Incremental, Inductive Model Checking

by

Zyad Hassan

B.S., Cairo University, Egypt, 2006

M.S., University of Colorado at Boulder, 2009

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer, and Energy Engineering

2014

This thesis entitled:
Incremental, Inductive Model Checking
written by Zyad Hassan
has been approved for the Department of Electrical, Computer, and Energy Engineering

Fabio Somenzi

Aaron R. Bradley

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Hassan, Zyad (Ph.D., Electrical Engineering)

Incremental, Inductive Model Checking

Thesis directed by Professor Fabio Somenzi and Professor Aaron R. Bradley

Model checking has become a widely adopted approach for the verification of hardware designs. The ever increasing complexity of these designs creates a continuous need for faster model checkers that are capable of verifying designs within reasonable time frames to reduce time to market. IC3, the recently developed, very successful algorithm for model checking safety properties, introduced a new approach to model checking: incremental, inductive verification (IIV). The IIV approach possesses several attractive traits, such as stability and not relying on high-effort reasoning, that make its usage in model checking very appealing, which motivated the development of another algorithm that follows the IIV approach for model checking ω-regular languages. The algorithm, Fair, has been shown to be capable of dealing with designs beyond the reach of its predecessors.

This thesis explores IIV as a promising approach to model checking. After identifying IIV's main elements, the thesis presents an IIV-based model checking algorithm for CTL: the first practical SAT-based algorithm for branching time properties. The algorithm, IICTL, is shown to complement state-of-the-art BDD-based CTL algorithms on a large set of benchmarks. In addition to fulfilling the need for a SAT-based CTL algorithm, IICTL highlights ways in which IIV algorithms can be improved; one of these ways is addressing counterexamples to generalization, which is explored in the context of IC3 and is shown to improve the algorithm's performance considerably. The thesis then addresses an important question: for properties that fall into the scope of more than one IIV algorithm, do these algorithms behave identically? The question is answered negatively, pointing out that the IIV framework admits multiple strategies and that there is a wide spectrum of possible algorithms that all follow the IIV approach. For example, all properties in the common fragment of LTL and CTL—an important class of properties—can be checked with Fair and IICTL. However, empirical evidence presented in the thesis suggests that neither algorithm is always superior to the other, which points out the importance of being flexible in deciding the strategy to apply to a given problem.

Dedication

To my parents, for all they did to make me who I am,

To my wife and son, for bringing so much love and joy into my life.

Acknowledgements

I am truly indebted to my advisors, Fabio Somenzi and Aaron Bradley, who taught me everything I know about model checking. I was first attracted to model checking through Fabio's outstanding "Introduction to Formal Verification" course. As an advisor, Fabio is always there for his students to discuss any issues that come up during research. He has truly been an advisor, a teacher, and a mentor. Aaron taught me to thoroughly think about concepts and ideas no matter how difficult they seemed. I have always admired and was inspired by Aaron's high ethical standards in research. Both Fabio and Aaron have been very patient and understanding. They never complained if I repeated the same questions over and over again. Instead, they always tirelessly addressed any misconceptions I had. I cannot thank them enough for all what they have done.

I would like to thank the other committee members Pavol Černý, Sriram Sankaranarayanan, and Niklas Sörensson for their valuable comments and suggestions on the work. I would also like to thank instructors at CU. Not only I learned a lot from them, but they were also great sources of inspiration. Sriram Sankaranarayanan especially helped broaden my view of formal verification.

It has been a pleasure to have the company of Michael Dooley, Saqib Sohail, Yan Zhang, and Arlen Cox in the VLSI lab. I have especially benefited from hours of discussion with Michael whose work was the closest to mine. Michael was also kind enough to provide feedback on my writing in several occasions.

I would also like to thank Mentor Graphics's formal team, especially Jeremy Levitt, for the internship opportunities I have been given. The internships have helped my PhD tremendously.

I am very grateful to my wife, Yassmin, without whom this thesis would not have been possible. Yassmin has always been there for me and has supported me in every step of my PhD. Whenever I was going through a busy period, she would take care of all our matters regardless of how busy she herself was. I am also very grateful to my parents for the continuous encouragement and support. I thank my dad for instilling in me the love of math and learning. I thank my mom for providing me with so much love and care, and for spending countless hours studying with me since my early years and up to college.

I am grateful to my friends in Boulder, especially Mohamed Nofal and Abdelati Hawwari, who have always stood by me and were like my family throughout my stay in Boulder.

Special thanks to Laura Nash from the writing center who has diligently revised my thesis and provided me with many comments that helped me improve my writing.

Above all, I thank Allah for His Guidance, and for helping me pass successfully through the most difficult periods.

Contents

Tables

Figures

Chapter 1

Introduction

Integrated circuits (ICs) have become ubiquitous in today's world. They exist in all electronic devices from cell phones and computers, to engine control units in cars and electronic control systems in aircrafts, to medical devices such as pacemakers. ICs have become very complex with the number of transistors on an IC reaching a few billion. However, with the increase in the complexity of the design comes an increase in the number of design errors. For example, in 2001, Intel reported finding more than 7,800 bugs in its Pentium 4 microprocessor prior to tapeout—a 350% increase in the number of bugs over its predecessor, the Pentium Pro [9]. Not only can design errors have serious effects, erroneous ICs are often difficult and costly to replace. Thus, IC designers often allocate a great portion of the design budget to verifying the correctness of the design[1] .

The correctness of a design is established by verifying that every behavior it exhibits conforms to the specifications. Ideally, the verification process (a) is automatic, i.e., requires minimal human input, (b) achieves high coverage, i.e., analyzes as many behaviors of the design as possible, and (c) is fast enough to allow meeting time-to-market constraints. Model checking [33, 52] is a set of rigorous mathematical techniques that mostly meets those requirements by performing an exhaustive search of the design's state space looking for behaviors that violate the specifications. The only inputs model checking requires from the user are a description of the design in some modeling language, and its specifications in some property specification language; it neither requires input stimulus (which is required by simulation), nor requires guiding the proof (which is required by theorem

[1] The International Technology Roadmap for Semiconductors (ITRS) estimates that functional verification consumes 75% of the system-on-chip design resources [39].

provers). The usefulness of model checking has been established by its ability to detect subtle bugs that escape testing.

Since its introduction in the early 80's, model checking has undergone considerable advancement. On the theoretical side, researchers have defined different classes of properties and analyzed their model checking complexities. On the practical side, careful engineering and clever heuristics have been applied in designing model checkers to overcome the high complexity of the algorithms and make worst case complexity scenarios as rare as possible. The practical advancement has transformed model checking from being of purely academic interest into a widely adopted approach in the microelectronics industry [9, 1, 8, 34].

The improvements in model checking technology have allowed model checkers to handle industrial-size designs [36, 7, 4]. Yet, many designs and properties of interest are still out of the reach of state-of-the-art model checkers in that such designs and properties require model checkers to run for prohibitively long periods of time. Thus, the quest for faster model checkers is still ongoing.

Modern model checkers rely on a collection of proof engines. A proof engine is an implementation of a model checking algorithm, which typically applies some form of state-space exploration to find a counterexample to the property, or prove the non-existence of such a counterexample. The earliest model checking algorithms were explicit: they dealt with individual states of the model [26, 27], which limited their applicability to relatively small designs. This limitation created a need for an alternative approach that can deal with large designs. As a result, symbolic model checking [21, 45] was proposed circa 1990 to fulfill this need. Rather than dealing with individual states, symbolic model checking algorithms manipulate sets of states. The first symbolic model checking algorithm [21] represented sets of states using binary decision diagrams (BDDs) [19] which extended the reach of model checkers to designs with more than 10^{20} states. However, a problem with BDD-based algorithms is that they often do more work than is required[2] . This has led to the

[2] A by-product of constructing a BDD for a Boolean function is the determination of the satisfiability and validity of that function!

introduction of another class of symbolic model checking algorithms that are based on satisfiability (SAT) checking. Enabled by the advancement in SAT solver technology, SAT-based algorithms manipulate propositional formulae, typically in conjunctive normal form (CNF), and employ SAT solvers to check the satisfiability of such formulae. SAT-based algorithms have proven to be more robust than BDD-based ones and are capable of efficiently dealing with designs far beyond the reach of BDD-based algorithms.

Bounded model checking (BMC) [12] is the first SAT-based model checking algorithm. BMC checks for the existence of a counterexample of length k by formulating a SAT query through unrolling the circuit k times. An unsatisfiable answer from the SAT solver triggers BMC to increment k. In this form, BMC is incomplete: it is only capable of finding counterexamples. Nevertheless, it can be made complete in various ways; for example, it can be augmented with a termination check to determine if a bound has been reached that is high enough to conclude the absence of a counterexample for any depth. The bound could be based on the (recurrence) diameter [12] or the reachability (recurrence) diameter [41, 23]. The bound can often be reduced by resorting to an induction check [13, 54, 5, 6]. An alternative approach to making BMC complete is to derive interpolants [28] from unsatisfiable formulae to compute overapproximations of reachable states [47]. If the overapproximations reach a fixpoint (which can be checked via a SAT query), the property holds.

Common to all BMC-based methods is the need to unroll the circuit with increasing depth, although approaches based on induction and interpolation resort to unrolling less often. A fundamental limitation of unrolling-based algorithms is the exponential increase in the difficulty of the SAT instances with the unrolling depth; unless the solution is determined at a sufficiently small depth, the SAT solving times become prohibitively long, increasing the likelihood that the algorithm runs out of time. While interpolation often finds the solution at much shallower depths, a blow-up often occurs in the size of the interpolants which also increases the difficulty of the SAT instances, albeit in a different way.

To overcome this limitation, Bradley developed a SAT-based method in 2010 that does not rely on unrolling [17, 14]. This method, called IC3, instead performs many relatively easy SAT calls whose complexity does not increase much over time. Working with easy SAT calls gives it the ability to continuously make steady progress towards determining a solution for the model checking instance.

Similar to interpolation, IC3 attempts to find an inductive strengthening of the property, but does so in an incremental fashion through deriving lemmas that hold relative to previously derived ones. The lemmas are derived on demand to block states that interfere with the inductiveness of the property. In deriving lemmas, IC3 draws its reasoning strength from inductive generalization [15]. These characteristics have made IC3 the best stand-alone model checking algorithm developed so far [18]. Indeed, IC3 by itself lost by a small margin to two highly-tuned multi-engine model checkers in the 2010 Hardware Model Checking Competition [37], triggering the addition of IC3 engines to most model checkers in the following year.

IC3's approach to model checking is an incremental, inductive one. Its approach is incremental because of how it constructs a proof from simple lemmas and builds upon existing lemmas when deriving new ones. Its inductiveness draws from using inductive reasoning in deriving lemmas. The success of IC3 gives great promise to incremental, inductive verification (IIV). Indeed, an IIV-based algorithm for model checking progress properties (Fair) [16] has been shown to complement existing BDD-based algorithms and state-of-the-art safety algorithms applied to liveness-to-safety [11] converted models. Section 1.1 gives an overview of IIV.

1.1 Overview of Incremental, Inductive Verification

Incremental, inductive verification (IIV) is an approach for verification introduced by IC3. IIV has proven to be a successful approach as demonstrated by IC3 and its counterpart for ω-regular properties, Fair. Its success with these two algorithms suggests that it should be considered when designing new model checking algorithms. The elements of the IIV approach are the following:

- **Abstraction:** A suitable abstraction of the system is maintained and is refined on demand. The choice of abstraction dictates the workings of the IIV-based algorithm and could greatly impact its performance. The coarser the abstraction, the simpler the algorithm usually is. For IC3, the system is abstracted as a set of over-approximations to states reachable within i steps. This is a finer abstraction than the one used by IC3's predecessor, FSIS [15], which only abstracts the system as a single set of potentially reachable states. This inflexibility of FSIS is what makes it inferior to IC3. For Fair, the system is abstracted in the form of potentially reachable states and arenas of strongly connected component (SCC)-closed sets.

- **Analyzing Individual Counterexamples:** When the current abstraction is not strong enough to support the property (which occurs in the case of IC3 when none of the over-approximate sets forms an inductive strengthening of the property, and in the case of Fair when the set of potentially reachable states still contains fair SCCs), a specific reason—generally a set of states—for why this is the case is examined. If analysis concludes that such a reason is invalid, i.e., is not admitted by the system, the abstraction is refined accordingly. In the process, the analysis may need to examine other related reasons. In the case of IC3, the reason examined is a counterexample to induction (CTI). The analysis carried out tries to prove this CTI unreachable within k steps. If the analysis fails, predecessors of that CTI which are believed to be reachable are examined. In the case of Fair, the existence of a skeleton, a set of states that fall in a single arena that satisfy all fairness constraints, indicate that the criterion is not yet satisfied. Proving that any skeleton-state is unreachable or that any pair of states cannot be connected is sufficient to rule out that skeleton.

- **Generalization:** Before refining the abstraction, the analysis is generalized to produce a stronger refinement. Generalization uses inductive reasoning and is a crucial component of IIV because it greatly reduces the number of counterexamples. However, there is a tradeoff between the strength of the generalization and the speed of the procedure. IIV strikes a balance through fixing the domain of generalization—in particular to one in which a generalization can be efficiently computed, and attempting to find the strongest possible generalization in that domain. For example, in IC3, a proof of the unreachability of a CTI within k steps is generalized to other states in order to reduce the number of CTIs that have to be examined. The generalization of the proof is kept in the form of a single clause and a minimal-size clause is sought. In Fair, a proof of unreachability of a skeleton-state from the initial states or from another skeleton-state is generalized to include other states. The purpose here is to reduce the number of skeletons to analyze. Fair limits the effort spent on generalization by restricting the generalization domain to a CNF with state variables only.

- **Incrementality:** IIV uses the incremental approach of [44]. In contrast to the monolithic approach, the incremental approach uses low-effort reasoning to derive pieces of the proof. This makes the algorithm more effective in dealing with large systems for which the high-effort reasoning used by monolithic approaches is likely to become a bottleneck.

  Another advantage of the incremental approach is that any fact that is to be derived does not need to hold by itself; it only needs to hold relative to already proven facts. This feature turns out to be especially important to overcome the limitation of the fixed generalization domain. In particular, not assuming already proven facts could lead to the failure of generalization due to the impossibility of expressing the fact in the fixed domain.

1.2 Thesis Contributions

This thesis takes advantage of the incremental, inductive verification (IIV) approach to advance the state-of-the-art in model checking. In particular, this thesis makes the following contributions:

- It develops an IIV, SAT-based algorithm for model checking CTL properties—the first practical SAT-based algorithm for branching time logics. The algorithm, called IICTL, follows the IIV approach outlined in Section 1.1. In particular, IICTL abstracts the system by over- and under-approximations of states satisfying each subformula of the CTL formula. The individual counterexamples IICTL examines are **undecided states**: ones that are in the over-approximation but not in the under-approximation of a certain subformula. To decide a state, IICTL carries out a query depending on the CTL operator. Once a state is decided, the result is generalized to other states. Experiments in Chapter 3 show that IICTL is complementary to BDD-based approaches on a large set of benchmarks.

- It enhances an existing IIV algorithm, IC3, by improving its generalization procedure. The improved generalization procedure addresses **counterexamples to generalization** (CTGs), which are states that cause a certain strengthening of a clause to fail. Addressing CTGs is shown to reduce the depth of IC3's priority queue, resulting in a performance gain to the algorithm as confirmed through testing the improved procedure in two independent implementations of IC3.

- It compares the strategies of IC3 and Fair to that of IICTL; despite all being IIV algorithms, the strategies of such algorithms greatly differ. This comparison points out that IIV is a framework that admits multiple strategies and that the IIV–LTL approach (IC3 and Fair), and the IIV–CTL approach (IICTL) are two extreme points on a spectrum of possible IIV algorithms.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 defines terms used in the thesis, and gives an overview of the IC3 and Fair algorithms. Chapter 3 describes IICTL, the IIV-based algorithm for CTL properties. Chapter 4 lists the different types of generalizations needed by IIV algorithms and outlines procedures for efficiently carrying such generalizations out. Chapter 5 describes IC3's improved generalization procedure. Chapter 6 analyzes the behavior of different IIV algorithms on properties in the common fragment of CTL and LTL in order to understand the differences between the strategies that these algorithms employ. Finally, Chapter 7 concludes the thesis.

Chapter 2

Preliminaries

This chapter reviews the definitions of terms, and the descriptions of algorithms referred to throughout the thesis. Section 2.1 defines finite-state systems, which are used as models for hardware designs for model checking purposes. Section 2.2 defines invariance properties and describes induction, the classical approach for proving invariance properties and a principal component of the incremental, inductive verification (IIV) approach. In Sections 2.3 and 2.4, two of the popular logics used for specifying properties of finite-state systems—CTL and LTL—are described. Section 2.5 defines Büchi automata, which are used for model checking LTL properties. Sections 2.6 and 2.7 give an overview of the two existing IIV-based algorithms, IC3 and Fair, which deal with invariance and ω-regular properties, respectively. Finally, model checking algorithms described in this thesis are SAT-based; Section 2.8 explains common techniques employed by SAT-based algorithms to use SAT solvers efficiently.

2.1 Finite-State Systems

Hardware designs at the register-transfer level (RTL) can be modeled as finite-state systems to which model checking algorithms described in this thesis are applied. A **finite-state system** is represented as a tuple $S : \langle \bar{i},\ \bar{x},\ I(\bar{x}),\ T(\bar{x}, \bar{i}, \bar{x}'),\ \mathcal{B} \rangle$ consisting of primary inputs $\bar{i}$, state variables $\bar{x}$, a propositional formula $I(\bar{x})$ describing the initial configurations of the system, a propositional formula $T(\bar{x}, \bar{i}, \bar{x}')$ describing the transition relation, and a set $\mathcal{B} = \{B_1(\bar{x}), \ldots, B_\ell(\bar{x})\}$ of Büchi fairness constraints. Primed state variables $\bar{x}'$ represent the next state. A state of the system is an assignment of Boolean values to all variables $\overline{x}$ and is described by a **cube** over $\overline{x}$, which, generally, is a conjunction of literals, each **literal** a variable or its negation. A **subcube** of a cube q is a conjunction of a subset of q's literals. A **clause** is a disjunction of literals. A **subclause** $d \subseteq c$ is a clause d whose literals are a subset of c's literals.

An assignment s to all variables of a formula F either satisfies the formula, $s \models F$, or falsifies it, $s \not\models F$. If s is interpreted as a state and $s \models F$, then s is an F**-state**. A set of states D satisfies a formula F if all states in D satisfy F: $\forall s \,.\, s \in D \Rightarrow s \models F$. A propositional function is the characteristic function of a set of states; thus, when there is no ambiguity, propositional functions and sets of states are used interchangeably. A formula F **implies** another formula G, written $F \Rightarrow G$, if every satisfying assignment of F satisfies G. The (in)validity of $F \Rightarrow G$ is established by querying a SAT solver for the unsatisfiability of $F \wedge \neg G$.

The transition structure of a finite-state system is assumed to be complete. That is, every state has at least one successor on every input: $\forall \overline{x}, \overline{i} \,.\, \exists \overline{x}' \,.(\overline{x}, \overline{i}, \overline{x}') \models T$. A **path** in S, $s_0, s_1, s_2, \ldots$, which may be finite or infinite in length, is a sequence of states such that for each adjacent pair (s_i, s_{i+1}) in the sequence, $\exists \overline{i}.(s_i, \overline{i}, s'_{i+1}) \models T$. A **strongly connected component** (SCC) of a finite-state system S is a maximal set of states C in S such that there is a path from every state in C to every other state in C. For a path $s_0, s_1, s_2, \ldots$ in S, if $s_0 \models I$, then the path is a **run** of S. A state that appears in some run of the system is **reachable**. For an infinite path π, $\pi(i)$ denotes the i-th state in the sequence, i.e., $\pi(i) = s_i$, and π^i denotes the suffix of π starting at s_i. An infinite path $s_0, s_1, s_2, \ldots$ is **fair** if, for every $B \in \mathcal{B}$, infinitely many s_i satisfy B, $s_i \models B$; if $s_0 \models I$ then it is a **fair run** or **computation** of S. S's language is **empty** if it has no fair runs.

2.2 Invariance Properties and Induction

There are different types of properties that can be used for specifying finite-state systems. Invariance properties are the simplest type of properties; they state that something must hold in every state of the system. Formally, an **invariance property** $P(\overline{x})$, a propositional formula, asserts that only P-states are reachable. P is **invariant** for the system S (that is, S-invariant) if indeed only P-states are reachable. If P is not invariant, then there exists a finite **counterexample** run $s_0, s_1, \ldots, s_k$ such that $s_k \not\models P$. An invariance property $P(\overline{x})$ is **inductive** if

(1) (**initiation**) every initial state satisfies the property: $I(\overline{x}) \Rightarrow P(\overline{x})$; and

(2) (**consecution**) every transition from a P-state leads to a P-state: $P(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{x}') \Rightarrow P(\overline{x}')$.

While an inductive property P is invariant, the converse is not necessarily true. In such cases, it is customary to seek an **inductive strengthening** of P, which is a formula F such that $F \wedge P$ is inductive.

An assertion F is **inductive relative to** another assertion G, possibly containing primed variables, if

(1) every initial state satisfies F: $I(\overline{x}) \Rightarrow F(\overline{x})$; and

(2) F satisfies consecution under assumption G:
$G(\overline{x}, \overline{x}') \wedge F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{x}') \Rightarrow F(\overline{x}')$.

2.3 Computational Tree Logic

Computational Tree Logic (CTL [25, 52]) is a branching-time temporal logic. Its formulae are inductively defined over a set A of atomic propositions which in this thesis are taken to be propositional functions defined over the state variables $\overline{x}$. Every atomic proposition is a CTL formula. In addition, if φ and ψ are CTL formulae, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\mathsf{EX}\,\varphi$, $\mathsf{E}\,\psi\,\mathsf{U}\,\varphi$, and $\mathsf{EG}\,\varphi$. Additional operators are defined as abbreviations. In particular, $\mathsf{EF}\,\varphi$ abbreviates $\mathsf{E}(\varphi \vee \neg\varphi)\,\mathsf{U}\,\varphi$, $\mathsf{AX}\,\varphi$ abbreviates $\neg\,\mathsf{EX}\,\neg\varphi$, $\mathsf{AG}\,\varphi$ abbreviates $\neg\,\mathsf{EF}\,\neg\varphi$, and $\mathsf{AF}\,\varphi$ abbreviates $\neg\,\mathsf{EG}\,\neg\varphi$. A model of a CTL formula is a finite-state system S. Satisfaction of a CTL formula at state s_0 of S is then defined as follows:

$S, s_0 \models a$	iff $s_0 \models a$ for $a \in A$
$S, s_0 \models \neg\varphi$	iff $S, s_0 \not\models \varphi$
$S, s_0 \models \varphi \wedge \psi$	iff $S, s_0 \models \varphi$ and $S, s_0 \models \psi$
$S, s_0 \models \mathsf{EX}\,\varphi$	iff $\exists$ a fair path $s_0, s_1, \ldots$ of S such that $S, s_1 \models \varphi$
$S, s_0 \models \mathsf{EG}\,\varphi$	iff $\exists$ a fair path $s_0, s_1, \ldots$ of S such that for $i \geq 0$, $S, s_i \models \varphi$
$S, s_0 \models \mathsf{E}\,\psi\,\mathsf{U}\,\varphi$	iff $\exists$ a fair path $s_0, s_1, \ldots$ of S such that there exists $i \geq 0$ for which $S, s_i \models \varphi$, and for $0 \leq j < i$, $S, s_j \models \psi$.

Then $S \models \varphi$ if $\forall s\,.(s \models I) \Rightarrow (S, s \models \varphi)$. That is, S models formula φ if all its initial states do. In model S, the set of states that satisfy φ is written $[\![\varphi]\!]$.

The fact that every CTL formula is interpreted as a set of states makes model checking easier than for the more expressive CTL*. Working bottom-up on the parse graph of φ, the standard symbolic CTL model checking algorithm [45] annotates each node with a set of states. Boolean connectives are dealt with in the obvious way, while temporal operators are handled with fixpoint computations. The bottom-up approach is also known as **global** model checking. In contrast, **local** model checking [42, 55, 10, 29] proceeds top-down. A local model checker starts from the goal of proving that initial state s satisfies φ and applies inference rules to reformulate the goal as a list of subgoals in terms of subformulae of φ and states in the vicinity of s. While local model checking can sometimes prove a property without examining most of a system's states, in its basic formulation it does not play to the strengths of symbolic algorithms. For that reason, local model checkers for finite-state systems tend to employ explicit search.[1]

2.4 Linear Time Logic

Formulae of Linear Time Logic (LTL [51]) are defined inductively over a set A of atomic propositions which in this thesis are taken to be propositional functions defined over the state variables $\overline{x}$. Every atomic proposition is a formula. In addition, if φ and ψ are LTL formulae, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\mathsf{X}\,\varphi$, $\varphi\,\mathsf{U}\,\psi$, and $\varphi\,\mathsf{R}\,\psi$. Additional operators are defined as abbreviations.

[1] Some BDD-based model checkers incorporate elements of local algorithms. For instance, the CTL model checker in VIS [57] uses top-down **early termination conditions** to define conditions that a safe approximation of a set of states must satisfy. However, it is still fundamentally a bottom-up algorithm.

$\mathsf{F}\,\varphi$ abbreviates $(\varphi \vee \neg\varphi)\,\mathsf{U}\,\varphi$, and $\mathsf{G}\,\varphi$ abbreviates $(\varphi \wedge \neg\varphi)\,\mathsf{R}\,\varphi$. A model of an LTL formula is a finite-state system S. Satisfaction of an LTL formula along path π of S is defined as follows:

$S, \pi \models a$	iff $\pi(0) \models a$ for $a \in A$
$S, \pi \models \neg\varphi$	iff $S, \pi \not\models \varphi$
$S, \pi \models \varphi \wedge \psi$	iff $S, \pi \models \varphi$ and $S, \pi \models \psi$
$S, \pi \models \mathsf{X}\,\varphi$	iff $S, \pi^1 \models \varphi$
$S, \pi \models \varphi\,\mathsf{U}\,\psi$	iff there exists $i \geq 0$ for which $S, \pi^i \models \psi$ and for $0 \leq j < i$, $S, \pi^j \models \varphi$
$S, \pi \models \varphi\,\mathsf{R}\,\psi$	iff for all $i \geq 0$, $S, \pi^i \models \psi$, or there exists $j \geq 0$ such that $S, \pi^j \models \varphi$ and for all $i, 0 \leq i \leq j, S, \pi^i \models \psi$.

