FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Software Verification with IC3 via Abstraction and Interpolation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Johannes Birgmeier
Matrikelnummer 0902998

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Weissenbacher, Georg; Assistant Prof. Dipl.-Ing. D.Phil.
Mitwirkung: Veith, Helmut; Univ.Prof. Dipl.-Ing. Dr.techn.

Wien, 04.12.2013

	(Unterschrift Verfasser)	(Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Software Verification with IC3 via Abstraction and Interpolation

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Johannes Birgmeier
Registration Number 0902998

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Weissenbacher, Georg; Assistant Prof. Dipl.-Ing. D.Phil.
Assistance: Veith, Helmut; Univ.Prof. Dipl.-Ing. Dr.techn.

Vienna, 04.12.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Johannes Birgmeier
Vinzenzgasse 24/8, 1180 Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I would like to thank my advisor, Georg Weissenbacher, for his continued support and countless valuable suggestions during the development of this thesis. Apart from this, his special achievements include an unfaltering commitment to provide reviews, answering emails at ungodly hours and staying calm when we experienced general confusion about certain aspects of the algorithms involved in this work. These commitments grant him the highest honors in the advisors' hall of fame, the whereabouts of which are known only to the initiated.

I am deeply indebted to Aaron Bradley, who during a visit to the Vienna University of Technology in June 2013 came up with the idea of how to extend IC3 to software transition systems, and provided continued support throughout the development of this work whilst entirely disregarding any lack of apparent incentives to do so. A number of ideas elaborated in this thesis were proposed by him.

Last but certainly not least, I would like to thank Helmut Veith for his decision to invite me into his FORSYTE group in the first place, being an inspiration for what it means to be a researcher, providing guidance in critical hours and spreading good spirits during group meetings.

Abstract

Software model checking is an approach to formal software verification based on reasoning about the states a program can be in. A software model checker can prove that certain properties hold on a given program. Properties expressing that certain states must never be reached during a run are called *safety properties*.

In this work, we aim to construct a model checker that can prove or refute safety properties on certain programs. The approach for model checking is based on the principle of incremental, inductive model checking. An incremental, inductive model checker proves safety properties by incrementally constructing a description of a set of states that the program can never leave and all of which are *safe*.

The model of the program that the checker operates on is the *transition system*. The transition systems we derive from software programs are expressed as first-order formulas over the theory of quantifier-free linear integer arithmetic. Such transition systems operate on infinitely many states, since all first-order constants in the transition system can be interpreted as an arbitrary integer value.

The method we develop in this work is based on the IC3 model checker ([10]). IC3 can prove properties only on systems that comprise finitely many states. Since we are aiming at proving safety properties on systems working with infinitely many states, we reduce the problem to the finite-state case by applying Boolean predicate abstraction on the state space. The reduced state space is called the *abstract domain*.

However, Boolean predicate abstraction is subject to the difficulty of choosing a fitting set of predicates that will allow the algorithm to prove safety properties. We approach the problem by starting with a set of heuristically determined predicates, and adding new instances to the abstract domain predicates as needed during the run of the algorithm, thus *refining* the abstract domain.

In the model checker we developed, refinement is predominantly triggered by *spurious counterexamples*: The abstract domain admits transitions which could not occur in the original state space. In order to eliminate such spurious counterexamples, we refine the abstract domain with predicates extracted from Craig interpolants ([20, 21]) over a formula that describes the infeasibility of the spurious counterexample. Thus, our approach to abstraction is an instance of *counterexample-guided abstraction refinement* ([18, 19]).

The result of this thesis is IC3-CEGAR, an incremental, inductive model checker that is capable of proving or disproving safety properties on certain software programs.

Kurzfassung

Software-Modellprüfung ist ein Ansatz zur Verifikation von Software-Programmen, der auf Schlussfolgerungen über Programmzustände aufgebaut ist. Ein Software-Modellprüfer kann beweisen oder widerlegen, dass bestimmte Eigenschaften für Software-Programme gelten. Eigenschaften, die beschreiben, dass gewisse Zustände während der Programmausführung nie vorkommen dürfen, heißen *Sicherheitseigenschaften*.

In dieser Arbeit wird ein Modellprüfungsalgorithmus entwickelt, der Sicherheitseigenschaften für gewisse Programme beweisen und widerlegen kann. Der Modellprüfungsansatz basiert auf dem Prinzip der induktiven, inkrementellen Modellprüfung. Ein induktiver, inkrementeller Modellprüfungsalgorithmus beweist Sicherheitseigenschaften, indem er nach und nach eine Beschreibung einer Zustandsmenge aufbaut, welche das Programm während der Ausführung nie verlassen kann, und die alle *sicher* sind.

Die Modelle der Programme, auf denen der Modellprüfungsalgorithmus arbeitet, sind sogenannte *Übergangssysteme*. Die in dieser Arbeit vorgestellen Übergangssysteme werden als prädikatenlogische Formeln über der Theorie der quantorenfreien linearen ganzzahligen Arithmetik beschrieben. Derartige Übergangssysteme arbeiten auf unendlich vielen Zuständen, da jede prädikatenlogische Konstante im Übergangssystem als beliebige ganze Zahl interpretiert werden kann.

Der in dieser Arbeit entwickelte Ansatz basiert auf dem IC3-Modellprüfungsalgorithmus ([10]). IC3 beweist Eigenschaften allerdings nur auf Übergangssystemen, die auf endlich vielen Zuständen arbeiten. Da wir allerdings Sicherheitseigenschaften für Systeme beweisen wollen, die unendliche viele Zustände bearbeiten, reduzieren wir das Problem auf den endlichen Fall, indem wir Prädikatenabstraktion auf den Zustandsraum anwenden. Der reduzierte Zustandsraum heißt auch *abstrakte Domäne*.

Allerdings ist es im Zuge von Prädikatenabstraktion schwierig, herauszufinden, welche Prädikate es dem Algorithmus ermöglichen, Sicherheitseigenschaften zu beweisen. Wir gehen an das Problem heran, indem wir mit einer Menge von heuristisch gewählten Prädikaten für die abstrakte Domäne anfangen, und während der Ausführung des Modellprüfungsalgorithmus neue Prädikate zur Menge hinzufügen, um die abstrakte Domäne zu *verfeinern*.

In unserem Modellprüfungsalgorithmus wird die abstrakten Domäne verfeinert, wenn *unechte Gegenbeispiele* gefunden werden: Das heißt, die abstrakte Domäne erlaubt Übergänge, die im ursprünglichen Übergangssystem nicht vorkommen könnten. Um derartige unechte Gegenbeispiele zu beseitigen, wird die abstrakte Domäne mit Prädikaten verfeinert, die aus Craig-Interpolanten ([20, 21]) entnommen werden. Die Formeln, aus denen die Craig-Interpolanten

entnommen werden, beschreiben, warum das Gegenbeispiel im konkreten Übergangssystem nicht vorkommen kann. Daher ist unser Ansatz ein Beispiel für *Counterexample-Guided Abstraction Refinement* ([18, 19]).

Das Ergebnis dieser Masterarbeit ist IC3-CEGAR, ein inkrementeller, induktiver Modellprüfungsalgorithmus, der Sicherheitseigenschaften auf gewissen Softwareprogrammen beweisen oder widerlegen kann.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Formal Software Verification

Formal Software Verification is a name for a collection of formal techniques whose goal it is to prove or disprove the correctness of a program in that it fulfills certain well-defined properties.

In order to check whether programs are bug-free, engineers typically perform *testing* on the software they have produced. To this end, *test cases* are written; these test cases usually run a program with a number of pre-specified inputs and check whether the outputs for these pre-specified inputs are correct. If a test run is successful, then it has been proved that the program does not contain bugs for these specific inputs. However, there are many other inputs, and possibly infinitely many, that are left untested, and may result in a bug if the program is run on them.

In contrast to testing, formal verification can be used to prove that a program is bug-free according to the specification for *any* input. It is, of course, impossible to run a program with infinitely many inputs to check whether it contains a bug for any of them. Therefore, sophisticated formal verification techniques have been developed that allow to prove or disprove the correctness of a program with different approaches.

Because formal verification is such a powerful technique, and because it is usually harder to prove a program correct than to find bugs by running a number of test cases, formal software verification is computationally expensive. In the early days of computing, proofs of program correctness were constructed manually by computer scientists. Formal systems such as Hoare Logic ([31]) allowed computer scientists to prove that a program would return the correct output given the right input.

More recently, techniques have begun to emerge that would allow the *automatic* or *computer-aided* formal verification of software programs: Such automatic verifiers are programs that take as input other programs and either prove them correct or return a counterexample to correctness. It is in general undecidable to prove whether a program is correct. This means that all automatic verifiers can will fail on some programs: They cannot prove whether the program is correct or incorrect. In such a case, the automatic verifier will either continue to run forever or terminate with a message that the program cannot be either proved correct or proved faulty. The result that there is no program that can prove all programs correct or incorrect is one of the oldest results in computer science; it is provable by a reduction from the Halting Problem, which was shown to be undecidable by Alan Turing in 1936 ([46]).

Since formal software verification, whether manual or automatic, is such a costly technique, it is less commonly applied than simple software testing. However, formal software verification bears a distinct advantage over simple testing in certain environments: Formal software verification is necessary if the correctness of a program is critical, and it must be ascertained that the program will fail under no circumstances.

Testers will usually not succeed in finding those very rare corner cases in which inputs cause catastrophic bugs, as there are usually millions of millions of possible inputs. However, in safety-critical environments it is paramount to find these inputs, or to prove that no such inputs exist. Indeed, even in programs that are less safety critical (such as common Microsoft Windows device drivers) it is desirable to prove that no inputs will make the program crash. Crashes in Windows device drivers cause the infamous blue screen of death. In order to reduce the number of crashes due to device driver errors, Microsoft created a package called SLAM ([2–4, 6]) that is used as the engine of the Static Driver Verifier (SDV).

There are less harmless cases of bugs caused by software errors that resulted in massive property damage or death. Two of the more well-known examples include:

- The Ariane 5 disaster: Due to unverified re-use of certain parts in the rocket's control code involving a data type conversion, it exploded on its first flight.

- Therac-25: A software error in this radiation therapy machine caused it to distribute massive overdoses of radiation in certain cases; at least six people are believed to have died from them.

In both of these cases, formal software verification could have prevented the disasters, albeit it would have taken some time and effort to prove the programs incorrect.

Since the great computation cost associated with formal software verification is a hindrance in the way of making it more commonplace, there is a continuing effort in accelerating automated software verification tools. In order to measure the performance of software verifiers, a competition exists: The Software Verification Competition, or SV-COMP, which measures the performance of software verifiers in multiple categories. The formal verification method we developed in this thesis beats a gold-medal winner of the SV-COMP 2013 on some benchmarks, thus making a step in the direction of establishing formal software verification as a feasible software engineering method.

Model Checking

Model Checking is an automated formal verification technique that is rooted in the problem of determining whether a temporal logic formula is a model of a temporal logic structure. It was introduced by E. Allen Emerson and Edmund Clarke, as well as Joseph Sifakis and Jean-Pierre Queille in the early 1980s ([16, 17, 25, 42, 44, 45]). Temporal verification of programs reaches back even earlier, with notable works by Zohar Manna and Amir Pnueli ([34, 40, 41]).

Model Checking is a method that works by automatically reasoning about the *states* of a system. When software programs are executed, they are in a well-defined state at each moment. This state is determined by the point of execution in the program (also known as the "program counter"), and the values of variables in the program.

One important class of properties to prove for a software program are *safety* properties. Such properties define which states of the program are *safe*, i.e., they are defined to cause no unintended consequences. All states that are not safe are *unsafe*. The software verifier presented in this thesis is a model checker capable of proving safety for software programs by showing that a program can never reach an unsafe state during execution.

Model Checking and Invariants

In order to make the problem statement tractable, we will present a very high-level clarification of the methodology employed in order to prove safety by model checking.

What our software verifier seeks to find is a so-called *inductive invariant* that suffices to prove safety. Such an inductive invariant is a description of a set of program states. This set of states must fulfill the following properties:

- It must include the initial state, i.e., state that the program starts in.
- If the program is in a state inside the inductive invariant, it cannot ever reach a state outside the inductive invariant.
- All of the states in the inductive invariant are safe.

Intuitively, it is clear why a set of states that fulfill the following properties suffices to prove that all of a program's reachable states are safe. It is, of course, undecidable to find a description of such a set of states for every program. However, finding an inductive invariant is possible for many programs that occur in practice.

Main Contribution

The main contribution of this thesis is a model checker that is capable of proving safety by coming up with an inductive invariant. Without inductivity, it is impossible to prove safety for programs over integer arithmetic, as the program needs to be checked for correctness by trying infinitely many inputs.

The main advantages of inductive software verification over model checking techniques such as binary decision diagrams (BDDs, [12, 35]), bounded model checking ([8]) or interpolating verification ([36, 37]) can be summarized as follows:

- **Inductive model checking is capable of proving safety:** In contrast to bounded model checking techniques, which can only prove that there are now runs that lead to an error in up to a fixed number of steps, inductive model checkers such as the one describe here are able to prove that the error is unreachable in *any* number of steps.

- **Inductive model checking is fast at calculating invariants that are good enough for proving safety:** In contrast to BDD-based model checking, which proves safety by calculating the smallest set of reachable states and proving that these reachable states do not overlap with error states, inductive verification will find a set of states that is does not overlap with the error states, contains all reachable states and is inductive.

  This results in a relaxation of the requirements for proving safety: It is computationally expensive to construct the precise set of reachable states. Inductive model checkers such as the one described here may come up with the precise set of reachable states as an inductive invariant, but they may also come up with a different invariant that suffices for proving safety.

- **Inductive model checking will pursue concrete counterexamples to the error and target its effort on proving that concrete traces to the error are not feasible:** Interpolating software verifiers incrementally constructs over-approximations of sets of states reachable in up to a certain number of steps that do not overlap with the error. Interpolating verifiers rely on the assumption that the over-approximations of reachable states will become general enough to prove that the program can never leave these sets of states in any number of steps. This assumption will often not hold.

  In contrast, inductive model checkers such as the one described here will make a targeted effort at trying to prove that there are no concrete runs to the error by incrementally finding new partial runs to the error and showing that these runs are infeasible.

Counterexamples

In case the model checker finds that a program cannot be proved correct, it will come up with a so-called *counterexample trace*. A counterexample trace is a sequence of states of the program that could occur during an actual run that finally leads to an unsafe state. It is possible to determine the exact program inputs that lead to an unsafe state; thus, it is easy to verify that a counterexample trace is actually feasible.

1.2 Technical Problem Statement

The problem we are aiming to solve is twofold:

- We aim at constructing a model checker that is capable of finding inductive invariants for software programs that suffice to prove given safety properties correct for the program.

- If a property does not hold on a given program, a counterexample trace should be printed.

Deciding whether programs are correct is undecidable in general; therefore, we do not make any termination guarantees for our model checker.

The underlying principle is based on the method of *IC3* (**I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness). IC3 is a model checking algorithm designed for hardware model checking. It can be adapted, however, to software model checking by employing a number of techniques, the most notable of which is *Boolean predicate abstraction*.

The main problem in extending IC3 to software programs is that software model checking usually deals with an infinite state space. That is, the model checker has to reason about an infinite number of possible states the program could be in. This is in contrast to hardware model checking, where the state space is both finite and the variables that describe a given state are can only take a binary value. In software model checking, the variables that describe states will usually be of different types, notably integers. In the approach described, we deal with programs that only contain integer variables, as many other variable types can be reduced to integers.

CHAPTER 2

Background

2.1 Prerequisites

The definitions in this section are loosely based on [23], especially regarding the notation.

