3CFA only uses the control-flow automaton statically to split the frame sequences and determine predecessors and successors, rather than dynamically unrolling the control-flow into an abstract reachability tree (ART). Therefore IC3CFA can apply the equality-based inductivity check of IC3 and does not require expensive coverage checks for ARTs.

Apart from the search phase of IC3CFA, we presented a lifting of the generalization procedure of IC3 to IC3CFA in Section 4.3. Based on this basic generalization we introduced a variety of techniques to improve the generalization procedure, some of which are also applicable to other IC3-style verification algorithms. From the set of all improvements, in particular the generalization based on subcubes of the weakest precondition in the predecessor frame and the caching of generalizations with generalization contexts have proven to be highly efficient as evaluated in Chapter 5.

Despite the already strong performance of IC3CFA, we believe that there is still room for improvement in future work:

As presented in Section 5.1, the current implementation of IC3CFA relies on an inlining of function calls to create a single procedure without remaining function calls to execute the search phase. While this allows us to analyse programs with simple function calls, it does exhibit a number of drawbacks. On the one hand, the resulting inlined function can grow extremely large, due to pasting the called function into the call side. Such a blow-up in turn disables modularity, compared to an approach that would analyse each function on its own such as e.g. procedure summaries, thus preventing scalability. On the other hand, the approach of inlining functions in the call side can not be applied for functions with recursive calls, since the inliner will enter an infinite loop. Recursion however is very uncommon for PLC software. For these reasons, an interprocedural verification approach would be preferable over simple inlining, but is also much more involved. Nevertheless, we see large potential to use the intraprocedural IC3CFA algorithm in an interprocedural context, by augmenting it with a sort of *Meta-IC3* that handles the call structure: On a high level, each function in a software system represents an information transformer from input parameters to returned results - assuming side effect freedom. In a search phase similar to IC3, the *Meta-IC3* would consider each function call as an edge in a space of global system states. In a backward search, *Meta-IC3* would start with a concrete CTI of the system in some function f and compute the corresponding bad inputs using IC3CFA on this function. The resulting *hyper-CTI* can only be reached by functions that call f, such that all of these functions represent predecessor hyper-states and have to be analyzed using the intraprocedural IC3CFA. If on the other hand an application of IC3CFA shows that the *hyper-CTI* C is not reachable, it constructs a strengthening F for all, but in particular the initial location, resulting in a partial *Hoare triple* (F, f, C) that can be stored, therefore iteratively constructing partial Hoare triples that eventually suffice in order to prove the correctness of the property, if no counterexample trace exists.

A similar approach could also be used for an adaptation of IC3CFA that allows verifying PLC code over arbitrary cycle lengths and proving that a violation is not reachable over any number of executions. In fact, the procedure would be a special case of the above, where we assume that there exists one recursive function that calls itself at the end of each cycle. However, even this approach is not fully complete for PLC code, since input values are dependent on actions of previous cycles in a way that is determined by the physical environment. As long as such environment models are not part of the system model, PLC code verification will always expose only a limited precision.

Bibliography

Audemard, G., Lagniez, J., and Simon, L. (2013). "Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction". In: *SAT*. Vol. 7962. Lecture Notes in Computer Science. Springer, pp. 309–317.

Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., and Tang, A. (2015). "Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar". In: *IEEE Trans. Software Eng.* 41.7, pp. 620–638.

Baier, C. and Katoen, J. (2008). *Principles of Model Checking.* MIT Press.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., and Tinelli, C. (2011). "CVC4". In: *CAV*. Vol. 6806. Lecture Notes in Computer Science. Springer, pp. 171–177.

Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M. E., and Sebastiani, R. (2009). "Software model checking via large-block encoding". In: *FMCAD*. IEEE, pp. 25–32.

Beyer, D., Keremoglu, M. E., and Wendler, P. (2010). "Predicate abstraction with adjustable-block encoding". In: *FMCAD*. IEEE, pp. 189–197.