Then $S \models \varphi$ if $\forall \pi \,.\, \pi(0) \models I \Rightarrow S, \pi \models \varphi$. That is, S models formula φ if every infinite path of S that starts with an initial state models φ.

2.5 Büchi Automata

A (generalized) Büchi automaton [20] is a tuple $\langle \Sigma, Q, q_0, \delta, \mathcal{F} \rangle$, where $\Sigma = 2^A$ is the input alphabet defined over a set of atomic propositions A which in this thesis are taken to be propositional functions over the state variables $\overline{x}$, Q is the finite set of states, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $\mathcal{F} \subseteq 2^Q$ is the set of acceptance conditions. A **run** of the automaton, ρ is an infinite sequence of states $q_0, q_1, q_2, \ldots$ such that for $i \geq 0, (q_i, \sigma_i, q_{i+1}) \in \delta$. A run ρ is accepting if for every $F \in \mathcal{F}$, there is at least one F-state that occurs infinitely often in ρ. A Büchi automaton can be encoded as a finite-state system.

LTL is less expressive than Büchi automata [40, 35]: every LTL formula is representable by a Büchi automaton. Building on this fact, Vardi and Wolper devised an automata-theoretic approach to model checking LTL formulae which checks whether a finite-state system S satisfies a given LTL formula φ by constructing a Büchi automaton that accepts $\neg\varphi$, composing it with S, and checking the composition for language emptiness [56].

2.6 An Overview of IC3

IC3 [17, 14] is a model checking algorithm for invariance properties. The algorithm operates in a demand-driven manner, generating relatively inductive lemmas in response to states that interfere with the inductiveness of the property. Lemma generation proceeds incrementally until an inductive strengthening is discovered or the lemmas guide the backward search to a counterexample trace. IC3 is SAT-based but, in contrast to other SAT-based approaches, poses numerous, relatively easy SAT queries that arise from considering single steps of a transition relation. This style of using a SAT solver keeps IC3's memory footprint small.

IC3 maintains a sequence of overapproximations F_i to sets of states reachable within i steps, for $0 \leq i \leq k$, where $k = 1$ initially. Each F_i is a conjunction of the property P with an initially empty set of clauses. For each $k > 0$, IC3 refines the F_i's for $i \leq k$ as needed to prove inductiveness of P relative to F_k. This refinement is property-driven: for every counterexample to the inductiveness (CTI) of the property, which is an F_k-state with a $\neg P$-successor, IC3 derives a clause to block the CTI. If successful, IC3 applies induction to generalize the clause to block many more states than the CTI alone.[2] IC3 then adds the generalized clause to F_i for all $i \leq k$.

If unsuccessful in blocking a CTI, IC3 explores (transitive) predecessors of the CTI to derive supporting strengthening clauses until the original CTI can itself be addressed relative to F_k. This exploration of concrete predecessors is guided by a priority queue of pairs of states and frame indices: (s, i) represents the obligation that state s must be inductively excluded relative to F_i, i.e., proved unreachable for at least $i+1$ steps. Obligations are handled in lowest-index-first order, guaranteeing termination. IC3 aggressively generalizes from states: once it addresses (s, i) by finding a clause $c \subseteq \neg s$ that is inductive relative to some F_j, $j \geq i$, IC3 adds obligation $(s, j + 1)$ to the queue if $j < k$. This aggressive strategy not only facilitates early discovery of mutually inductive clauses, it also allows IC3 to find deep counterexamples even when k is small.

When no CTIs remain (for F_k), IC3 checks each clause of each F_i to determine if it can

[2] IC3's inductive generalization procedure is described in Section 4.1.1.

be propagated forward, i.e., if it has become inductive relative to F_i since its creation because of subsequent strengthening of F_i. In the process, IC3 determines whether any F_i has become an inductive strengthening of the property, in which case the property is declared to hold. If not, it increments k and seeds the new frontier F_k with all clauses that are inductive relative to F_{k-1}. This process continues until IC3 finds an inductive strengthening of the property or finds a counterexample by following a sequence of CTIs back to an initial state.

Listing 2.1: IC3 pseudocode.

```
 1 bool IC3($S$, $P$):
 2   if $I \not\Rightarrow P$ or $I \wedge T \not\Rightarrow P'$:
 3     return false
 4   $F_0$ := $I$, $F_1$ := $P$
 5   for $k$ := 1 to $\infty$:
 6     $F_{k+1}$ := $P$
 7     while $F_k \wedge T \not\Rightarrow P'$:
 8       $queue$ := (CTI, $k-1$) { CTI is satisfying assignment from line 7 }
 9       while $queue$ not empty:
10         ($s$, $i$) = pop($queue$) { Prioritized by second element of pair.
11                                     Lower values have higher priorities. }
12         $j$ := $\max(\{-1\} \cup \{j : i \leq j \leq k$ and $F_j \wedge \neg s \wedge T \Rightarrow \neg s'\})$
13         if $j = -1$:
14           if $i = 0$: { Counterexample }
15             return false
16           with $(F_i \wedge \neg s)$−state $t$: { $t$ is predecessor of $s$ from line 12 }
17             { Add $t$ to queue }
18             push($queue$, ($t$, $i-1$))
19         else:
20           $\hat{s}$ = MIC($s$, $j$)
21           $clauses(F_l)$ := $clauses(F_l) \cup \neg\hat{s}$ for $1 \leq l \leq j+1$
22           if $j < k$:
23             push($queue$, ($s$, $j+1$))
24     { Propagate clauses }
25     for $i$ := 1 to $k$:
26       foreach Clause $c$ in $clauses(F_i)$:
27         if $F_i \wedge c \wedge T \Rightarrow c'$:
28           $clauses(F_{i+1})$ := $clauses(F_{i+1}) \cup c$
29       if $clauses(F_i)$ = $clauses(F_{i+1})$ { $F_i$ is an inductive strengthening }
30         return true
```

The pseudocode for IC3 is shown in Listing 2.1. The algorithm first checks for the existence of 1- and 2-state counterexamples (line 2). If none exist, it initializes the first two overapproximating assertions F_0 and F_1 to I and P, respectively (line 4).[3] The algorithm continues to increment k (effectively creating a new overapproximation) as long as none of the current F_i's for $i \leq k$ forms an inductive strengthening of the property, and no counterexamples are found. The loop on lines 7–23 refines the overapproximations through addressing CTIs and their transitive predecessors. Line 12 finds the highest level at which the negation of a CTI is inductive. On the one hand, if $\neg s$ is not inductive relative to any F_j for $i \leq j \leq k$, an $(F_i \wedge \neg s)$-predecessor t is added to the queue (line 18), unless $i = 0$ which signals the completion of a trace from a bad state back to an initial state. On the other hand, if $\neg s$ is inductive relative to some F_j for $j \geq i$, the proof obligation has been handled. In this case, $\neg s$ is generalized (line 20) and the generalized clause is added to all F_l's for $1 \leq l \leq j+1$. The loop on lines 25–30 checks if any F_i now forms an inductive strengthening.

Several algorithms described in this thesis pose reachability queries to determine whether a target is reachable from a source via some (possibly constrained) path. For this purpose, we define a function, $\mathsf{reach}(S, C, F, G)$, that accepts a system S, a set of constraints $C(\overline{x}, \overline{x}')$ on the transition relation, an initial condition F, and a target G. reach returns either a counterexample run from an F-state to a G-state on a path with C-states,[4] or an assertion $P(\overline{x})$, inductive relative to C, separating F from G. This function can be implemented using IC3 (or any other safety model checker) by setting the system's initial condition $I = F$, its transition relation $T = T_S \wedge C$, and the property $P = \neg G$.

2.7 An Overview of Fair

Fair [16] is an algorithm that answers the question of whether the language of a given finite-state system S is empty. Fair arrives at the answer by examining a sequence of skeletons; each skeleton is a set of states that satisfy all fairness constraints of S. For each skeleton, Fair checks whether the skeleton forms a reachable cycle, i.e., that the skeleton-states are reachable from an initial state, and that every skeleton-state can reach all other skeleton-states. To check whether

[3] Note that F_0 is exact and is never refined by the algorithm.

[4] Note that an F-state that is also a G-state constitutes a counterexample since every state on the path to a G-state (there are none in this case) is (vacuously) a C-state.

the skeleton forms a reachable cycle, Fair chooses a cyclic order for the skeleton-states, and checks whether every state in the order can reach the next, and whether one of the skeleton-states is reachable from some initial state.[5] Reachability is checked using an invariant model checker like IC3. If all reachability queries succeed, then there is a fair run and the language of S is not empty. Otherwise, Fair tries to learn information about S from the failed reachability query that guides the future choice of skeletons. The type of information that Fair learns is dependent on the type of the reachability query: whether it is a **stem query** that checks whether a skeleton-state is reachable from some initial state, or a **cycle query** which checks whether a skeleton-state is reachable from another. From a failed stem query, Fair learns about unreachable S-states, and from an unreachable cycle query, Fair learns about SCC-closed regions in S. Every skeleton chosen in the future must be from states that are not known to be unreachable, and must be from the same SCC-closed region. The non-existence of such a skeleton indicates that the language is empty.

Listing 2.2: Fair pseudocode.

```
1  bool Fair($S$):
2    $R$ := $\top$, $\mathcal{W}$ := $\emptyset$
3    while $(\bigwedge_{i\in\{1,\ldots,l\}} B_i(\overline{x}^i) \wedge R(\overline{x}^i)) \wedge \bigwedge_{W\in\mathcal{W}}[(\bigwedge_{i\in\{1,\ldots,l\}} W(\overline{x}^i)) \vee (\bigwedge_{i\in\{1,\ldots,l\}} \neg W(\overline{x}^i))]$:
4      $(s_0,\ldots,s_{l-1})$ := $(\overline{x}^1,\ldots,\overline{x}^l)$
5      if $\neg$reach($S$, $R$, $I$, $s_0$): {with proof $P$}
6        $R$ := $R \wedge P$
7        continue
8      if $\neg$reach($S$, $R$, $s_i$, $s_{i\oplus 1}$) for any $0 \leq i \leq l-1$: {with proof $P$}
9        $\mathcal{W}$ := $\mathcal{W} \cup P$
10       continue
11     return false
12   return true
```

The pseudocode for a basic version of the algorithm is shown in Listing 2.2 (see [16] for a more complete pseudocode). Initially, all states are presumed reachable (hence $R = \top$) and the set of inductive walls $\mathcal{W}$ is empty. Line 3 checks whether any skeletons exist given the current reachability information R and the set of walls $\mathcal{W}$. To that purpose, l copies of the system's state

[5] Note that constraining the path to R-states is not necessary for correctness but helps the safety model checker converge faster through constraining its search space.

variables $\overline{x}$ are created that would represent the l skeleton-states. The first big conjunction in the SAT query of line 3 ensures that each fairness constraint B_i is satisfied by a skeleton-state $s_i = \overline{x}^{i+1}$ that is not known to be unreachable. The second big conjunction ensures that for each inductive wall, all skeleton-states lie on the same side of the wall. If the query is unsatisfiable, the language is empty. Otherwise, reachability queries are performed to check whether the skeleton-states can form a reachable fair cycle. The stem query on line 5 checks if s_0 is reachable from some initial state. If not, the inductive proof returned by the safety model checker is added to R. If s_0 is reachable, the algorithm proceeds to perform cycle queries (line 8). If any query fails, the inductive proof is added to the set of walls.

The IIV-based algorithm for model checking CTL properties described in Chapter 3 poses queries to determine whether a reachable fair cycle exists. For that, it is convenient to define a function $\mathsf{fair}(S, C, F)$ that accepts a system S (possibly with fairness constraints $\{B_1, \ldots, B_\ell\}$), a set of constraints $C(\overline{x}, \overline{x}')$ on the transition relation, and an initial condition F. Function fair returns either an F-reachable fair cycle, or an inductive assertion $P(\overline{x})$, where $F \Rightarrow P$, describing a set of states that lack reachable fair cycles. Function fair can be implemented using the Fair algorithm through setting the system's initial condition to F and its transition relation to $T \wedge C$.

2.8 Efficient Usage of Incremental SAT Solvers

SAT-based model checking algorithms pose many queries to a SAT solver. The different queries often share a common set of clauses, e.g., the transition relation clauses. Incremental SAT solvers, for example zChaff [48] and MiniSat [32], efficiently handle such usage pattern by providing mechanisms to add clauses to an existing SAT database, and keeping the conflict clauses between the different calls to the solver in order to avoid repeating the work needed to derive those clauses.

In some cases, clauses added to the SAT database are to be asserted in some queries and disabled in others. This is achieved through the usage of **activation literals**. An activation literal is a "fresh" variable that is added to a clause that would need to be enabled in some queries and disabled for others. Such a clause $c = l_1 \vee \ldots \vee l_n$ is added to the SAT database as $\neg a \vee l_1 \vee \ldots \vee l_n$, where a is an activation literal, so that one can then assert a or $\neg a$ depending on whether it is desired to enable or disable c, respectively. To allow asserting a literal or a set of literals for a single query, incremental SAT solvers provide a mechanism for passing a **list of unit assumptions**, which they conjoin with their current clause database for a specific query.

If the SAT instance turns out to be UNSAT, incremental SAT solvers also provide a mechanism by which a subset of the unit assumptions necessary to make the instance UNSAT can be extracted.

Chapter 3

Incremental, Inductive CTL Model Checking

Incremental, inductive verification (IIV) algorithms construct proofs by generating lemmas based on concrete hypothesis states. Through inductive generalization, a lemma typically provides significantly more information than is required to address the hypotheses. A principle of IIV is that each lemma holds **relative to** previously generated lemmas (hence the term **incremental**), so that the difficulty of lemma generation is fairly uniform throughout execution. Because lemmas are generated in response to concrete hypothesis states, property-directed abstraction is achieved. The safety model checker IC3 [17, 14] and the progress model checker Fair [16] are both incremental, inductive model checkers. IC3 generates stepwise relatively inductive clauses in response to states that lead to property violations. Fair generates inductive information about reachability and SCC-closed sets in response to sets of states that together satisfy every fairness constraint. This chapter describes an incremental, inductive model checker, IICTL, for deciding CTL properties of finite state systems, possibly with fairness constraints.

An investigation into an IIV model checker for CTL properties is important for several reasons. First, CTL is a historically significant specification language. Second, some properties like **resetability** ($\mathsf{AG}\,\mathsf{EF}\,p$ in CTL) require branching time semantics. Third, on properties in the fragment common to CTL and LTL, traditional CTL algorithms are sometimes superior to traditional LTL algorithms. CTL model checking is inherently hierarchical in that a CTL property can be analyzed according to its parse graph. In the context of IIV, the strategy that IICTL applies to such properties is different than that applied by Fair. Finally, CTL offers a conceptual challenge that previous IIV algorithms, IC3 and Fair, do not address: branching time semantics. In particular, CTL motivates generalizing counterexample traces in addition to using proof-based generalization.

The first approaches to SAT-based CTL model checking [3, 46] were global algorithms that leveraged the ability of CNF formulae and Boolean circuits to be reasonably sized in some cases when BDDs are not. These approaches differ from IICTL, which is an incremental, local algorithm. A few attempts [50, 60, 59, 49] have been made to extend bounded model checking to branching time; they are all restricted to universal properties, though, and they have not received an extensive experimental evaluation. Their effectiveness thus remains unclear.

The lack of practical SAT-based model checkers for CTL is the gap that IICTL fills. The success of IC3 and Fair motivates following the IIV approach in the design of a SAT-based CTL model checker. IICTL builds on traditional parse graph-based analyses, except that it eschews the standard global, or bottom-up, approach in favor of a task-directed top-down strategy. IICTL assigns a task to a node in the parse graph to **decide** whether a state satisfies the subformula rooted at this node. States for which tasks are created—task states—are those on which the truth or falsity of the CTL property depends; the first task states are the initial states of the system for which the root node decides whether they satisfy the CTL formula. In the process of making a decision, a node can in turn generate a set of tasks for its children, and so on. Depending on the root operator of the node, it applies a SAT solver (for EX nodes), a safety model checker such as IC3 (for EU nodes), or a fair cycle finder such as Fair (for EG nodes), to investigate the status of the task states. Once it reaches a conclusion, it generalizes the witness—either a proof or a counterexample trace—to extend the decision reached for the task state to other states.

Section 3.1 gives an overview of IICTL. Section 3.2 describes the algorithm in detail. Section 3.3 proves total correctness of the algorithm. Section 3.4 discusses an essential component of IICTL: generalizing decisions. Then Section 3.5 describes a refinement to the basic algorithm, while Section 3.6 describes the additions for handling fairness constraints. Section 3.7 presents the results of an implementation of IICTL within the IImc model checker [38].

3.1 An Overview of IICTL

Global, or fixpoint-based, CTL model checkers compute exact sets of states satisfying each subformula of the CTL formula, hence the need to proceed bottom-up on the formula's parse graph. IICTL instead maintains overapproximations and underapproximations of the set of states satisfying each subformula. The approximations are represented using propositional formulae and are initialized bottom-up, leveraging properties of CTL temporal operators, for example that states that satisfy $\mathsf{EG}\,\psi$ are a subset of those that satisfy ψ (hence the overapproximation of $\mathsf{EG}\,\psi$-states can be initially set to the overapproximation of ψ-states), and that all states that satisfy φ also satisfy $\mathsf{E}\,\psi\,\mathsf{U}\,\varphi$ (hence the underapproximation of $\mathsf{E}\,\psi\,\mathsf{U}\,\varphi$-states can be initially set to the underapproximation of φ-states), and so on.

Using these approximations, IICTL attempts to answer questions that relate to the truth or falsity of the property. In some cases, the answer to a question can be inferred from the current approximations, and in other cases, refinement of the approximations is necessary to arrive at an answer. The first question IICTL asks is: does every initial state satisfy the formula? To infer an answer with only approximations to the actual set of states satisfying the formula available, IICTL resorts to asking two questions (via performing SAT queries):

(1) Is every initial state contained in the underapproximation of the states that satisfy the formula?

(2) Is there an initial state not contained in the overapproximation of the states that satisfy the formula?

On the one hand, if the answer to the first question is yes, then every initial state definitely satisfies the formula, which indicates that the property holds. On the other hand, nothing can be deduced from a negative answer: an initial state not contained in the underapproximation may still satisfy the property. IICTL, thus, proceeds to the second question. A positive answer to the second question—an initial state that is not contained in the overapproximation—implies that such initial state definitely does not satisfy the formula, and thus the property fails. But, again, a negative answer does not necessarily imply that the property holds: a state that is contained in the overapproximation is not guaranteed to satisfy the formula. Thus, when both questions are answered negatively, nothing can be inferred; an initial state s is not contained in the underapproximation but every initial state (including s) is contained in the overapproximation. Such a state s is therefore an undecided state with respect to the current approximations. Answering the original question—does every initial state satisfy the formula?—depends on whether s satisfies the formula. Hence, IICTL proceeds by creating a task to decide whether s satisfies the formula. The task is assigned to the node for which s is undecided—the root node in this case.

A node decides a state by performing an action that depends on the node's type: CTL temporal operators perform an appropriate query, and Boolean connectives push the task down to their children. Consider for example an EX node: for s to satisfy $\mathsf{EX}\,\varphi$, it must have a successor to a φ-state, which can be determined via a SAT query. However, φ-states are not precisely known: only approximations of them are. Therefore, IICTL is instead forced to perform two SAT queries to determine whether s satisfies $\mathsf{EX}\,\varphi$: a **lower bound query** to determine if s has a successor that satisfies φ's underapproximating formula, and an **upper bound query** to determine if s has a successor that satisfies φ's overapproximating formula. Similar to before, a positive answer to the lower bound query immediately determines that s satisfies the formula, and a negative answer to the upper bound query immediately determines that it does not. However, when the lower bound query is answered negatively and the upper bound query is answered positively, there is a successor to s (say t) in the overapproximation of φ, but no successors in its underapproximation (not even t). This indicates that t is undecided for φ and that deciding t is necessary for deciding s. The node labeled φ is therefore tasked with deciding t. Once t is decided and the approximations are updated, the upper and lower bound queries involving s are repeated[1] . The same two-query approach applies to EU and EG nodes, except that SAT queries are replaced with reachability and fair-cycle queries, respectively.

[1] In some cases, repeating both queries is not necessary.

An important aspect of IICTL (as well as other IIV algorithms) is the generalization of conclusions. For example, once it is determined whether t satisfies φ, this result is generalized to include other states, which, otherwise, IICTL may have had to decide later. The way to generalize a conclusion depends on the type of query performed (a SAT, a reachability, or a fair-cycle query) and on the conclusion (whether a proof or a counterexample). Chapter 4 presents procedures for generalization for the different situations that arise in IICTL.

Several refinements of the basic algorithm described above are possible. One of the most important is to maintain an approximation of states reachable from the initial states of the system. This approximation, initially $\top$, is refined using inductive invariants generated by the safety model checker, and is used to constrain all queries—which often speeds them up significantly—and as don't care information during generalization—which often allows much stronger generalizations.

3.2 Algorithm

The input to IICTL consists of a finite-state system S and the parse graph of a CTL formula φ. Each node of the parse graph is a natural number v and is labeled with a token from φ. Node 0 is the root of the graph. The formula rooted at v is denoted by ψ_v, so that, in particular, $\psi_0 = \varphi$. IICTL annotates each node v with two propositional formulae over the state variables: U_v and L_v that are used to compute an upper bound formula $\mathcal{U}_v$ and a lower bound formula $\mathcal{L}_v$ (discussed later) which approximate the satisfying set $[\![\psi_v]\!]$ of the formula ψ_v in S. Initial approximations are computed bottom-up as shown in Table 3.1. A global approximation of the states of S reachable from the initial states is maintained as inductive propositional formula R. Initially, $R = \top$; that is, all states are presumed reachable[2] .

Throughout execution, IICTL maintains the following invariant[3] :

$$[\![R \wedge U_v \wedge L_v]\!] \subseteq [\![R \wedge \psi_v]\!] \subseteq [\![R \wedge U_v]\!] \ . \tag{3.1}$$

[2] R can also be initialized with reachability information available from a previous IICTL run or derived by a different proof engine.

[3] This invariant is weaker than the more intuitive one, $[\![L_v]\!] \subseteq [\![\psi_v]\!] \subseteq [\![U_v]\!]$. Maintaining the weaker form is a key enabler to much stronger generalizations in IICTL.

Table 3.1: Initial bounds for IICTL

ψ_v	L_v	U_v
$a \in A$	a	a
$\neg\psi_i$	$\neg U_i$	$\neg L_i$
$\psi_i \wedge \psi_j$	$L_i \wedge L_j$	$U_i \wedge U_j$

ψ_v	L_v	U_v
$\mathsf{EX}\,\psi_i$	$\bot$	$\top$
$\mathsf{E}\,\psi_j\,\mathsf{U}\,\psi_i$	L_i	$U_i \vee U_j$
$\mathsf{EG}\,\psi_i$	$\bot$	U_i

All states of the left set definitely satisfy ψ_v, thus the set underapproximates $[\![R \wedge \psi_v]\!]$; all states not in the right set definitely do not satisfy ψ_v or are unreachable, thus the set overapproximates $[\![R \wedge \psi_v]\!]$. A state s of the system S such that $s \models R \wedge U_v$ but $s \not\models R \wedge U_v \wedge L_v$—together, $s \models R \wedge U_v \wedge \neg L_v$—is **undecided** for ψ_v. The algorithm incrementally refines the approximations by considering undecided states until either every initial state of S is determined to satisfy φ, proving $S \models \varphi$, or an initial state $\hat{s}$ is found such that $\hat{s} \not\models U_0$, proving $S \not\models \varphi$.

Let $\mathcal{L}_v = R \wedge U_v \wedge L_v$ designate the **lower bound** states: those states that are known to satisfy ψ_v. Let $\mathcal{U}_v = R \wedge U_v$ designate the **upper bounds** states: those states that are not known not to satisfy ψ_v. Invariant (3.1) is then written $[\![\mathcal{L}_v]\!] \subseteq [\![R \wedge \psi_v]\!] \subseteq [\![\mathcal{U}_v]\!]$. Finally, let $\mathcal{A}_v = \mathcal{U}_v \wedge \neg\mathcal{L}_v = R \wedge U_v \wedge \neg L_v$ designate the undecided states of node v.

If ever $I \wedge \neg U_0$ becomes satisfiable, then IICTL concludes that $S \not\models \varphi$: not even the overapproximation $\mathcal{U}_0$ of φ contains all I-states, so neither can φ itself. If instead $I \wedge \neg(L_0 \wedge U_0)$ becomes unsatisfiable, then $S \models \varphi$: the underapproximation $\mathcal{L}_0$ of φ contains all I-states, so φ itself must as well.

Otherwise, one or more initial undecided states must be **decided**. At the top level, a witness s to the satisfiability of $I \wedge U_0 \wedge \neg L_0$ is undecided; it is decided by calling the recursive function decide with arguments s and 0, the root of the parse tree of φ, which eventually returns true if $S, s \models \varphi$ and false otherwise. In general, decide(t, v) return true iff $S, t \models \psi_v$. A call to decide(t, v) can update L_v or U_v (or both) so that state t becomes decided for ψ_v; moreover, the call can trigger a cascade of recursive calls that update the bounds of descendants of v and, crucially, may decide many states besides t. The pseudocode for decide in Listing 3.1 provides structure to the following discussion.