In this section, the IC3 algorithm for hardware model checking is described. Understanding the principle of the algorithm for the finite state case is paramount for understanding the workings of the algorithm for the infinite state case.

Transition Systems and Inductive Sets of States

Definition 1 (Transition System) *A transition system M is defined as a tuple $\langle S, T\rangle$, where S is a finite set of states and $T \subseteq S \times S$ is a transition relation.*

The set $I \subset S$ is a set of initial states. The set $E \subset S$ is a set of error states, with $P = S \setminus E$ being its complement. A system is correct if no error state can be reached by starting from an initial state and stepping through the transition relation.

Given a set of states, the post-image operator $post : 2^S \mapsto 2^S$ returns the set of states reachable in one step from the input states: $post(Q) = \{s' \in S \mid s \in Q \text{ and } (s, s') \in T\}$. Let $post^0(Q) = Q$ and $post^{i+1} = post(post^i(Q))$: Then $post^n(Q)$ describes the states reachable in n steps from any state in Q. In a similar way, the pre-image operator $pre : 2^S \mapsto 2^S$ return the set of states from which any of the states in Q is reachable within one step: $pre(Q) = \{s \in S \mid s' \in Q \text{ and } (s, s') \in T\}$.

Definition 2 (Inductive Sets) *An* inductive *set of states is a set P with $post(P) \subseteq P$. A set of states P satisfies* initiation *if $I \subseteq P$. A set of states P is called an* inductive invariant *if it is inductive and satisfies initiation.*

The set of states R_I that is reachable from an initial state in any number of steps is called *strongest inductive invariant*. In a correct system, there is also a *weakest inductive invariant* W_E, which is the largest set of states from which no state in E is reachable.

Both R_I and W_E can be defined as follows using the fixed point operator μ:

$$R_I = \mu Q.(I \cup post(Q))$$
$$W_E = S \setminus \mu Q.(E \cup pre(Q))$$

Suppose we had an approximate post-image operator $\hat{post} : 2^S \mapsto 2^S$ that over-approximates the states reachable in one-step: $post(Q) \subset \hat{post}(Q)$ for all sets of states Q. Using this approximate post-image operator, we can calculate an approximate set of reachable states $\hat{R}_I$:

$$\hat{R}_I = \bigcup_{i \geq 0} \hat{post}^i(I)$$

If it turns out that $\hat{R}_I \cap E = \emptyset$, then the error is not reachable from an initial state, since the set $\hat{R}_I$ can only be a super-set of the set of actually reachable states R_I. So the correctness of a transition system can be proven by finding an apt over-approximation of the set of reachable states.

Propositional and First-Order Logic

The definitions on propositional logic in this section follow the descriptions in [47].

Propositional Logic We follow the conventional definition of propositional logic over a set of propositional variables X and the atomic constants $\top$ (for $true$) and $\bot$ (for $false$). A propositional *atom* is a variable or one of the atomic constants $\top$ and $\bot$. We use the logical connectives $\wedge, \vee, \Rightarrow$, and $\neg$ (denoting conjunction, disjunction, implication, and negation). A literal is a variable or the negation of a variable.

A *clause* is a disjunction of literals. The empty clause $\square$ evaluates to $\bot$. The negation of a clause is a conjunction of literals and is called a *cube*. A formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses.

An *interpretation* of a propositional formula is an assignment of truth values to propositional variables. A *model* of a formula is an interpretation under which the formula evaluates to $true$. A formula is *valid* if every interpretation is a model of it. A formula is *satisfiable* if it has at least one model, and *unsatisfiable* otherwise.

Definition 3 **($\models$)** *For two formulas f and g, $f \models g$ if every model of the formula f is also a model of the formula g. This is also written as "f models g".*

The definitions on first-order logic in this section follow the descriptions in [47] and [27].

First-Order Logic First-order logic is defined over a set of variables, functions, and predicates. The set of a formula's predicate and function symbols forms the *vocabulary* of a formula. The logical connectives are the same as in propositional logic. First-order formulas may be quantified by a universal ($\forall$) or existential ($\exists$) quantifier. A formula is *quantifier-free* if it does not contain quantifiers, and *ground* if it does not contain free variables. A *literal* is either a ground atom or the negation of a ground atom.

An *interpretation* of a first-order formula consists of a *domain* and an *interpretation function*. The domain is an arbitrary set of individuals. The interpretation function defines function return values and predicate truth values for given input tuples. The definitions of models, validity and satisfiability are the same as in propositional logic.

A 0-ary function symbol is called a *constant*. A 0-ary predicate is called a *Boolean constant*. A first-order word built of function symbols and variables is called a *term*. $\wp(t_1, \ldots, t_n)$ is called an *atom* if $\wp$ is a predicate symbol and $t_1, \ldots, t_n$ are terms.

Definition 4 (First-Order Theory) *A first-order theory consists of a signature σ defining valid predicate and function symbols, and a set of axioms, which are Boolean expressions set to hold over these predicate and function symbols.*

Definition 5 (Linear Integer Arithmetic) Linear Integer Arithmetic *is a first-order theory consisting of the two constant symbols* 0 *and* 1 *as well as the binary function symbol* $+$ *and the binary predicate symbol* $\leq$*. Linear integer arithmetic is sufficiently expressive to build linear equations/inequalities.*

When solving linear integer arithmetic with an SMT solver, the interpretation domain is defined as the set of integers $\mathbb{Z}$, the $+$ function is defined as addition and the $\leq$ predicate is defined in its usual meaning.

A linear equation/inequality is a predicate $\sum_i a_i x_i + c \circ 0$, where $\circ \in \{=, \leq\}$, all x_i are constants and all a_i as well as c are integers. (Using negation, the predicates $=, \leq$ suffice to express all of $=, \neq, <, >, \leq$ and $\geq$.) Linear equations/inequalities can be expressed using linear integer arithmetic.

The multiplication function $*$ is a short-hand for repeated addition in linear integer arithmetic, as there are no multiplications of two variables in the theory.

Other predicates such as $=$, $<$ and $\geq$ are defined as short-hands for Boolean formulas over the predicate symbol $\leq$; for instance, the binary predicate $a = b$ is defined as the formula $a \leq b \wedge b \leq a$.

Sets of states are described by a predicate $\wp$ in propositional or first-order logic. The set of states described by a formula is equivalent to the models of the formula: $\{s \in S \mid s \models \wp\}$.

Definition 6 (State/Formula Equivalence) *In the propositional case, a state s is equivalent to a vector of assignments of propositional constants to $true$ or $false$. In this case, the state s is also equivalent to a propositional conjunction of literals $\bigwedge_i \ell_i$, where each constant whose value is $false$ in the state occurs negated and each constant whose value is $true$ in the state occurs in plain form.*

In the first-order case, a state s is equivalent to a vector of assignments of first-order constants to elements of the first-order domain. In this case, the state s is also equivalent to a first-order conjunction of equivalences $\bigwedge_i c_i = v_i$, where each constant is set equal with the domain element it is assigned to.

SAT and SMT Solvers

Definition 7 (SAT Solver) *A SAT solver is a program which, given a propositional formula f, can decide whether the formula is satisfiable or unsatisfiable. In case the formula f is satisfiable, the SAT solver can produce a model of the formula.*

Definition 8 (SMT Solver) *An SMT solver is a program which, given a first-order formula f that uses the vocabulary of a particular theory T (such as linear integer arithmetic), can decide whether the formula is satisfiable or unsatisfiable modulo theory T. In case the formula f is satisfiable, the SMT solver can produce a model of the formula.*

AllSAT and the Unsatisfiable Core

Definition 9 (Unsatisfiable Core) *Given an unsatisfiable formula ϕ consisting of the conjuncts $\{\phi_1, \phi_2, \dots\}$, the unsatisfiable core of ϕ is a subset $\phi^c \subseteq \{\phi_1, \phi_2, \dots\}$ such that ϕ^c is also unsatisfiable.*

Definition 10 (AllSAT) *Given a formula ψ, an* AllSAT *query over the formula returns all the models of ψ. If a number of* important *atoms are defined for the call, the all models of the important atoms of the formula will be returned for which a satisfying assignment of the remaining function symbols, constants and variables exists.*

SAT and SMT solvers may support the output of unsatisfiable cores given unsatisfiable formulas and AllSAT queries.

2.2 IC3 – The Finite State Case

The description of IC3 in this section is derived from the original IC3 papers [10] and [11].

Hardware Verification Prerequisites

The state space of a digital hardware element can be described as a set of vectors $\vec{x} = x_1, x_2, \dots, x_n$ of n propositional variables. Inspired by the notion of an underlying digital hardware element, such binary variables are called *latches* in the common IC3 literature, and we will follow this usage. One particular state of the hardware element is a vector of length n where each latch is set to either 1 or 0.

The notation we will use to describe one state is the following: A state $\vec{x} = x_1, x_2, \dots, x_n$ of n propositional variables set to values $v_1, v_2, \dots, v_n$ is described as $(x_1 = v_1 | x_2 = v_2 | \dots | x_n = v_n)$. Following the definition 6, this state is equivalent to the following propositional formula:

$$\bigwedge_i \left\{ \begin{array}{ll} x_i & \text{if } v_i = true \\ \neg x_i & \text{if } v_i = false \end{array} \right\}.$$

The transition relation T is usually encoded by a formula $T(\vec{x}, \vec{x'})$. A state s leads to a states s' iff the tuple models the transition relation:

$$(s, s') \in T \Leftrightarrow s, s' \models T(\vec{x}, \vec{x'}).$$

Any primed variable always refers to the state of the system after one transition, while unprimed variables refer to the current state of the system. A pair of states $(\vec{x}, \vec{x'})$ that is in T is called a pair of *consecutive* states.

Definition 11 (Primed formulas) *A primed formula is defined as the original formula after all the occurring variables have been replaced by their primed counterparts:*

$$F(\vec{x})' = F([\vec{x'}/\vec{x}]),$$

where $[\vec{x'}/\vec{x}]$ means that all occurrences of unprimed variables $\vec{x}$ are to be replaced by primed variables $\vec{x'}$.

The set P is encoded by a propositional formula $P(\vec{x})$. The model checker tries to either prove that no error state can be reached from any initial state, or to find a series of consecutive states that starts from an initial state and ends in an error state. Such a series is called a *counterexample trace*. An arbitrary series of consecutive states is called a *trace*. A formal definition of trace follows that makes use of the equivalence between states and propositional formulas as defined in definition 6:

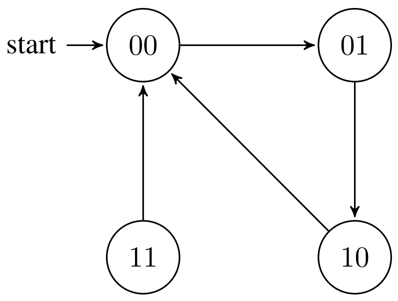
Definition 12 (Trace) *A* trace *is a sequence of states $\langle s_0, s_1, s_2, \ldots, s_n\rangle$ such that for all $0 \leq i < n$, $s_i \wedge T \wedge s'_{i+1}$ is satisfiable. A full counterexample trace is a trace for which the last state s_n overlaps with the error: $s_n \wedge \neg P$ is satisfiable and s_0 is an initial state: $s_n \wedge I$ is satisfiable. If the formula $s_i \wedge T \wedge s'_j$ is satisfiable for two states s_i and s_j, this is also written as $s_i \to s_j$.*

Hardware Verification Example

As a running example, suppose we have a system that contains just two latches, x_1 and x_2. The state space of this element is $\{(x_1 = 0|x_2 = 0), (x_1 = 0|x_2 = 1), (x_1 = 1|x_2 = 0), (x_1 = 1|x_2 = 1)\}$. Since it is clear that the first 0 or 1 describes the state of the latch x_1, while the second 0 or 1 describes the state of the latch x_2, we will from now own omit the variable names when describing individual states, and just write a sequence of 0s or 1s to describe one state. Thus, the state space of our example element is $\{00, 01, 10, 11\}$.

Suppose that in our running example system, the initial state set is the set $\{00, 01\}$. Thus, $I(\vec{x})$ can be described by the propositional formula $\neg x_1$. The models of this formula directly correspond to the given state set.

Suppose that our running example system is a counter that always counts from zero to two and then restarts again at zero. The transition relation can be visualized as the following:



The transition relation $T(\vec{x}, \vec{x'})$ is well-defined by the following formula in propositional logic: $(x_1 \Rightarrow \neg x'_1 \wedge \neg x'_2) \wedge (\neg x_1 \wedge \neg x_2 \Rightarrow \neg x'_1 \wedge x'_2) \wedge (\neg x_1 \wedge x_2 \Rightarrow x'_1 \wedge \neg x'_2)$.

Consecution and Initiation in Symbolic Notation

Given a transition relation T, an initial condition I and an arbitrary formula F:

- F satisfies **initiation** iff $I \Rightarrow F$.
- F satisfies **consecution** iff $F \wedge T \Rightarrow F'$.
- F satisfies **consecution relative to an arbitrary formula** $G(\vec{x})$ iff $F \wedge G \wedge T \Rightarrow F'$.

Outline of the Algorithm

IC3 tries to prove that all reachable states of the system satisfy the property to prove, or stated otherwise: that the set of reachable states is a subset of the property. In order to do this, the algorithm tries to construct a formula $F(\vec{x})$ that satisfies initiation, consecution, and implies the property P to prove.

The IC3 algorithm finds such a formula F by refining and extending a sequence $F_0, F_1, \ldots, F_k$ of formulas that describe over-approximations of the sets of states reachable in up to $0, 1, \ldots, k$ steps. In terms of the previously-defined *post* operator,

$$F_i \supseteq \bigcup_{j \leq i} post^j(I).$$

Any such F_i is called a *frame*. Since the set of states reachable in 0 steps is known as the initial condition I, it always holds that $F_0 = I$. The index i of a frame F_i is called the *level* of the frame. The frame F_k is called the *frontier*.

Initially, the sequence is very short, with $k = 1$. The algorithm increases k as necessary such that frames are added to the end of the sequence. The algorithm then continues to make the over-approximations of states reachable in up to k steps more precise; that is, these formulas $F_1, \ldots F_k$ describe ever smaller sets of states as the algorithm progresses. Making a frame more precise is called *strengthening*.

The following four invariants always hold on the sequence $F_0, F_1, \ldots F_k$:

Inv1 $I = F_0$: The states reachable in 0 steps are equal to the initial set of states.

Inv2 $F_i \Rightarrow F_{i+1}$ for $0 \leq i < k$: There are fewer states reachable in up to i steps than in up to $i+1$ steps.

Inv3 $F_i \Rightarrow P$ for $0 \leq i \leq k$: The states reachable in up to i steps all satisfy the property to prove.

Inv4 $F_i \wedge T \Rightarrow F'_{i+1}$ for $0 \leq i < k$: The states reachable in up to $i+1$ steps are at most one transition away from the states reachable in up to i steps.

Now suppose that IC3 arrives at a point where some frame F_i describes the same set of states as the frame F_{i+1}, that is: $F_i \equiv F_{i+1}$. Then by the four invariants it holds that:

- $F_0 = I$ by **Inv1**, $I \Rightarrow F_1 \Rightarrow F_2 \cdots \Rightarrow F_i$ by **Inv2**: So initiation is satisfied for F_i.
- $F_i \wedge T \Rightarrow F'_{i+1}$ by **Inv4**, and since $F_i \equiv F_{i+1}$ it also holds that $F_i \wedge T \Rightarrow F'_i$: So consecution is satisfied for F_i.
- $F_i \Rightarrow P$ by **Inv3**: So the formula implies the property to prove.

Thus the algorithm has found an inductive invariant in the formula F_i. For the finite state case, it is guaranteed that IC3 will eventually find an i with $F_i \equiv F_{i+1}$.