Biere, A. (2014). "Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the SAT Solver Lingeling". In: *POS@SAT*. Vol. 27. EPiC Series in Computing. EasyChair, p. 88.

Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. (1999). "Symbolic Model Checking without BDDs". In: *TACAS*. Vol. 1579. Lecture Notes in Computer Science. Springer, pp. 193–207.

Biere, A., Heule, M., Maaren, H. van, and Walsh, T., eds. (2009). *Handbook of Satisfiability.* Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press.

Birgmeier, J., Bradley, A. R., and Weissenbacher, G. (2014). "Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)". In: *CAV*. Vol. 8559. Lecture Notes in Computer Science. Springer, pp. 831–848.

Bjørner, N. and Janota, M. (2015). “Playing with Quantified Satisfaction”. In: *LPAR (short papers)*. Vol. 35. EPiC Series in Computing. EasyChair, pp. 15–27.

Boole, G. (1853). *Investigation of The Laws of Thought On Which Are Founded the Mathematical Theories of Logic and Probabilities*. George Boole’s collected logical works. Walton and Maberly.

Bradley, A. R. (2011). “SAT-Based Model Checking without Unrolling”. In: *VMCAI*. Vol. 6538. Lecture Notes in Computer Science. Springer, pp. 70–87.

Bradley, A. R. (2012). “Understanding IC3”. In: *SAT*. Vol. 7317. Lecture Notes in Computer Science. Springer, pp. 1–14.

Bradley, A. R. and Manna, Z. (2006). “Verification Constraint Problems with Strengthening”. In: *ICTAC*. Vol. 4281. Lecture Notes in Computer Science. Springer, pp. 35–49.

Bradley, A. R. and Manna, Z. (2007a). “Checking Safety by Inductive Generalization of Counterexamples to Induction”. In: *FMCAD*. IEEE Computer Society, pp. 173–180.

Bradley, A. R. and Manna, Z. (2007b). *The Calculus of Computation - Decision Procedures with Applications to Verification*. Springer.

Bruni, R. (2003). “Approximating minimal unsatisfiable subformulae by means of adaptive core search”. In: *Discrete Applied Mathematics* 130.2, pp. 85–100.

Chockler, H., Ivrii, A., Matsliah, A., Moran, S., and Nevo, Z. (2011). “Incremental formal verification of hardware”. In: *FMCAD*. FMCAD Inc., pp. 135–143.

Cimatti, A. and Griggio, A. (2012). “Software Model Checking via IC3”. In: *CAV*. Vol. 7358. Lecture Notes in Computer Science. Springer, pp. 277–293.

Cimatti, A., Griggio, A., Mover, S., and Tonetta, S. (2014). “IC3 Modulo Theories via Implicit Predicate Abstraction”. In: *TACAS*. Vol. 8413. Lecture Notes in Computer Science. Springer, pp. 46–61.

Cimatti, A., Griggio, A., Schaafsma, B., and Sebastiani, R. (2013). “The MathSAT5 SMT Solver”. In: *TACAS*. Vol. 7795. LNCS. Springer.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). “Model Checking and Abstraction”. In: *ACM Trans. Program. Lang. Syst.* 16.5, pp. 1512–1542.

Clarke, E. M., Grumberg, O., and Peled, D. A. (2001). *Model Checking*. MIT Press.

Colón, M., Sankaranarayanan, S., and Sipma, H. (2003). “Linear Invariant Generation Using Non-linear Constraint Solving”. In: *CAV*. Vol. 2725. Lecture Notes in Computer Science. Springer, pp. 420–432.

Cook, S. A. (1971). “The Complexity of Theorem-Proving Procedures”. In: *STOC*. ACM, pp. 151–158.

Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., and Vardi, M. Y. (2001). “Benefits of Bounded Model Checking at an Industrial Setting”. In: *CAV*. Vol. 2102. Lecture Notes in Computer Science. Springer, pp. 436–453.