Listing 3.1: Basic version of the main recursive function.

```
bool decide($t$ : state, $v$ : node):
  **if** $t \models R \wedge U_v \wedge L_v$: **return true** {*already decided: $S, t \models \psi_v$*}
  **if** $t \not\models R \wedge U_v$: **return false** {*already decided: $S, t \not\models \psi_v$*}
  **match** $\psi_v$ **with**:
  $\psi_u \wedge \psi_w$: [**update** $L_v$, $U_v$ := $L_u \wedge L_w$, $U_u \wedge U_w$]
    **return** decide($t$, $u$) $\wedge$ decide($t$, $w$)
  $\neg\psi_u$: [**update** $L_v$, $U_v$ := $\neg U_u$, $\neg(L_u \wedge U_u)$]
    **return** $\neg$decide($t$, $u$)
  EX$\,\psi_u$:
    **if** $t \wedge \mathcal{U}_v \wedge T \wedge U'_u$ **is unsat**: {*with $\hat{t} \subseteq t$ from core*}
      $U_v$ := $U_v \wedge \neg$generalize($\hat{t}$)
      **return false**
    **else**: {*with $t$–successor $s$*}
      **if** $t \wedge T \wedge L'_u \wedge U'_u$ **is sat**:
        $L_v$ := $L_v \vee$ generalize($t$)
        **return true**
      **else**:
        decide($s$, $u$)
        **return** decide($t$, $v$)
  E$\,\psi_u\,$U$\,\psi_w$: [**update** $L_v$, $U_v$ := $L_v \vee L_w$, $U_v \wedge (U_u \vee U_w)$]
    **if** $\neg$reach($S$, $U_u \wedge U_v \wedge R \wedge U'_v$, $t$, $U_w$): {*with proof $P$*}
      $U_v$ := $U_v \wedge \neg$generalize($P$)
      **return false**
    **else**: {*with trace $s_0 = t$, $s_1$, ..., $s_n$*}
      **if** reach($S$, $\mathcal{L}_u \wedge U_v \wedge U'_v$, $t$, $L_v \wedge U_v$): {*with trace $\bar{r}$*}
        $L_v$ := $L_v \vee$ generalize($\bar{r}$)
        **return true**
      **elif** decide($s_i$, $u$), $0 \le i < n$, **and** decide($s_n$, $w$) **are true**:
        $L_v$ := $L_v \vee$ generalize($s_0, \ldots, s_n$)
        **return true**
      **else**: **return** decide($t$, $v$)
  EG$\,\psi_u$: [**update** $U_v$ := $U_v \wedge U_u$]
    **if** $\neg$fair($S$, $U_v \wedge R \wedge U'_v$, $t$): {*with assertion $P$*}
      $U_v$ := $U_v \wedge \neg$generalize($P$)
      **return false**
    **else**: {*with trace $s_0 = t$, ..., $s_k$, ..., $s_n$, $s_k$*}
      **if** fair($S$, $L_u \wedge U_v \wedge R \wedge L'_u \wedge U'_v$, $t$): {*with trace $\bar{r}$*}
        $L_v$ := $L_v \vee$ generalize($\bar{r}$)
        **return true**
      **elif** decide($s_i$, $u$), $0 \le i \le n$, **are true**:
        $L_v$ := $L_v \vee$ generalize($s_0, \ldots, s_n$)
        **return true**
      **else**: **return** decide($t$, $v$)
```

3.2.1 Atomic Propositions and Boolean Nodes.

According to Table 3.1, no state can be undecided for a propositional node because the initial approximations are exact; since decide is never called with a state that does not satisfy R, in the case that v is a propositional node, one of the conditions of lines 2–3 holds.

If $\psi_v = \psi_u \wedge \psi_w$, the following invariant is maintained:

$$U_v = U_u \wedge U_w \quad \text{and} \quad L_v = L_u \wedge L_w \ . \tag{3.2}$$

If t is undecided on entry, then recurring on nodes u and w decides t for v (line 6). The **update** statement (line 5; also lines 7, 20, and 32) indicates that L_v and U_v should be updated whenever a child's bound is updated during recursion. It does not express an invariant.

If $\psi_v = \neg\psi_u$, the following invariant is maintained:

$$U_v = \neg(L_u \wedge U_u) \quad \text{and} \quad L_v = \neg U_u \ . \tag{3.3}$$

If t is undecided on entry, then recurring on node u decides t for v (line 8).

3.2.2 EX Nodes.

If $\psi_v = \mathsf{EX}\,\psi_u$, then the undecided question is whether t has a successor satisfying ψ_u. IICTL executes two SAT queries in order to answer this question. First, it executes an **upper bound query**. Naively, this query is $t \wedge T \wedge U_u'$, which asks whether t has a U_u-successor. However, for better generalization, the following is used instead (line 10):

$$t \wedge \mathcal{U}_v \wedge T \wedge U_u' \ . \tag{3.4}$$

If unsatisfiable, the core reveals cube $\hat{t} \subseteq t$ such that all $\hat{t}$-states (including t) lack U_u-successors (and thus ψ_u-successors) or are unreachable. U_v is then updated to $U_v \wedge \neg\hat{t}$ (line 11)—no $\hat{t}$-state is a ψ_u-state (or it is an unreachable ψ_u-state).

However, if query (3.4) is satisfiable, the witness reveals successor U_u-state s (line 13). A **lower bound query** is executed next (line 14):

$$t \wedge T \wedge L_u' \wedge U_u' \ . \tag{3.5}$$

If satisfiable, then t itself has been decided: it definitely has a ψ_u-successor, since it has a $(U_u \wedge L_u)$-successor (recall invariant (3.1)). Forall-exists generalization (Section 4.2) then produces a cube $\hat{t} \subseteq t$ of states that definitely have ψ_u-successors (or are unreachable), and L_v is updated to $L_v \vee \hat{t}$ (line 15). If the query is unsatisfiable (line 17), then state s is undecided for u. In this case, $\mathsf{decide}(s,\ u)$ is called (line 18), which results in updates to at least one of U_u and L_u, which is a form of progress. The entire process iterates until t is decided (line 19).

3.2.3 EU Nodes.

An EU-node, $\psi_v = \mathsf{E}\, \psi_u \,\mathsf{U}\, \psi_w$, maintains the following invariant:

$$[\![\mathcal{L}_w]\!] \subseteq [\![\mathcal{L}_v]\!] \ , \ [\![\mathcal{L}_v]\!] \subseteq [\![\mathcal{L}_u]\!] \cup [\![\mathcal{L}_w]\!] \ , \ [\![\mathcal{U}_w]\!] \subseteq [\![\mathcal{U}_v]\!] \ , \ [\![\mathcal{U}_v]\!] \subseteq [\![\mathcal{U}_u]\!] \cup [\![\mathcal{U}_w]\!] \ . \tag{3.6}$$

The undecided question is whether t has a ψ_u-path to a ψ_w-state. To answer this question, it executes two reachability queries using an engine capable of returning counterexample traces and inductive proofs, such as IC3 [17, 14] (see Section 2.6 for a description of reach).

The **upper bound query** asks whether t leads to a U_w-state (line 21). The following query determines if it can reach a U_w-state via a U_u-path:

$$\mathsf{reach}(S,\ U_u \wedge U_v \wedge R \wedge U_v',\ t,\ U_w)\ . \tag{3.7}$$

The transition relation constraint $U_u \wedge U_v \wedge R \wedge U_v'$ mixes the necessary (U_u) with the optimizing $(U_v \wedge R \wedge U_v')$. If the query is unsatisfiable, the returned inductive proof P shows that no U_w-state can be reached via a potentially reachable $(U_u \wedge U_v)$-path, deciding at least t and leading to the update of U_v to $U_v \wedge \neg P$ (line 22). If query (3.7) is satisfiable, let $s_0 = t, s_1, \ldots, s_n$ be the returned counterexample trace (line 24).

Lower bound queries are executed next (line 25). decide asks whether t can reach a known ψ_v-state via a known ψ_u-path:

$$\mathsf{reach}(S,\ \mathcal{L}_u \wedge U_v \wedge U_v',\ t,\ L_v \wedge U_v)\ . \tag{3.8}$$

The target set has those states that are known to have ψ_u-paths to ψ_w-states. If the query is satisfiable, t also has a ψ_u-path to a ψ_w-state. Forall-exists generalization (Chapter 4) produces a set of states F, including t, that definitely have ψ_u-paths to ψ_w-states or are unreachable. L_v is updated with F (line 26).

However, if the query is unsatisfiable, then attention returns to the trace $s_0, \ldots, s_n$ of the upper bound query (3.7) to decide whether its states satisfy the appropriate subformulae (lines 28–31). Each s_i, $0 \le i < n$, is queried for node u, and s_n is queried for node w. If all states of the trace are decided positively (line 28), then t is decided positively for v; therefore, L_v is expanded by the generalization of the trace (line 29). If one of the states is decided negatively, the upper and lower bound queries are iterated until t is decided (line 31): either a trace is found, or the nonexistence of such a trace is proved.

3.2.4 EG Nodes.

An EG-node, $\psi_v = \mathsf{EG}\,\psi_u$, maintains the following invariant:

$$[\![\mathcal{L}_v]\!] \subseteq [\![\mathcal{L}_u]\!] \quad \text{and} \quad [\![\mathcal{U}_v]\!] \subseteq [\![\mathcal{U}_u]\!] \ . \tag{3.9}$$

The undecided question is whether there exists a reachable fair cycle all of whose states are ψ_u-states. To answer this question, it executes two **fair cycle queries** using an engine capable of returning (1) fair cycles and (2) inductive reachability information describing states that lack a reachable fair cycle. The function $\mathsf{fair}(S, C, F)$ defined in Section 2.7 can be used for this purpose.

The **upper bound query** asks whether a reachable fair cycle whose states satisfy U_u exists. The constraint on the transition relation uses U_v because states of a counterexample should potentially be $\mathsf{EG}\,\psi_u$ states (line 33):

$$\mathsf{fair}(S,\ U_v \wedge R \wedge U_v',\ t) \ . \tag{3.10}$$

If the query is unsatisfiable, the returned inductive assertion P describes states, including t, that do not have reachable fair cycles (line 33); hence, U_v is updated to $U_v \wedge \neg P$ (line 34). Otherwise, a reachable fair cycle $s_0 = t, \ldots, s_k, \ldots, s_n, s_k$ is obtained (line 36).

Before exploring the trace, a **lower bound query** is executed (line 37) to determine whether a reachable fair $\mathcal{L}_u$-cycle exists[4] :

$$\mathsf{fair}(S,\ \mathcal{L}_u \wedge \mathcal{L}'_u,\ t)\ . \tag{3.11}$$

If it is satisfiable, the resulting run is generalized (Chapter 4) to a formula F, and L_v is updated to $L_v \vee F$ (line 38).

Otherwise, the reachable fair cycle from query (3.10) is considered (line 40). If all s_i proved to be ψ_u-states, decide finishes as with a satisfiable lower bound query (lines 41–42). Otherwise, the exploration updates U_v, so that some progress is made, and the process is iterated (line 43).

Even if generalize were to return what it is given, the sound updates to L_v (lines 15, 26, 29, 38, 41) and U_v (lines 11, 22, 34), combined with the progress guaranteed by each call to decide, make the basic version of IICTL correct.

3.3 Proof of Correctness

This section proves the correctness of IICTL for a system with no fairness constraints (i.e., $\mathcal{B} = \emptyset$). Section 3.6 describes how IICTL deals with fairness constraints, and extends this proof of correctness to systems with fairness constraints.

The following lemmas state the correctness of reach and fair, which derive from the correctness of the safety and fair cycle model checkers they invoke.

Lemma 1. $\mathsf{reach}(S, C, F, G)$ *terminates and returns* **true** *iff* $\exists s\,.(s \models F) \wedge (S, s \models \mathsf{E}\, C \,\mathsf{U}\, G)$*. When it returns* **false***, it also returns an assertion* $P(\overline{x})$ *such that*

$$F(\overline{x}) \Rightarrow P(\overline{x})\ ,$$

$$P(\overline{x}) \wedge C(\overline{x}, \overline{x}') \wedge T(\overline{x}, \overline{i}, \overline{x}') \Rightarrow P(\overline{x}'),\ \textit{and}$$

$$P(\overline{x}) \Rightarrow \neg G(\overline{x})\ ,$$

[4] Note that the fair cycle query could potentially be avoided by asking if known ψ_v-states are reachable from t via a ψ_u-path: $\mathsf{reach}(S,\ \mathcal{L}_u \wedge \mathcal{L}'_u,\ t,\ \mathcal{L}_v)$.

Lemma 2. $\mathsf{fair}(S, C, F)$ *terminates and returns* **true** *iff* $\exists s\,.(s \models F) \wedge (S, s \models \mathsf{EG}\,C)$. *When it returns* **false**, *it also returns an assertion* $P(\overline{x})$ *such that*

$$F(\overline{x}) \Rightarrow P(\overline{x}) \ ,$$

$$P(\overline{x}) \wedge C(\overline{x}, \overline{x}') \wedge T(\overline{x}, \overline{i}, \overline{x}') \Rightarrow P(\overline{x}'), \textit{ and}$$

$$\forall s\,.(s \models P) \Rightarrow (S, s \not\models \mathsf{EG}\,C) \ .$$

The following lemmas establish that IICTL maintains invariant (3.1). In particular, Lemma 3 proves the initial bounds satisfy the invariant.

Lemma 3. *The bounds initialized according to Table 3.1 satisfy invariant* (3.1).

Proof. We prove by structural induction that the initial bounds satisfy the stronger invariant:

$$[\![L_v]\!] \subseteq [\![\psi_v]\!] \subseteq [\![U_v]\!] \ .$$

For the base case, the bounds on an atomic proposition a are exact:

$$L_v = \psi_v = U_v = a \ ,$$

and thus obviously satisfy the invariant.

For the inductive step, consider the following cases:

Case $(\psi_v = \neg\psi_u)$. In this case, $L_v = \neg U_u$ and $U_v = \neg L_u$. By the inductive hypothesis, $L_u \Rightarrow \psi_u$, which implies that $\neg\psi_u \Rightarrow \neg L_u$, and therefore $\psi_v \Rightarrow U_v$. Also, the inductive hypothesis states that $\psi_u \Rightarrow U_u$, which implies that $\neg U_u \Rightarrow \neg\psi_u$, from which $L_v \Rightarrow \psi_v$ follows.

Case $(\psi_v = \psi_u \wedge \psi_w)$. In this case, the initial bounds are $L_v = L_u \wedge L_w$ and $U_v = U_u \wedge U_w$. By the inductive hypothesis, $L_u \Rightarrow \psi_u$ and $L_w \Rightarrow \psi_w$. Therefore, $L_u \wedge L_w \Rightarrow \psi_u \wedge \psi_w$. It follows that $L_v \Rightarrow \psi_v$. By similar arguments, $\psi_v \Rightarrow U_v$ can be proven.

Case $(\psi_v = \mathsf{EX}\,\psi_u)$. The trivial bounds $L_v = \bot$ and $U_v = \top$ obviously satisfy the invariant.

Case $(\psi_v = \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w)$. The initial bounds are $L_v = L_w$ and $U_v = U_u \vee U_w$. First, consider a state s such that $s \models L_v$. We have

$$s \models L_v \iff s \models L_w \ .$$

By the inductive hypothesis, $s \models \psi_w$. The semantics of CTL (see Section 2.3) imply that $s \models \mathsf{E}\,\varphi\,\mathsf{U}\,\psi_w$ for any assertion φ, and in particular $\varphi = \psi_u$. Therefore

$$s \models L_v \implies s \models \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w \implies s \models \psi_v \ .$$

For the upper bound, consider a state s such that $s \not\models U_v$. Thus,

$$s \models \neg U_v \implies s \models \neg U_u \wedge \neg U_w \implies s \models \neg U_u \text{ and } s \models \neg U_w \implies s \not\models U_u \text{ and } s \not\models U_w.$$

From the inductive hypothesis, it follows that

$$s \not\models \psi_u \text{ and } s \not\models \psi_w \ .$$

From the semantics of CTL,

$$s \not\models \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w \ ,$$

and therefore

$$s \not\models \psi_v \ .$$

Case ($\psi_v = \mathsf{EG}\,\psi_u$). The trivial lower bound obviously satisfies the invariant. For the upper bound $U_v = U_u$, consider a state s such that $s \not\models U_v$,

$$s \not\models U_v \implies s \not\models U_u \implies s \not\models \psi_u \implies s \not\models \mathsf{EG}\,\psi_u \implies s \not\models \psi_v \ .$$

$\square$

The main reason IICTL maintains invariant (3.1) rather than its stronger, more intuitive form ($[\![L_v]\!] \subseteq [\![\psi_v]\!] \subseteq [\![U_v]\!]$) is that (3.1) allows expanding $[\![L_v]\!]$ beyond $[\![\psi_v]\!]$, and shrinking $[\![U_v]\!]$ beyond $[\![\psi_v]\!]$. The following two lemmas prove the sufficient conditions for generalization to maintain invariant (3.1).

Lemma 4. *If an assertion P satisfies $\forall s\,.(s \models P) \Rightarrow (s \not\models \psi_v \wedge R \wedge U_v)$ (i.e., $s \not\models \psi_v \wedge \mathcal{U}_v$), then updating U_v through conjoining $\neg P$ maintains invariant* (3.1).

Proof. Let U_v^{old} and U_v^{new} be the values of U_v before and after the update, respectively. Before the update, invariant (3.1) holds, thus,

$$R \wedge \psi_v \Rightarrow R \wedge U_v^{old} \ . \tag{3.12}$$

After the update, the consequent becomes $R \wedge U_v^{new} = R \wedge U_v^{old} \wedge \neg P$. Every state that gets removed is either a $\neg\psi_v$-state, and is thus not an $(R \wedge \psi_v)$-state, or a $\neg(R \wedge U_v^{old})$-state, which by (3.12) is also not an $(R \wedge \psi_v)$-state. Thus, the invariant is maintained. ☐

Lemma 5. *If an assertion W satisfies $\forall s\,.(s \models W) \Rightarrow (s \models \psi_v \vee \neg R \vee \neg U_v \vee L_v)$ (i.e., $s \models \psi_v \vee \neg \mathcal{A}_v$), then updating L_v through disjoining W maintains invariant* (3.1).

Proof. Let L_v^{old} and L_v^{new} be the values of L_v before and after the update, respectively. Since invariant (3.1) holds before the update, then:

$$R \wedge U_v \wedge L_v^{old} \Rightarrow R \wedge \psi_v \ . \tag{3.13}$$

By updating L_v to $L_v^{new} = L_v^{old} \vee W$, the antecedent becomes $R \wedge U_v \wedge (L_v^{old} \vee W) = (R \wedge U_v \wedge L_v^{old}) \vee (R \wedge U_v \wedge W)$. By 3.13, the first disjunct satisfies the invariant. For the second disjunct, a W-state s can be one of the following:

Case $(s \models \psi_v)$. In this case, the second disjunct also satisfies $R \wedge \psi_v$.

Case $(s \models \neg R$ or $s \models \neg U_v)$. In either case, the second disjunct reduces to $\bot$ and thus vacuously satisfies $R \wedge \psi_v$.

Case $(s \models L_v^{old})$. By (3.13), s also satisfies $R \wedge \psi_v$.

Thus, the invariant is maintained. ☐

Section 3.4 shows that generalization in IICTL produces assertions that satisfy the conditions of Lemma 4 (for proofs) or Lemma 5 (for counterexamples). This is stated in the following lemma:

Lemma 6. *Given a proof (a counterexample),* generalize *produces an assertion that satisfies the condition of Lemma 4 (Lemma 5).*

Generalization, in fact, satisfies a stronger condition, but the conditions of Lemmas 4 and 5 are sufficient for proving partial correctness of decide as stated in the following lemma:

Lemma 7. *The recursive function* decide(t, v) *updates the bounds of node* v *while maintaining invariant* (3.1)*, and returns* **true** *iff* $S, t \models R \wedge \psi_v$*.*

Proof. The proof is by induction on the structure of ψ_v. For the base case, v can only be an atomic proposition node, $\psi_v = a$. Since no updates occur to the bounds of atomic proposition nodes, L_v and U_v remain at their initial value of a. Because of that, the conditions on lines 2 and 3 reduce to $t \models R \wedge a$ and $t \not\models R \wedge a$. Clearly, t must satisfy one of the two conditions, and therefore, decide returns **true** iff $t \models R \wedge \psi_v$.

For the inductive step, state t is either decided or undecided for node v. If it is decided, one of the conditions on lines 2 or 3 is satisfied, and if it is not, decide proceeds to decide t depending on its type. If the condition on line 2,

$$t \models R \wedge U_v \wedge L_v \ ,$$

is satisfied, by the inductive hypothesis, since no updates to the bounds have been made in the current decide call, invariant (3.1) holds, which implies that $t \models R \wedge \psi_v$. Thus, decide returns **true**. Otherwise, if the condition on line 3:

$$t \not\models R \wedge U_v$$

is satisfied, then, again, the inductive hypothesis implies that $t \not\models R \wedge \psi_v$, and therefore decide returns **false**.

If the conditions on line 2 and 3 are not satisfied, t is undecided:

$$t \models R \wedge U_v \wedge \neg L_v \ , \tag{3.14}$$

so decide proceeds to match ψ_v, and one of the following cases occurs:

Case $(\psi_v = \neg\psi_u)$. In this case, decide(t, v) returns **false** iff decide(t, u) returns **true**. By the inductive hypothesis, decide(t, u) returns **true** iff $t \models R \wedge \psi_u$, i.e., t is potentially reachable and satisfies ψ_u. But if it satisfies ψ_u then it must not satisfy $\psi_v = \neg\psi_u$. Therefore, $t \not\models \psi_v$, and consequently, $t \not\models R \wedge \psi_v$. Therefore, returning **false** is correct. For the bound updates on line 7, the inductive hypothesis guarantees that $\mathsf{decide}(t,u)$ updates the bounds of ψ_u such that invariant (3.1) is maintained. Now, consider a state s such that $s \models R \wedge U_v \wedge L_v$. We have:

$$\begin{aligned} s \models R \wedge U_v \wedge L_v &\iff s \models R \wedge (\neg L_u \vee \neg U_u) \wedge \neg U_u \\ &\iff s \models R \wedge \neg U_u \end{aligned}$$

But if s satisfies $\neg U_u$, by the inductive hypothesis, it satisfies $\neg(R \wedge \psi_u)$, which implies that satisfies $\neg(R \wedge \neg\psi_v)$. But, by the previous argument, $s \models R$, and thus, $s \models R \wedge \psi_v$. Therefore, $[\![R \wedge U_v \wedge L_v]\!] \subseteq [\![R \wedge \psi_v]\!]$.

Next, consider a state s such that $s \models R \wedge \psi_v$. Then,

$$s \models R \wedge \psi_v \iff s \models R \wedge \neg\psi_u$$

By the inductive hypothesis, if $s \not\models \psi_u$, it must also be the case that $s \not\models R \wedge U_u \wedge L_u$, i.e., $s \not\models R \wedge \neg U_v$. But since s is potentially reachable, $s \models R \wedge U_v$. Therefore, $[\![R \wedge \psi_v]\!] \subseteq [\![R \wedge U_v]\!]$. Thus, the updates on line 7 maintain invariant (3.1).

Case ($\psi_v = \psi_u \wedge \psi_w$). In this case, $\mathsf{decide}(t,v)$ returns **true** iff both $\mathsf{decide}(t,u)$ and $\mathsf{decide}(t,w)$ return **true**. By the inductive hypothesis, $\mathsf{decide}(t,u)$ and $\mathsf{decide}(t,w)$ return **true** iff $t \models R \wedge \psi_u$ and $t \models R \wedge \psi_w$, which holds iff $t \models R \wedge \psi_u \wedge \psi_w$, which is equivalent to $t \models R \wedge \psi_v$.

For the bound updates, the inductive hypothesis implies that invariant (3.1) holds for ψ_u and ψ_w. Thus, v's new lower bound: $R \wedge U_v \wedge L_v \Leftrightarrow R \wedge U_u \wedge U_w \wedge L_u \wedge L_w \Rightarrow R \wedge \psi_u \wedge \psi_w \Leftrightarrow R \wedge \psi_v$ satisfies the invariant. A similar argument applies to v's new upper bound.

Case ($\psi_v = \mathsf{EX}\,\psi_u$). If $t \wedge T \wedge U_u'$ is **unsat**, then none of the successors of t is a U_u-state. By the inductive hypothesis, none of the successors of t are ψ_u-states. Therefore, $t \not\models \mathsf{EX}\,\psi_u$, and consequently $t \not\models R \wedge \psi_v$; accordingly, decide returns **false**. The addition of $\mathcal{U}_v$ to the query does not affect its satisfiability since by (3.14), $t \models \mathcal{U}_v$. However, the addition of $\mathcal{U}_v$ may reduce the subset of t's literals that are used by the solver to prove the query **unsat**: every $\hat{t}$-state can either be a $\neg\mathcal{U}_v$-state or a $\neg\psi_v$-state. But then, $\hat{t}$ satisfies the condition of Lemma 4. In addition, by Lemma 6, $\mathsf{generalize}(\hat{t})$ also satisfies the condition of Lemma 4, and thus, the update maintains invariant (3.1).

Otherwise, if $t \wedge T \wedge L'_u \wedge U'_u$ is **sat** (line 14), then, by the inductive hypothesis, the $(L_u \wedge U_u)$-successor of t also satisfies ψ_u. It follows from the semantics of CTL that $t \models \mathsf{EX}\,\psi_u$. But by (3.14), $t \models R$. Therefore, $t \models R \wedge \psi_v$, and decide returns **true**. By Lemma 6, $\mathsf{generalize}(t)$ satisfies the condition of Lemma 5, and thus, invariant (3.1) is maintained.

If neither $t \wedge \mathcal{U}_v \wedge T \wedge U'_u$ is **unsat** nor $t \wedge T \wedge L'_u \wedge U'_u$ is **sat**, $\mathsf{decide}(s,\, u)$ is called with the state s that is a successor to t, and that satisfies[5] :

$$s \models R \wedge U_u \text{ and } s \not\models R \wedge U_u \wedge L_u \ .$$

and is thus undecided for node u. By the inductive hypothesis, $\mathsf{decide}(s,\, u)$ returns **true** iff $s \models R \wedge \psi_u$, and either adds s to L_u—in which case $t \wedge T \wedge L'_u \wedge U'_u$ becomes **sat**—or removes it from U_u—in which case $t \wedge \mathcal{U}_v \wedge T \wedge U'_u$ either becomes **unsat** or provides another satisfying assignment. Assuming this process terminates (see Lemma 8), one of the conditions on line 10 or 14 is eventually satisfied in which case $\mathsf{decide}(t,\, v)$ returns **true** iff $t \models R \wedge \psi_v$.