The two figures 2.1 and 2.2 outline some of the basic principles of IC3.

Detailed Description of the Algorithm

A graphical outline of the algorithm is given in the figure 2.3.

The formulas F_i are always stored in conjunctive normal form (CNF), so they are conjunctions of *clauses*. A clause is a disjunction of *literals*. A literal is an atom or the negation of an atom. The negation of a clause is a conjunction of literals and is called a *cube*.

We will now give an account of the basic workings of the algorithm, clarifying the following:

- How IC3 strengthens frames
- When IC3 adds a new frame to the end of the sequence
- When it checks whether two frames are equal

Note that, for clarity, the presentation in this thesis omits many implementation details and various optimizations that make the algorithm run faster in practice.

Any of the queries in the following description can be answered by a SAT solver.

1. **The base cases:**
   a) Set $k = 0$.
   b) Check whether the initial states satisfy the property ($I \Rightarrow P$); if not, return **false** and an initial state that does not satisfy it.

Figure 2.1: Discovering paths to the error in order to strengthen frames. The states inside F_1 and F_2 will need to be excluded: they lead to the error. If there is an I-predecessor to the state inside F_1, then a feasible counterexample trace from an initial state to the error has been found. If the state inside F_1 has no I-predecessor, it can be generalized and excluded from F_1, thus strengthening the frame. Subsequently, if the state inside F_2 has no further F_1-predecessor, it can be generalized and excluded from F_2.

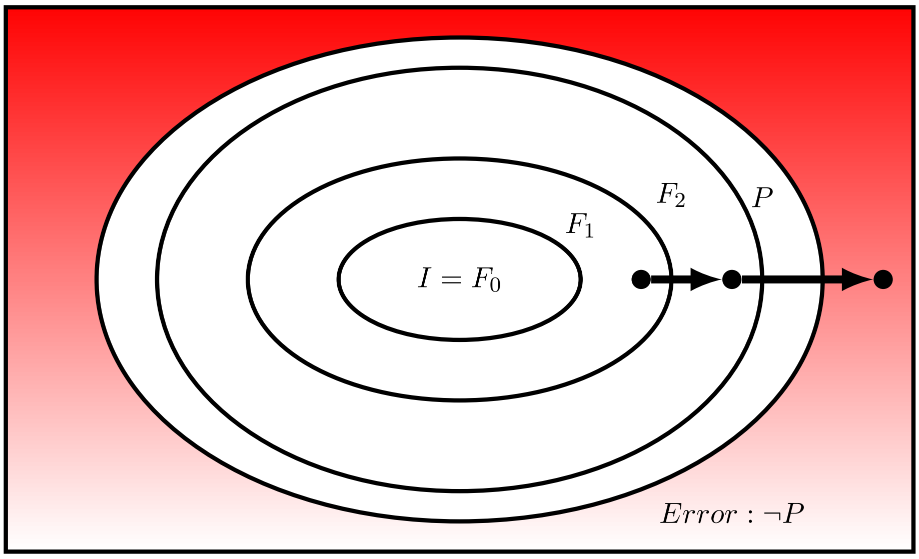
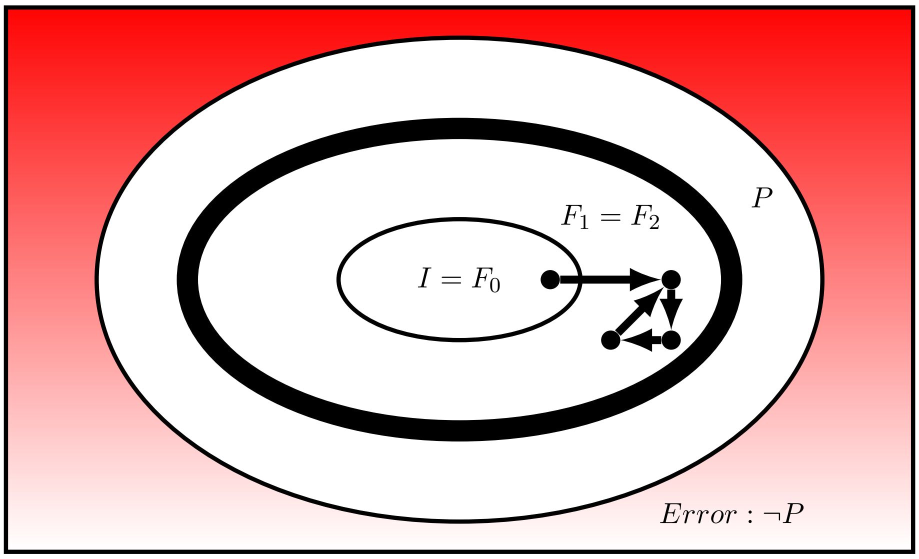
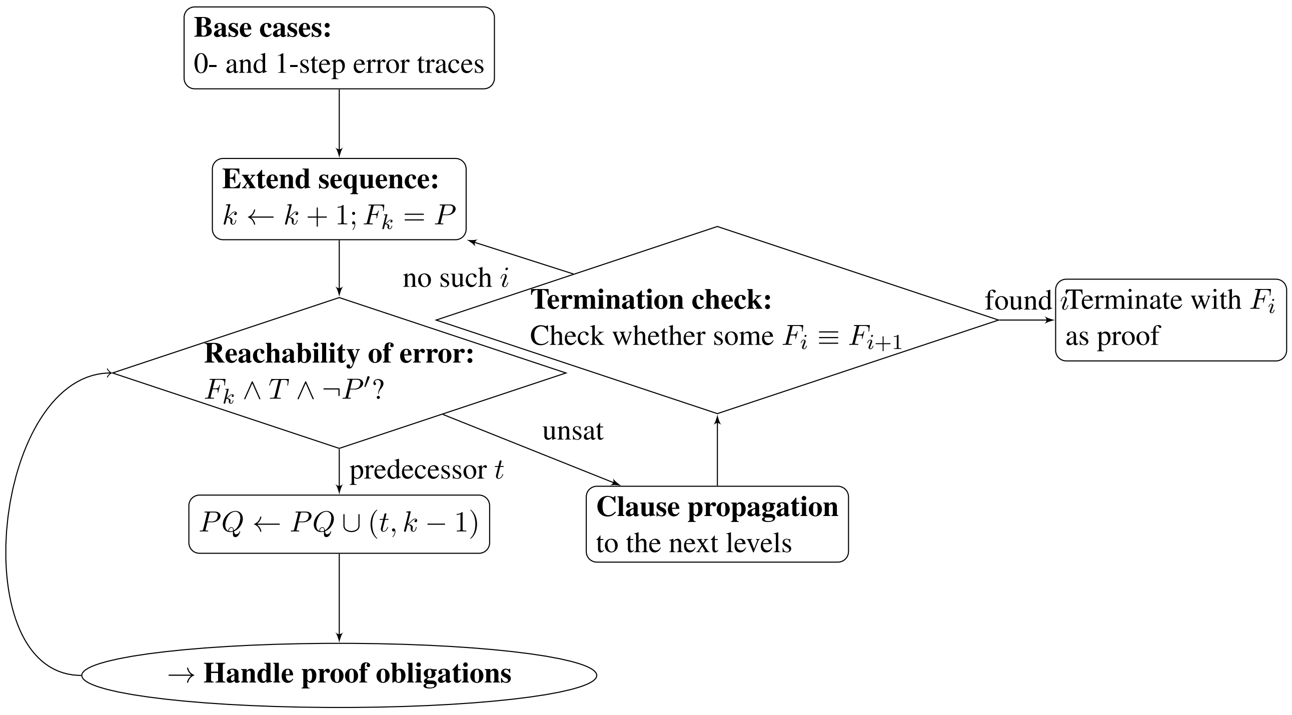


Figure 2.2: Termination: F_1 and F_2 have been sufficiently strengthened and a fixed point has been discovered. No path starting from the initial state set can ever leave the fixed point F_1 (with $F_1 = F_2$)



.

Figure 2.3: The general outline of the algorithm. PQ is the proof obligations priority queue.



Base cases: 0- and 1-step error traces
Extend sequence: $k \leftarrow k+1; F_k = P$
no such i
Termination check: Check whether some $F_i \equiv F_{i+1}$
found i
Terminate with F_i as proof
Reachability of error: $F_k \wedge T \wedge \neg P'$?
unsat
predecessor t
$PQ \leftarrow PQ \cup (t, k-1)$
Clause propagation to the next levels
$\rightarrow$ **Handle proof obligations**

c) Check whether the error (= the negation of the property) can be reached in one transition from an initial state ($I \wedge T \Rightarrow P'$); it yes, return **false** and a transition from the initial state to the error.

2. **Extending the sequence:**

   Set $k \leftarrow k+1$

3. Set $F_k = P$.

4. **Strengthening the frames (...making the over-approximations more precise):**

   a) Let there be a priority queue that stores pairs $\langle s, i \rangle$, where s is a state and i is a level. Such a pair is called a *proof obligation*. The priority queue operates in lowest-level-first output order.

   b) **Reachability of error:**

   Check whether it is possible to go from the frontier to the error in one step: $F_k \wedge T \Rightarrow \neg P'$.

   c) If the error cannot be reached from the frontier in one step, go to the **Clause propagation** phase (5).

   d) If the error is reachable from the frontier in one step, extract a state s in F_k that leads to the error. Put s into the priority queue at level $k-1$.

In order to expunge s from the level k, invariant **Inv4** must be shown to hold on the strengthened frames $F_k \wedge \neg s$ and $F_{k-1} \wedge \neg s$. This means checking whether $\neg s \wedge F_{k-1} \wedge T \Rightarrow \neg s'$, otherwise it is not certain that $F_{k-1} \wedge T \Rightarrow F'_k$. Putting s into the priority queue at level $k-1$ means that $\neg s$ will be checked for consecution relative to level $k-1$.

Checking F_{k-2} and all the lower frames for consecution is not necessary because s must already be excluded from the frames F_{k-1} downwards: s directly leads to the error, but F_{k-1} cannot lead to the error within one step, otherwise the frontier would not be at the current k. So s must already have been excluded from the frame F_{k-1} at an earlier point.

e) **Handle the proof obligations**, as described in the next subsection (2.2).

f) Return to step **Reachability of error** (4b).

5. **Clause propagation to the next frame:**

   For $0 \leq i < k$:

   a) For each clause c in F_i, check whether the clause will also hold in the next transition: $F_i \wedge T \Rightarrow c'$. If yes, add c to F_{i+1}.

6. **Checking for termination:**

   Check whether there is some F_i that contains exactly the same clauses as F_{i+1}. In such a case, the algorithm terminates with **true** and outputs the frame F_i as a proof of correctness. (Recall that F_i is an inductive invariant in this case.) If there is no such F_i, re-start at step **Extending the sequence** (2).

Handling Proof Obligations

A pseudo-code outline of this description is given in algorithm 1.

1. Peek the state-level pair with the lowest level from the priority queue. (There might be multiple states at the same lowest level; in this case, the algorithm may choose any of them.)

   Given a state-level pair $\langle s, i\rangle$ that was peeked from the queue, the algorithm tries to exclude the state s from all frames up to F_{i+1} by conjoining the clause $\neg s$, or a clause that implies $\neg s$, to the formulas $F_1, \ldots, F_{i+1}$. (It will become clear that the proof obligation is stored as $\langle s, i\rangle$ and not $\langle s, i+1\rangle$ because the algorithm checks for consecution of $\neg s$ relative to frame F_i.)

   Of course, whenever adding a clause to any frame via conjunction, heed must be taken in order to ensure that the invariants **Inv1**-**Inv4** still hold on the sequence of frames. We will now describe how the algorithm strengthens the frames while maintaining the four invariants.

   Why not just conjoin $\neg s$ to the level F_{i+1}, but also all the lower levels? Recall that invariant **Inv2** requires that $F_j \Rightarrow F_{j+1}$ for $0 \leq j < k$. Then why not the level F_0? Recall

Algorithm 1 Outline of handling proof obligations. I is the initial condition. T is the transition relation. F_i is the frame at level i.

```
 1: function VOID HANDLEOBLIGATIONS($PQ$: proof obligations priority queue)
 2:     while $PQ$.size > 0 do
 3:         $\langle s, i \rangle \leftarrow PQ$.PEEK                    ▷ $\langle s, i \rangle$ is the obligation with the lowest $i$
 4:         if $I \wedge s$ is satisfiable then
 5:             Terminate with trace from $s$ to $\neg P$
 6:         end if
 7:         if $\neg s \wedge F_i \wedge T \Rightarrow \neg s'$ then               ▷ $\neg s$ satisfies consecution at level $i$?
 8:             $PQ$.POP                       ▷ Eliminate proof obligation $\langle s, i \rangle$ from $PQ$
 9:             $c \leftarrow$ GENERALIZE($\neg s$)                 ▷ GENERALIZE might be CTGDOWN
10:             $j \leftarrow max\{j \mid \neg s \wedge F_j \wedge T \Rightarrow \neg s'\}$   ▷ Find max. $j$ relative to which $c$ satisfies
    consecution
11:             for $l \leftarrow 1 .. j + 1$ do
12:                 $F_l \leftarrow F_l \wedge c$
13:             end for
14:             if $i < j \leq k$ then
15:                 $PQ \leftarrow PQ \cup \langle s, j \rangle$                    ▷ Push proof obligation forward
16:             end if
17:         else
18:             SatModel $mod$ ←SATMODEL($\neg s \wedge F_i \wedge T \wedge s'$)
19:             Predecessor $t \leftarrow$ unprimed variable assignments from $mod$
20:             if $i = 0$ then
21:                 Terminate with trace from $t$ to $\neg P$
22:             end if
23:             $PQ \leftarrow PQ \cup \langle t, i - 1 \rangle$
24:         end if
25:     end while
26: end function
```

that $I = F_0$ and s is not an initial state; thus conjoining $\neg s$ to I would not change F_0 anyway. In fact, **Inv1** requires there must never be a strengthening of the initial states.

2. If $i! = 0$, skip this step. Otherwise, check whether the state s to exclude from level i is an initial state ($I \wedge s$ is satisfiable). If it is, we cannot exclude s from any level, as the invariant **Inv1** would be violated. However, in this case, there is a trace from the initial state s to the error. (This trace must exist, since s is an initial state and was discovered by walking backwards from the error. Storing the successor state along with a proof obligation ensures that the discovery of the trace is trivial.) In this case, terminate the algorithm with **false** and a trace from s to the error.

3. Assume that s is not an initial state. Then s must be excluded from all levels up to $i + 1$. To this end, it is necessary to add $\neg s$ to the frames $F_1, F_2, \ldots, F_{i+1}$. Doing so preserves the invariant **Inv2**. Since $F_i \Rightarrow P$ for all i in the first place, the invariant **Inv3** is preserved

by conjoining $\neg s$ to any frame. In order to observe the invariant **Inv4**, the clause $\neg s$ must satisfy consecution relative to the frame F_i: $\neg s \wedge F_i \wedge T \Rightarrow \neg s'$.