Davydov, G., Davydova, I., and Kleine Büning, H. (1998). “An Efficient Algorithm for the Minimal Unsatisfiability Problem for a Subclass of CNF”. In: *Ann. Math. Artif. Intell.* 23.3-4, pp. 229–245.

Dijk, T. van and Pol, J. van de (2017). “Sylvan: multi-core framework for decision diagrams”. In: *STTT* 19.6, pp. 675–696.

Dijkstra, E. W. (1975). “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8, pp. 453–457. DOI: 10.1145/360933.360975. URL: http://doi.acm.org/10.1145/360933.360975.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall. URL: http://www.worldcat.org/oclc/01958445.

Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Paxson, V. (2014). “The Matter of Heartbleed”. In: *Internet Measurement Conference*. ACM, pp. 475–488.

Eén, N., Mishchenko, A., and Brayton, R. K. (2011). “Efficient implementation of property directed reachability”. In: *FMCAD*. FMCAD Inc., pp. 125–134.

Eén, N. and Sörensson, N. (2003). “An Extensible SAT-solver”. In: *SAT*. Vol. 2919. Lecture Notes in Computer Science. Springer, pp. 502–518.

Flanagan, C. and Saxe, J. B. (2001). “Avoiding exponential explosion: generating compact verification conditions”. In: *POPL*. ACM, pp. 193–205.

Fleischner, H., Kullmann, O., and Szeider, S. (2002). “Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference”. In: *Theor. Comput. Sci.* 289.1, pp. 503–516.

Garg, P., Löding, C., Madhusudan, P., and Neider, D. (2014). “ICE: A Robust Framework for Learning Invariants”. In: *CAV*. Vol. 8559. Lecture Notes in Computer Science. Springer, pp. 69–87.

Gershman, R., Koifman, M., and Strichman, O. (2008). “An approach for extracting a small unsatisfiable core”. In: *Formal Methods in System Design* 33.1-3, pp. 1–27.

Griggio, A. and Roveri, M. (2016). “Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking”. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 35.6, pp. 1026–1039.

Günther, H., Laarman, A., and Weissenbacher, G. (2016). "Vienna Verification Tool: IC3 for Parallel Software - (Competition Contribution)". In: *TACAS*. Vol. 9636. Lecture Notes in Computer Science. Springer, pp. 954–957.

Gurfinkel, A. and Ivrii, A. (2015). "Pushing to the Top". In: *FMCAD*. IEEE, pp. 65–72.

Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. A. (2015). "The SeaHorn Verification Framework". In: *CAV (1)*. Vol. 9206. Lecture Notes in Computer Science. Springer, pp. 343–361.

Guthmann, O., Strichman, O., and Trostanetski, A. (2016). "Minimal unsatisfiable core extraction for SMT". In: *FMCAD*. IEEE, pp. 57–64.

Hassan, Z., Bradley, A. R., and Somenzi, F. (2013). "Better generalization in IC3". In: *FMCAD*. IEEE, pp. 157–164.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2002). "Lazy abstraction". In: *POPL*. ACM, pp. 58–70.

Hoder, K. and Bjørner, N. (2012). "Generalized Property Directed Reachability". In: *SAT*. Vol. 7317. Lecture Notes in Computer Science. Springer, pp. 157–171.

Huang, G. (1995). "Constructing Craig Interpolation Formulas". In: *COCOON*. Vol. 959. Lecture Notes in Computer Science. Springer, pp. 181–190.

Itzhaky, S., Bjørner, N., Reps, T. W., Sagiv, M., and Thakur, A. V. (2014). "Property-Directed Shape Analysis". In: *CAV*. Vol. 8559. Lecture Notes in Computer Science. Springer, pp. 35–51.

Kars, P. (1996). "Formal Methods in the Design of a Storm Surge Barrier Control System". In: *European Educational Forum: School on Embedded Systems*. Vol. 1494. Lecture Notes in Computer Science. Springer, pp. 353–367.