Case ($\psi_v = \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$). If the upper bound query, $\mathsf{reach}(S,\, R \wedge U_u,\, t,\, R \wedge U_w)$, returns **false**, then by Lemma 1, $t \not\models \mathsf{E}(R \wedge U_u)\,\mathsf{U}(R \wedge U_w)$. By the inductive hypothesis, $t \not\models \mathsf{E}(R \wedge \psi_u)\,\mathsf{U}(R \wedge \psi_w)$. However, by (3.14), $t \models R$, and because the transition structure is complete, it must be the case that $t \models \mathsf{AG}\,R$. This implies that $t \not\models \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$, and thus $t \not\models R \wedge \psi_v$. Therefore, decide returns **false**. The addition of $U_v \wedge U'_v$ to the constraints of the reach query on line 21 does not affect its result: every state on a ψ_u-path to a ψ_w-state satisfies $\mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$, and therefore, satisfies $\mathsf{E}\,\psi_v\,\mathsf{U}\,\psi_w$. Since U_v characterizes the potential ψ_v-states, adding $U_v \wedge U'_v$ does not affect the result of the reach query. Also notice that adding R to the target of the reach query is redundant because R is an inductive assertion, and thus, the successor of every R-state is also an R-state. For the bound update on line 22, the inductive hypothesis, and Lemmas 1 and 6 guarantee that P and

[5] Note that since $t \models R$, and s is a t-successor, $s \models R$

$\mathsf{generalize}(P)$ satisfy the condition of Lemma 4. Thus, the update maintains invariant (3.1).

Suppose the upper bound query at line 21 returns **true**. If the lower bound query, $\mathsf{reach}(S, R \wedge U_u \wedge L_u, t, R \wedge U_w \wedge L_w)$, returns **true**, then by Lemma 1, $t \models \mathsf{E}(R \wedge U_u \wedge L_u)\,\mathsf{U}(R \wedge U_w \wedge L_w)$, and therefore also satisfies $\mathsf{E}(R \wedge \psi_u)\,\mathsf{U}(R \wedge \psi_w)$ by the inductive hypothesis, which consequently implies that it satisfies $\mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$. Thus, $t \models \psi_v$. Because $t \models R$, $t \models R \wedge \psi_v$ and decide returns **true**. The same reasoning above applies to why it is sound to add $U_v \wedge U_v'$ to the constraints and remove R from the target. Finally, by Lemma 6, the update to the lower bound on line 26 maintains invariant (3.1).

If neither the query on line 21 returns **false** nor the one on line 25 returns **true**, then decide is called on every state of the counterexample trace of the satisfiable upper bound query. Similar reasoning to the one for the EX case shows that when the process terminates, one of the conditions on line 21 or line 25 is triggered.

Case $(\psi_v = \mathsf{EG}\,\psi_u)$*.* If the upper bound query, $\mathsf{fair}(S, R \wedge U_u, t)$, returns **false**, then by Lemma 2, $t \not\models \mathsf{EG}(R \wedge U_u)$. By the inductive hypothesis, $(R \wedge \psi_u) \Rightarrow (R \wedge U_u)$, it follows that $t \not\models \mathsf{EG}(R \wedge \psi_u)$. But since, by (3.14), $t \models R$, and since the transition structure is complete, it must be the case that $t \models \mathsf{AG}\,R$. The latter fact, together with $t \not\models \mathsf{EG}(R \wedge \psi_u)$, implies that $t \not\models \mathsf{EG}\,\psi_u$, and consequently $t \not\models \psi_v$; thus, $\mathsf{decide}(t, v)$ returns **false**. The addition of $U_v \wedge U_v'$ to the constraints of the fair query on line 33 does not affect its result because every state on a path that is a witness to the satisfaction of $t \models \mathsf{EG}\,\psi_u$, also satisfies $\mathsf{EG}\,\psi_u$ and thus, must satisfy $\mathsf{EG}\,\psi_v$. Finally, by noticing that the update on line 32 guarantees that $U_v \Rightarrow U_u$, U_u is redundant and is therefore not included in the query on line 33.

Before returning, decide updates U_v to $U_v \wedge \neg P$ where P is the assertion returned by fair. By Lemma 2, P satisfies:

$$\forall s\,.(s \models P) \Rightarrow (S, s \not\models \mathsf{EG}(U_u \wedge R))\ .$$

But because a reachable state s satisfies $\mathsf{AG}\,R$, it follows that every P-state is either unreachable or does not satisfy $\mathsf{EG}\,\psi_u$; formally:

$$\forall s\,.(s \models P) \Rightarrow [(s \models R) \Rightarrow (S, s \not\models \mathsf{EG}\,\psi_u)]\ .$$

from which it can be concluded that

$$\forall s\,.(s \models P) \Rightarrow (S, s \not\models \psi_v \wedge R)]\ .$$

Thus, P satisfies the condition of Lemma 4 and consequently invariant (3.1) is maintained.

Suppose the upper bound query returns **true**. If the lower bound query, $\mathsf{fair}(S,\ L_u \wedge U_u \wedge R,\ t)$, returns **true**, then by Lemma 2, $t \models \mathsf{EG}(L_u \wedge U_v \wedge R)$, which by the inductive hypothesis implies that $t \models \mathsf{EG}(R \wedge \psi_u)$. By the CTL semantics, t must also satisfy $\mathsf{EG}\,\psi_u$. Finally, because t is potentially reachable, it follows that $t \models R \wedge \mathsf{EG}\,\psi_u$, and consequently decide returns **true**. By reasoning similar to the one above, the addition of $U_v \wedge L'_u \wedge U'_v$ to the query does not affect the result of the lower bound query.

Since every state on the trace resulting from the satisfiable fair query satisfies ψ_v, it satisfies the condition of Lemma 5, and therefore adding it to the lower bound maintains invariant (3.1).

□

Lemma 8. *$\mathsf{decide}(t, v)$ terminates for any state t and node v.*

Proof. The proof is by induction on the structure of ψ_v. For the base case, atomic proposition nodes, one of the conditions on lines 2 or 3 is satisfied as argued in the proof of Lemma 7, and thus decide terminates without making any recursive calls.

For the inductive step, if the conditions on lines 2 or 3 is satisfied, decide terminates. Otherwise, consider the following cases:

Case ($\psi_v = \neg\psi_u$). By the inductive hypothesis, $\mathsf{decide}(t,\ u)$ terminates. In addition, since t is undecided for v, it must be undecided for u, and thus the recursive call makes progress, and consequently $\mathsf{decide}(t,\ v)$ also does.

Case ($\psi_v = \psi_u \wedge \psi_w$). By the inductive hypothesis, $\mathsf{decide}(t,\ u)$ and $\mathsf{decide}(t,\ w)$ terminate. In addition, because t is undecided for v, it must be undecided for at least one of its children. Thus, one of the recursive decide calls is guaranteed to make progress. $\mathsf{decide}(t,\ v)$, thus, also makes progress.

Case ($\psi_v = \mathsf{EX}\, \psi_u$). If the condition on line 10 or the one on line 14 is satisfied, $\mathsf{decide}(t,\ v)$ terminates. Otherwise, the satisfying assignment s from the upper bound query must be an undecided state for node u. By the inductive hypothesis, the call to $\mathsf{decide}(s,\ u)$ terminates. In addition, Lemma 7 guarantees that s is decided for node u, i.e., is either added to L_u or removed from U_u. The change to the bounds of ψ_u is monotonic and affects one of the queries on line 10 or 14. Since the system has a finite number of states, $\mathsf{decide}(t,\ v)$ can only make a finite number of calls before one of the conditions on line 10 or line 14 is satisfied. Therefore, $\mathsf{decide}(t,\ v)$ terminates.

Case ($\psi_v = \mathsf{E}\, \psi_u\, \mathsf{U}\, \psi_w$). If the condition on line 21 or the one on line 25 is satisfied, $\mathsf{decide}(t,\ v)$ terminates. Otherwise, at least one of the states on the counterexample trace from the upper bound query is undecided for one of ψ_v's children. Thus, at least one of the calls on line 28 is guaranteed to make progress. As in the case of the EX, this changes at least one the queries on lines 21 and 25. Thus, by the same finiteness argument, after a finite number of calls, either the condition on line 21 or the one on line 25 is satisfied.

Case ($\psi_v = \mathsf{EG}\, \psi_u$). Similar reasoning to the EU case applies.

□

Theorem 1. *IICTL terminates and returns* **true** *iff* $S \models \varphi$.

Proof. The top-level function of IICTL calls $\mathsf{decide}(t,\ 0)$ for every undecided initial state t. Since there are a finite number of initial states, it follows from Lemmas 7 and 8 that eventually, either an initial state is found not to satisfy ψ_0 in which case the property fails, or all initial states are found to satisfy ψ_0 in which case the property holds. □

Example 1. Consider resetability, $\varphi = \mathsf{AG}\,\mathsf{EF}\, p = \neg\, \mathsf{EF} \neg\, \mathsf{EF}\, p$, whose parse graph, with initial

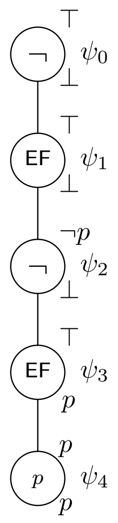


Figure 3.1: $\mathsf{AG}\,\mathsf{EF}\,p$

upper and lower bounds is shown in Figure 3.1. Because the initial states are undecided for 0, IICTL chooses some initial state s and calls decide(s, 0), which in turn calls decide(s, 1). To determine if s is a ψ_1-state, decide queries a safety model checker for the existence of a path from s to U_2, i.e., to a $\neg p$-state. If none exist, inductive proof P is returned, and U_1 is updated by $\neg P$. If counterexample trace $s, \ldots, t$ is found, decide asks whether a path from s to L_2 exists, which is currently impossible. The disagreement between U_2 and L_2 on t triggers calls to decide(t, 2) and decide(t, 3). With equal bounds for node 4, only one reachability query is needed. If t cannot reach p (case 1), the inductive proof eliminates t from U_3 and adds it to L_2. Then s can reach a ψ_2-state, deciding s for 1 positively, and $s, \ldots, t$ are added to L_1. Finally, s is removed from node 0, indicating failure of the property.

Otherwise (case 2), the discovered trace at node 3 is generalized to F, included in L_3, and eliminated from U_2. Then the upper bound reachability query of node 1 is repeated, asking for the existence of a path from s to a $\neg p \wedge \neg F$-state. The procedure continues until either case 1 occurs (failure), or until this query fails, establishing at least that $s \models \psi_0$. Then, decide is invoked again for node 1 with a remaining undecided initial state if any exist, or success of the property is declared.

3.4 Generalization

3.4.1 Generalizing Proofs from Upper Bound Queries

Proofs from upper bound queries provide generalization in one direction: unsatisfiable cores for EX-nodes, inductive unreachability proofs for EU-nodes, and inductive reachability information from fair cycle queries for EG-nodes. Chapter 4 discusses ways to manipulate such proofs to obtain better generalizations.

3.4.2 Generalizing Counterexamples from Lower Bound Queries

An essential aspect of making IICTL work in practice is the ability to generalize from counterexample traces. Chapter 4 describes a powerful generalization technique for traces, called **forall-exists generalization**. This section explains how the procedure is applied in the context of IICTL.

Forall-exists generalization provides two functions: $\mathsf{generalize}(s, G, D)$ and $\mathsf{generalize}(\bar{r}, C, G, D)$ where s is a state, $\bar{r}$ is a sequence of states that form a path, C, G, and D are propositional formulae that denote a path-constraint, a target, and a don't care set. The semantics of the functions are as follows:

- $\mathsf{generalize}(t, G, D)$: Given a state t that is a witness to a satisfiable EX query, $\mathsf{generalize}(t, G, D)$ returns a cube $\hat{t}$ whose literals are a subset of those of t (i.e., $\hat{t} \subseteq t$), such that every $\hat{t}$-state either satisfies D or has a G-successor; formally, $\forall \overline{x} \,.\, \hat{t}(\overline{x}) \rightarrow (\overline{x} \models D) \vee (\overline{x} \models \mathsf{EX}\, G)$.

- $\mathsf{generalize}(\bar{r}, C, G, D)$: Given a trace $\bar{r} = s_0, s_1, \ldots, s_n$ that is a witness to a satisfiable EU or EG query, $\mathsf{generalize}(\bar{r}, C, G, D)$ generalizes $\bar{r}$ to a sequence of cubes $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n$ where $\forall 0 \leq i \leq n \,.\, \hat{s}_i \subseteq s_i$, such that every state $\hat{s}_i$-state either satisfies D or is on a C-path to a G-state; formally: $\forall 0 \leq i \leq n, \overline{x} \,.\, \hat{s}_i(\overline{x}) \rightarrow (\overline{x} \models D) \vee (\overline{x} \models \mathsf{E}\, C\, \mathsf{U}\, G)$.

The implementation of the two $\mathsf{generalize}$ functions is described in Chapter 4. The remainder of this section describes how to use the functions to generalize traces from satisfiable EX, EU, and EG queries.

3.4.2.1 EX Nodes.

For an EX-node, $\psi_v = \mathsf{EX}\,\psi_u$, a satisfiable lower bound query (see (3.5)) gives a satisfying assignment t such that $t \models \mathsf{EX}(L_u \wedge U_u)$. It is desired to generalize t to a cube $\hat{t}$ such that every $\hat{t}$-state also has a $(L_u \wedge U_u)$-successor. This can be achieved by calling $\mathsf{generalize}(t, \mathcal{L}_u, \bot)$. For the returned cube $\hat{t}$, it is guaranteed that every $\hat{t}$-state satisfies $\mathsf{EX}\,\mathcal{L}_u$, which implies that it satisfies $\mathsf{EX}\,\psi_u$. Thus L_v can be updated to $L_v \vee \hat{t}$. One observation that allows a stronger generalization is that since $\mathcal{L}_v$ conjoins L_v with R and U_v, it is safe to include in L_v states that are not in R or are not in U_v without violating invariant 3.1. Why is this necessary? With $\mathsf{generalize}(t, \mathcal{L}_u, \bot)$, dropping a literal from t could fail if the expanded cube has a mixture of states that have $\mathcal{L}_u$-successors, $\neg R$-states, and $\neg U_v$-states. Thus, allowing the inclusion of $\neg R$-states and $\neg U_v$-states could result in a successful generalization that would fail otherwise. This can be achieved by passing $\neg\mathcal{A}_v = \neg(R \wedge U_v \wedge \neg L_v)$ as the third argument to $\mathsf{generalize}$. The resulting cube $\hat{t}$ is such that every $\hat{t}$-state either has an $\mathcal{L}_u$-successor, is unreachable (such states will be filtered by conjoining L_v with R), is not in U_v (such states will be filtered by conjoining L_v with U_v), or is already in L_v. Thus, $\hat{t}$ satisfies the condition of Lemma 5, and thus, maintains the truth of invariant 3.1.

3.4.2.2 EU Nodes.

For an EU-node, $\psi_v = \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$, a satisfiable lower bound query (see (3.8)) gives a witness $\bar{r} = s_0, s_1, \ldots, s_n$ such that $\forall 0 \leq i < n \,.\, s_i \models \mathcal{L}_u \wedge U_v$ and $s_n \models L_v \wedge U_v$. By invariant 3.1, $\forall 0 \leq i \leq n \,.\, s_i \models \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$. Calling $\mathsf{generalize}(\bar{r}, \mathcal{L}_u, \mathcal{L}_v, \bot)$ returns a generalized trace of cubes $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n$ such that every $\hat{s}_i$-state has an $\mathcal{L}_u$-path to an $\mathcal{L}_v$-state. But since every $\mathcal{L}_v$-state satisfies $\mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$, then every $\hat{s}_i$-state also does. Similar to the case of EX-nodes, $\neg\mathcal{A}_v$ can be passed as a don't care set to allow stronger generalizations.

3.4.2.3 EG Nodes.

For an EG-node, $\psi_v = \mathsf{EG}\,\psi_u$, a satisfiable lower bound query (see (3.11)) gives a witness $\bar{r} = s_0, s_1, \ldots, s_n$ such that $\forall 0 \leq i \leq n \,.\, s_i \models \mathsf{EG}\,\mathcal{L}_u$, which by invariant 3.1 implies that $\forall 0 \leq i \leq$

$n\,.\,s_i \models \mathsf{EG}\,\psi_u$. Calling $\mathsf{generalize}(\bar{r}, \mathcal{L}_u, \mathcal{L}_v, \mathcal{A}_v)$ returns a generalized trace of cubes $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n$ such that every $\hat{s}_i$-state has an $\mathcal{L}_u$-path to an $\mathcal{L}_v$-state. But since every $\mathcal{L}_v$-state is on an infinite path of ψ_u-states (i.e., satisfies $\mathsf{EG}\,\psi_u$), then every $\hat{s}_i$-state also is.

3.5 Early Termination

One refinement is immediate. To detect early termination, each time some node u's L_u or U_u is updated, its parent v is notified, and the proper update is made to its L_v and U_v, as explained in Section 3.2. If there is a (semantic) change in at least one of L_v and U_v, then the upward propagation continues. If the root node is modified so that a termination criterion is met ($I \wedge \neg U_0$ is satisfiable or $I \wedge \neg(L_0 \wedge U_0)$ is unsatisfiable), then the proof is complete. Consider the property of Example 1. If it fails, there is at least one trace leading from an initial state to a state s that falsifies $\mathsf{EF}\,p$. The outer EF node would direct IICTL to find such a trace, after which the upper bound query of the inner EF-node would return a proof that s cannot reach a p-state. As soon as the proof is generated, it is evident that the property is false.

3.6 Fairness

Fairness in CTL cannot be handled completely within the logic itself. Instead, model checkers must be able to handle fairness constraints algorithmically when deciding whether a state satisfies an EG formula, a task that IICTL accomplishes by passing the constraints to fair. To show that finite paths computed for other types of formulae can be extended to fair paths, it suffices to show that they end in states that satisfy $\mathsf{EG}\,\top$. Hence, it is customary in BDD-based CTL model checkers to pre-compute the states that satisfy $\mathsf{EG}\,\top$ and constrain the targets of EU and EX computations to them [45].

IICTL instead tries to decide the fairness of as few states as possible. To achieve that, it computes from the given CTL formula φ a modified formula $\tau(\varphi)$ recursively defined as follows:

$$\begin{aligned} \tau(p) &= p & \tau(\mathsf{EG}\,\varphi) &= \mathsf{EG}\,\tau(\varphi) \\ \tau(\neg\varphi) &= \neg\tau(\varphi) & \tau(\mathsf{EX}\,\varphi) &= \mathsf{EX}(\tau(\varphi) \wedge \psi) \\ \tau(\varphi_1 \wedge \varphi_2) &= \tau(\varphi_1) \wedge \tau(\varphi_2) & \tau(\mathsf{E}\,\varphi_1\,\mathsf{U}\,\varphi_2) &= \mathsf{E}\,\tau(\varphi_1)\,\mathsf{U}(\tau(\varphi_2) \wedge \psi)\ , \end{aligned}$$

where

- p is an atomic proposition, and
- $\psi = \top$ if φ is a positive Boolean combination of EX, EU and EG formulae; $\psi = \mathsf{EG}\,\top$ otherwise.

For example, $\tau(\mathsf{AG}\,\mathsf{EF}(p \wedge \neg q)) = \tau(\neg\,\mathsf{EF}\,\neg\,\mathsf{EF}(p \wedge \neg q)) = \neg\,\mathsf{EF}(\neg\,\mathsf{EF}((p \wedge \neg q) \wedge \mathsf{EG}\,\top) \wedge \mathsf{EG}\,\top)$, while $\tau(\mathsf{AG}\,\mathsf{AF}\,p) = \tau(\neg\,\mathsf{EF}\,\mathsf{EG}\,\neg p) = \neg\,\mathsf{EF}\,\mathsf{EG}\,\neg p$.

While the definition of $\tau(\varphi)$ is closely related to the one implicitly used by most BDD-based model checkers—the difference is that in the latter, ψ always equals $\mathsf{EG}\,\top$. This definition minimizes checks for fairness by taking into account that every path with a fair suffix is fair.

For instance, in the case of $\mathsf{AG}\,\mathsf{AF}\,p$, IICTL does not check whether any state satisfies $\mathsf{EG}\,\top$ because the states that satisfy $\mathsf{EG}\,p$ are known to be fair. For the resetability property $\mathsf{AG}\,\mathsf{EF}(p \wedge \neg q)$, however, a state that satisfies $p \wedge \neg q$ is not assumed to satisfy the inner EF node unless it is proved fair.

When applying IICTL to a system S with fairness constraints, the fairness constraints are ignored in the EX and EU computation, but they are passed to fair. This is possible because the transformation from φ to $\tau(\varphi)$ above guarantees that the target of an EX or EU computation is always fair.

3.7 Results

The IICTL algorithm has been implemented in the IImc model checker [38], and it has been evaluated on a set of 79 models (mostly from [57]) for a total of 1245 CTL properties (1058 passing and 187 failing). Of the 79 models, 22 had fairness constraints. The performance of IICTL is compared to that of the BDD-based CTL model checker in VIS-2.4 [57] (with and without preliminary reachability analysis). The experiments have been run on a machine with 2.8 GHz Intel Core i7 CPUs and 9 GB of RAM. A timeout of 900 s was imposed on all runs.

The total run times for the complete collection were: 1952 minutes for IICTL; 5513 minutes and 3390 minutes for VIS with and without reachability analysis. Table 3.2 shows for each of the three CTL algorithms the numbers of timeouts (TO) and the numbers of properties that could be solved by only one technique (US). Only the models for which timeouts occurred are listed. While IICTL obtains the lowest number of timeouts and the highest number of unique solutions, it is apparent that the three methods have different strengths and thus are complementary. This point is further brought out by the plots of Figure 3.2 that shows the comparison of IICTL to VIS in the form of scatter plots.

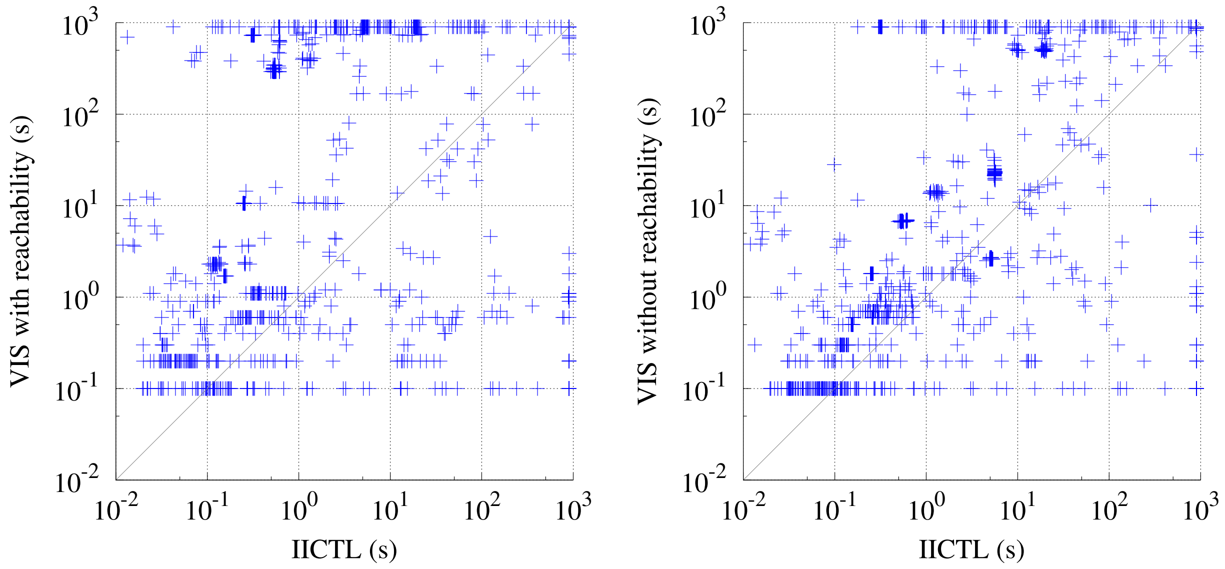


Figure 3.2: Comparing IICTL to competing techniques

Table 3.2: Timeouts and Unique Solves (nr = no reachability)

model	IICTL		VIS		VIS-nr	
	TO	US	TO	US	TO	US
abqueues4					1	
abqueues8	7	11	16		18	
abqueues8m	1		7			1
am2901		1	2		1	
am2910			4			
amba16			8			
amba32			130			
andersonSQ	3					
blackjack	1	13	13	1	50	
bufferAlloc	9					
CAB	6			6	11	
checkers	45	15	60		60	
fifteen	6	2	8		8	
gcd	2					
newnim	5					
palu			4			
redCAB	5				3	
retherRTF					1	
retherRTFfair					3	
rgraph	1					
simple16	2	14	15	1	17	
simple8	1					
soap	10		10		10	
swap	2					
twoQ	3		1			
vcordic	1		1		1	
viper			3			
vsa16a	1		6		1	
vsaR			11			
vsyst	1		1		1	
total	112	56	300	8	186	1

Chapter 4

Generalization in Incremental, Inductive Verification

Generalization is a crucial component of IIV. Without generalization, IIV algorithms reduce to state-enumerating strategies. IIV algorithms generalize facts they discover about the system to produce a more thorough refinement of the abstraction. Each type of property—hence every IIV algorithm—has its own generalization requirements. In some cases, an IIV algorithm may need to generalize a claim that has been proven to hold in the system, and in other cases, it may need to generalize a counterexample that explains why a claim does not hold. This chapter discusses different types of generalizations and ways in which they can be carried out.

Table 4.1 summarizes the generalizations required by the different IIV algorithms. For each generalization, the table lists the section of this chapter that describes a procedure for carrying out this specific generalization.

Section 4.1 describes procedures for generalizing proofs, and Section 4.2 describes procedures for generalizing counterexamples. Procedures described in this chapter are assumed to have access to the components of the system S that is being model checked, in particular its initial states I, its transition relation T, and its fairness constraints $\{B_1, \ldots, B_l\}$.