- If the clause $\neg s$ satisfies consecution relative to level F_i:
  - Obliterate $\langle s, i \rangle$ from the priority queue.
  - **Clause Generalization:**
    Try to find a more general clause c with $c \Rightarrow \neg s$ that satisfies initiation and consecution. This basically works by systematically trying to drop literals from $\neg s$ and checking whether the result still satisfies the two conditions.
  - Find the maximum j such that c still satisfies consecution relative to F_j. Then add the clause c to the frames $F_1, \ldots, F_{j+1}$. Why may c be added to level F_{j+1}? Because c is inductive at level j, meaning that $c \wedge F_j \wedge T \Rightarrow c'$. Adding c up to F_{j+1} preserves all the invariants, notably **Inv4**.
  - If $i < j \leq k$, add the proof obligation $\langle s, j \rangle$ to the priority queue. Why this? It is known that s is part of an error trace suffix. While s has been successfully excluded up to level $j + 1$, it is certain that s will turn up again at a higher k unless either the algorithm terminates first or it is taken care of at an earlier point. In this way, IC3 tries to find sets of mutually inductive clauses. Experiments validate that this is a good heuristic: It is better not to wager that the algorithm terminates before s turns up again, so it is beneficial to exclude s at the highest level currently possible.
- If the clause $\neg s$ does not satisfy consecution relative to level F_i:
  - Then there is a predecessor t that directly leads into s. (The consecution query $\neg s \wedge F_i \wedge T \Rightarrow \neg s'$ effectively asks for such an F_i-predecessor; if there is none, the clause $\neg s$ satisfies consecution.)
  - If $i = 0$, is is certain that t must be in F_0 and therefore is an initial state; that is, there is a trace from an initial state to the error. Terminate the algorithm with **false** and the counterexample trace.
  - Otherwise, add the proof obligation $\langle t, i - 1 \rangle$ to the priority queue: The algorithm aims to prove consecution for $\neg t$ at level $i - 1$, thus ensuring that t is purged from any level up to i. The argument is the same as in step 5.

4. If the priority queue is not empty, handle the next proof obligation (go to step 1). Otherwise, return to the main algorithm (see section 2.2).

Clause Generalization

One of the most important parts of IC3 is the generalization of states. Generalization in IC3 means: given a cube s whose associated clause $\neg s$ satisfies initiation and consecution at a certain level i, find a clause c with $c \Rightarrow \neg s$ that also satisfies initiation and consecution at the same level. Finding such a strong clause c is paramount for the performance of IC3: If only single states

s are excluded from specific levels, IC3 degenerates to explicit-state model checking, which is undesirable considering the state explosion problem.

The basic idea behind generalization is to take a cube q, drop a literal from the cube, obtaining a cube $\hat{q}$, and then see whether $\neg\hat{q}$ still satisfies initiation and consecution. If it does, the procedure can be repeated with $\hat{q}$. If $\neg\hat{q}$ does not satisfy initiation, then a different literal is dropped from q. If $\neg\hat{q}$ does not satisfy consecution, the situation gets complicated. The easiest thing to do is to try and drop a different literal. However, there is a number of tricks worth performing when a literal cannot to be dropped.

The process of dropping literals is continued until no more literals can be dropped such that consecution and initiation still hold. The result is a *minimal inductive subclause* that, in addition to excluding s from up to the level $i + 1$, excludes similar states that are not reachable in up to $i + 1$ steps either. The algorithm in which literals are systematically dropped and the resulting clauses are checked for consecution is called MIC.

The most sophisticated approach so far that deals with literals that cannot to be dropped is called CTGDOWN and is described in [30]. CTGDOWN accelerates IC3 markedly in comparison with applying only MIC and joining generalized cubes with counterexamples to generalization. Since the algorithm is such an important part of IC3, we will provide a short explanation of this approach to find a minimal inductive subclause. Pseudo-code describing the algorithm is given in listing 2. The algorithm handles cubes, the negation of which are clauses. Given a cube q, the algorithm finds a minimal inductive subclause of $\neg q$.

The function MIC drops a literal ℓ from the cube q, obtaining $\hat{q}$ (line 6) and then calls CTGDOWN to perform the necessary checks on $\hat{q}$ (line 7). If CTGDOWN returns $true$, MIC continues to drop literals from the shortened cube. Otherwise, it continues with the unabridged cube.

The function CTGDOWN that determines whether some cube is a proper generalization first checks whether the clause $\neg\hat{q}$ satisfies initiation (line 16). If it does not, the cube is not a proper generalization and MIC continues without dropping the literal ℓ. Otherwise, CTGDOWN checks whether the cube satisfies consecution at the given level i (line 19). If it does, the cube is a proper generalization and MIC continues with the shortened cube.

If the cube $\hat{q}$ does not satisfy consecution, a *counterexample to generalization* is extracted: This is a state s in the current frame F_i that is not in $\hat{q}$, but leads into $\hat{q}$ in one step. The existence of this state s impedes the dropping of literal ℓ from q at the level i.

CTGDOWN then continues to exclude s recursively at level i, generalizing s in the process if possible. Since trying to exclude s at level i might result in a large number of recursive calls, the recursion depth is bounded in CTGDOWN.

In the case that s (or a generalized version of s) is successfully excluded at level i, CTGDOWN re-iterates and checks again whether the cube $\hat{q}$ is a proper generalization. This might result in the next discovery of a counterexample to generalization. In order to preclude the handling of a disproportionate number of counterexamples to generalization, the number of such states to handle is bounded by maxCTGs.

In the case that s can not be excluded at level i, CTGDOWN *joins* the cube $\hat{q}$ with s. (This is written as $\hat{q} \leftarrow \hat{q} \sqcap s$ in the algorithm.) Joining $\hat{q}$ with s means that only the literals that occur both in $\hat{q}$ and $\hat{s}$ are kept in $\hat{q}$. This is necessary because it is known that any generalization of $\hat{q}$

Algorithm 2 MIC with ctgDown. The presentation closely follows [30].

```
function VOID MIC($q$: cube ref, $i$: level)
    MIC($q$, $i$, 1)
end function
function VOID MIC($q$: cube ref, $i$: level, $d$: recDepth)
    for each Literal $\ell$ in $q$ do
        $\hat{q} \leftarrow q \setminus \ell$
        if CTGDOWN($\hat{q}$, $i$, $d$) then
            $q \leftarrow \hat{q}$
        end if
    end for
end function
function BOOL CTGDOWN($\hat{q}$: cube ref, $i$: level, $d$: recDepth)
    $ctgs \leftarrow 0$
    $joins \leftarrow 0$
    while $true$ do
        if $I \not\Rightarrow \neg\hat{q}$ then
            return $false$
        end if
        if $F_i \wedge \neg\hat{q} \wedge T \Rightarrow \neg\hat{q}'$ then
            return $true$
        end if
        if $d >$ maxDepth then
            return $false$
        end if
        State $s \models F_i \wedge \neg\hat{q}$, $s$ leads into $\neg\hat{q}$
        if $(ctgs <$ maxCTGs$)$ and $(i > 0)$ and $(I \Rightarrow \neg s)$ and $F_{i-1} \wedge \neg s \wedge T \Rightarrow \neg s'$ then
            $ctgs \leftarrow ctgs + 1$
            for $j \leftarrow i$ to $k$ do
                if $F_j \wedge \neg s \wedge T \not\Rightarrow \neg s'$ then
                    break
                end if
            end for
            MIC($s, j - 1, d + 1$)
            for $l \leftarrow 1 .. j + 1$ do
                $F_l \leftarrow F_l \wedge \neg s$
            end for
        else if $joins <$ maxJoins then
            $ctgs \leftarrow 0$
            $joins \leftarrow joins + 1$
            $\hat{q} \leftarrow \hat{q} \sqcap s$
        else
            return $false$
        end if
    end while
end function
```

that does not include s will have s as a counterexample to generalization. The only chance to make $\hat{q}$ inductive without explicitly excluding s from level i is to enlarge $\hat{q}$ such that it includes s. In the next iteration, the enlarged $\hat{q}$ is checked for initiation and consecution. This continues until some enlarged $\hat{q}$ finally satisfies all the properties, in which case the enlarged $\hat{q}$ is a proper generalization, or it does not satisfy initiation anymore, in which case the literal ℓ cannot be dropped from q in the first place, and MIC continues with dropping a different literal.

The constants maxCTGs, maxDepth and maxJoins govern some of the behavior of CTG-DOWN. The algorithm never excludes more than maxCTGs counterexamples to generalization for a given clause $\hat{q}$, it does not handle counterexamples to generalization beyond a recursion depth of maxDepth and it does not join more than maxJoins CTGs with $\hat{q}$.

2.3 The Infinite State Case

Until now, we have described the IC3 algorithm for transition relations that reason over a finite state space, as it occurs in hardware verification. In this section, we will describe the prerequisites for extending IC3 to software model checking for transition relations over linear integer arithmetic. We will describe how to construct transition relations from software programs and will outline techniques needed for the adaptation of IC3 to software model checking.

Control Flow Graphs

A *control flow graph (CFG)* is a tuple $\langle L, G\rangle$, where L is a set of program counter values and $G \subseteq L \times C \times Ops \times L$ is a set of edges that lead from one program counter value to the next. Ops is a set of operations that modify the state of the program. C is a set of Boolean formulas called "guards". There is a special program counter value $pc_0 \in L$ that serves as the initial node of the CFG. The following formulas are assumed to hold on a CFG:

$$\bigwedge\{\bigwedge\{\neg c_1 \vee \neg c_2 \mid \langle l \times c_1 \times \_ \times l_1'\rangle \in G, \langle l \times c_2 \times \_ \times l_2'\rangle \in G, l_1' \neq l_2'\} \mid l \in L\}$$

This states that guards on edges emerging from the same node in the CFG must never be simultaneously true for any variable assignment.

$$\bigwedge\{\bigvee\{c \mid \langle l \times c \times \_ \times l'\rangle \in G\} \mid l \in L\}$$

This states that all the guards emerging from a single node in the CFG must be a valid formula if disjoined.

The formula that encodes the transition relation of a program is usually obtained by translating the labeled control flow graph of the program to a first-order formula modulo a theory such as linear integer arithmetic. The variables of the program are defined as first-order constants, with a primed and an unprimed version of each constant. There is a special constant that serves as the program counter.

User input is usually modeled as a non-deterministic assignment to a variable. In the transition relation, a non-deterministic assignment is obtained by eliminating all constraints on this variable, thereby granting the SMT solver liberty to assign any value to the variable in a satisfiability call. It is assumed that if a variable is set non-deterministically at a given program counter location, it does not occur in any guard expression at this program counter location.

Given a CFG, the symbolic transition relation $T(\vec{x}, \vec{x'})$ in first-order logic (where $\vec{x}$ is a vector of variables, including the program counter pc, and $\vec{x'}$ is the corresponding set of primed variables) is given by

$$\begin{aligned} T(\vec{x}, \vec{x'}) = \\ & \bigwedge \{((pc = l) \land c) \Rightarrow (pc' = l' \land \vec{x'} = op(\vec{x})) \mid l \in L, (l \times c \times op \times l') \in G\} \\ & \land pc \geq 0 \land pc \leq max(l) \end{aligned} \tag{2.1}$$

CFG Translation Example

As an example, consider the following program with two variables:

```
int a = 0;
int b = *;
while (a < b) ++a;
assert(a == 10);
```

Of course, the assertion at the end of the program may be violated, since b can take non-positive values. The labeled control flow graph for this program is given in figure 2.4.

This labeled control flow graph might be encoded in a first-order formula modulo the theory of linear integer arithmetic as follows:

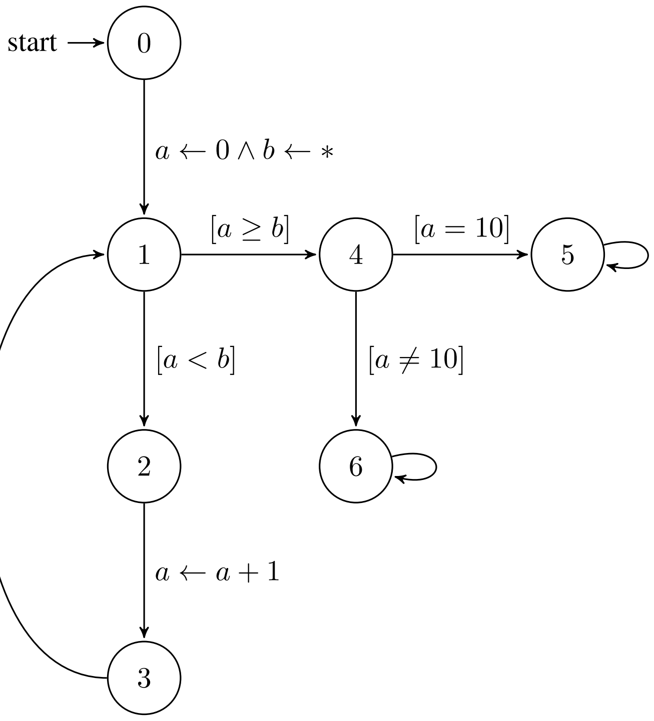
$$\begin{array}{llllllll}
(pc = 0 & & & \Rightarrow & pc' = 1 & \land & a' = 0) & \land \\
(pc = 1 & \land & a < b & \Rightarrow & pc' = 2 & \land & a' = a \land b' = b) & \land \\
(pc = 2 & & & \Rightarrow & pc' = 3 & \land & a' = a + 1 \land b' = b) & \land \\
(pc = 3 & & & \Rightarrow & pc' = 1 & \land & a' = a \land b' = b) & \land \\
(pc = 1 & \land & a \geq b & \Rightarrow & pc' = 4 & \land & a' = a \land b' = b) & \land \\
(pc = 4 & \land & a = 10 & \Rightarrow & pc' = 5 & \land & a' = a \land b' = b) & \land \\
(pc = 4 & \land & a \neq 10 & \Rightarrow & pc' = 6 & \land & a' = a \land b' = b) & \land \\
(pc = 5 & & & \Rightarrow & pc' = 5 & \land & a' = a \land b' = b) & \land \\
(pc = 6 & & & \Rightarrow & pc' = 6 & \land & a' = a \land b' = b) & \land \\
(pc \geq 0 & \land & pc \leq 6) & & & & &
\end{array}$$

Note that the non-deterministic assignment to b is modeled by removing any b' constraints from the transition from $pc = 0$ to $pc = 1$.

The initial condition is modeled as $pc = 0$. The property to prove is modeled as $pc \neq 6$, stating that there must not be any path from the initial state, where $pc = 0$, to a state where $pc = 6$.

For this thesis, we wrote a compiler that translates from a simple C-like languages to first-order formulas modulo linear integer arithmetic similar to the one given above.

Figure 2.4: Labeled control flow graph for an example program. The numbers in the circles are the values of the program counter. Note that in transitions where assignments to variables are missing, the variable is assumed to keep the value it had in the last transition.



Lattices

The concept of lattices will be needed in the context of abstraction, so we introduce it here.

Definition 13 (Partially Ordered Set) *A* partially ordered set $(L, \sqsubseteq)$, *or* poset, *consists of a set L together with a binary relation $\sqsubseteq$. The following axioms hold for the binary relation $\sqsubseteq$:*

- $\forall a \in L : a \sqsubseteq a$
- $\forall a, b \in L :$ *If* $(a \sqsubseteq b)$ *and* $(b \sqsubseteq a)$, *then* $(a = b)$
- $\forall a, b, c \in L :$ *If* $(a \sqsubseteq b)$ *and* $(b \sqsubseteq c)$, *then* $(a \sqsubseteq c)$

Definition 14 (Supremum, Infimum) *Let $(L, \sqsubseteq)$ be a partially ordered set and S is an arbitrary subset of L, then:*

- *An element u of L is an upper bound of S if $s \sqsubseteq u$ for each $s \in S$.*
- *An upper bound u of S is the* supremum *of S if $u \sqsubseteq x$ for each upper bound x of S.*
- *An element l of L is a lower bound of S if $l \sqsubseteq s$ for each $s \in S$.*

- *A lower bound l of S is said to be its* infimum *if $x \sqsubseteq l$ for each lower bound x of S.*

Definition 15 (Lattice) *A* lattice *is a partially ordered set in which every two elements have a supremum and an infimum.*

Any set of Boolean formulas $\mathbb{P}$ together with the logical implication operator $\Rightarrow$ forms a lattice: $\sqsubseteq$ is given by

$$\forall a, b \in \mathbb{P} : (a \sqsubseteq b) \text{ iff } (a \Rightarrow b).$$

Abstraction

The description of abstraction in this section is based on [5]. In the infinite state case, logical formulas are given as SMT formulas rather than propositional logic formulas. An SMT (**S**atisfiability **M**odulo **T**heory) formula is a formula in a specific fragment of first-order logic. The model checker described in this thesis works with formulas over linear integer arithmetic. There is a number of SMT solvers that can decide satisfiability queries on SMT formulas.