Kleine Büning, H. (2000). "On subclasses of minimal unsatisfiable formulas". In: *Discrete Applied Mathematics* 107.1-3, pp. 83–98.

Komuravelli, A., Bjørner, N., Gurfinkel, A., and McMillan, K. L. (2015). "Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays". In: *FMCAD*. IEEE, pp. 89–96.

Komuravelli, A., Gurfinkel, A., and Chaki, S. (2014). "SMT-Based Model Checking for Recursive Programs". In: *CAV*. Vol. 8559. Lecture Notes in Computer Science. Springer, pp. 17–34.

Krajícek, J. (1997). "Interpolation Theorems, Lower Bounds for Proof Systems, and Independence Results for Bounded Arithmetic". In: *J. Symb. Log.* 62.2, pp. 457–486.

Kroening, D. and Strichman, O. (2008). *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Lee, S. and Sakallah, K. A. (2014). "Unbounded Scalable Verification Based on Approximate Property-Directed Reachability and Datapath Abstraction". In: *CAV*. Vol. 8559. Lecture Notes in Computer Science. Springer, pp. 849–865.

Leino, K. R. M. (2005). "Efficient weakest preconditions". In: *Inf. Process. Lett.* 93.6, pp. 281–288.

Löding, C., Madhusudan, P., and Neider, D. (2016). "Abstract Learning Frameworks for Synthesis". In: *TACAS*. Vol. 9636. Lecture Notes in Computer Science. Springer, pp. 167–185.

Manna, Z. and Pnueli, A. (1995). *Temporal Verification of Reactive Systems - Safety*. Springer.

Marques-Silva, J., Heras, F., Janota, M., Previti, A., and Belov, A. (2013). "On Computing Minimal Correction Subsets". In: *IJCAI*. IJCAI/AAAI, pp. 615–622.

McMillan, K. L. (2003). "Interpolation and SAT-Based Model Checking". In: *CAV*. Vol. 2725. Lecture Notes in Computer Science. Springer, pp. 1–13.

Mertens, T. (2016). "Efficient reuse of learnt information for control-flow oriented IC3 algorithms". Master thesis. RWTH Aachen University.

Moura, L. M. de and Bjørner, N. (2008). "Z3: An Efficient SMT Solver". In: *TACAS*. Vol. 4963. Lecture Notes in Computer Science. Springer, pp. 337–340.

Nadel, A. (2010). "Boosting minimal unsatisfiable core extraction". In: *FMCAD*. IEEE, pp. 221–229.

Nadel, A., Ryvchin, V., and Strichman, O. (2013). "Efficient MUS extraction with resolution". In: *FMCAD*. IEEE, pp. 197–200.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis*. Springer.

Oh, Y., Mneimneh, M. N., Andraus, Z. S., Sakallah, K. A., and Markov, I. L. (2004). "AMUSE: a minimally-unsatisfiable subformula extractor". In: *DAC*. ACM, pp. 518–523.

Papadimitriou, C. H. and Wolfe, D. (1988). "The Complexity of Facets Resolved". In: *J. Comput. Syst. Sci.* 37.1, pp. 2–13.

Prinz, F. (2016). "Generalisation methods for control-flow oriented IC3 algorithms". Master thesis. RWTH Aachen University.

Pudlák, P. (1997). "Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations". In: *J. Symb. Log.* 62.3, pp. 981–998.

Sankaranarayanan, S., Sipma, H. B., and Manna, Z. (2005). "Scalable Analysis of Linear Systems Using Mathematical Programming". In: *VMCAI*. Vol. 3385. Lecture Notes in Computer Science. Springer, pp. 25–41.

Sebastiani, R. (2007). "Lazy Satisability Modulo Theories". In: *JSAT* 3.3-4, pp. 141–224.

Sheeran, M., Singh, S., and Stålmarck, G. (2000). "Checking Safety Properties Using Induction and a SAT-Solver". In: *FMCAD*. Vol. 1954. Lecture Notes in Computer Science. Springer, pp. 108–125.