4.1 Generalizing Proofs

IIV algorithms prove many lemmas on their way to deciding the property. This section describes procedures for generalizing such lemmas. Section 4.1.1 describes a procedure for generalizing a relatively inductive clause. Section 4.1.2 explains how to generalize a state that cannot

IIV Algorithms	Fact Discovered	Type	Required Generalization	Section
IC3	A clause $\neg q$ is relatively inductive: $I \Rightarrow \neg q$, and $F_i \wedge \neg q \wedge T \Rightarrow \neg q'$	Proof	A subclause $\neg\hat{q} \subseteq \neg q$ that is relatively inductive: $I \Rightarrow \neg\hat{q}$, and $F_i \wedge \neg\hat{q} \wedge T \Rightarrow \neg\hat{q}'$	4.1.1
Fair	A state s is unreachable from the initial states: $I \not\models \mathsf{EF}\, s$	Proof	A set of states that are unreachable from the initial states: $I \not\models \mathsf{EF}\, G$ where $s \Rightarrow G$	4.1.3.1
Fair	A state s cannot reach another, t: $s \not\models \mathsf{EF}\, t$	Proof	A set of states that cannot reach t: $G \not\models \mathsf{EF}\, t$ where $s \Rightarrow G$	4.1.3.2, 4.1.3.3
IICTL	A state s cannot reach a target in one step: $s \not\models \mathsf{EX}\, \varphi$	Proof	A cube $\hat{s} \subseteq s$ of states that cannot reach the target in one step: $\hat{s} \not\models \mathsf{EX}\, \varphi$	4.1.2
IICTL	A state s cannot reach a target via a constrained path: $s \not\models \mathsf{E}\, \varphi\, \mathsf{U}\, \psi$	Proof	A set of states that cannot reach the target via a constrained path $G \not\models \mathsf{E}\, \varphi\, \mathsf{U}\, \psi$ where $s \Rightarrow G$	4.1.3.2
IICTL	A state s is not on a fair path: $s \not\models \mathsf{EG}\, \varphi$	Proof	A set of states that are not on a fair path $G \not\models \mathsf{EG}\, \varphi$ where $s \Rightarrow G$	4.1.4
IICTL	A state s can reach a target in one step: $s \models \mathsf{EX}\, \varphi$	Counterexample	A cube $\hat{s} \subseteq s$ of states that can reach the target in one step: $\hat{s} \models \mathsf{EX}\, \varphi$	4.2.1
IICTL	A state s can reach a target via a constrained path: $s \models \mathsf{E}\, \varphi\, \mathsf{U}\, \psi$	Counterexample	A set of states that can reach the target via a constrained path: $G \models \mathsf{E}\, \varphi\, \mathsf{U}\, \psi$ where $s \Rightarrow G$	4.2.2
IICTL	A state s is on a fair path: $s \models \mathsf{EG}\, \varphi$	Counterexample	A set of states are on a fair path: $G \models \mathsf{EG}\, \varphi$ where $s \Rightarrow G$	4.2.2

Table 4.1: Types of generalizations required by IIV algorithms.

reach a target in one step into a cube of states that all lack successors into the target. Section 4.1.3 describes procedures for manipulating a proof of unreachability through strengthening, weakening, or minimizing the proof CNF. Finally, Section 4.1.4 describes a procedure for generalizing a state that is not on a fair path to a set of states that are all not on fair paths.

4.1.1 A State's Negation is Relatively Inductive

Listing 4.1: IC3 generalization procedure.

```
1  Cube MIC($q$: Cube, $i$: Level):
2    $L$ := literals($q$)
3    foreach Literal $l$ in $L$:
4      $\hat{L}$ := down($L \setminus \{l\}$, $i$)
5      if $\hat{L} \neq \emptyset$:
6        $L$ := $\hat{L}$
7    return $\bigwedge_{l \in L} l$
8
9  LiteralSet down($L$: LiteralSet, $i$: Level):
10   while true:
11     $q$ := $\bigwedge_{l \in L} l$
12     if $I \not\Rightarrow \neg q$:
13       return $\emptyset$
14     if $F_i \wedge \neg q \wedge T \Rightarrow \neg q'$:
15       return $L$
16     with $(F_i \wedge \neg q)$–state $s$: {$s$ is the satisfying assignment from line 14}
17       $L_s$ := literals($s$)
18       $L$ := $L \cap L_s$
```

When IC3 proves a counterexample to induction (CTI) unreachable within $i+1$ steps (by proving that the CTI's negation is inductive relative to F_i), it tries to extend this conclusion to other states. It does so by expanding the CTI cube s (or equivalently finding a subclause of $\neg s$) using induction to guide the dropping of literals. IC3's generalization procedure (called MIC for minimal inductive clause [15]) is described in Listing 4.1.[1] Given a cube q that represents a state or a set of states that have been proven unreachable in $i+1$ steps, the procedure attempts to drop each literal in turn from q, calling down to validate each potential expansion of the cube (and, as

[1] The original MIC procedure described in [15] uses two procedures internally: up and down. However, the up procedure is not central to this thesis so we omit it for clarity.

a side effect, to further expand the cube). If down reports that the literal cannot be dropped, MIC returns it to q.

Given the literals of a cube q, the down procedure finds the largest subcube $\hat{q} \subseteq q$ that does not contain any initial states, and whose states are unreachable in one step from $(F_i \wedge \neg\hat{q})$-states (or equivalently, the largest subclause $\neg\hat{q} \subseteq \neg q$ that is inductive relative to F_i). Procedure down returns the literals of the subcube, if found, and $\emptyset$ if such a subcube does not exist. For a cube's states to be unreachable, all their predecessors must also be. Thus, starting with q, down checks if any q-state is reachable from an $(F_i \wedge \neg q)$-state. If there is one, its predecessor must also be proven unreachable; down, therefore, includes the predecessor in the cube and recurs. But, as a result of representing the set of states to be proven unreachable in the form of a single cube (which is important for efficiency), including a predecessor in the cube results in the inclusion of other states as a side effect. If at some point, an initial state gets included, the procedure declares failure.

Denoting the cube at iteration j by q_j (line 11), each fixpoint iteration queries the SAT solver for the existence of an $(F_i \wedge \neg q_j)$-predecessor to some q_j-state. The procedure concludes if no such predecessor exists. Otherwise, down must include the predecessor $(F_i \wedge \neg q_j)$-state s in the cube. Including s is done by taking the literals common to q_j and s (line 18), which is equivalent to finding the largest subcube whose equivalent set of states includes all q_j-states as well as s. The number of literals in the cube thus strictly decreases in every iteration, effectively expanding the cube q_j and consequently its set of states.

The following lemma states the correctness of the MIC procedure.

Lemma 9 ([15]). *Given a cube q and the index i of an overapproximate set F_i such that $I \Rightarrow \neg q$ and $F_i \wedge \neg q \wedge T \Rightarrow \neg q'$, MIC returns a cube $\hat{q} \subseteq q$ such that $I \Rightarrow \neg\hat{q}$ and $F_i \wedge \neg\hat{q} \wedge T \Rightarrow \neg\hat{q}'$*

4.1.2 A State Cannot Reach a Target in One Step

In IICTL, if an upper bound EX query is unsatisfiable—a task state s does not have φ-successors for some target assertion φ—it is desirable to generalize this to other states by finding a subcube $\hat{s} \subseteq s$ such that no $\hat{s}$-state has a φ-successor. As pointed out in Section 2.8, one can immediately obtain from the SAT query a subcube of s that satisfies this condition by passing s as a list of unit assumptions to the SAT solver, and requesting the subset of assumptions needed to make the query UNSAT. However, the subset of assumptions returned by the SAT solver is not guaranteed to be minimal. Therefore, one can obtain a better generalization by iterating over the remaining literals and attempting to drop each of them, querying the SAT solver every time if the query is still UNSAT. If it is, the subset of assumptions can be extracted. Otherwise, the literal is added back to the cube.

4.1.3 A State Cannot Reach a Target

Both Fair and IICTL invoke a proof-producing safety model checker through a function, reach(S, C, F, G), that accepts a finite-state system S, and propositional formulae for the constraint $C(\overline{x}, \overline{x}')$, the initial states $F(\overline{x})$, and the target $G(\overline{x})$. The function reach checks whether a G-state is reachable from an F-state via a path constrained by C. Fair calls reach for its stem and cycle queries, and IICTL calls reach for its upper and lower bound EU queries, for queries that determine whether a counterexample to generalization is reachable (see Section 4.2), and indirectly through calling Fair. A negative answer from reach is accompanied by a 1-step inductive strengthening of the property, that is, an assertion H such that

$$F \Rightarrow H$$

and

$$\neg G \wedge C \wedge H \wedge T \Rightarrow \neg G' \wedge H'.$$

The safety model checker used by IIV algorithms described in this thesis is IC3. IC3 implements reach(S, C, F, G) by setting its initial condition I to reach's F argument, its property P to $\neg G$, and its transition relation T to $T_S \wedge C$, where T_S is S's transition relation. The inductive strengthening H produced by IC3 is a CNF that is formed of the clauses in IC3's l-th overapproximation F_l, where l is the level at which IC3 converged. Depending on the algorithm and the particular query that is answered via reach, it might be desirable to **strengthen**, **weaken**, or **minimize** the CNF for F_l. Procedures for carrying out each of these generalizations, and the contexts in which they are needed are described next.

In what follows, let $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ be the set of clauses in F_l, i.e., $F_l = \bigwedge_{c_i \in \mathcal{C}} c_i$.

4.1.3.1 Proof Strengthening

For an unreachable stem query in Fair, the inductive strengthening returned by reach is an overapproximation of the states reachable from the initial states. Future skeletons that Fair examines must fall within this overapproximation. Thus, having an overapproximation that is as tight as possible can reduce the number of skeletons that Fair has to examine.

A CNF can be strengthened by either strengthening the individual clauses or by adding new clauses. The former is easier to carry out algorithmically. Listing 4.2 describes a procedure that strengthens each clause of the CNF by applying the MIC procedure to it. The level parameter of MIC is passed the value l so that induction is applied relative to F_l.

Listing 4.2: Strengthening a proof of unreachability.

```
ClauseSet strengthenProof($\mathcal{C}$: ClauseSet)
  do:
    $changed$ := false
    foreach Clause $c_i$ in $\mathcal{C}$:
      $oldSize$ := $|c_i|$
      $\neg c_i$ := MIC($\neg c_i$, $l$)
      if $|c_i| < oldSize$:
        $changed$ := true
  while $changed$
  return $\mathcal{C}$
```

Strengthening a clause c_i can enable strengthening another, c_j, even if a previous attempt of strengthening c_j failed. This is due to the strengthened induction hypothesis in the consecution query. Thus, the proof strengthening procedure iterates until a full pass on all clauses does not produce any further strengthening.

The following lemma establishes the correctness of strengthenProof.

Lemma 10. *Applying the procedure described in Listing 4.2 to an inductive strengthening F_l of a* *property P, produces an assertion $\hat{F}_l$ that is also an inductive strengthening of P, and is at least as strong as F_l.*

Proof. By Lemma 9, every call to MIC with a clause c is guaranteed to produce a relatively inductive clause $\hat{c}$. Because $\hat{c}$ is at least as strong as c, the relative inductiveness of all other clauses is maintained. □

4.1.3.2 Proof Weakening

An unreachable upper bound EU query in IICTL indicates that a state s does not satisfy a subformula, and thus can be removed from the subformula's upper bound. In such a case, the proof from IC3 represents states that all do not satisfy the subformula. By weakening such a proof, the set of states can be enlarged.

A CNF can be weakened by either dropping clauses or expanding them, i.e., adding more literals. The former can be done algorithmically using a procedure similar to MIC but applied to clauses instead of literals. The procedure is described in Listing 4.3.

Listing 4.3: Proof weakening procedure.

```
ClauseSet weakenProof($\mathcal{C}$: ClauseSet):
  foreach Clause $c$ in $\mathcal{C}$:
    $\hat{\mathcal{C}}$ := downClauses($\mathcal{C} \setminus \{c\}$)
    if $\hat{\mathcal{C}} \neq \emptyset$:
      $\mathcal{C}$ := $\hat{\mathcal{C}}$
  return $\mathcal{C}$

ClauseSet downClauses($\mathcal{C}$: ClauseSet):
  while true:
    if $P \wedge (\bigwedge_{c_i \in \mathcal{C}} c_i) \wedge T \not\Rightarrow P'$:
      return $\emptyset$
    if $P \wedge (\bigwedge_{c_i \in \mathcal{C}} c_i) \wedge T \Rightarrow P' \wedge (\bigwedge_{c_i \in \mathcal{C}} c_i')$:
      return $\mathcal{C}$
    with $(\neg(P' \wedge (\bigwedge_{c_i \in \mathcal{C}} c_i')))$-state $t$:
      foreach Clause $c$ in $\mathcal{C}$:
        if $t \models \neg c$:
          $\mathcal{C} := \mathcal{C} \setminus \{c\}$
```

The procedure iterates over the clauses of $\mathcal{C}$ and attempts to drop them one at a time. For each clause c, the procedure downClauses (line 8) is called with $\mathcal{C}\setminus\{c\}$ (line 3) to determine whether c is necessary. Given a set of clauses $\mathcal{C}$, downClauses returns a maximal subset of clauses $\hat{\mathcal{C}} \subseteq \mathcal{C}$ such that $\bigwedge_{c_i \in \hat{\mathcal{C}}} c_i$ is an inductive strengthening of the property, if it exists. First, downClauses checks if the property is inductive relative to the reduced set of clauses (line 10). If it is not, then the reduced set of clauses is no longer an inductive strengthening, and cannot become one through dropping more clauses; downClauses returns the empty set indicating that c is a necessary clause. Otherwise, downClauses checks whether the reduced set of clauses is inductive relative to the property[2] (line 12). If it is, the set is returned. Otherwise, the satisfying assignment of the next-state, t', is examined: t is a state that is not in the weakened proof, but is reachable from a state in the weakened proof. For the weakened proof to obey consecution, t must be included in the set. This is done by dropping every clause c that t violates; the loop in lines 15–17 drops such clauses.

The usage of activation literals is important for efficiency. For this procedure, an activation literal is created for each clause, which is used to activate or deactivate that clause. Doing so is useful for two purposes:

(1) It prevents having to modify the CNF in the SAT database between the different queries.

(2) By passing the activation literals as a set of unit assumptions to the SAT solver in the consecution query on line 12, if the query is unsatisfiable, literals that are not in the unsatisfiable core correspond to clauses in $\mathcal{C}$ that are not necessary for consecution to hold. Such clauses can be excluded from $\mathcal{C}$ before it is returned on line 13.

Lemma 11. *Applying the procedure described in Listing 4.3 to an inductive strengthening F_l of a property P, produces an assertion $\hat{F}_l$ that is also an inductive strengthening of P, and that is at least as weak as F_l.*

Proof. The set of clauses returned by the procedure obviously satisfies consecution since downClauses

[2] Note that while P was proven in line 10 to be inductive relative to $\bigwedge_{c_i \in \mathcal{C}} c_i$, including P' in the consequent is useful for the purposes of extracting an unsatisfiable core that represents a subset of $\mathcal{C}$ that obeys consecution.

only returns a subset of the clauses if they form an inductive strengthening of the property. It remains to prove that they also satisfy initiation. But since the procedure only drops clauses, and because if $I \Rightarrow \bigwedge_{c_i \in \mathcal{C}} c_i$, then for $\hat{\mathcal{C}} \subseteq \mathcal{C}$, $I \Rightarrow \bigwedge_{c_i \in \hat{\mathcal{C}}} c_i$, then the returned clauses also satisfy initiation. $\square$

4.1.3.3 Proof Minimization

The size of a CNF is determined by the number of clauses and literals it contains. The procedures described in Sections 4.1.3.1 and 4.1.3.2 drop literals and clauses, respectively, from a CNF and thus can be iterated to reduce its size. This is useful to apply for unreachable cycle queries in Fair, where minimizing the size of the proof can reduce the difficulty of all future queries. Iterating the two procedures until convergence results in a prime CNF, one in which no literal or clause can be dropped without losing inductiveness.

4.1.4 A State is not on a Fair Path

IICTL (see Chapter 3) invokes Fair to check whether $s \models \mathsf{EG}\,\varphi$ for an undecided state s. If Fair concludes that $s \not\models \mathsf{EG}\,\varphi$, it is desirable to generalize this result to states other than s, i.e., to find an assertion G such that

$$\forall \overline{x}\,.\,G(\overline{x}) \Rightarrow (\overline{x} \not\models \mathsf{EG}\,\varphi).$$

Fair concludes that $s \not\models \mathsf{EG}\,\varphi$ when it refines its set of potentially reachable states, R, to the point where the skeleton query becomes UNSAT indicating that R lacks fair SCCs. In this case, all R-states—not just s—are not on a fair path. Furthermore, states that lack fair SCCs can be generalized more by weakening R using a procedure similar to the one described in Section 4.1.3.2 but that guarantees that weakening does not result in including skeletons. The procedure is described in Listing 4.4.

Similar to the procedure described in Listing 4.3, this procedure tries to drop the clauses one by one. For each dropped clause, downFairClauses finds the maximal inductive subset of clauses that is skeleton-free and thus fair-SCC-free. Checking if the subset of clauses is skeleton-free (line

Listing 4.4: Procedure for generalizing a Fair proof.

```
1  ClauseSet weakenFairProof($\mathcal{C}$: ClauseSet):
2    foreach clause $c$ in $\mathcal{C}$:
3      $\hat{\mathcal{C}}$ = downFairClauses($\mathcal{C} \setminus \{c\}$)
4      if $\hat{\mathcal{C}} \neq \emptyset$:
5        $\mathcal{C}$ := $\hat{\mathcal{C}}$
6    return $\mathcal{C}$
7
8  ClauseSet downFairClauses($\mathcal{C}$: ClauseSet):
9    while true:
10     if $\left(\bigwedge_{i\in\{1,\ldots,l\}} B_i(\overline{x}^i) \wedge \bigwedge_{c_i\in\mathcal{C}} c_i \wedge \bigwedge_{W\in\mathcal{W}}\left[\left(\bigwedge_{i\in\{1,\ldots,l\}} W(\overline{x}^i)\right) \vee \left(\bigwedge_{i\in\{1,\ldots,l\}} \neg W(\overline{x}^i)\right)\right]\right.$:
11       return $\emptyset$
12     if $\left(\bigwedge_{c_i\in\mathcal{C}} c_i\right) \wedge T \Rightarrow \left(\bigwedge_{c_i\in\mathcal{C}} c_i'\right)$:
13       return $\mathcal{C}$
14     with $(\neg\left(\bigwedge_{c_i\in\mathcal{C}} c_i'\right))$-state $t$:
15       foreach clause $c$ in $\mathcal{C}$:
16         if $t \models \neg c$:
17           $\mathcal{C} := \mathcal{C} \setminus \{c\}$
```

10) uses the skeleton query described in Section 2.7. The information needed by the query is stored in the state of the Fair solver.

Lemma 12 states the correctness of this procedure.

Lemma 12. *Applying the procedure in Listing 4.4 to an assertion G for which $\forall\overline{x}\,.\,G(\overline{x}) \Rightarrow (\overline{x} \not\models \mathsf{EG}\,\varphi)$ produces an assertion H such that $\forall\overline{x}\,.\,H(\overline{x}) \Rightarrow (\overline{x} \not\models \mathsf{EG}\,\varphi)$, and that is at least as weak as G.*

Proof. Similar to the proof of Lemma 11, downFairClauses guarantees that the returned clause set is both inductive and skeleton-free. In addition, since the procedure only drops clauses, the CNF of the resulting clause set is at least as weak as the original. □

4.2 Generalizing Counterexample Traces

Unlike IC3 and Fair, generalizing counterexample traces is an essential component of IICTL. In particular, when IICTL discovers via a lower bound query that a task state satisfies a subformula, it is important to generalize this discovery to other states such that they can all be added to the corresponding node's lower bound. Chapter 3 describes generalizing traces from satisfiable EX, EU, and EG queries in terms of two procedures: $\mathsf{generalize}(t, G, D)$ and $\mathsf{generalize}(\bar{r}, C, G, D)$. This section describes the implementation of those two procedures.

We consider the problem of generalizing a trace, where in the case of an EX, the trace consists of two states. A first approach to generalize a given trace $s_0, i_0, s_1, i_1, \ldots, s_{n-1}, i_{n-1}, s_n$ with interleaved states and primary input values, is to use the unsatisfiable cores of the query $s_j \wedge i_j \wedge T \wedge \neg s'_{j+1}$ in which the literals of s_j are used as assumptions, to reduce s_j to a subcube. This query is applied starting with $j = n - 1$ down to 0 [53, 22]. One can also seek a **minimal** unsatisfiable core by dropping each literal if that does not make the query satisfiable. The (minimal) unsatisfiable core describes a set of states that include s_j, and that all have transitions on i_j to s_{j+1}. This restriction to predecessors with i_j-transitions limits the generalization power of this approach. For greater generalization power, **forall-exists generalization** is introduced in this section.

The overall idea of forall-exists trace generalization is to (1) select a cube c of the trace, (2) flip a literal of c to obtain $\bar{c}$, and (3) decide whether all $\bar{c}$-states satisfy the formula that the states of the trace satisfy. If they are, c can be replaced with the resolvent of c and $\bar{c}$, that is, the cube obtained by dropping the literal of step (2). This process continues until no further literal of the trace can be dropped.

Selecting the cube (step (1)) and one of its literals (step (2)) can be heuristically guided. The following describes step (3) of the procedure for the two $\mathsf{generalize}$ functions.

4.2.1 $\mathsf{generalize}(t, G, D)$

Given a state t, a target G, and a don't care assertion D, $\mathsf{generalize}(t, G, D)$ returns a cube $\hat{t} \subseteq t$ such that $\forall \overline{x} \,.\, \hat{t}(\overline{x}) \rightarrow (\overline{x} \models D) \vee (\overline{x} \models \mathsf{EX}\, G)$. Thus, if the candidate cube $\bar{c}$ obtained by flipping a literal of a subcube $c \subseteq t$ satisfies:

$$\forall \overline{x} \,.\, \bar{c}(\overline{x}) \rightarrow D(\overline{x}) \vee \exists \overline{i}, \overline{x}' \,.\, T(\overline{x}, \overline{i}, \overline{x}') \wedge G(\overline{x}') \ , \tag{4.1}$$

then $\bar{c}$ satisfies the postcondition of $\mathsf{generalize}(t, G, D)$.

The challenge with checking condition (4.1) is quantifier alternation. Rather than using a general QBF solver, forall-exists adopts a strategy in which two queries are executed iteratively. The first SAT query is the following:

$$\bar{c} \wedge \neg D \wedge T \wedge \neg G' \ . \tag{4.2}$$

It asks whether any "care" $\bar{c}$-state has a $\neg G$ successor. If the query is unsatisfiable, then every $\bar{c}$-state is either a D-state (a don't care) or has **all** its successors in G, i.e.,

$$\forall \overline{x} \,.\, \bar{c}(\overline{x}) \rightarrow D(\overline{x}) \vee \forall \overline{i} \,.\, \exists \overline{x}' \,.\, T(\overline{x}, \overline{i}, \overline{x}') \wedge G(\overline{x}') \ . \tag{4.3}$$

This is clearly stronger than (4.1), and therefore $\bar{c}$ also satisfies (4.1).

If, however, query (4.2) is satisfiable, then there exists a care $\bar{c}$-state s with at least one successor outside of G. If s only has $\neg G$-successors, then it represents a counterexample to (4.1). This is checked via a second SAT query:

$$s \wedge T \wedge G' \ . \tag{4.4}$$

If the query is satisfiable, then s does not prevent the generalization to $\bar{c}$. Query (4.2) can be repeated after adding the clause $\neg s$. However, before doing so, it is desirable to find states other than s that also do not prevent the generalization so that the procedure does not have to repeat queries (4.2) and (4.4) too many times. Finding states similar to s can be achieved by extracting from query (4.4) the satisfying assignment j for the primary inputs, and performing the following SAT query:

$$s \wedge \neg D \wedge j \wedge T \wedge \neg G' \ . \tag{4.5}$$

This query is unsatisfiable, since on j, s goes to a G-state[3] . The set of literals of s that do not appear in the unsatisfiable core can be dropped from s to obtain subcube $\hat{s}$. Each $\hat{s}$-state is either in D or goes to G under input j; $\hat{s}$-states therefore satisfy condition (4.1) and need not be reconsidered

[3] While query (4.5) is unsatisfiable even without the conjunction of $\neg D$, the presence of $\neg D$ allows generalizations of s that include D-states.

when checking (4.2). The procedure, thus, proceeds by repeating query (4.2) after adding to it the clause $\neg\hat{s}$.

If query (4.4) is unsatisfiable, s is considered a **counterexample to generalization** (CTG). It explains why $\bar{c}$ is not a valid generalization: a $\bar{c}$-state (s in this case) is neither a don't care state nor has a G-successor. However, all is not lost: the question remains whether s is even reachable. If it is not reachable, then it is a don't care state for the purposes of IICTL. Because generalization is unnecessary for correctness but necessary for (practical) completeness, answering this question requires balancing computational costs against the potential benefits of greater generalization. There are three reasonable approaches to addressing the question: (1) ignore it, obtaining immediate speed at the cost of generalization; (2) apply a semi-decision procedure for reachability, such as the MIC procedure of FSIS and IC3 [15, 14] to establish whether there is a subclause of $\neg s$ that is inductive; (3) apply a full reachability procedure such as IC3[4] .

With approach (3), if IC3 finds that s is reachable (hence $\bar{c}$ does not satisfy (4.1)), it is useful to cache this result by adding s to a set of states known to be reachable. Henceforth, whenever a cube $\bar{c}$ is considered as part of generalization, $s \models \bar{c}$ is first tested; if so, then query (4.4) is immediately applied. If this query is satisfiable because G has expanded, then s is no longer a CTG and can be removed from the list. This reuse of known reachable states during generalization significantly mitigates the cost of approach (3) on some benchmarks.

4.2.2 $\mathsf{generalize}(\bar{r}, C, G, D)$

$\mathsf{generalize}(\bar{r}, C, G, D)$ generalizes a given trace $\bar{r}$ such that every state in the generalized trace either satisfies the don't care assertion D, or is on a C-path to a G-state ($\forall 0 \le i \le n, \overline{x} \,.\, \hat{s}_i(\overline{x}) \rightarrow (\overline{x} \models D) \vee (\overline{x} \models \mathsf{E}\, C \,\mathsf{U}\, G)$). The condition:

$$\forall \overline{x} \,.\, \bar{c}(\overline{x}) \rightarrow D(\overline{x}) \vee G(\overline{x}) \vee C(\overline{x}) \wedge \exists \bar{i}, \overline{x}' \,.\, T(\overline{x}, \bar{i}, \overline{x}') \wedge G(\overline{x}') \tag{4.6}$$

is sufficient for the candidate cube $\bar{c}$ to satisfy the postcondition of $\mathsf{generalize}(\bar{r}, C, G, D)$: every $\bar{c}$-state is either a don't care, a G-state, or a C-state with a G-successor.

[4] Option 2 is the default in the implementation.