Unfortunately, the word "predicate" has a different connotation in the context of abstraction than in the context of first-order logic vocabulary. In the area of abstraction, a predicate is simply a Boolean formula that may be built from many predicates in the first-order sense. Since the use of "predicate" for an arbitrary Boolean formula is so widespread, we adopt the ambiguous usage. Henceforth, the word "predicate" will denote an arbitrary Boolean formula in the context of abstraction.

Sets of states are usually described by predicates. Assume that $\mathbb{P}$ is a set of predicates. If a state satisfies a predicate, it is said to be *in* the predicate.

The set $\mathbb{P}$ divides the state space into equivalence classes. In a domain of size n, an equivalence class can be described by a bit-vector $\overline{s}$ of length n. Such a bit-vector is called an *abstract state*.

The *abstract domain* is the set of all abstract states for a given set of predicates $\mathbb{P}$:

$$\mathbb{D} = 2^{(\{0,1\}^n)}.$$

There is a partial ordering on the abstract domain given by subset inclusion.

The function $\alpha : 2^S \mapsto 2^{\mathbb{D}}$ that maps from the *concrete domain* (the set of sets of states) to the abstract domain is called the *abstraction function*. It is given by

$$Q \mapsto \{\langle v_1, \ldots, v_n \rangle \mid Q \cap \{s \mid s \models v_1\wp_1 \wedge \cdots \wedge v_n\wp_n\} \neq \emptyset\},$$

where $0\wp_i = \neg\wp_i$ and $1\wp_i = \wp_i$.

The function $\gamma : 2^{\mathbb{D}} \mapsto 2^S$ that performs the reverse operation is called the *meaning function* and is given by

$$V \mapsto \{s \mid \exists\langle v_1, \ldots, v_n \rangle \in V : s \models v_1\wp_1 \wedge \cdots \wedge v_n\wp_n\}.$$

Definition 16 (Most Precise Abstraction) *Given a set of predicates $\mathbb{P}$, the* most precise abstraction *of a formula ϕ is a formula ψ built of predicates in $\mathbb{P}$ such that $\phi \Rightarrow \psi$ and there is no formula ψ_2 built of predicates in $\mathbb{P}$ such that $\psi_2 \not\equiv \psi$ and $\phi \Rightarrow \psi_2 \wedge \psi_2 \Rightarrow \psi$.*

An abstract domain may admit *spurious transitions* between two abstract states. In the scope of this thesis, a spurious transition is defined as follows: There is a concrete state s that has no predecessor inside a particular frame F_i, but the most precise abstraction $\overline{s}$ does have a predecessor inside the frame F_i.

As will be described in the next chapters, it is sometimes necessary to add predicates to the set $\mathbb{P}$, in order to eliminate spurious transitions. This process is called *refinement*. After refinement, there will be concrete states that formerly were in the same abstract states, but then are in different abstract states.

CFG Translation Example contd.

The state of a program is a vector of variable assignments. A program with the variables a and b might be in the state (or *concrete state*, as opposed to abstract state) $s : (pc = 2|a = 5|b = 11)$. Following the definition 6, this state is equivalent to the first-order formula $pc = 2 \wedge a = 5 \wedge b = 11$.

Since a and b are integers, it is clear that the program can be in infinitely many concrete states. (The values the program counter can take are usually bounded.) In order to apply an IC3 derivative to a software program described in SMT terms, it is necessary to reduce the state space to a finite size. We do this by Boolean abstraction as described below.

Analogously to the notation in the finite state case, we will use the notation $(x_1 x_2 \dots x_n)_a$, where $x_1, \dots, x_n \in \{0, 1\}$ to describe an abstract state. $x_i = 0$ means that the i^{th} predicate occurs negated in the conjunction, and x_i means that the i^{th} predicate occurs in its plain form in the conjunction. An abstract state comprises multiple, and up to infinitely many, concrete states.

Take a program with the two variables a and b. Let the set of predicates of the abstract domain be $\mathbb{P} = \{\wp_1 : a < 4, \wp_2 : b < 5\}$ and let the abstract domain be $\mathbb{D} = 2^{\mathbb{P}}$. The abstract state $(10)_a$ comprises the set of states that are models of the formula $(a < 4) \wedge \neg(b < 5)$. The state $s_1 : (pc = 0|a = 0|b = 7)$ is in $\wp_1$ but not in $\wp_2$, so $s_1 \models (10)_a$. The same holds for state $s_2 : (pc = 0|a = 1|b = 7)$. Thus, as long as the abstract domain stays this way, the algorithm will treat these two states in the same way.

CHAPTER 3

Technical Contributions & Implementation

This section describes our implementation of the IC3 algorithm for software verification and details various optimizations and improvements that make the algorithm run decidedly faster. We will call the algorithm for software verification **IC3-CEGAR** from now on.

3.1 Description of IC3 for Software Verification by Counterexample-Guided Abstraction Refinement

In the last chapter, we described the outline of IC3 for propositional systems. In this section, we will explain how to adapt the algorithm to the verification of systems over linear integer arithmetic SMT formulas.

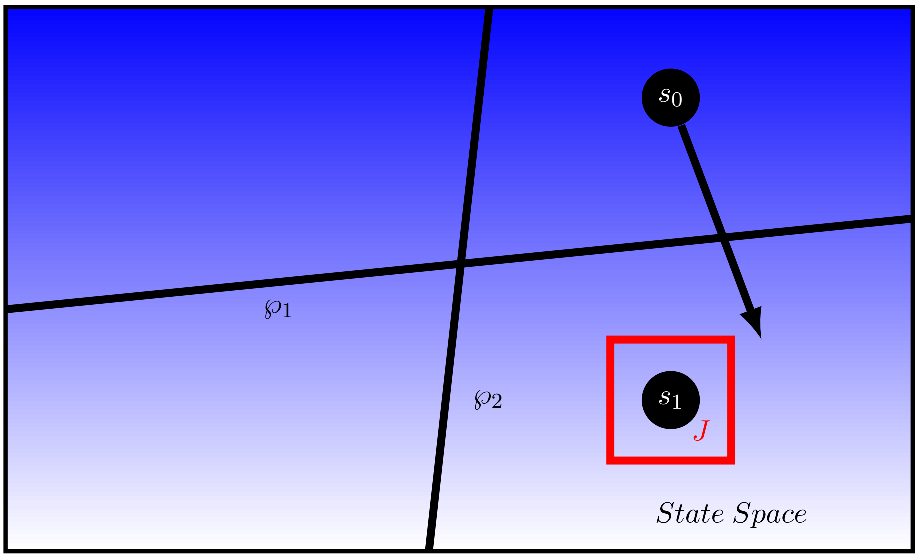
Some basic adaptations to the algorithm are:

- An SMT solver will be used instead of a SAT solver.
- Concrete states need to be stored as vectors of variable assignments (or interpretations) instead of bit vectors.
- Abstract states are stored as vectors of abstract domain predicates. (In practice, abstract states are stored as vectors of indices to abstract domain predicates. If an index is negated, the predicate is assumed to be negated in the abstract state. This representation allows for fast computation and straightforward refinement of the abstract domain.)

The framework around the algorithm that is necessary for handling the SMT solver instances, the different concrete and abstract states etc., expands quite significantly due to the overhead of dealing with a more complicated logic. The basic algorithm, however, does not need to be modified dramatically. The most important changes occur in the handling of proof obligations.

Figure 3.1 outlines some of the features used in IC3-CEGAR.

Figure 3.1: Infinitely large concrete state space. Two predicates ($\wp_1, \wp_2$) splitting the state space, thus obtaining an abstract state space. Two concrete states s_0 and s_1. Spurious transition from the abstract state $\overline{s_0}$ to the abstract state $\overline{s_1}$. Craig interpolant J for refinement.



The Data Structures Involved

Assume that any data structures whose description is not updated in this section remains unchanged in IC3-CEGAR.

The Sequence of Frames

As in the finite state case, IC3-CEGAR maintains a sequence of formulas $F_0, F_1, \ldots, F_k$ that describe over-approximations of states reachable in up to $0, 1, \ldots, k$ steps. These formulas are still basically in conjunctive normal form; this means that they are a conjunction of clauses. A clause in IC3-CEGAR is a negation of an abstract state.

Since an abstract domain predicate can be any SMT formula, no F_i is guaranteed to be in CNF insofar as the abstract domain predicates can be an arbitrary Boolean formula. However, the internals of a predicate are not of interest to the algorithm. If the abstract domain predicates' internal structure is neglected, each frame F_i is in CNF.

Proof Obligations

In the simplest version of IC3-CEGAR, a proof obligation is, again, a tuple $\langle s, i\rangle$, where s is a concrete state and i is a level.

The Abstract Domain

The predicates of the abstract domain are stored as follows:

- There is a vector that holds all predicates of the domain as SMT terms. This vector allows for easy indexing of predicates and easy refinement: New predicates simply get pushed to the back of the vector after refinement.
- Additionally, there is a lattice data structure that stores predicates. Any set of logical formulas can be sorted in a lattice with $\sqsubseteq$ being defined as the logical implication $\Rightarrow$ (cf. section 2.3).

  The lattice data structure represents this mathematical concept in memory. Obviously, re-shuffling the lattice data structure becomes necessary when predicates are added to the domain. The data structure allows for easy discovery of redundant predicates (which are then skipped), as well as for fast discovery of stronger and weaker predicates.

The predicate lattice supports the following three operations:

- The operation PARENTS takes a predicate $\wp$ as argument and returns all abstract domain predicates that are implied by $\wp$ and not implied by some predicate that is implied by $\wp$.
- The operation CHILDREN takes a predicate $\wp$ as argument and returns all abstract domain predicates that imply $\wp$ and do not imply some predicate that implies $\wp$.
- The operation ADD takes a predicate as argument and adds it to the predicate lattice, adjusting pointers where necessary.

Handling Proof Obligations

The basic outline of the algorithm as given in figure 2.3 stays the same. The main changes to the algorithm occur during the handling of proof obligations. The handling of proof obligations in IC3-CEGAR are described in this section.

A pseudo-code outline of this description is given in algorithm 3.

1. Peek the state-level pair with the lowest level from the priority queue.

   Given a state-level pair $\langle s, i \rangle$ that was peeked from the queue, the algorithm will now try to exclude the state s from all frames up to F_{i+1} by conjoining the clause $\neg s$, or a clause that implies $\neg s$, to the formulas $F_1, \ldots, F_{i+1}$.
2. If $i \neq 0$, skip this step. Otherwise, check if the state s to exclude from level i is an initial state. In this case, terminate the algorithm with **false** and a trace from s to the error.
3. Assume that s is not an initial state. Then s must be excluded from all levels up to $i + 1$.
4. Compute the most precise abstraction $\overline{\overline{s}} = \alpha(s)$. α is the abstraction function which, given a set of Boolean formulas, computes the most precise formula of a set of concrete states by returning a formula in disjunctive normal form whose literals are the given Boolean formulas. A more detailed explanation about the implementation of the abstraction function is given in the following section “Abstraction of States” (3.3). Therefore, let $\overline{\overline{s}} = \overline{s_0} \vee \overline{s_1} \vee \cdots \vee \overline{s_y}$. (It will become clear in the following section “Lifting of states”

Algorithm 3 Outline of handling proof obligations in IC3-CEGAR. I is the initial condition. T is the transition relation. F_i is the frame at level i.

```
 1: function VOID HANDLEOBLIGATIONS($PQ$: proof obligations priority queue)
 2:     while $PQ$.size > 0 do
 3:         $\langle s, i \rangle \leftarrow PQ$.PEEK                    ▷ $\langle s, i \rangle$ is the obligation with the lowest $i$
 4:         if $i = 0$ and $I \wedge s$ is satisfiable then
 5:             Terminate with trace from $s$ to $\neg P$
 6:         end if
 7:         $\overline{s} = \overline{s_0} \vee \cdots \vee \overline{s_y} \leftarrow \alpha(s)$
 8:         set<clause ref> $generalizations$
 9:         level $minLevel \leftarrow k + 1$
10:         for each $x \leftarrow 0 .. y$ do
11:             if $\exists c \in generalizations$ with $\overline{s_x} \Rightarrow c$ then
12:                 continue
13:             end if
14:             if $\neg\overline{s_x} \wedge F_i \wedge T \Rightarrow \neg\overline{s_x}'$ then          ▷ $\neg\overline{s_x}$ satisfies consecution at level $i$?
15:                 $c \leftarrow$ GENERALIZE($\neg\overline{s_x}$)              ▷ GENERALIZE might be CTGDOWN
16:                 $j \leftarrow max\{j \mid \neg\overline{s_x} \wedge F_j \wedge T \Rightarrow \neg\overline{s_x}'\}$       ▷ Find max. $j$ relative to which $c$
    satisfies consecution
17:                 for $l \leftarrow 1 .. j + 1$ do
18:                     $F_l \leftarrow F_l \wedge c$
19:                 end for
20:                 $minLevel \leftarrow min(j, minLevel)$
21:             else if $\neg s \wedge F_i \wedge T \Rightarrow \neg s$ then  ▷ There is a spurious transition from $s$ at level $i$
22:                 REFINEABSTRACTDOMAIN($\langle s, i \rangle$)
23:                 break
24:             else
25:                 SatModel $mod \leftarrow$SATMODEL($\neg s \wedge F_i \wedge T \wedge s'$)
26:                 Predecessor $t \leftarrow$ unprimed variable assignments from $mod$
27:                 if $i = 0$ then
28:                     Terminate with trace from $t$ to $\neg P$
29:                 end if
30:                 $PQ \leftarrow PQ \cup \langle t, i - 1 \rangle$
31:                 break
32:             end if
33:         end for
34:         if all $\overline{s_x}$ were successfully eliminated then
35:             $PQ$.POP                         ▷ Eliminate proof obligation $\langle s, i \rangle$ from $PQ$
36:             if $i < minLevel \leq k$ then
37:                 $PQ \leftarrow PQ \cup \langle s, minLevel \rangle$                ▷ Push proof obligation forward
38:             end if
39:         end if
40:     end while
41: end function
```

(3.2)) that the proof obligation s may describe multiple concrete states. Thus, the most precise abstraction of s is a disjunction of abstract states.)