Siemens AG (2018a). *Safety position, standstill, direction and speed detection*. `https://support.industry.siemens.com/cs/document/49221879/safety-position-standstill-direction-and-speed-detection?dti=0&lc=en-WW`. [Online; accessed 3-May-2018].

Siemens AG (2018b). *Safety position, standstill, direction and speed detection (Documentation)*. `https://cache.industry.siemens.com/dl/files/879/49221879/att_870336/v3/49221879_F-Position_DOC_V20_en.pdf`. [Online; accessed 3-May-2018].

Strichman, O. (2000). "Tuning SAT Checkers for Bounded Model Checking". In: *CAV*. Vol. 1855. Lecture Notes in Computer Science. Springer, pp. 480–494.

Suda, M. (2013). "Triggered Clause Pushing for IC3". In: *CoRR* abs/1307.4966.

Tonetta, S. (2009). "Abstract Model Checking without Computing the Abstraction". In: *FM*. Vol. 5850. Lecture Notes in Computer Science. Springer, pp. 89–105.

Tseitin, G. S. (1968). "On the complexity of derivation in propositional calculus". In: *Studies in Constructive Mathematics and Mathematical Logic* 2.115-125, pp. 10–13.

Vizel, Y. and Gurfinkel, A. (2014). "Interpolating Property Directed Reachability". In: *CAV*. Vol. 8559. Lecture Notes in Computer Science. Springer, pp. 260–276.

Vizel, Y., Weissenbacher, G., and Malik, S. (2015). "Boolean Satisfiability Solvers and Their Applications in Model Checking". In: *Proceedings of the IEEE* 103.11, pp. 2021–2035.

Vojnar, T. and Beyer, D. (2018). *Competition on Software Verification (SV-COMP)*. https://sv-comp.sosy-lab.org/.

Welp, T. and Kuehlmann, A. (2013). "QF BV model checking with property directed reachability". In: *DATE*. EDA Consortium San Jose, CA, USA / ACM DL, pp. 791–796.

Witt, B. I., Baker, F. T., and Merritt, E. W. (1993). *Software Architecture and Design: Principles, Models, and Methods*. New York, NY, USA: John Wiley & Sons, Inc.

Zhang, J., Li, S., and Shen, S. (2006). "Extracting Minimum Unsatisfiable Cores with a Greedy Genetic Algorithm". In: *Australian Conference on Artifi-*

cial Intelligence. Vol. 4304. Lecture Notes in Computer Science. Springer, pp. 847–856.

Prior Publications

Lange, T. (2013). “Code-Based Model Minimization for PLC Code Verification”. Master thesis. RWTH Aachen University.

Lange, T., Neuhäußer, M. R., and Noll, T. (2013). “Speeding Up the Safety Verification of Programmable Logic Controller Code”. In: *Haifa Verification Conference*. Vol. 8244. Lecture Notes in Computer Science. Springer, pp. 44–60.

Lange, T., Neuhäußer, M. R., and Noll, T. (2015). “IC3 Software Model Checking on Control Flow Automata”. In: *FMCAD*. IEEE, pp. 97–104.

Lange, T., Prinz, F., Neuhäußer, M. R., Noll, T., and Katoen, J.-P. (2018). “Improving Generalization in Software IC3”. In: *SPIN*. LNCS. To be published. Springer.

Schommer, J. F., Franke, D., Lange, T., and Kowalewski, S. (2012). “Load Balancing for Cross Layer Communication”. In: *COMPSAC Workshops*. IEEE Computer Society, pp. 476–481.

2016-03	Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs
2016-04	Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution
2016-05	Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization
2016-06	Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud
2016-07	Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis
2016-08	Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features
2016-09	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic
2016-10	Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering
2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity
2017-04	Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05	Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06	Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07	Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++
2017-08	Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09	Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing

2018-01 *	Fachgruppe Informatik: Annual Report 2018
2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04	Andreas Ganser: Operation-Based Model Recommenders
2018-05	Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems
2018-06	Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.