To address (4.6) without a QBF solver, several queries are executed iteratively. First, a necessary condition for the satisfaction of (4.6) is checked, which is that every $\bar{c}$-state has to be a D-state, a G-state, or a C-state. This is addressed with the following query:

$$\bar{c} \wedge \neg D \wedge \neg G \wedge \neg C \ . \tag{4.7}$$

If satisfiable, a care $\bar{c}$-state s is neither a G-state nor a C-state, and therefore cannot satisfy (4.6). s is therefore a CTG that can be analyzed for reachability. A reachable CTG ends consideration of $\bar{c}$. For every unreachable CTG, its negation is added as a clause to (4.7). Once query (4.7) becomes unsatisfiable, every (remaining) $\bar{c}$-state is a don't care state, a G-state, or a C-state. $\bar{c}$-states that are don't care states or G-states certainly satisfy (4.6); C-states, on the other hand, do not satisfy (4.6) unless they have G-successors. Therefore, focus turns to the existence of G-successors for all such $\bar{c}$-states:

$$\bar{c} \wedge \neg D \wedge \neg G \wedge T \wedge \neg G' \ . \tag{4.8}$$

If unsatisfiable, $\bar{c}$-states that are neither D-states nor G-states (and therefore by the unsatisfiability of (4.7) must be C-states) only have G-successors, which, again, is a stronger condition than (4.6). Therefore, $\bar{c}$ satisfies (4.6), and generalization is complete.

Otherwise, a witness state s exists; it is checked for G-successors:

$$s \wedge T \wedge G' \ . \tag{4.9}$$

If the query is satisfiable, then there exists s-successor state t and input j such that $t \models G$ and $(s, j, t') \models T$. In this case, the following query is unsatisfiable:

$$s \wedge \neg D \wedge \neg G \wedge j \wedge T \wedge (\neg C \vee \neg G') \ . \tag{4.10}$$

Its unsatisfiable core reveals a cube $\hat{s} \subseteq s$ that satisfies (4.6)[5] , and can therefore be added as a blocking clause to (4.8). Adding the blocking clause eliminates s as a counterexample to query (4.8), and therefore the query can be repeated. If query (4.9) is unsatisfiable, then s is a CTG to be handled as described for generalize(t, G, D).

[5] Note that while s is a C-state, the $\neg C$ term in the disjunction is necessary to guarantee that states in the unsatisfiable core that are neither D-states nor G-states not only have G-successors, but are also C-states.

Chapter 5

Better Generalization in IC3

One of the key components of IC3—as well as other IIV algorithms—is inductive generalization. While IC3 has an element of explicit state model checking in that it examines individual states called counterexamples to induction (CTIs), inductive generalization makes IC3 symbolic allowing it to handle huge state spaces. IC3's success on a model, thus, hinges on its ability to generalize facts that it discovers from considering specific states. The effectiveness of generalization depends on the connectivity of a model's state graph and its encoding. The overapproximate nature of inductive generalization causes it to fail frequently for some encodings, and for some models independent of their encoding. Unsuccessful generalization forces IC3 to examine more individual states.

Addressing counterexamples to generalization (CTGs) has proven to be a crucial aspect of effective generalizations in the context of IICTL, the IIV algorithm for CTL properties (see Chapter 4). In particular, addressing CTGs often allows generalizations to succeed that would have otherwise failed, helping alleviate the restriction caused by fixing the domain of generalization. Moreover, every successful generalization can have a ripple effect that allows other generalizations to succeed, and so on. This chapter examines what CTGs in IC3 represent, and proposes ways to address them. Addressing CTGs is shown to greatly enhance the performance of IC3 in two independent implementations of the algorithm.

Consider the state graph in Figure 5.1, where 000 is the initial state, 001 is the bad state. This model has two counterexamples to the inductiveness of the property: 110 and 100, two good

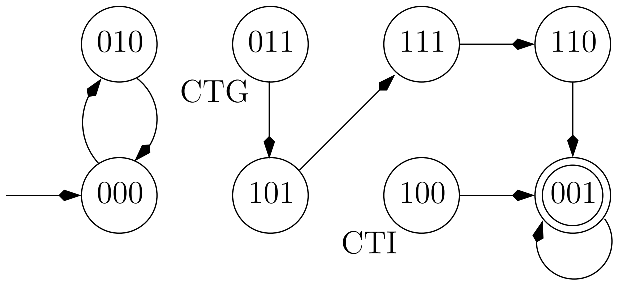


Figure 5.1: Failure to generalize a clause.

states with a bad successor. Suppose state 100 is the first CTI that IC3 finds. Since this state does not have predecessors, its negation is inductive, so that IC3 concludes it is unreachable. The unreachability of this state is a specific fact that IC3 next tries to generalize in order to prove that other states are unreachable as well. It does so by attempting to drop as many literals as possible which, in this case, is not possible for any of the literals of 100. For example, if IC3 attempts to drop the third literal, the negation of the resulting cube 10$-$, where $-$ indicates a don't care, is not inductive because of the transition from 011 to 101. If there is a cube whose negation is inductive and excludes both 100 and 101, that cube must also include 101's predecessor, 011. However, the smallest cube that includes all three states is $- - -$, which includes the initial state and whose negation is therefore not inductive. Similar reasoning shows that IC3 also cannot drop either the first or the second literal. Thus the strongest clause that can be derived through generalization only blocks the CTI itself. IC3 then has to prove that the other CTI (110) is unreachable without having learned much from the first CTI.

A state that hinders a generalization attempt (011 in the example) is a **counterexample to generalization** (CTG): it prevents dropping a literal (the third in the example), i.e., generalizing to a larger cube. Despite being itself unreachable, state 011 causes the inclusion of an initial state into the cube that covers both it and 10$-$, which in turn causes generalization to fail. In this case, it is useful to focus some effort on the CTG rather than only on the CTI. Since the negation of the CTG is inductive, IC3 can block it by adding the CTG's negation as a clause to the reachability sets. Then, with the CTI's predecessor blocked, generalization succeeds in dropping the third literal of 101. Indeed, the second literal can be dropped as well, as all predecessors of the cube $1 - -$ are blocked. This further expansion takes care of state 110 as well, ending the proof to the property.

This example motivates the improved generalization procedure described in this chapter. The proposed procedure addresses CTGs that appear during the generalization of some CTI-derived relatively inductive clause. CTGs are often deep backward reachable states, and addressing them reduces the depth of the explicit backward search IC3 performs and allows stronger inductive generalizations.

The proposed generalization procedure is evaluated within the implementations of IC3 in the model checkers IImc [38] and ABC [2]. Both show considerable improvement on Hardware Model Checking Competition (HWMCC) benchmarks [37].

Several improvements orthogonal to the generalization method presented here have been described for IC3. Ternary simulation [31] and SAT-based [22] methods of enlarging CTI cubes significantly improve running time. A scheme for integrating lazy abstraction with IC3 has also been developed [58].

Section 5.1 describes the proposed generalization procedure. Section 5.2 presents the results of IC3 with improved generalization on the HWMCC 2010–2012 benchmark suites. To confirm the intended purpose of addressing CTGs—reducing IC3's explicit backward search, Section 5.3 analyzes the behavior of IC3 with the improved procedure and compares it to its behavior with the standard generalization procedure.

5.1 Addressing Counterexamples-to-Generalization

5.1.1 Presentation of the Procedure

In IC3's down procedure (see Section 4.1.1), if induction does not hold with an expanded cube q_j due to the existence of an $(F_i \wedge \neg q_j)$-predecessor s to a q_j-state, the procedure proceeds by dropping from q_j all literals not in s. Keeping only the common literals of q_j and s provides an overapproximating union over state sets—a **join** in the cube lattice. While this operation responds to the need to include the q_j-predecessor s in the state set described by q_{j+1}, it also typically brings in other F_i-states. Therefore, even when all q_0-states are unreachable, down (eventually) fails if, through overapproximation, it incorporates a reachable state.

If down fails, state s is a possible cause of its failure. State s is a **counterexample to generalization** (CTG) since it is encountered in the context of dropping a literal (in MIC) in order to generalize a cube. Unlike CTIs, states brought in as a result of dropping a literal or joining cubes are not necessarily backward reachable from the error. On the one hand, if s is backward reachable—and it represents a set of deep backward reachable states—then addressing it could save IC3 from having to explicitly traverse the state graph from the error state to s. On the other hand, if s is neither backward nor forward reachable, it could still obstruct generalization: when it is joined with q_j to form q_{j+1}, it could cause the inclusion of a reachable state. Nevertheless, IC3 would never attempt directly to block s since it only generalizes from backward reachable states. Yet blocking s, rather than joining with it, could enable generalization to succeed, thereby helping IC3 produce stronger clauses and potentially find a proof faster.

The arguments presented motivate the potential benefits of addressing CTGs. A generalization procedure that addresses CTGs, ctgDown, is presented in Listing 5.1. Similarly to down (Section 4.1.1), ctgDown first checks whether $\neg q$ is inductive (lines 16–19). However, if it is not inductive, ctgDown does not immediately join q with the discovered predecessor s. Rather, it attempts to block s at level i by proving $\neg s$ inductive relative to F_{i-1} (line 23). If this attempt succeeds, ctgDown tries to block s at higher levels (lines 25–27), and then strengthens the clause at the highest level relative to which it was found to be inductive by applying MIC (line 28). Having addressed one cause for the non-inductiveness of $\neg q$, ctgDown returns its attention to q.

To maintain its focus on the main goal of strengthening $\neg q$, ctgDown considers at most maxCTGs CTGs between joins (line 23). If the limit is exceeded or a CTG is not found to be inductive, the CTG is joined with q (line 33). New states brought in as a result of the join present an opportunity to explore behaviors farther from the error, so ctgDown re-enables considering CTGs by resetting the number of allowable CTGs to maxCTGs (line 31) to allow examining potentially

Listing 5.1: Proposed generalization procedure.

```
Cube MIC($q$: Cube, $i$: Level):
  return MIC($q$, $i$, 1)

Cube MIC($q$: Cube, $i$: Level, $d$: RecDepth):
  $L$ := literals($q$)
  foreach Literal $l$ in $L$:
    $\hat{L}$ := ctgDown($L \setminus \{l\}$, $i$, $d$)
    if $\hat{L} \neq \emptyset$:
      $L$ := $\hat{L}$
  return $\bigwedge_{l \in L} l$

LiteralSet ctgDown($L$: LiteralSet, $i$: Level, $d$: RecDepth):
  $ctgs$ := 0
  while true:
    $q$ := $\bigwedge_{l \in L} l$
    if $I \not\Rightarrow \neg q$:
      return $\emptyset$
    if $F_i \wedge \neg q \wedge T \Rightarrow \neg q'$:
      return $L$
    with $(F_i \wedge \neg q)$-state $s$:
      if $d >$ maxDepth:
        return $\emptyset$
      if $ctgs <$ maxCTGs and $i > 0$ and $(I \Rightarrow \neg s)$ and $(F_{i-1} \wedge \neg s \wedge T \Rightarrow \neg s')$:
        $ctgs$ := $ctgs + 1$
        for $j$ := $i$ to $k$:
          if $F_j \wedge \neg s \wedge T \not\Rightarrow \neg s'$:
            break
        $s$ := MIC($s$, $j-1$, $d+1$)
        $clauses(F_j)$ := $clauses(F_j) \cup \neg s$
      else:
        $ctgs$ := 0
        $L_s$ := literals($s$)
        $L$ := $L \cap L_s$
```

deeper CTGs.

Since ctgDown calls MIC, the version of MIC associated with ctgDown monitors the recursion depth through its d parameter. The recursion depth is initialized by the wrapper function to 1 (line 2) and updated by the call to MIC from ctgDown (line 28). At a recursion depth of 1, ctgDown examines CTGs that are encountered during generalization of a CTI-induced clause. For larger depths, an encountered CTG is one that interferes with the generalization of a CTG-induced clause. A parameter, maxDepth, limits the effort spent on addressing CTGs of CTG-induced clauses. When this limit is exceeded (line 21), CTGs are not examined, and joins are disabled; as a result, ctgDown fails immediately if $\neg q$ is not inductive (line 22).

5.1.2 Discussion

The limit on handling CTGs, which gets reset after every join, results in an interesting pattern of state exploration. IC3 itself explores the state space in an explicit manner backward from the error through its priority queue. A state s in the priority queue can reach the error in a relatively few number of steps. If IC3 is forced to consider a predecessor of s, then it is known that the predecessor can reach the error too. In contrast, when MIC is applied to s, the first step is to drop a literal, enlarging the represented state set. In ctgDown, up to maxCTGs times, predecessors of the enlarged cube are then considered as CTGs. They are likely to be backward reachable; they are also likely to be about as close to an error as s is[1] .

Eventually maxCTGs is exhausted, forcing a join. Predecessors to the enlarged cube are then considered as CTGs. These predecessors are less likely to be backward reachable but more likely to be "farther" from an error than s. Deep backward reachable states may be particularly valuable to prove unreachable from the initial states early because this may facilitate proving other states unreachable. This cycle can continue for several iterations, each iteration exploring states that are increasingly far from the error but at the cost of being increasingly likely not to be able to reach

[1] While there are models for which this assumption is invalid, the fraction of state bits of a large digital system that changes at each clock cycle is often less than one tenth. This fraction supports the view that similarity between states decreases with their distance in the state graph.

the error. Further iterations of dropping literals by MIC add layers of likelihoods of depth and backward reachability to the state exploration.

Another behavior worth noting is that unlike down, the effort spent on a ctgDown call that in the end fails, is not always wasted. When down fails, the only gained information is that the dropped literal is actually required. In contrast, ctgDown may successfully handle some CTGs on the way to failing to prove the inductiveness of the given cube. These CTG-derived lemmas could well prove useful in addressing the overall model checking problem.

In early attempts at considering CTGs, we investigated a scheme that delayed the handling of CTGs as much as possible. Rather than prioritizing the direct handling of CTGs over joining with them, it aggressively joined and only handled CTGs upon failure. If $\neg q_j$ failed initiation, the last-encountered CTG s was addressed directly. Successful elimination of s would enable the reconsideration of q_{j-1}; failure would cause the CTG leading from q_{j-2} to q_{j-1} to be addressed instead, and so on. This version was inferior to ctgDown, possibly because too much effort was put into addressing states that were either not actually backward reachable or too removed from the original CTI to be relevant to the generalization effort. While ctgDown explores CTGs in an outwardly expanding set from the error, the unsuccessful variant explored CTGs in an inwardly contracting set.

While these explanations assume characteristics of a state space that need not hold for a given model, they offer an intuitive motivation for using ctgDown instead of down: with some trade-offs, it jumps to deep states, complementing IC3's conservative top-level behavior. Section 5.3 compares, empirically, the behavior of IC3 with down versus with ctgDown.

5.2 Results

In this section, IC3 with ctgDown is compared empirically to existing standard implementations of IC3. The new procedure was implemented within the IC3 engines in IImc v1.3 [38] and in ABC `vbb0deac` (Apr 3, 2013) [2]. The following parameter values were used in the experiments for both implementations of ctgDown: $\mathsf{maxDepth} = 1$ and $\mathsf{maxCTGs} = 3$[2] . The implementations of ctgDown differ from the pseudocode of Listing 5.1 in the following respects:

- In the IImc implementation, the consecution call in line 26 was implemented as a call to down. This change enables blocking a CTG at a (higher) level at which its negation is not inductive but contains an inductive subclause. The experiments are inconclusive with regards to which version is better.
- In the ABC implementation, the CTG cube is expanded through ternary simulation before it is checked for inductiveness (line 20) [31].

Note that, unlike IImc's standard implementation of IC3, ABC's standard implementation does not employ down in its generalization procedure; in particular, it never joins. Thus, experiments with IImc compare the effects of ctgDown against down, while experiments with ABC compare the effects of ctgDown against ABC's generalization procedure. While a variant of ctgDown in which joins are disabled can be implemented, experiments with ABC (whose details are not reported here) show that this variant of ctgDown is inferior to full ctgDown.

The benchmark suite was gathered from the HWMCC 2010–2012 benchmarks [37]—with one exception. Backward-encoded BEEM models (distinguished by the names `beem*`i`b`j) were replaced with their corresponding "functional" versions (also available from [37]). The backward encoding of these models involves two features[3] :

(1) Serial exists-step transition relation [30]: this feature adds "shortcut" transitions to the state graph.

(2) Reverse relational encoding: the transition relation is inverted, and the initial states are swapped with the bad states. The latch updates are directly from primary inputs, and a `valid` bit is added to track whether a state is backward reachable in the original design.

[2] Generally, small values for maxCTGs (2–5) gave the best performance. For higher values, IC3 tended to derive too many clauses and lose property focus.

[3] See `http://fmv.jku.at/aiger/README.beemaigs` for details.

IImc has a "reverse" option to invert the transition relation and exchange the initial and error states. With this option, IImc typically works better on reverse-encoded models and worse on forward-encoded ones. A possible explanation is that a clause is a natural logical means of describing a design intention; moreover, conjunctions of clauses capture local arguments. In contrast, disjunctions of cubes—which is what IC3 produces from the forward perspective on reverse-encoded designs—are less capable of capturing local arguments. For this reason, with both the functional and the backward encodings of these models available, one would never choose to use the backward encoding with IC3. Conclusions drawn from data based on such benchmarks are misguided.

As a preprocessing step, IImc's `sr` simplification tactic was applied to each benchmark. Then, IImc and ABC with and without the new generalization procedure were run on the simplified benchmarks only invoking their IC3 engines. No other features of IImc or ABC—e.g., multi-threading, other proof engines, or more powerful simplification techniques—were used. Each benchmark was run for up to 900 seconds. To account for variability, each benchmark was run five times with different random seeds. The experiments were performed on two identical machines with four 2.80 GHz Intel cores and 9 GBs of memory. The full results can be found at `http://vlsi.colorado.edu/fmcad13`.

A comparison between the performance of IC3 with and without ctgDown is presented in Figures 5.2–5.4. Figure 5.2 shows cactus plots for IImc and ABC and Figures 5.3 and 5.4 show scatter plots. All the plots use the results of the median runs. For the easier models, the use of ctgDown does not typically reduce CPU time (Figures 5.3 and 5.4).

Detailed results by benchmark family are presented in Table 5.1. Benchmark families with at least 60 benchmarks are listed separately. The remaining benchmarks are included in the "other" category. The "Solved" columns show the minimum, median, and maximum number of solved instances over the five runs. The "Time" columns reports the median CPU time in seconds. Overall, IImc and ABC with ctgDown solve 17 and 27 more instances, respectively, than their

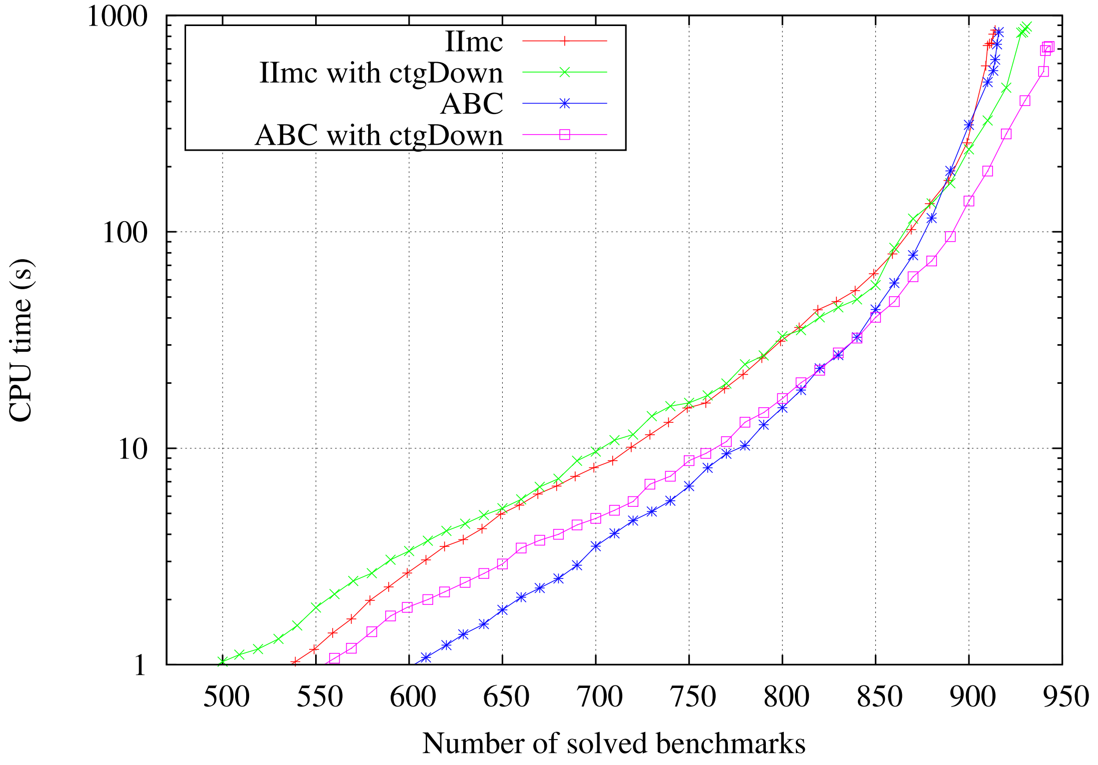


Figure 5.2: Cactus plot comparing the performance of IImc and ABC with and without `ctgDown`.

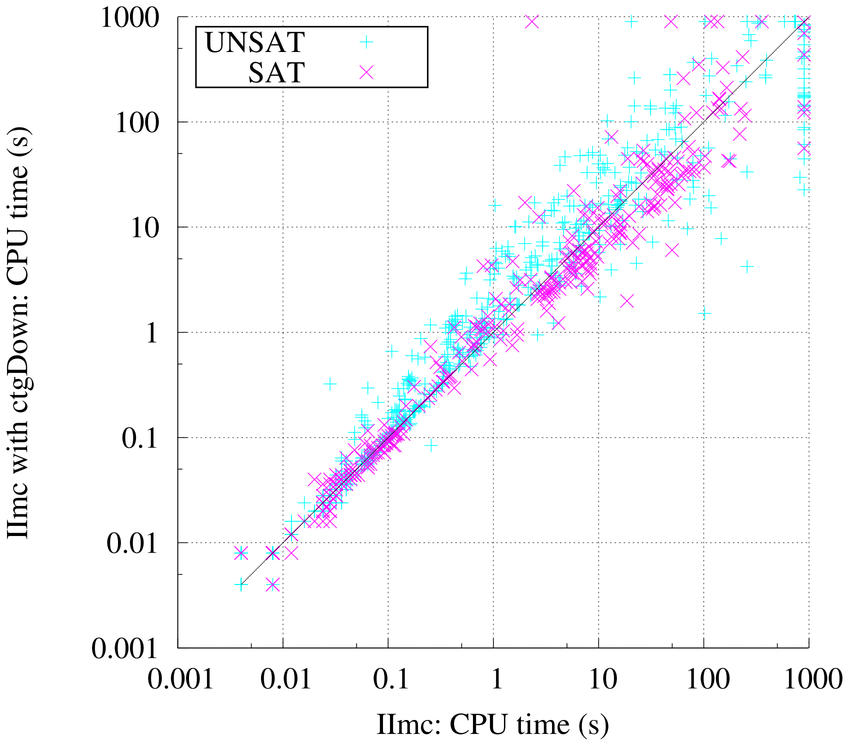


Figure 5.3: IImc scatter plot

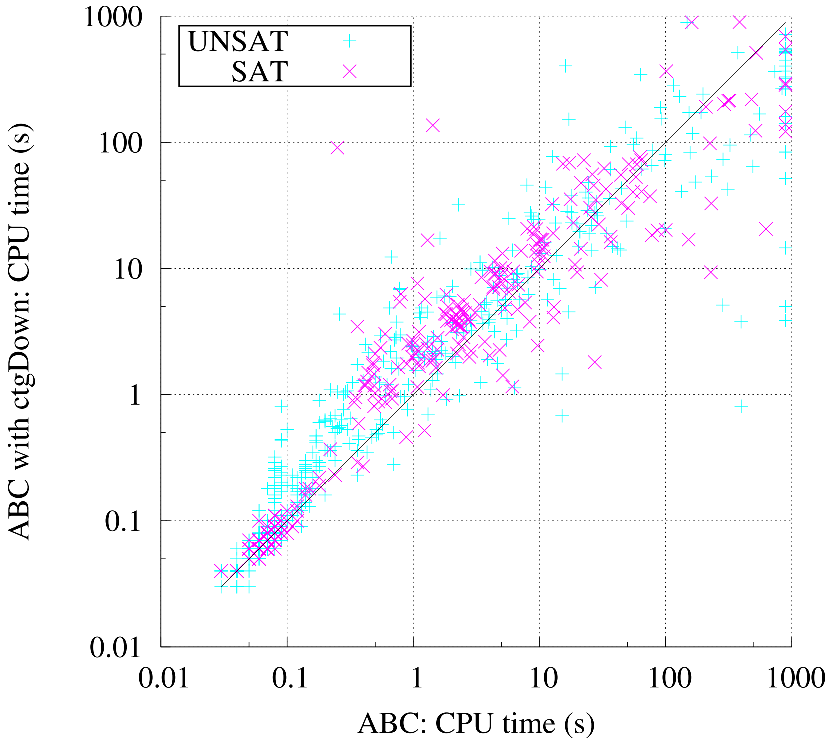


Figure 5.4: ABC scatter plot

standard counterparts[4] . The same trend was observed when the timeout was increased to one hour: IImc and ABC solved 17 and 24 more instances respectively.

5.3 Analysis of IC3's Behavior

An observed weakness in IC3 with down is that, on some models, it handles long chains of states explicitly rather than symbolically. ctgDown is intended to address this weakness by addressing CTGs to accelerate IC3's exploration of deep backward reachable states while still maintaining its characteristic focus on the property. As discussed, CTGs interfere with generalizing from CTIs and so are worthwhile candidates for blocking with generalization, even when they are not backward reachable. Through measuring several metrics, this section presents an analysis that suggests that ctgDown achieves its intended behavior. It highlights differences in the behavior of IC3 with the standard (down) and improved (ctgDown) generalization procedures. The data in this section were collected from IImc's IC3 runs. Data collected from ABC's runs also support the observations made.

Data points for scatter plots in this section are divided into two categories: those for which IC3 performs better with ctgDown, marked by a $\times$ in the plots, and those for which IC3 performs better with down, marked by a $+$.