5. For each of the abstract states $\overline{s_x}$, loop over the the following:

   - If $\neg\overline{s_x}$ implies a clause c that was excluded in a previous loop iteration, continue the loop with $\overline{s_{x+1}}$.
   - It is known that $\overline{s_x}$ does not overlap with the initial condition, since it is part of an abstraction of a concrete state s that is not an initial state. (Recall that the abstract domain predicates always contain the initial condition in order to ensure that the most precise abstraction of any concrete non-initial state does not overlap with the initial condition.)
   - If the clause $\neg\overline{s_x}$ satisfies consecution relative to level F_i:
     - **Clause Generalization:** Try to find a more general clause c with $c \Rightarrow \neg\overline{s_x}$ that still satisfies initiation and consecution.
     - Find the maximum j such that c still satisfies consecution relative to F_j. Then add the clause c to the frames $F_1, \ldots, F_{j+1}$.
   - If the clause $\neg\overline{s_x}$ does not satisfy consecution relative to level F_i:
     - Check if the negation of the concrete state s satisfies consecution at level i. If it does not satisfy consecution either, then there is a concrete predecessor t that directly leads into s. If $i = 0$, then t must be an initial state, so terminate with a trace from t to the error. Otherwise, add the proof obligation $\langle t, i-1\rangle$ to the priority queue. Then break the $\overline{s_x}$ loop, as it is known that an obligation at a lower level exists and must be handled first.
     - If $\neg s$ satisfies consecution at level i, but the abstract clause $\neg\overline{s_x}$ does not, then the abstract domain is too coarse: There is a concrete state in F_i that leads into $\overline{s_x}$, but there is no concrete predecessor of s in F_i. Thus, the abstract domain admits a spurious transition and abstract domain refinement is triggered at this point.
       Because refinement of the abstract domain is triggered when a *counter-example to induction* occurs, this refinement strategy is a variation on the well-known **C**ounter**e**xample-**G**uided **A**bstraction **R**efinement ([18, 19]).
       A number of refinement strategies are discussed below. After refinement, break the $\overline{s_x}$ loop, as the most precise abstraction of s has changed.

6. If the whole abstraction of s, i.e., all the abstract states $\overline{s_x}$, were successfully excluded up to level $i+1$, obliterate $\langle s, i\rangle$ from the priority queue. In this case, and if $i < j \leq k$, add the proof obligation $\langle s, j\rangle$ to the priority queue, where j is the minimum level among all the levels to which generalizations of any $\overline{s_x}$ could be pushed.

7. If the priority queue is not empty, handle the next proof obligation. Otherwise, return to the main algorithm.

3.2 Lifting of States

Lifting states in finite-state transition systems in a propositional symbolic encoding was described in [13, 24] and applied to IC3 for hardware verification. We describe how to adapt the approach to IC3-CEGAR.

Algorithm 3 shows how the algorithm extracts a predecessor t and adds it to the list of proof obligations. So far, we have assumed that t is a single concrete state: t is a vector of assignments to all variables that in the program (including the program counter). Such a vector of assignments describes exactly one state of the program.

However, consider that t is extracted for its feature of being a state that leads into its successor s in one step. There might be a lot of states very similar to t that also lead into s. Consider the following program:

```
   int x, y;
1: if (x > 0)
2:   ++x;
   else
3:   assert(false);
```

...where the numbers `1`, `2` and `3` are program counter locations. Assume that IC3-CEGAR checks whether it is possible to reach the error $\neg P : pc = 3$ in one step — it is: Amongst others, the state $t : pc = 1, x = -1, y = 3$ leads to $\neg P$ in one step.

Observe that the error can be reached in one step from t regardless of the value of y. Instead of adding t to the proof obligations queue, thus taking care of the specific case where $pc = 1, x = -1, y = 3$, the *lifted state* $pc = 1, x = -1$ should rather be added to the proof obligation queue. This would ensure that any state with $pc = 1, x = -1$ is excluded from the current frame, regardless of the value of y.

The process of *lifting* drops variable assignments from a concrete state t in such a way that the resulting *lifted state* $\hat{t}$ still leads into the successor state s. Lifting a state works as follows:

1. Let ϕ be a formula describing either the error $\neg P$ (in case the algorithm checks for the reachability of the error) or a given successor state s.

2. Let t be the concrete predecessor to be lifted. It is known that t leads into ϕ in exactly one transition.

3. Let T be the transition relation, symbolically encoded as an SMT formula.

4. Perform the query $t \wedge T \wedge \neg\phi'$ on an SMT solver. Since the single concrete state t is known to lead into ϕ, this query must always be unsatisfiable.

5. Extract the unsatisfiable core from the last call. From the unsatisfiable core, take all unprimed variable assignments and put them into a vector $\hat{t}$ of variable assignments. These unprimed variable assignments are parts of the concrete state t. Since the unsatisfiable core contains a subset of the input predicates of a call that are sufficient for unsatisfiability, the call $\hat{t} \wedge T \wedge \neg\phi'$ is unsatisfiable.

6. All states described by $\hat{t}$ lead into ϕ in exactly one step, as evidenced by the fact that $\hat{t} \wedge T \Rightarrow \phi'$ is valid (which, in turn, follows from the unsatisfiability of the above query).

$\hat{t}$ is a lifted state of the program: a lifted state is a vector of variable assignments just like an ordinary concrete state. But in contrast to a single concrete state, a lifted state may describe up to an infinite number of single concrete states.

Since all concrete states encompassed by $\hat{t}$ lead into either the error $\neg P$ or the successor state s, all of them must be excluded from the level $i - 1$ in order exclude s or the error at level i. Thus, instead of adding the possibly very specific concrete state t at level $i - 1$ to the priority queue PQ, the lifted state $\hat{t}$ is added as an obligation at level $i - 1$.

The most precise abstraction of a single concrete state is a single abstract state. However, the most precise abstraction of a lifted state, which may describe infinitely many states, is a disjunction of abstract states. This explains why the handling of proof obligations as given in listing 3 must handle multiple abstract states for a single proof obligation.

Using lifted states instead of concrete states in the priority queue accelerates the algorithm markedly. In addition, the produced counterexample traces are also easier to parse for humans. Consider the following program, where $\star$ denotes a non-deterministic assignment to a variable:

```
     int main() {
       int m, n, i, j, k;
0:     m = *;
1:     n = *;
2:     i = 0;
3:     if (n <= m) {
4:       return -1;
       }
5:     while (m < n) {
6:       ++m;
7:       ++i;
       }
9:     assert(m == (n + 1));
}
```

A lifted trace to the program, as generated by IC3-CEGAR, might looks like this (modified slightly from the raw output for clarity of presentation):

```
.pc=0
.pc=1, main.m=-1
.pc=2, main.m=-1, main.n=0
.pc=3, main.m=-1, main.n=0
.pc=5, main.m=-1, main.n=0
.pc=6, main.m=-1, main.n=0
.pc=7, main.m=0,  main.n=0
.pc=5, main.m=0,  main.n=0
.pc=9, main.m=0,  main.n=0
```

In this trace, there values of the variables i, j and k are neglected, as they are not essential for reaching the error.

Contrast this with the following non-lifted trace:

```
.pc=0, main.i=0, main.j=15, main.k=16, main.m=-3, main.n=0
.pc=1, main.i=0, main.j=15, main.k=16, main.m=-1, main.n=0
.pc=2, main.i=0, main.j=15, main.k=16, main.m=-1, main.n=0
.pc=3, main.i=0, main.j=15, main.k=16, main.m=-1, main.n=0
.pc=5, main.i=0, main.j=15, main.k=16, main.m=-1, main.n=0
.pc=6, main.i=0, main.j=15, main.k=16, main.m=-1, main.n=0
.pc=7, main.i=0, main.j=15, main.k=16, main.m=0,  main.n=0
.pc=5, main.i=1, main.j=15, main.k=16, main.m=0,  main.n=0
.pc=9, main.i=1, main.j=15, main.k=16, main.m=0,  main.n=0
```

The situation becomes worse when there are more variables in an incorrect program.

Lifting Non-Deterministic Assignments

Lifting a state over a transition system in which a non-deterministic assignment to variables occur poses a problem: IC3 checks whether it is possible to reach s in one step. If it is, a predecessor t is extracted. Now a lifted version of t should be added to the priority queue at level $i - 1$. In order to lift, the query $t \wedge T \wedge \neg s'$ is posed. This query is not unsatisfiable, for a state that is in $\neg s$ can also be reached from a state similar to t that contains a different value for the variable that was non-deterministically assigned.

In order to lift a state t at a program counter location where non-deterministic assignments to variables occur, it is necessary to store which variables are non-deterministically assigned at t's program counter value. Before lifting the predecessor t, all non-deterministic assignments must be eliminated from the successor s before performing the lifting query. The resulting lifted state $\hat{t}$ may still (non-deterministically) lead into s in exactly one transition, and therefore needs to be excluded.

In addition, since it is known that the values of non-deterministically assigned variables in t must be irrelevant, these variable assignments can also be dropped from t before performing the lifting query. (Recall that variables which are non-deterministically assigned at a given program counter location must not be used in the Boolean guard expressions at this program counter location.) The query must still stay unsatisfiable and yield a lifted predecessor $\hat{t}$ that must be excluded because it may lead into the successor s in one transition.

To illustrate this with an example, let the current proof obligation $s : pc = 5, a = 3, b = 0$ at level i.

```
4: a = *;
5: ...;
```

So in order to lift t it has to be known that at $pc = 4$, a non-deterministic assignment to a occurs. Then it poses the modified query $(pc = 4, b = 0) \wedge T \wedge \neg(pc = 5, b = 0)$. The unsatisfiable core of this query reveals a lifting $\hat{t}$ that may lead into s.

3.3 Abstraction of States

The method we use for abstracting lifted states is based on [33]. Because of the large size of the abstract domain, which often includes hundreds of predicates, and because of abstract state generalization, we modified the approach.

Given a set of abstract domain predicates $\mathbb{P}$, a *Boolean formula* over $\mathbb{P}$ is either a predicate of $\mathbb{P}$ or a conjunction, disjunction or negation of Boolean formulas over $\mathbb{P}$.

Given a ground formula ϕ, the approach described in [33] computes the strongest Boolean formula $\alpha_{\mathbb{P}}(\phi)$ over the set of predicates $\mathbb{P}$ for which $\phi \Rightarrow \alpha_{\mathbb{P}}(\phi)$. The authors show that

$$\alpha_{\mathbb{P}}(\phi) \equiv \bigvee \{\overline{s} \mid \overline{s} \text{ is an abstract state over } \mathbb{P} \text{ and } \overline{s} \wedge \phi \text{ is satisfiable.}\}$$

(Recall that an abstract state is a conjunction of all predicates in $\mathbb{P}$, where each predicate can occur in either plain or negated form.) Thus, $\alpha_{\mathbb{P}}(\phi)$ computes the most precise abstraction of ϕ.

The authors of [33] continue to show how this most precise abstraction $\alpha_{\mathbb{P}}(\phi)$ can be constructed from an AllSAT call given the set of predicates $\mathbb{P}$ and the formula ϕ:

For each predicate $\wp_i \in \mathbb{P}$, introduce a Boolean constant b_i. Construct the formula ψ as

$$\phi \wedge \bigwedge \{b_i \Leftrightarrow \wp_i \mid \wp_i \in \mathbb{P}\}$$

and feed ψ into an AllSAT call with the Boolean constants b_i being the important atoms. $\alpha_{\mathbb{P}}(\phi)$ is constructed from the AllSAT-models by converting each model into an abstract state and disjuncting these abstract states:

$$\alpha_{\mathbb{P}}(\phi) \equiv \bigvee \{\overline{s} \mid \text{Given a model over all } b_i, \overline{s} = \bigwedge \left\{ \begin{array}{ll} \wp_i & \text{if } b_i = true \\ \neg\wp_i & \text{if } b_i = false \end{array} \right\}\}.$$

Implementation

Our implementation of the abstraction function builds on the shown principle.

The Predicates Chosen for Abstraction

Recall that most states to be abstracted are lifted, that is, they are a vector of assignments to a subset of the program's variables. The current abstract domain will likely contain a number of predicates that do not contain any of the lifted state's variables. Including such predicates in abstraction would lead to an exponential blowup in the number of abstract states per abstraction: If there are n predicates in the abstract domain, of which m predicates do not contain any of the lifted state's variables, then each of these m predicates must occur in both negated and plain form for each abstract state over the $n - m$ other predicates. This means an increase by a factor of 2^m in the number of models per AllSAT call.

In order to avoid this exponential blowup, we do not feed all predicates of the current abstract domain into the abstraction function α as shown above, but only such predicates that contain at least one variable that is set by the lifted state.

Aborting the Abstraction

A second important point about the implementation is that computing all models for a given AllSAT call is often unnecessary. Abstracting lifted states is currently needed when handling proof obligations and in CTGDOWN. In both cases, the loop that iterates over the abstract states is sometimes terminated prematurely. In such cases, computing all the abstract states a priori is a waste of resources.

Aborting an AllSAT call prematurely is supported by some SMT solvers: Instead of computing all the models for a given formula and returning them, SMT solvers may provide a hook for a callback function that is called each time a model is computed. Depending on the return value of the callback function, the SMT solver continues to compute models or aborts. In our current implementation, none of the loops over the abstract states in HANDLEOBLIGATIONS and CTGDOWN is implemented exactly as shown; instead, the loop body that iterates over the $\overline{s_x}$ is passed as a callback to the AllSAT solver such that the AllSAT call may be aborted as soon as possible.

An additional optimization that allows the abortion of the AllSAT call while handling proof obligations: When performing abstract state generalization, the current proof obligation might be completely excluded by a generalized clause before all the models of the AllSAT call have been generated. In this case it is also correct to abort the AllSAT call. A separate solver checks whether the current proof obligation has been covered by the generalized clauses so far. If this is the case, the AllSAT call is aborted.

3.4 Refinement Strategies

The previous section clarified at which point in the algorithm the abstract domain was refined. We will now show three possible ways of refining the abstract domain in order to eliminate spurious transitions that obstruct the elimination of states from frames.

Refinement by Interpolation

The first refinement technique is based on Craig Interpolation ([20,21]).

Definition 17 *An interpolant for a valid first-order formula $A \Rightarrow B$ is a first-order formula J such that $A \Rightarrow J$, $J \Rightarrow B$ and and the vocabulary of J contains only symbols that are both in the vocabulary of A and in the vocabulary of B.*

In order to use Craig interpolation for the refinement of the domain, consider the following: There is a state s in a frame F_i for which $\neg s \wedge F_i \wedge T \Rightarrow \neg s'$, but $\neg \overline{s} \wedge F_i \wedge T \not\Rightarrow \neg \overline{s}'$.

Divide the formula $\neg s \wedge F_i \wedge T \Rightarrow \neg s'$ into the two parts

$$A : \neg s \wedge F_i \wedge T$$

and

$$B : \neg s'.$$

Since $A \Rightarrow B$, there is a Craig interpolant J' with $A \Rightarrow J'$ and $J' \Rightarrow B$. (We call the interpolant J' because it contains only primed atoms, since the B side contains only primed atoms.)

Since $J' \Rightarrow B$, it holds that $J' \Rightarrow \neg s'$. The unprimed converse of that is $s \Rightarrow \neg J$. The abstract domain is refined by $\neg J$. Also, we know that $s \Rightarrow \overline{s}$. Therefore, $\overline{s} \wedge \neg J$ is an abstraction of s. So the new most precise abstraction of s is a formula $\overline{s}_2$, where

$$s \Rightarrow \overline{s}_2 \Rightarrow \overline{s} \wedge \neg J.$$

After this refinement of the abstract domain has been performed, the consecution query on the abstract state will succeed:

$$F_i \wedge \neg \overline{s}_2 \wedge T \Rightarrow F_i \wedge \neg s \wedge T \Rightarrow J',$$

therefore (by weakening the consequent)

$$F_i \wedge \neg \overline{s}_2 \wedge T \Rightarrow (\neg \overline{s} \vee J') \Rightarrow \neg \overline{s}_2.$$

By refining with J, the consecution query succeeds for both s and the new abstraction $\overline{s}_2$, and the algorithm continues to eliminate s from the appropriate levels.

Implementation Issues

The presented approach was implemented using the MathSAT5 SMT solver ([15]), which provides support for computing interpolants over linear integer arithmetic. The interpolants generated by MathSAT are the only source of new abstract domain predicates in this approach.

The composition of the abstract domain is highly important for the algorithm in general, as a good abstract domain can lead to great speed-ups, while a bad abstract domain or a bad refinement strategy can lead to non-termination.

Considering that the formula $\neg s'$ alone is an interpolant, it is not surprising that MathSAT turns up with interpolants that contain many variable assignments. An ideal interpolant is weaker than $\neg s \wedge F_i \wedge T$ and stronger than $\neg s$.

The definition of half-space is needed below, so it is given here.

Definition 18 (Half-Space) *A* half-space *is a linear inequality. An* open half-space *takes the form $\sum_i a_i x_i + c < 0$, where all x_i are constants and all a_i as well as c are integers. Similarly, a* closed half-space *takes the form $\sum_i a_i x_i + c \leq 0$.*

Any negated unprimed interpolant that is added to the domain will usually be a conjunction of many predicates, notably of half-spaces (and equalities, which are equivalent to a conjunction of two closed half-spaces). For eliminating the particular spurious transition the algorithm is concerned with, it does not matter whether the whole conjunction is added or the individual predicates that form the conjunction. Therefore the domain is not simply refined with $\neg J$, but instead $\neg J$ is split along all of the outermost conjunctions. The predicates that form the conjunction are added to the abstract domain individually. This technique greatly increases the expressiveness of the domain, since the formula $\neg J$ is usually a very strong formula that cannot be reused in most future abstractions, but the parts of the conjunction are usually quite weak and will be heavily reused in future abstractions.

Reusing predicates as far as possible instead of repeatedly refining is important because a large abstract domain is one of the primary reasons of slowdown of the algorithm: Amongst other reasons, abstracting concrete states and generalizing clauses becomes exponentially slower with the size of the abstract domain.

Interpolation by "Beautiful Interpolants"

From the last subsection it should be clear that half-spaces are a desirable form of predicate to have in the abstract domain, at least in contrast to variable assignments and other types of strong predicates.

In order to fix this issue, we decided to try a different approach of building interpolants: The approach described in the "Beautiful Interpolants" paper ([1]).

Definition 19 (Beautiful Interpolant) *A beautiful interpolant is a disjunction of convex polytopes. A convex polytope is a conjunction of half-spaces. Therefore, a beautiful interpolant takes the form*

$$\bigvee_j \bigwedge_i LI_{ij},$$

where each LI_{ij} is a linear inequality. The beautiful interpolants approach applies heuristics in order to find an interpolant with i and j being as small as possible.

The paper [1] describes how to compute beautiful interpolants from given input formulas A and B. Since the returned interpolant is a disjunction, its negation is a conjunction. Again, we split the interpolant along the uppermost conjunctions and then added the resulting parts as predicates to the abstract domain for refinement.

The resulting performance was disappointing, however. It turned out that it took very long for beautiful interpolants to be computed, and that the resulting half-spaces were often impractical to use because of the large coefficients. The reason for this might be the structure of our interpolation calls: The A side is a large and structurally complex formula, while the B side is a very small formula that describes very few states.

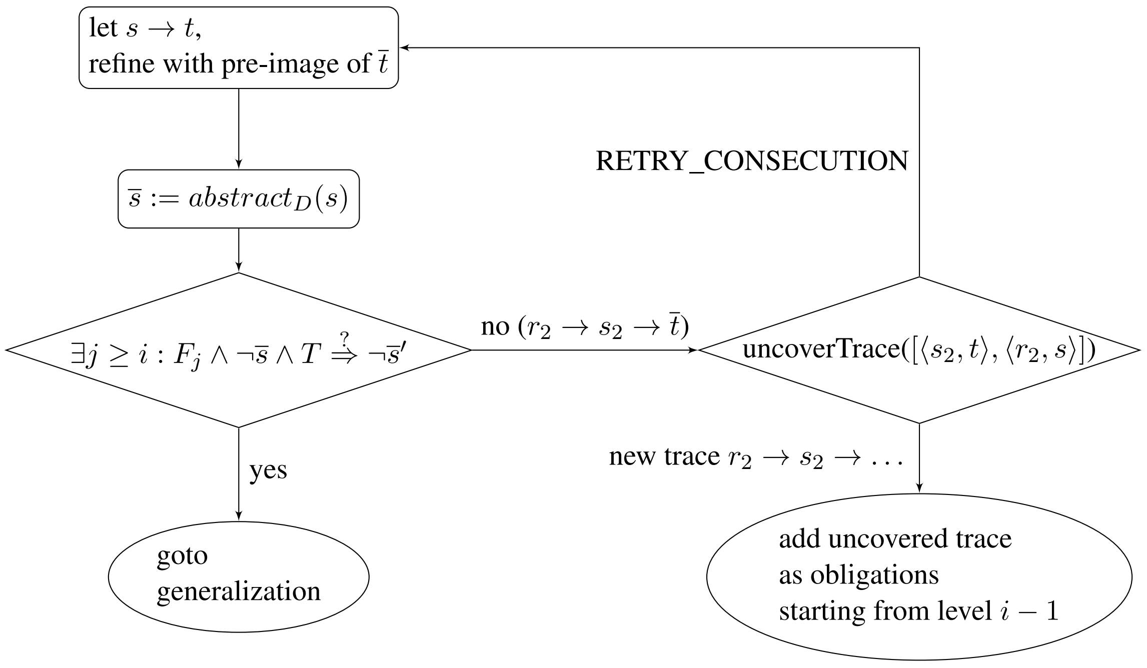
Refinement by Preimage Computation

We also tried a completely different refinement approach that relies on pre-image computation instead of interpolation.

The underlying notion is the following: The best way to make sure that no state leads to the error is to exclude the pre-image of the error from the frontier. Similarly, if a state t is on a trace to the error at level i, then the best way to make sure that no state leads into $\bar{t}$ exclude the pre-image of $\bar{t}$ at level $i-1$.

A simple recursive calculation of all pre-images until a fixed point is reached is, of course, intractable. Computing precise pre-images of the error and of abstract states requires $\exists$-quantifier elimination, as will be seen below.

Figure 3.2: Preimage-based refinement outline


let $s \to t$,
refine with pre-image of $\overline{t}$
$\overline{s} := abstract_D(s)$
$\exists j \geq i : F_j \wedge \neg\overline{s} \wedge T \overset{?}{\Rightarrow} \neg\overline{s}'$
no ($r_2 \to s_2 \to \overline{t}$)
uncoverTrace($[\langle s_2, t\rangle, \langle r_2, s\rangle]$)
RETRY_CONSECUTION
yes
new trace $r_2 \to s_2 \to \dots$
goto generalization
add uncovered trace as obligations starting from level $i - 1$

After the pre-image of the error or the abstract state $\overline{t}$ to exclude has been computed, the domain is refined with the predicates that occur in the pre-image. Note that this alone does not guarantee the elimination of the particular spurious counterexample $s \to t$; the complete algorithm is given below.

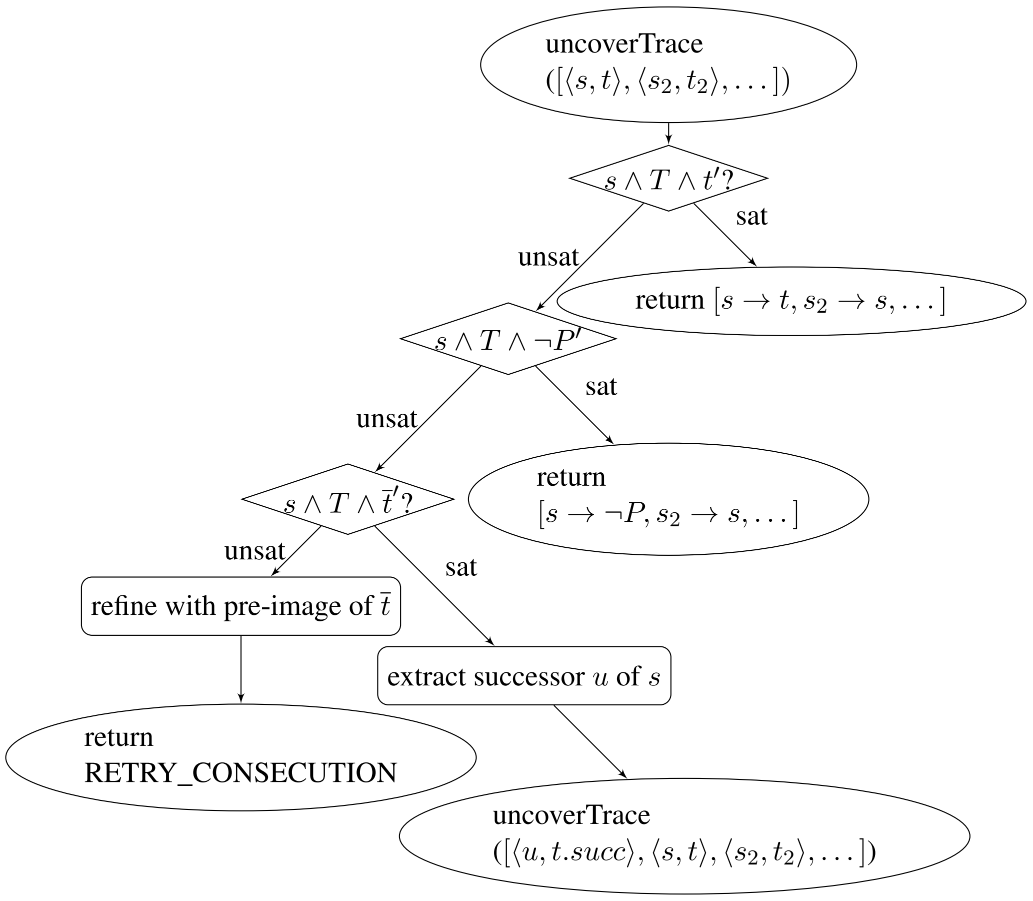
Explanation

The pre-image-based refinement scheme works as follows: Suppose there there is a state s with an abstraction $\overline{s}$ to exclude at level i. The abstract consecution query $\neg\overline{s} \wedge F_i \wedge T \Rightarrow \neg\overline{s}'$ fails, but the concrete consecution query $\neg s \wedge F_i \wedge T \Rightarrow \neg s'$ succeeds. So there is a spurious predecessor r that leads into $\overline{s}$ but not directly to s. Suppose that t is the successor of s.

Now the domain is refined with the pre-image of $\overline{t}$. The pre-image is obtained in one of the following ways:

- If the pre-image of the exact same $\overline{t}$ was computed in a previous refinement step, restore the pre-image from memory. All pre-images computed in refinement are kept in memory, as they do not take up much space, but eliminate the need to re-compute a previous pre-image.
- Otherwise, compute the pre-image by quantifier elimination over the formula $\overline{\exists x'.T(\vec{x}, \vec{x'}) \wedge t(\vec{x'})}$. The SMT solver Z3 ([22]) is capable of performing such a quantifier elimination over linear integer arithmetic.

Figure 3.3: Preimage-based refinement: Uncovering a new trace


uncoverTrace
$([\langle s,t\rangle,\langle s_2,t_2\rangle,\dots])$
$s \wedge T \wedge t'$?
sat
unsat
return $[s \to t, s_2 \to s, \dots]$
$s \wedge T \wedge \neg P'$
sat
unsat
return
$[s \to \neg P, s_2 \to s, \dots]$
$s \wedge T \wedge \overline{t}'$?
unsat
sat
refine with pre-image of $\overline{t}$
extract successor u of s
return
RETRY_CONSECUTION
uncoverTrace
$([\langle u, t.succ\rangle,\langle s,t\rangle,\langle s_2,t_2\rangle,\dots])$

Refining with the pre-image of $\overline{t}$ does not guarantee the abstract consecution query to hold. If it holds, then the abstract state $\overline{s}$ will be excluded at level i. If the abstract consecution query fails, a transition $r_2 \to s_2 \in \overline{s} \to \overline{t}$ is extracted. This transition needs to be eliminated in any case because the next goal will be to exclude $\overline{t}$ at level $i+1$, which will fail if there are any transitions leading into $\overline{t}$ (even if these transitions do not lead directly to t).

The algorithm now tries to uncover a trace starting from the path prefix $r_2 \to s_2$ that leads to the error. By the previous refinement, it is guaranteed that there exists at least a path prefix $r_2 \to s_2 \to t_2 \in \overline{t}$. Let u be the successor of t. The next step would be to find a concrete successor to t_2 in the abstraction $\overline{u}$. If no such concrete successor is found, the domain is refined with the (under-approximation of the) pre-image of $\overline{u}$ and the algorithm restarts by trying to exclude the abstraction of s at level i.

The uncovering of the trace continues until either a concrete trace is found from a predecessor of $\overline{s}$ to the error or the abstraction $\overline{s}$ satisfies consecution at level i. One of these cases is guaranteed to hold: The trace is consecutively refined with (under-approximations of) pre-images that recursively eliminate spurious predecessors, resulting either in the elimination of all predecessors or the finding of a non-spurious predecessor.

Each of these refinements is done only if there is a transition into an abstract state that does not lead to the successor to exclude. If there are such transitions into an abstract state,

- they must either be eliminated, otherwise the abstract consecution query will never hold on this state
- or they are part of an actual error trace and should be uncovered sooner rather than later.

However, it turned out that because of the following factors, the approach turned out to be inferior to refinement by interpolation:

- Pre-images are very expensive to compute.
- The resulting predicates are similar to what one gets by interpolation, so the abstract domain
- The phase of uncovering a new trace can lead to a number of refinement steps that is up to quadratic in the number of steps of the current trace. While this rarely happens, it is usually the case that a number of refinements must be made to eliminate the original spurious transition $s \rightarrow t$.

A previous attempt at applying inductive verification to software ([14]) also relies on pre-image computation. Instead of computing the exact pre-image, as is done with quantifier elimination, the approach computes an under-approximation of the pre-image. Despite the fact that computing such an under-approximation is much faster than performing quantifier elimination, we did not try it. The following challenges might occur when trying to implement pre-image based refinement using under-approximation of pre-images:

- Suppose there is a spurious transition $s \rightarrow t$. The under-approximation of the pre-image of $\overline{t}$ must refine the domain in such a way that this spurious transition is eliminated. When refining with the under-approximation of a pre-image, this is not the case anymore. In order for this to be the case, one has to make sure to get an under-approximation of the pre-image of $\overline{t}$ that refines the abstraction $\overline{s}$ of s, which is expensive again.
- If there is a fitting under-approximation: The predicates of the under-approximation of the pre-image might not be better than the ones gathered by the exact pre-image, resulting in no better performance.

Refinement State Mining

Motivation

Though the above refinement strategies suffice for correctness, they do not suffice for termination in many cases. We implemented an additional refinement strategy that we choose to call *Refinement State Mining*.

Non-termination of IC3-CEGAR is often caused by a diverging domain: The abstract domain is continually refined without ever achieving termination. Still, refining the domain is necessary in order for IC3-CEGAR to be correct. But the predicates generated by any of the discussed refinement strategies, notably by interpolation, may not be good enough to ensure termination. Refinement State Mining is a technique that extracts predicates not to eliminate a specific spurious transition, but to refine the domain with predicates which may eliminate transitions that are likely to occur in the future.

Algorithm 4 Refinement State Mining

```
function VOID REFINEMENTSTATEMINING($s$: concrete state, $S$: set of concrete states)
    $S \leftarrow S \cup \{s\}$
    for each $\{G \subseteq S |$ all $s \in A$ have the same $pc$ value and the same variables $\}$ do
        if $|G| <$ minPoints then
            continue
        end if
        Define concrete state set $R$;
        if MINESTATES($G$, $R$) then
            $S \leftarrow S \setminus R$
        end if
    end for
end function
function BOOL MINESTATES($G$, $R$: sets of concrete states)
    if $|G| \leq 2$ then
        return $false$
        Define concrete state set $U$
        if CREATE$Predicate$($G$, $U$) then
            $R \leftarrow G$
            return $true$
        else ▷ UNSAT core will contain a collection of points which together do not satisfy the predicate
            Let $u_1 \in U, U_2 = U \setminus \{u_1\}$
            if MINESTATES($G \setminus \{u_1\}$, $R$) then
                return $true$
            end if
            if MINESTATES($G \setminus U_2$, $R$) then
                return $true$
            end if
            return $false$
        end if
    end if
end function
```

Conceptual Overview

In IC3-CEGAR, refinement is triggered when there is a proof obligation $\langle s, i \rangle$, consisting of a state s and a level i, for which $\neg s \wedge F_i \wedge T \Rightarrow \neg s'$, but $\neg \overline{s} \wedge F_i \wedge T \not\Rightarrow \neg \overline{s'}$, where $\overline{s} = \alpha(s)$.