The first experiment compares the average distances of CTGs and CTIs from an error. To measure the depths of CTGs, exact BDD-based backward reachability is performed; the resulting "onion rings" can be used to compute the depth of a given state. Of the CTGs handled in these experiments, 42% were backward reachable. For CTIs, their depths are considered to be the length of the chains through which the CTIs were found; this length provides an upper bound on a CTI's actual backward depth. Figure 5.5 shows a plot for the average CTI depth against the average CTG depth for the 294 benchmarks for which the preliminary BDD-analysis managed to complete within 12 hours. The plot confirms that CTGs are typically deeper than CTIs—sometimes by

[4] Since the median is used, the sum of the gains for the individual families is not necessarily equal to the overall gain.

Table 5.1: Detailed results by benchmark family.

		IImc					**ABC**				
		Standard		With ctgDown			Standard		With ctgDown		
Family	Size	Solved	Time (s)	Solved	Gain	Time (s)	Solved	Time (s)	Solved	Gain	Time (s)
139	99	98/99/99	2524	99/99/99	0	1230	99/99/99	701	99/99/99	0	754
6s	120	18/19/22	93466	19/21/22	2	94211	19/23/24	88401	27/30/31	7	82941
beem	86	46/48/49	38149	47/50/51	2	39594	50/51/53	34098	54/56/57	5	31191
bob	149	121/122/125	25804	120/120/122	*(2)*	28679	122/123/124	24292	122/124/127	1	24083
intel	60	22/23/23	35004	29/30/31	7	31153	23/23/23	35665	25/26/27	3	34249
pdt	350	330/331/332	19291	336/336/337	5	15469	327/329/329	22162	333/333/333	4	18120
other	280	270/271/272	11947	272/274/275	3	11463	269/270/271	12591	272/274/274	4	10359
Total	1144	910/913/917	226790	924/930/932	**17**	222460	914/916/919	218906	936/943/944	**27**	201417

several orders of magnitude. The plot also indicates that ctgDown helps in the cases where IC3 is forced to explore deep CTIs.

Next, several metrics of IC3 runs were analyzed to understand when the proposed generalization procedure helps or harms the performance of IC3. The metrics are the maximum length of traces from states in the priority queue to an error; the average size of derived clauses; the convergence level, i.e., the level at which a proof or a counterexample is found; and the average number of clauses derived per level.

Plots comparing IC3 with and without the proposed generalization procedure on the four metrics are shown in Figures 5.6a–5.5d. The same information is presented with box-and-whisker plots in Figure 5.4e. The plots report the ratio of each metric with ctgDown to without.

Figure 5.6a indicates a significant reduction in the depth of the explicit search performed by IC3 when ctgDown is used. Statistics indicate an average reduction of 22.3% in the depth of IC3's explicit search over all benchmarks. A higher reduction in the depth of the search often indicates better performance for IC3 as confirmed by the non-overlapping notches in the box plot, which indicate a significant difference in the median depth ratios between cases with better and those with worse performance.

The point in the lower right corner of Figure 5.6a (benchmark eijks420) represents an extreme case in which IC3 with ctgDown proved the property with very little explicit backward search; with down, the depth of the priority queue went up to 2049.

Figure 5.6b points out ctgDown's ability to produce stronger CTI-induced clauses. Again, a stronger clause indicates improved performance. On average, ctgDown drops 14% more literals than down, which is statistically significant as indicated by the box plot.

A characteristic of the new procedure is that it often increases the convergence level of IC3, as indicated in Figure 5.5c. This potentially undesirable side effect is probably attributable to the aggressiveness of ctgDown in deriving clauses to block CTGs—which, again, need not actually be backward reachable. In contrast, the standard procedure only derives clauses in response to truly backward reachable states. A clause that blocks a forward reachable state is certainly not

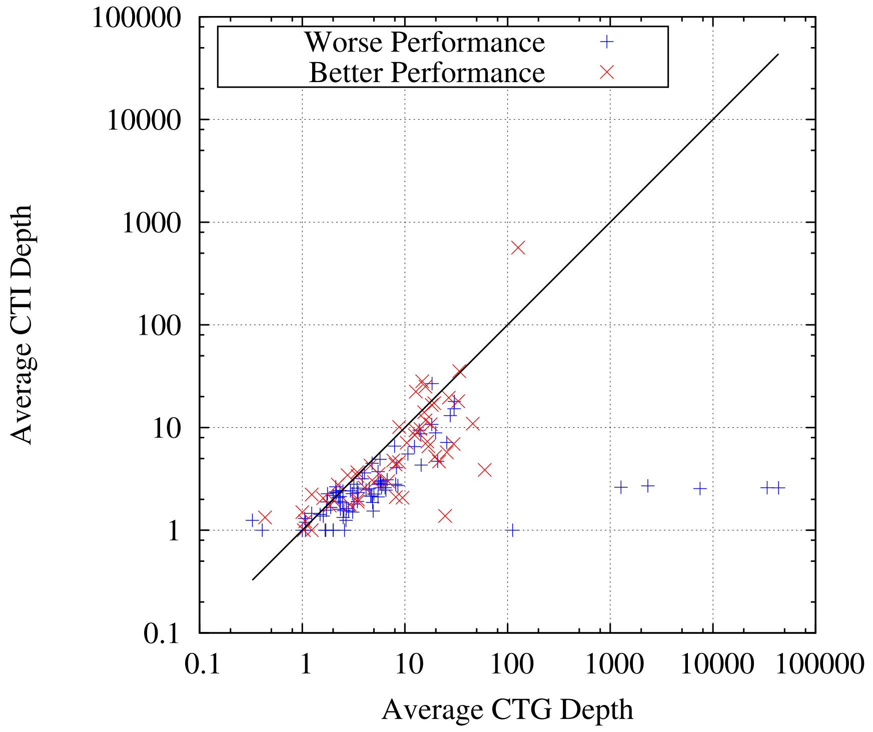
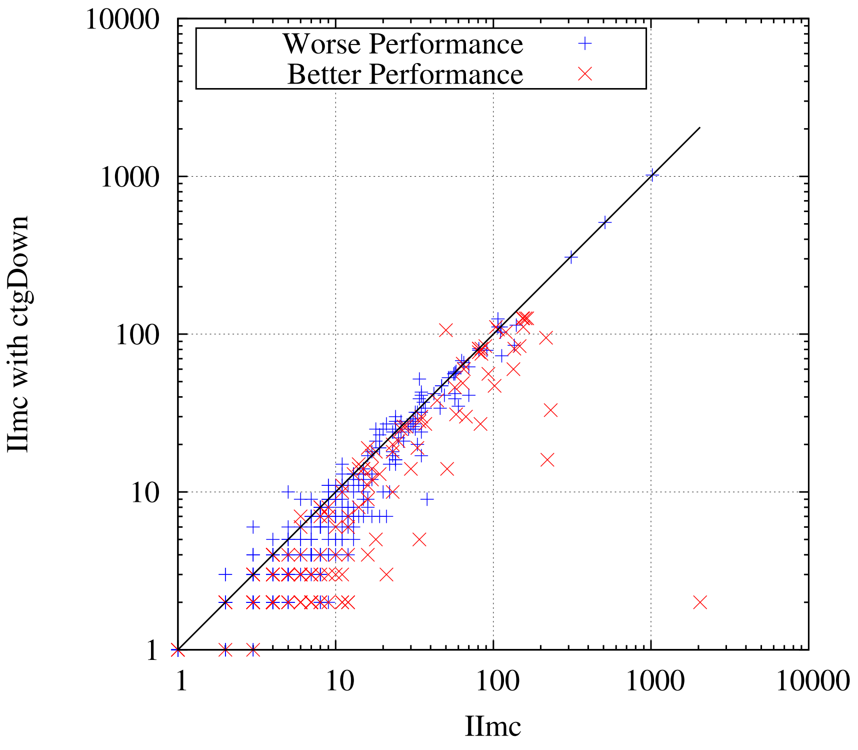
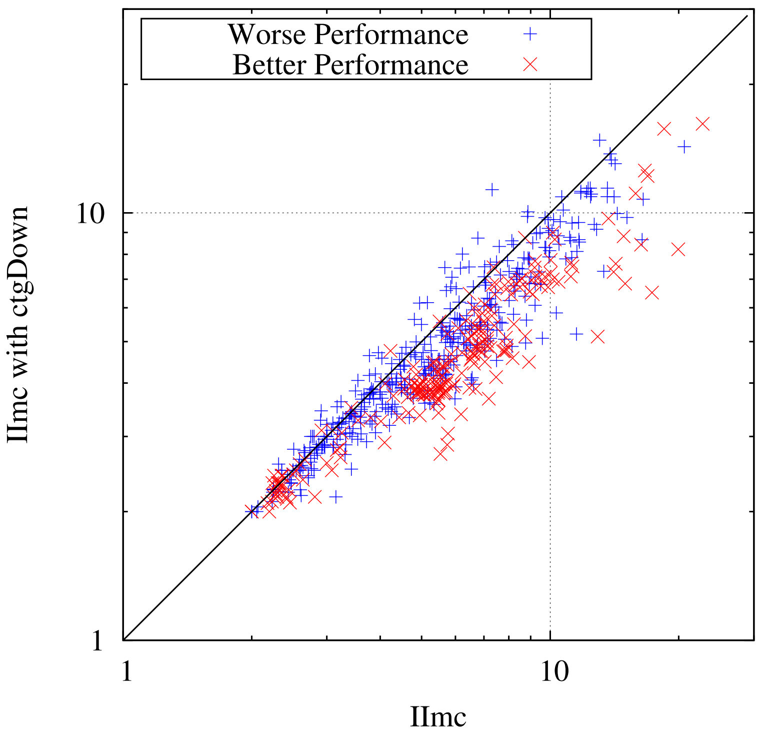


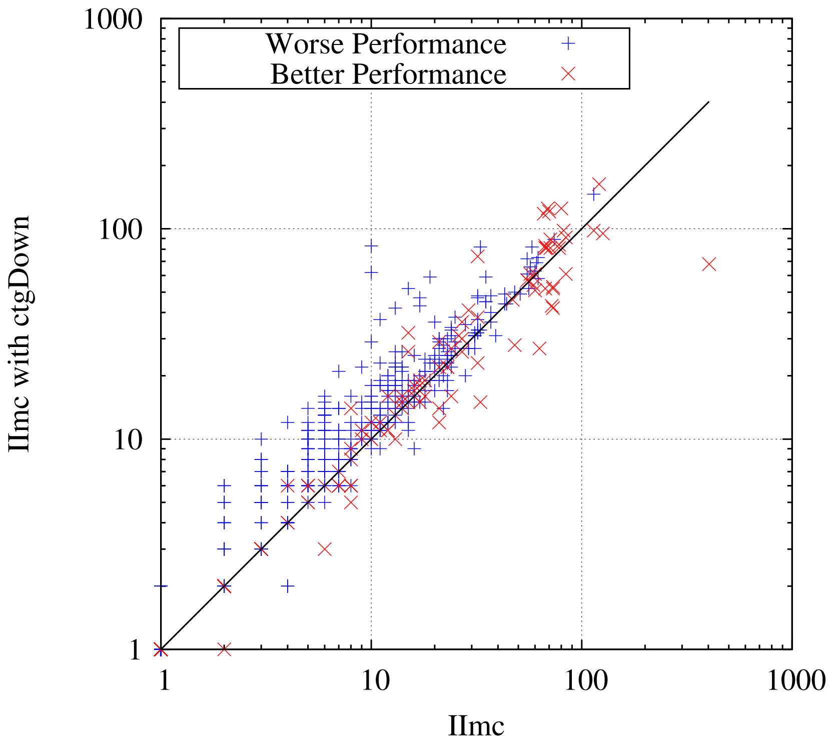
Figure 5.5: A comparison between the depths of CTIs and CTGs.



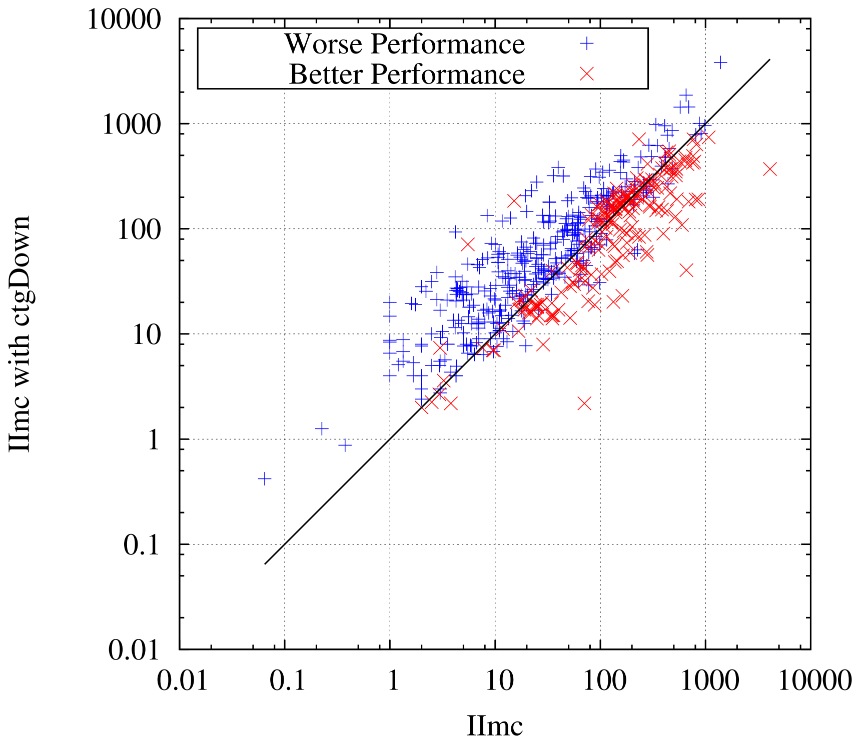
(a) Maximum depth of priority queue.



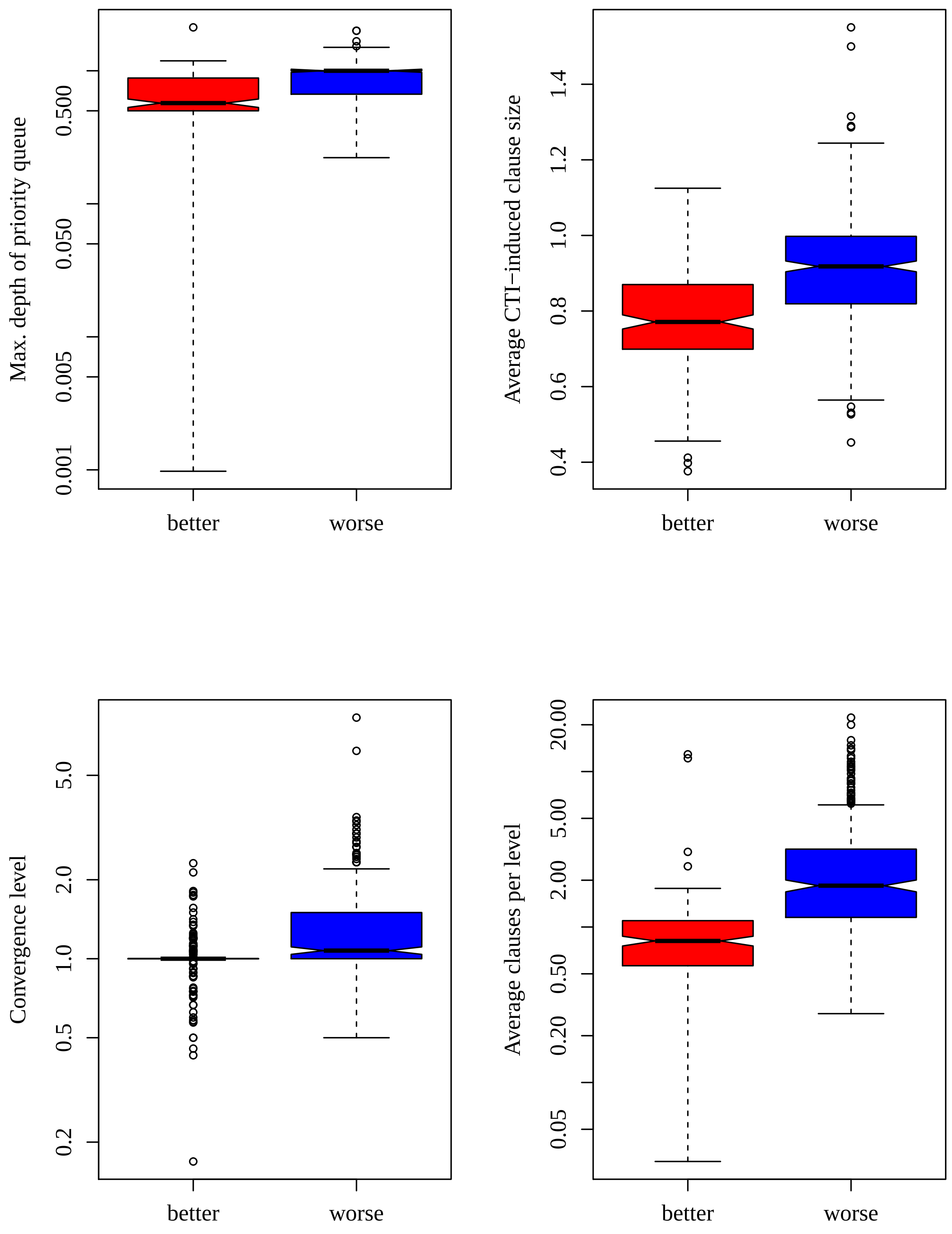
(b) Average CTI-induced clause size.



(c) Convergence level.



(d) Average clauses per level.



(e) Box plots for the ratios of the metrics shown in parts a–d.

Figure 5.4: Analyzing the effects of `ctgDown` on the IImc runs.

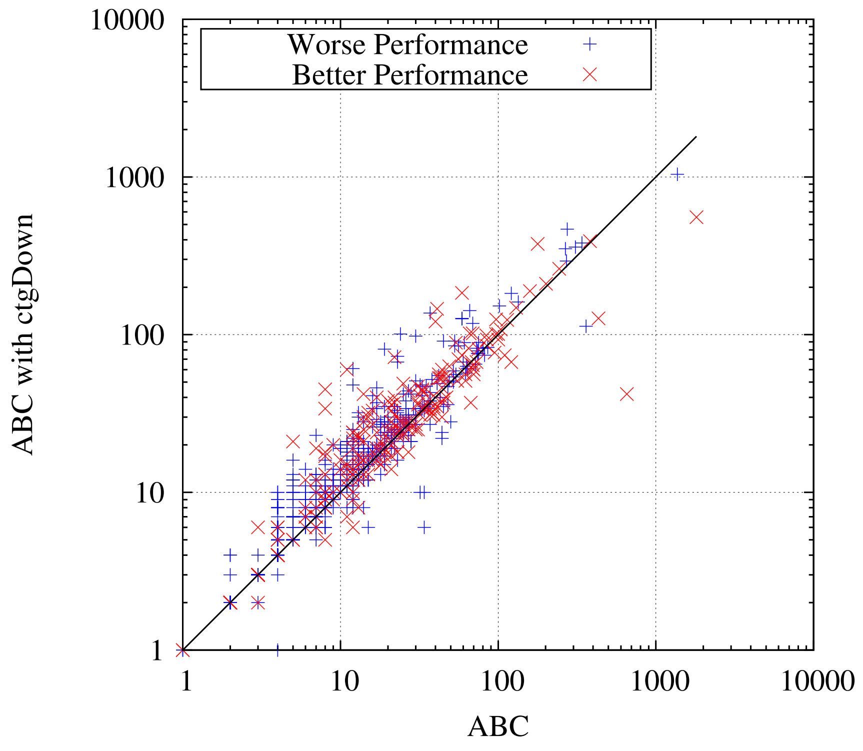


Figure 5.5: Convergence level.

inductive and thus cannot appear in the final inductive strengthening. Such clauses can cause overstrengthening of the F_i's forcing IC3 to propagate to higher levels in order to drop the clauses. Points to the far right in Figure 5.5 represent cases in which such behavior is exhibited: although CTGs are much deeper than CTIs, the percentage of handled CTGs that are forward reachable is higher than average causing overstrengthening. Also, as Figure 5.5c shows, a higher convergence level is significantly correlated with worse performance. Similar observations hold for ABC with ctgDown as Figure 5.5 indicates. The box plot in Figure 5.4e shows that 75% of the runs in which ctgDown was beneficial did not increase the convergence level. In contrast, for 75% of the runs that did not benefit from ctgDown, the convergence level was higher. On the other hand, statistics indicate that the increase in convergence level only occurs for passing properties; for 75% of the failing properties, the convergence level is not affected.

Points on the y-axis in Figure 5.5c correspond to benchmarks for which IC3 with down converges at level 1 while IC3 with ctgDown converges at higher levels. A characteristic behavior of IC3 with down is that clauses generated at level 1 are globally inductive until IC3 is forced to step back to level 0. Subsequently, all derived clauses have the support of clauses that were generated relative to F_0, and thus need not be globally inductive. Aggressive handling of CTGs interferes with this initial behavior. To overcome this behavior, a variant implementation was tried in which CTG handling was disabled until IC3 was forced to step back to level 0. IC3 with this variant ctgDown then converged at level 1 on these benchmarks; however, the performance difference across the benchmark suite was insignificant.

Finally, Figure 5.5d and the corresponding box plot indicate a clear correlation between the performance difference and the average number of clauses derived per level. An excessive number of clauses derived to block CTGs is often accompanied by longer runtimes.

Chapter 6

Strategies of Incremental, Inductive Verification Algorithms

The incremental, inductive CTL algorithm developed in this thesis (IICTL) expands the portfolio of incremental, inductive verification (IIV) algorithms. Despite being IIV-based, IICTL's strategy differs considerably from IC3 and Fair. This chapter sheds some light on the strategies employed by IIV algorithms in an attempt to develop a deeper understanding of the IIV approach.

Both IC3 and Fair can be viewed as **goal-driven**: they start from the goal—a bad state in the case of IC3, and a skeleton of a fair cycle in the case of Fair—and proceed by checking whether the goal is achievable given facts known about the system—for IC3, whether the bad state is reachable, and for Fair, whether the skeleton can be connected to a reachable fair cycle. If they determine that the goal is not achievable, they proceed to a different one after deriving as much information as possible to guide the future selection of goals. In contrast, IICTL uses a **data-driven** strategy: starting from facts known about the system—its initial states—and guided by the structure of the CTL parse graph, IICTL checks whether the system can evolve in a manner that constitutes a violation of the CTL property.

This difference in strategies between IC3 and Fair vs. IICTL becomes more apparent on properties that are in the scope of both approaches. In what follows, we give two examples of such properties and describe the behavior of the two approaches on them.

Example 1. Consider the LTL property $\varphi = \mathsf{G}(p \rightarrow \mathsf{G}\, q)$, which is equivalent to the CTL property $\mathsf{AG}(p \rightarrow \mathsf{AG}\, q)$. (See Appendix A for a proof of their equivalence.) The LTL property is a safety one and thus can be model checked using IC3. Applying IC3 requires constructing a Büchi automaton

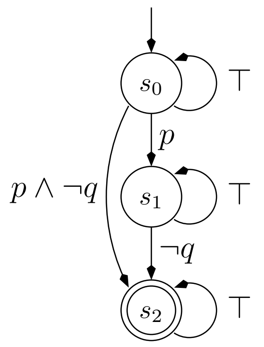


Figure 6.1: Büchi automaton for $\mathsf{F}(p \wedge \mathsf{F}\, \neg q)$

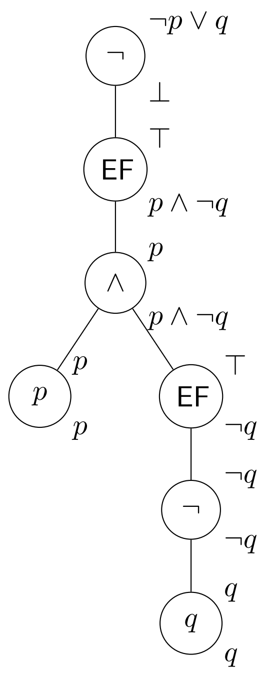


Figure 6.2: CTL parse graph for $\mathsf{AG}(p \rightarrow \mathsf{AG}\, q)$

that accepts the negation of the LTL formula and composing the automaton with the system. The automaton that accepts $\neg\varphi = \mathsf{F}(p \wedge \mathsf{F}\, \neg q)$ is shown in Figure 6.1. The automaton has one accepting state: s_2. IC3's task is to check whether an s_2-state is reachable in the composed system,[1] which it does by examining CTIs and attempting to prove them unreachable within k steps or extending a counterexample trace from them back to some initial state. In this case, CTIs are either $(s_1 \wedge \neg q)$-states or $(s_0 \wedge p \wedge \neg q)$-states.

Since the property is expressible in CTL, it can also be model checked using IICTL. Unlike IC3, IICTL operates on the system directly but uses the parse graph of the CTL formula to infer the queries to perform. The parse graph of $\mathsf{AG}(p \rightarrow \mathsf{AG}\, q) = \neg\, \mathsf{EF}(p \wedge \mathsf{EF}\, \neg q)$ is shown in Figure 6.2 with the initial upper and lower bounds shown to the right of each node. The bounds are initialized according to Table 3.1. IICTL starts by checking if any of the initial states does not satisfy the

[1] A simple yet effective optimization for safety properties is to instead consider the precondition of the accepting state(s) as the reachability target. For this example, the target would be $(s_0 \wedge p \wedge \neg q) \vee (s_1 \wedge \neg q)$. This strengthens the inductive hypothesis from $\neg s_2$, which does not provide any useful information, to $(\neg s_0 \vee \neg p \vee q) \wedge (\neg s_1 \vee q)$, which through including system variables can help prune unreachable states.

upper bound of the root node; if so, the property fails. Otherwise, it proceeds by carrying out an upper bound reachability query to check for the existence of a counterexample trace from some initial state to a state satisfying p; the absence of such a trace indicates that the property passes. If a p-state is reachable, a lower bound query checks whether any $(p \wedge \neg q)$-state is reachable; any such reachable state indicates a violation of the property. If no $(p \wedge \neg q)$-state is reachable, the reachable p-state t from the upper bound query is undecided for the $\wedge$-node and is passed down the right branch of the graph. IICTL next attempts to complete the counterexample trace by checking whether t can reach any $\neg q$-state. If no $\neg q$-states are reachable from t, an inductive proof that excludes t is used to refine the upper bound of the bottom EF node, triggering a refinement to the corresponding bound of the $\wedge$-node, and forcing IICTL to search for a reachable p-state different from t.

This example illustrates the differences between the goal-driven strategy of IC3 and the data-driven strategy of IICTL. Whereas IICTL proceeds forward from the initial states and breaks down the reachability problem into two (supposedly easier) problems, IC3—enabled by the composition of the automaton with the model—uses one **global** query that proceeds backwards from the bad states, pruning out unreachable states until it either finds a counterexample trace or proves the absence of any.

When would one approach work better than the other? On the one hand, if the model is such that the property passes vacuously because no p-states are reachable, IICTL would immediately find the proof through the first reachability query. On the other hand, depending on the structure of the unreachable state-space, IC3 might struggle to discover this fact. In a different model in which every p-state—whether reachable or not—only has p successors and $p \Rightarrow q$, IC3 is likely to discover the inductiveness of p quickly because it proceeds backwards and use it to prove the property. IICTL can also arrive at this fact, depending on how well it is able to generalize a proof of unreachability of a particular p-state to any $\neg q$-state. However, it may take it several iterations of finding reachable p-states that cannot reach a $\neg q$-state before it converges. This discussion points out that the structure of the model could give an edge to one approach over the other.

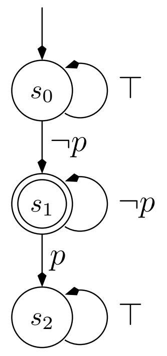


Figure 6.3: Büchi automaton for $\mathsf{F}\,\mathsf{G}\,\neg p$

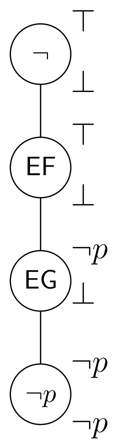


Figure 6.4: CTL parse graph for $\mathsf{AG}\,\mathsf{AF}\,p$

Example 2. Consider the LTL property $\mathsf{G}\,\mathsf{F}\,p$ whose CTL equivalent is $\mathsf{AG}\,\mathsf{AF}\,p$. (See Appendix A for a proof.) The Büchi automaton for the negation of the LTL property is shown in Figure 6.3. Fair can be used to determine whether a fair cycle exists in the composition of the model with the automaton. Fair picks a fair state, an s_1-state in the given automaton, and attempts to complete a fair cycle by checking whether this state is reachable and can reach itself. If so, the property is violated. Otherwise, an inductive proof excludes the s_1-state—and many other states—from future consideration. This process continues until either a fair cycle is found or the reachable state space is found to contain none.

The parse graph for the equivalent CTL formula is shown in Figure 6.4. IICTL first performs a query that checks for any reachable $\neg p$-state; the absence of one indicates that the property holds. Otherwise, it proceeds by checking if the reached $\neg p$-state t is on a cycle in which every state satisfies $\neg p$. If so, a counterexample is found. Otherwise, the inductive proof is used to exclude t from the upper bound of the EG-node and the EF query is repeated.

Similar to the previous example, this example contrasts the goal-driven approach of Fair to the data-driven approach of IICTL. Since both incoming edges to state s_1 in the automaton are labeled $\neg p$, the s_1-state that Fair picks must be a $\neg p$-state. Fair then checks whether this state is reachable. In contrast, IICTL starts by checking for **any** reachable $\neg p$-state. If the model has no reachable $\neg p$-states, IICTL would conclude with one reachability query. At the other extreme, suppose the model does have reachable $\neg p$-states, but each of them is on a cycle of $\neg p$-states. Because IICTL checks for reachability through invoking a safety model checker, and because safety model checkers tend to produce short counterexamples, IICTL would likely refute the property faster than Fair, which randomly picks $\neg p$-states that have no guarantees of being of close proximity to the model's initial states, and thus might increase the difficulty of the reachability queries. However, for models in which it is hard to find a reachable $\neg p$-state but it is easy to prove that they are not on a cycle of $\neg p$-states (for instance, if every $\neg p$-state only has p-successors), then Fair would likely prove the property quickly.

CTL Formula	Equivalent LTL Formula	Safety or Progress
$\mathsf{AG}(p \rightarrow \mathsf{AG}\, q)$	$\mathsf{G}(p \rightarrow \mathsf{G}\, q)$	Safety
$\mathsf{AG}(p \rightarrow \mathsf{A}\, q\, \mathsf{W}\, r)$	$\mathsf{G}(p \rightarrow q\, \mathsf{W}\, r)$	Safety
$\mathsf{AG}(p \rightarrow \mathsf{AX}^n\, q)$	$\mathsf{G}(p \rightarrow \mathsf{X}^n\, q)$	Safety
$\mathsf{AG}(p \rightarrow \mathsf{AX}\,\mathsf{AG}\, q)$	$\mathsf{G}(p \rightarrow \mathsf{X}\,\mathsf{G}\, q)$	Safety
$\mathsf{AG}(p \rightarrow (q \rightarrow \mathsf{AX}\, r \wedge \neg q \rightarrow \mathsf{AX}\, \neg r))$	$\mathsf{G}(p \rightarrow (q \leftrightarrow \mathsf{X}\, r))$	Safety
$p \rightarrow \mathsf{AG}(q \rightarrow \mathsf{AX}\, r)$	$p \rightarrow \mathsf{G}(q \rightarrow \mathsf{X}\, r)$	Safety
$\mathsf{A}\, p\, \mathsf{U}\, q$	$p\, \mathsf{U}\, q$	Progress
$\mathsf{AG}\,\mathsf{AF}\, p$	$\mathsf{G}\,\mathsf{F}\, p$	Progress
$\mathsf{AG}(p \rightarrow \mathsf{AF}\, q)$	$\mathsf{G}(p \rightarrow \mathsf{F}\, q)$	Progress
$\mathsf{AG}(p \rightarrow \mathsf{AX}^n\,\mathsf{AF}\, q)$	$\mathsf{G}(p \rightarrow \mathsf{X}^n\,\mathsf{F}\, q)$	Progress

Table 6.1: The subset of properties from Section 3.7 expressible in both CTL and LTL.

The previous examples highlight cases in which one of the two IIV approaches—goal-driven or data-driven—may perform better than the other. However, it remains to see whether in practice one of the two approaches dominates the other. To determine if this is the case, we extracted the subset of properties from the experiments in Section 3.7 that are expressible in both CTL and LTL, and compared the performance of IICTL to that of IC3 on safety properties for models with no fairness constraints, and to Fair on the others. For 697 of the 1245 properties used in the experiments in Section 3.7, we could easily determine if they satisfy the syntactic restrictions of ACTL$^{\text{det}}$ and are thus known to be in CTL $\cap$ LTL (see Appendix A). Table 6.1 lists the CTL formulae that were found to be in CTL $\cap$ LTL, their equivalent LTL formulae, and whether they are safety or progress properties. Each property was given a timeout of 900 seconds.

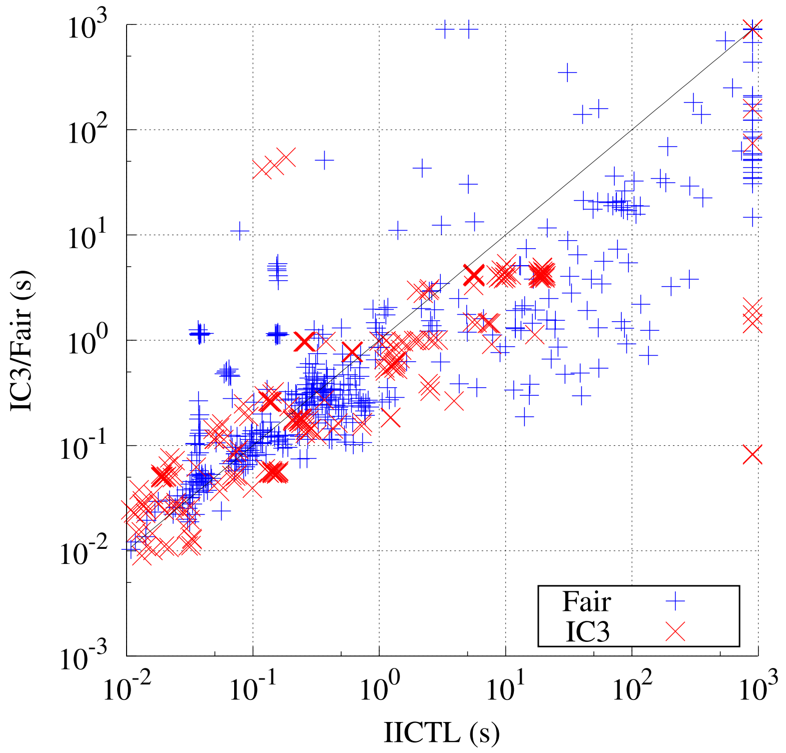


Figure 6.5: Comparing IICTL to IC3/Fair on properties in CTL ∩ LTL.

Figure 6.5 shows a scatter plot that compares the performance of the two approaches, where properties run by IC3 (281 out of the 697) are marked by a $\times$, and those run by Fair (416 out of the 697) are marked by a $+$. Whereas IICTL timed out on 45 properties, IC3 and Fair timed out on 17 properties only. The total CPU time for IICTL was 49058 seconds, and for IC3 and Fair was 21831 seconds. While IC3 and Fair performed better overall, and were >5% faster than IICTL on 227 properties, there were 75 properties for which IICTL was >5% faster than IC3 or Fair. These results indicate that neither approach dominates the other on properties in CTL ∩ LTL, and point out the potential advantage of having a flexible strategy; for example one which applies both approaches in parallel.

It is interesting to note that a data-driven, i.e., IICTL-like, approach can be applied to the automaton directly. Queries performed by the approach have a direct correspondence with those performed by IICTL on the CTL parse graph. For example, for the automaton shown in Figure 6.1, the approach first checks the viability of the transition from s_0 to s_2 through a query that checks for a reachable $(p \wedge \neg q)$-state. This query corresponds to the first lower bound EF query that IICTL carries out. If this transition is not viable, the only way the property could be violated is to reach an s_2-state via some s_1-state. Thus, a query is carried out to check for a reachable s_1-state, or equivalently a reachable p-state. This query corresponds to IICTL's upper bound EF query. If no s_1-states are reachable, the property holds. Otherwise, the reachable s_1-state t is potentially on a counterexample path. To see whether this is the case, a reachability query is performed that checks whether t can reach any s_2-state (equivalently any $\neg q$-state). This reachability query corresponds to the one that IICTL carries out to decide whether t satisfies the subformula rooted at the EF-node in the right branch of the CTL parse graph of Figure 6.2. A negative answer to this query triggers refinement of the label on the (s_0, s_1)-edge of the automaton, which forces the search for a reachable p-state other than t. This refinement of the label corresponds to IICTL's refinement of the upper bound of the EF node in the right branch of the CTL parse graph.

The above discussion also points out another important observation: IIV is a framework that admits multiple strategies. The goal-driven approach of IC3 and Fair and the data-driven approach of IICTL are the two extremes on a spectrum of possible IIV strategies. It remains open whether it is possible to develop hybrid algorithms that combine the best of the two approaches. Such a hybrid algorithm might not be feasible for the full CTL or LTL classes, but could be viable for fragments of these classes, for example the common fragment of CTL and LTL, or the LTL safety class.

Chapter 7

Conclusions

Model checking has witnessed several major developments in its history that have greatly contributed to efficiency and applicability: from the symbolic representation and manipulation of systems, to the employment of SAT solvers as reasoning engines, to automatic abstraction and refinement, to interpolation-based model checking, to IC3. This thesis attempted to, once again, push the efficiency boundaries of model checking through adopting the incremental, inductive verification (IIV) approach introduced by IC3. The thesis developed an IIV-based model checking algorithm for CTL properties, proposed improvements to IC3's generalization procedure, and pointed out the wide space of strategies that the IIV framework admits.

IICTL, the IIV-based algorithm for CTL model checking, fills a long-standing gap created with the introduction of bounded model checking for LTL, which is the absence of practical SAT-based algorithms for branching time properties. Similar to how IC3 outperformed other safety model checking algorithms and how Fair outperformed other fair-cycle detection algorithms, IICTL was shown to outperform existing CTL algorithms. This provides further evidence of the effectiveness of the IIV approach.

We then turned our attention in Chapter 5 to improving IC3 because of the potential high impact since IC3 is a core decision engine in Fair and IICTL and is an algorithm that deals with the most commonly used type of properties—safety properties. A point that was made repeatedly in the thesis is the central role of generalization in IIV. It is clear that improvements to the generalization procedures of IIV algorithms can significantly enhance their performance. The improvement to

IC3's generalization procedure presented in the thesis is motivated by the performance boost that addressing counterexamples to generalization (CTGs) provides to IICTL. CTGs, in the context of IC3, turn out to be deep backward reachable states and addressing them significantly reduces the depth of IC3's explicit traversal of the state graph.

Chapter 6 highlighted the difference in strategies between the existing IIV algorithms, IC3 and Fair, and the one introduced in this thesis, IICTL; whereas IC3 and Fair follow a goal-driven approach, IICTL follows a data-driven one. The chapter also gave empirical evidence that for some models, one strategy works better than the other, which points out the importance of being flexible in the usage of strategies.

Bibliography

[1] Magdy S. Abadir, Kenneth L. Albin, John Havlicek, Narayanan Krishnamurthy, and Andrew K. Martin. Formal verification successes at motorola. Formal Methods in System Design, 22(2):117–123, 2003.

[2] URL: `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[3] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In Tools and Algorithms for the Construction of Systems (TACAS), pages 411–425, 2000. LNCS 1785.

[4] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In Dominique Borrione and Wolfgang Paul, editors, Correct Hardware Design and Verification Methods, volume 3725 of Lecture Notes in Computer Science, pages 254–268. Springer Berlin Heidelberg, 2005.

[5] M. Awedh and F. Somenzi. Proving more properties with bounded model checking. In R. Alur and D. Peled, editors, Sixteenth Conference on Computer Aided Verification (CAV'04), pages 96–108. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[6] M. Awedh and F. Somenzi. Termination criteria for bounded model checking: Extensions and comparison. Electronic Notes in Theoretical Computer Science, 144(1):51–66, 2006. Presented at the Third International Workshop on Bounded Model Checking (BMC'05).

[7] I. Beer, S. Ben-David, C. Eisner, Y. Engel, R. Gewirtzman, and A. Landver. Establishing PCI compliance using formal verification: a case study. In Computers and Communications, 1995., Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on, pages 373–377, Mar 1995.

[8] Shoham Ben-David, Cindy Eisner, Daniel Geist, and Yaron Wolfsthal. Model checking at IBM. Form. Methods Syst. Des., 22(2):101–108, March 2003.

[9] Bob Bentley. Validating the intel pentium 4 microprocessor. In Proceedings of the 38th Annual Design Automation Conference, DAC '01, pages 244–248, New York, NY, USA, 2001. ACM.

[10] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for CTL*. In Logic in Computer Science (LICS), pages 388–397, San Diego, CA, June 1995.

[11] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. Electronic Notes in Theoretical Computer Science, 66(2), July 2002. Formal Methods for Industrial Critical Systems (FMICS'02).

[12] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99), pages 193–207, Amsterdam, The Netherlands, March 1999. LNCS 1579.

[13] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor unit using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, Eleventh Conference on Computer Aided Verification (CAV'99), pages 60–71. Springer-Verlag, Berlin, 1999. LNCS 1633.

[14] A. R. Bradley. SAT-based model checking without unrolling. In Verification, Model Checking, and Abstract Interpretation (VMCAI'11), pages 70–87, Austin, TX, January 2011. LNCS 6538.

[15] A. R. Bradley and Z. Manna. Checking safety by inductive generalization of counterexamples to induction. In Formal Methods in Computer Aided Design (FMCAD'07), pages 173–180, Austin, TX, November 2007.

[16] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang. An incremental approach to model checking progress properties. In Formal Methods in Computer Aided Design (FMCAD'11), pages 144–153, Austin, TX, November 2011.

[17] Aaron R. Bradley. k-step relative inductive generalization. Technical report, CU Boulder, March 2010. `http://arxiv.org/abs/1003.3649`.

[18] R. Brayton, N. Een, and A. Mishchenko. Continued relevance of bit-level verification research. In Proc. Usable Verification, pages 15–16, November 2010.

[19] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, C-35(8):677–691, August 1986.

[20] J. R. Büchi. On a decision method in restricted second order arithmetic. In Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science, pages 1–11. Stanford University Press, 1962.

[21] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In Proceedings of the Fifth Annual Symposium on Logic in Computer Science, pages 428–439, June 1990.

[22] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In Formal Methods in Computer Aided Design (FMCAD'11), pages 135–143, Austin, TX, November 2011.

[23] E. Clarke, D. Kröning, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In Verification, Model Checking, and Abstract Interpretation, pages 85–96, Venice, Italy, January 2004. Springer. LNCS 2937.

[24] E. M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In Proc. Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency, pages 428–438, Berlin, 1988. Springer-Verlag. LNCS 354.

[25] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Proceedings Workshop on Logics of Programs, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.

[26] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In Principles of Programming Languages (POPL), pages 117–126, January 1983.

[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. ACM Transaction on Programming Languages and Systems, 8(2):244–263, 1986.

[28] W. Craig. Linear reasoning. a new form of the Herbrand-Gentzen theorem. Journal of Symbolic Logic, 22(3):250–268, September 1957.

[29] X. Du, S. A. Smolka, and R. Cleaveland. Local model checking and protocol analysis. Software Tools for Technology Transfer, 3(1):219–241, November 1999.

[30] Jori Dubrovin, Tommi Junttila, and Keijo Heljanko. Exploiting step semantics for efficient bounded model checking of asynchronous systems. Science of Computer Programming, 77(10-11):1095–1121, 2012.

[31] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property-directed reachability. In Formal Methods in Computer Aided Design (FMCAD'11), pages 125–134, Austin, TX, November 2011.

[32] N Eén and N. Sörensson. An extensible SAT-solver. In Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), pages 502–518, S. Margherita Ligure, Italy, May 2003. Springer-Verlag. LNCS 2919.

[33] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. Science of Computer Programming, 2:241–266, 1982.

[34] Limor Fix. Fifteen years of formal property verification in intel. In Orna Grumberg and Helmut Veith, editors, 25 Years of Model Checking, volume 5000 of Lecture Notes in Computer Science, pages 139–144. Springer Berlin Heidelberg, 2008.

[35] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In Proceedings of the Symposium on Principles of Programming Languages, pages 163–173, 1980.

[36] A. Gupta. Formal hardware verification methods: A survey. Formal Methods in System Design, 1:151–238, 1992.

[37] Hardware model checking competition. `http://fmv.jku.at/hwmcc`.

[38] URL: `http://iimc.colorado.edu`.

[39] International Technology Roadmap for Semiconductors. URL: `http://www.itrs.net/Links/2011ITRS/Home2011.htm`, 2011.

[40] J. A. W. Kamp. Tense Logic and the Theory of Linear Order. PhD thesis, University of California at Los Angeles, 1968.

[41] D. Kröning and O. Strichman. Efficient computation of recurrence diameters. In Verification, Model Checking, and Abstract Interpretation, pages 298–309, New York, NY, January 2003. Springer. LNCS 2575.

[42] K. G. Larsen. Proof systems for Hennessy-Milner logic with recursion. Theoretical Computer Science, 72(2-3):265–288, 1990.

[43] M. Maidl. The common fragment of CTL and LTL. In Proc. 41th Annual Symposium on Foundations of Computer Science, pages 643–652, 2000.

[44] Z. Manna and A. Pnueli. Temporal Verification of Reactive Systems: Safety. Springer-Verlag, 1995.

[45] K. L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, Boston, MA, 1994.

[46] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, Fourteenth Conference on Computer Aided Verification (CAV'02), pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[47] K. L. McMillan. Interpolation and SAT-based model checking. In W. A. Hunt, Jr. and F. Somenzi, editors, Fifteenth Conference on Computer Aided Verification (CAV'03), pages 1–13. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[48] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In Proceedings of the Design Automation Conference, pages 530–535, Las Vegas, NV, June 2001.

[49] R. Oshman. Bounded model-checking for branching-time logic. Master's thesis, Technion, Haifa, Israel, June 2008.

[50] W. Penczek, B. Woźna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. Fundamenta Informaticae, 51(1-2):135–156, 2002.

[51] A. Pnueli. The temporal logic of programs. In IEEE Symposium on Foundations of Computer Science, pages 46–57, Providence, RI, 1977.

[52] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Proceedings of the Fifth Annual Symposium on Programming, 1981.

[53] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04), pages 31–45, Barcelona, Spain, March-April 2004. LNCS 2988.

[54] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt, Jr. and S. D. Johnson, editors, Formal Methods in Computer Aided Design, pages 108–125. Springer-Verlag, November 2000. LNCS 1954.

[55] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. Theoretical Computer Science, 89(1):161–177, 1991.

[56] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In Proceedings of the First Symposium on Logic in Computer Science, pages 322–331, Cambridge, UK, June 1986.

[57] URL: http://vlsi.colorado.edu/∼vis.

[58] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy abstraction and SAT-based reachability for hardware model checking. In Formal Methods in Computer-Aided Design (FMCAD'12), October 2012.

[59] B. Y. Wang. Proving $\forall\mu$-calculus properties with SAT-based model checking. In Formal Techniques for Networked and Distributed Systems (FORTE 2005), pages 113–127, Taipei, Taiwan, October 2005. LNCS 3731.

[60] B. Woźna. ACTL* properties and bounded model checking. Fundamenta Informaticae, 63(1):65–87, 2004.

Appendix A

Properties Expressible in Both ACTL and LTL

Maidl [43] characterized the fragment of properties expressible in both ACTL and LTL. A sufficient condition for an ACTL formula to be expressible in LTL is to satisfy the syntactic restriction of the class ACTL$^{\mathrm{det}}$ [43]. ACTL$^{\mathrm{det}}$ is inductively defined as follows:

Definition 1 (ACTL$^{\mathrm{det}}$)**.** Every atomic proposition is an ACTL$^{\mathrm{det}}$ formula. In addition, if φ and ψ are ACTL$^{\mathrm{det}}$ formulae, then so are $\varphi \wedge \psi$, $\mathsf{AX}\,\varphi$, $(p \wedge \varphi) \vee (\neg p \wedge \psi)$, $\mathsf{A}(p \wedge \varphi)\,\mathsf{U}(\neg p \wedge \psi)$, $\mathsf{A}(p \wedge \varphi)\,\mathsf{W}(\neg p \wedge \psi)$.

Theorem 2 ([43])**.** *Let φ be an ACTL formula. Then there exists an LTL formula ψ which is equivalent to φ iff φ can be expressed in ACTL^{det}.*

In the following theorem, φ^d denotes the formula resulting from dropping all path quantifiers from a CTL formula φ.

Theorem 3 ([24, 43])**.** *Given an ACTL formula φ, if $\varphi \in ACTL^{det}$, then the LTL formula φ^d is equivalent to φ.*

Claim 1. The ACTL formula $\varphi = \mathsf{AG}(p \rightarrow \mathsf{AG}\,q)$ is equivalent to the LTL formula $\mathsf{G}(p \rightarrow \mathsf{G}\,q)$.

Proof. First we show that φ is in ACTL$^{\text{det}}$ through the following equivalences[1] :

$$\begin{aligned}
\mathsf{AG}(p \to \mathsf{AG}\, q) &\Leftrightarrow \mathsf{AG}(p \to \mathsf{A}\, q \,\mathsf{W}\, \bot) \\
&\Leftrightarrow \mathsf{AG}(p \to \mathsf{A}(\top \wedge q)\,\mathsf{W}(\bot \wedge \bot)) \\
&\Leftrightarrow \mathsf{AG}(\neg p \vee \mathsf{A}(\top \wedge q)\,\mathsf{W}(\bot \wedge \bot)) \\
&\Leftrightarrow \mathsf{AG}(\neg p \vee (p \wedge \mathsf{A}(\top \wedge q)\,\mathsf{W}(\bot \wedge \bot))) \\
&\Leftrightarrow \mathsf{AG}((\neg p \wedge \top) \vee (p \wedge \mathsf{A}(\top \wedge q)\,\mathsf{W}(\bot \wedge \bot))) \\
&\Leftrightarrow \mathsf{A}((\neg p \wedge \top) \vee (p \wedge \mathsf{A}(\top \wedge q)\,\mathsf{W}(\bot \wedge \bot)))\,\mathsf{W}\,\bot \\
&\Leftrightarrow \mathsf{A}(\top \wedge ((\neg p \wedge \top) \vee (p \wedge \mathsf{A}(\top \wedge q)\,\mathsf{W}(\bot \wedge \bot))))\,\mathsf{W}(\bot \wedge \bot)
\end{aligned}$$

Finally, by Theorem 3, the equivalent LTL formula is $\varphi^d = \mathsf{G}(p \to \mathsf{G}\, q)$. □

Claim 2. The ACTL formula $\varphi = \mathsf{AG}\,\mathsf{AF}\, p$ is equivalent to the LTL formula $\mathsf{G}\,\mathsf{F}\, p$.

Proof. First we show that φ is in ACTL$^{\text{det}}$:

$$\begin{aligned}
\mathsf{AG}\,\mathsf{AF}\, p &\Leftrightarrow \mathsf{AG}\,\mathsf{A}\,\neg p\,\mathsf{U}\, p \\
&\Leftrightarrow \mathsf{AG}\,\mathsf{A}(\neg p \wedge \top)\,\mathsf{U}(p \wedge \top) \\
&\Leftrightarrow \mathsf{A}(\mathsf{A}(\neg p \wedge \top)\,\mathsf{U}(p \wedge \top))\,\mathsf{W}\,\bot \\
&\Leftrightarrow \mathsf{A}(\top \wedge (\mathsf{A}(\neg p \wedge \top)\,\mathsf{U}(p \wedge \top)))\,\mathsf{W}(\bot \wedge \bot)
\end{aligned}$$

By Theorem 3, the equivalent LTL formula is $\varphi^d = \mathsf{G}\,\mathsf{F}\, p$. □

[1] Note that both $\bot$ and $\top$ are in ACTL$^{\text{det}}$. The latter is equivalent to $p \wedge \neg p$, and the former is equivalent to $(p \wedge p) \vee (\neg p \wedge \neg p)$.