Assume that s is a vector of variable assignments that includes an assignment to the program counter pc. (Note that a proof obligation's state does not have to include an assignment to the program counter pc because of lifting; in practice, all lifted states still contain the program counter. Refinement state mining can be skipped for any proof obligation that does not contain the program counter.)

Refinement state mining works can be split in two phases:

- **Phase 1:** States used in an interpolating refinement call are grouped by program counter value (and stripped of the program counter assignment). All states that had the same program counter value are grouped into sets where each state shares the same assigned variables.

- **Phase 2:** Let G be a set of states that had the same program counter value and share the same assigned variables. If $|G| \geq$ minPoints (where minPoints is a predefined threshold with minPoints ≥ 2), then try to find a predicate that describes the states in G. The only predicate matching scheme currently implemented is trying to find a linear equality $\vec{a} * v\vec{a}r = c$, where $\vec{a}$ is a line vector of integer constants, c is an integer constants and $v\vec{a}r$ is a column vector of integer variables occurring in all states of G.

  The Refinement State Mining algorithm tries to find out if there is a linear equality that covers all states in G:
  $$\exists \vec{a}, c. \forall \vec{val} \in G.(\vec{a} * \vec{val} = c),$$
  where $\vec{val}$ is a column vector of the variable values in a given state.

  - **If fitting coefficients $\vec{a_i}, c$ cannot be found**, then the algorithm is re-started on a subset of the states in G unless $|G| \leq 2$, in which case the algorithm returns without finding a fitting predicate.
  - **If fitting coefficients $\vec{a_i}, c$ are found**, then the domain is refined with the linear equality $\vec{a} * v\vec{a}r = c$.

Instead of trying to fit a linear equality on a set of points, one could also try to fit predicates of different types on the set of points, such as non-linear predicates. We have only implemented the fitting of linear equalities.

Implementation

A sample implementation of the principle as shown above is given in algorithms 4 and 5.

The algorithm 4 starts in by grouping states previously used for interpolating refinement calls into sets by program counter value and by the assigned variables per state. If it determines that there are enough states in a particular set (where "enough" means "more than minPoints", and minPoints ≥ 2), then it calls a function that tries to find a predicate that will describe all

Algorithm 5 Fitting a linear equality on a set of points

```
1: function BOOL CREATELINEAREQUALITY($G$, $U$: sets of concrete states) Formula $f \leftarrow \top$
2:     for each state $s : G$ do
3:         LinearEquality $linEq_s \leftarrow \sum_i a_i v_i + c = 0$    ▷ where all $a_i$ and $c$ are free and all $v_i$ are instantiated with the values from $s$
4:         $f \leftarrow f \wedge linEq_s$
5:     end for
6:     Constraint $nz \leftarrow$ MKNOTZEROCONSTRAINT(variables that occur in $G$)
7:     $f \leftarrow f \wedge nz$    ▷ Assert that at least one of all $a_i$ or $c$ must be non-zero
8:     if $f$ is satisfiable then
9:         $\forall_i a_i, c \leftarrow$ SATMODEL($f$)
10:        LinearEquality $line \leftarrow \sum_i a_i x_i + c = 0$    ▷ where all $a_i$ and $c$ are instantiated with the SAT values from the model and all $x_i$ (which are the corresponding state variables) are constants
11:        abstractDomain.REFINE($line$)
12:        return $true$
13:    else
14:        $U \leftarrow$ UNSATCORE($f$)    ▷ The UNSAT core determines which linear equalities and corresponding states prevented the creation of a line through the given points
15:        return $false$
16:    end if
17: end function
```

given states. Such a function, which tries to fit a linear equality on a given set of states, is given in algorithm 5.

The algorithm 5 will either find a linear equality that describes all given states, or return with a set U of at least three states that cannot be described by a single linear equality.

In the first case, the abstract domain will be refined with the linear equality.

In the latter case, either $G \leq 2$, in which case the function returns without finding a fitting predicate. Otherwise, the original set of states is divided into two (possible overlapping) subsets: The first subset G_1 contains all points G
U, plus a subset U_1 of the states in U. The second subset G_2 contains all points G
U, plus a subset U_2 with $U_2 = U$
U_1. Refinement State Mining is re-started recursively on the sets G_1 and G_2.

Since the trivial linear equality $\vec{a} = \vec{0}, c = 0$ can be found for any set of points, this trivial solution is forbidden while trying to fit a linear equality on a set of states.

Related Work

Approaches similar to Refinement State Mining have been explored previously:

- The **Daikon** invariant detector ([26]) is a tool that discovers likely invariants by examining actual runs of a program. The runs of the program are analyzed and the tool tries to construct invariants that hold at specified points in the program.
- The approach described in [43] constructs linear and non-linear loop invariants by examining sets of concrete states that occur at specific points in the program. Invariants are generated using a guess-and-check algorithm.

Both of the approaches could be used to enhance the capabilities of IC3-CEGAR in future developments.

Lazy Refinement

So far, we have assumed that refinement occurs immediately when a spurious transition is discovered. However, it is not necessary for correctness to refine immediately when a spurious transition is detected: In this case, the algorithm could just continue without refining until a spurious trace from the initial state to the error is discovered. At this point, the algorithm must refine the domain in order to exclude the spurious transition. However, not all spurious transition will allow spurious traces from an initial state to the error. Since it is generally favorable to deal with a coarser domain, we experimented with *lazy refinement*: delaying refinement when discovering spurious transitions until either a spurious trace from the initial state to the error is discovered (in which case refinement is obligatory) or a spurious trace with too many spurious transitions is discovered.

The main difference in the data structures is that a proof obligation is now a triple $\langle s, i, n\rangle$ instead of a tuple $\langle s, i\rangle$. The third tuple argument n is counts the number of spurious transitions in the current proof obligations trace; if n exceeds the user-defined constant $\mathsf{maxSpurious}$, refinement is triggered regardless of whether a trace from the initial condition to the error was discovered or not.

If a spurious predecessor t is discovered, then t is a predecessor to the abstraction $\overline{s}$ but not to s. Therefore, t might not be inductive relative to level $i-2$, so the proof obligation for t is added at level 0.

If a trace from the initial state to the error is discovered that contains at least one spurious transition, then all proof obligations up to the spurious transitions are deleted and the abstract domain is refined such that this particular spurious transition is eliminated. A spurious transition is known to occur if $n > 0$ in a proof obligation $\langle s, i, n\rangle$.

Pseudo-code outlining the handling of proof obligations with lazy refinement is given in figures 6 and 7.

3.5 Abstract State Generalization

We employ minimal-inductive-clause generalization and the CTGDOWN algorithm to generalize abstract states. The finite-state case as introduced in [30] was described in section 2.2.

Algorithm 6 Outline of lazily refinement in IC3-CEGAR. I is the initial condition. T is the transition relation. F_i is the frame at level i.

```
1: function VOID HANDLEOBLIGATIONS($PQ$: proof obligations priority queue)
2:     while $PQ$.size > 0 do
3:         $\langle s, i, n\rangle \leftarrow PQ$.PEEK        ▷ $\langle s, i, n\rangle$ is the obligation with the lowest $i$
4:         if $i = 0$ and $I \wedge s$ is satisfiable then
5:             if $n = 0$ then
6:                 Terminate with trace from $s$ to $\neg P$
7:             end if
8:             BACKTRACKREFINE($\langle s, i, n\rangle$, $PQ$)        ▷ Refine with last known spurious
   transition and delete proof obligations up to this transition
9:             Continue
10:        end if
11:        $\overline{s} = \overline{s_0} \vee \cdots \vee \overline{s_y} \leftarrow \alpha(s)$
12:        set<clause ref> $generalizations$
13:        level $minLevel \leftarrow k + 1$
14:        for each $x \leftarrow 0 .. y$ do
15:            if $\neg$ ELIMINATEABSTRACTSTATE($\overline{s_x}$, $i$, $generalizations$, $minLevel$) then
16:                break
17:            end if
18:        end for
19:        if all $\overline{s_x}$ were successfully eliminated then
20:            $PQ$.POP        ▷ Eliminate proof obligation $\langle s, i, n\rangle$ from $PQ$
21:            if $i < minLevel \leq k$ then
22:                $PQ \leftarrow PQ \cup \langle s, minLevel, n\rangle$        ▷ Push proof obligation forward
23:            end if
24:        end if
25:    end while
26: end function
```

A variant of MIC and CTGDOWN that is applicable to the infinite state case is given in algorithm 8. The main difference is that concrete counterexamples to generalization (CTG) now need to be abstracted. Each of the abstractions is then treated as an abstract counterexample to generalization. These abstract counterexamples to generalization are then either successfully excluded at the given level, or they are joined to the cube $\hat{q}$ by the $\sqcap$ operator.

Our experiments with this variant of CTGDOWN indicate that good values for the max* constants are: maxDepth $= 1$, maxCTGs $= 3$ and maxJoins $= \infty$.

Geometric Generalization

In the infinite state case with an abstract domain, a certain type of generalization is possible that is not feasible in the finite-state case: geometric generalization.

Algorithm 7 Outline of eliminating abstract states in IC3-CEGAR with lazy refinement. I is the initial condition. T is the transition relation. F_i is the frame at level i.

```
function BOOL ELIMINATEABSTRACTSTATE($\overline{s_x}$: abstract state ref, $i$: level, $generalizations$: set<clause ref> ref, $minLevel$: level ref)
    if $\exists c \in generalizations$ with $\overline{s_x} \Rightarrow c$ then
        continue
    end if
    if $\neg\overline{s_x} \wedge F_i \wedge T \Rightarrow \neg\overline{s_x}'$ then    ▷ $\neg\overline{s_x}$ satisfies consecution at level $i$?
        $c \leftarrow$ GENERALIZE($\neg\overline{s_x}$)    ▷ GENERALIZE might be CTGDOWN
        $j \leftarrow max\{j \mid \neg\overline{s_x} \wedge F_j \wedge T \Rightarrow \neg\overline{s_x}'\}$    ▷ Find max. $j$ relative to which $c$ satisfies consecution
        for $l \leftarrow 1 \,..\, j+1$ do
            $F_l \leftarrow F_l \wedge c$
        end for
        $minLevel \leftarrow min(j, minLevel)$
        return $true$
    else if $\neg s \wedge F_i \wedge T \Rightarrow \neg s'$ then    ▷ There is a spurious transition into $\overline{s}$
        SatModel $mod$ ←SATMODEL($\neg\overline{s_x} \wedge F_i \wedge T \wedge \overline{s_x}'$)
        Spurious predecessor $t \leftarrow$ unprimed variable assignments from $mod$
        if $I \wedge t$ satisfiable or $n \geq$ maxSpurious then
            REFINEABSTRACTDOMAIN($\langle s, i\rangle$)
            return $false$
        else
            $PQ \leftarrow PQ \cup \langle t, 0, n+1\rangle$
            return $false$
        end if
    else
        SatModel $mod$ ←SATMODEL($\neg s \wedge F_i \wedge T \wedge s'$)
        Predecessor $t \leftarrow$ unprimed variable assignments from $mod$
        if $i = 0$ then
            if $n = 0$ then
                Terminate with trace from $t$ to $\neg P$    ▷ $t$ must be an initial state
            else
                BACKTRACKREFINE($\langle s, i, n\rangle$, $PQ$)    ▷ Refine with last known spurious transition and delete proof obligations up to this transition
                return $false$
            end if
        else
            $PQ \leftarrow PQ \cup \langle t, i-1, n\rangle$
            return $false$
        end if
    end if
end function
```

Algorithm 8 MIC with ctgDown for the infinite state case

```
function VOID MIC($q$: cube ref, $i$: level)
    MIC($q$, $i$, 1)
end function
function VOID MIC($q$: cube ref, $i$: level, $d$: recDepth)
    for each Literal $\ell$ in $q$ do
        $\hat{q} \leftarrow q \setminus \ell$
        if CTGDOWN($\hat{q}$, $i$, $d$) then
            $q \leftarrow \hat{q}$
        end if
    end for
end function
function BOOL CTGDOWN($\hat{q}$: cube ref, $i$: level, $d$: recDepth)
    $ctgs \leftarrow 0$
    $joins \leftarrow 0$
    while $true$ do
        if $I \wedge \hat{q}$ satisfiable then
            return $false$
        end if
        if $F_i \wedge \neg\hat{q} \wedge T \Rightarrow \neg\hat{q}'$ then
            return $true$
        end if
        if $d >$ maxDepth then
            return $false$
        end if
        State $s \models F_i \wedge \neg\hat{q}$, $s$ leads into $\neg\hat{q}$
        $\overline{s} = \overline{s_0} \vee \cdots \vee \overline{s_y} \leftarrow \alpha(s)$
        for each $\overline{s_x}$ in $\overline{s}$ do
            if $(ctgs <$ maxCTGs$)$ and $(i > 0)$
 and $(I \Rightarrow \neg\overline{s_x})$ and $F_{i-1} \wedge \neg\overline{s_x} \wedge T \Rightarrow \neg\overline{s_x}'$ then
                $ctgs \leftarrow ctgs + 1$
                for $j \leftarrow i$ to $k$ do
                    if $F_j \wedge \neg\overline{s_x} \wedge T \not\Rightarrow \neg\overline{s_x}'$ then
                        break
                    end if
                end for
                MIC($\overline{s_x}$, $j - 1$, $d + 1$)
                for $l \leftarrow 1 .. j$ do
                    $F_l \leftarrow F_l \wedge \neg\overline{s_x}$
                end for
            else if $joins <$ maxJoins then
                $ctgs \leftarrow 0$
                $joins \leftarrow joins + 1$
                $\hat{q} \leftarrow \hat{q} \sqcap \overline{s_x}$
            else
                return $false$
            end if
        end for
    end while
end function
```

Given a cube q, dropping a literal ℓ from q is the same as replacing the ℓ by $\top$ in q, obtaining $\hat{q}$. In previous sections, we discussed how to determine whether $\hat{q}$ is a proper generalization of q.

Consider the case that $\hat{q} = q \setminus \ell$ turns out not to be a proper generalization of q after the CTGDOWN call. Instead of simply forfeiting on ℓ, the literal might be replaced not by the weakest possible predicate $\top$, but by a literal $\pounds$ that is weaker that ℓ but stronger than $\top$, thus generalizing q. The new cube $\hat{q} = q \cup \{\pounds\} \setminus \wp$ is then checked via CTGDOWN.

Where could this weaker literal $\pounds$ come from? The easiest way to look for such a literal is to search among the existing predicates of the abstract domain.

- In the case that $\ell = \wp$, where $\wp$ is some predicate in the abstract domain, look for a predicate $\wp_2$ with $\wp \Rightarrow \wp_2$ and replace $\wp$ by $\wp_2$ in q, obtaining $\hat{q}$. Then check $\hat{q}$ by CTGDOWN.
- In the case that $\ell = \neg\wp$, where $\wp$ is some predicate in the abstract domain, look for a predicate $\wp_2$ with $\wp_2 \Rightarrow \wp$ and replace $\neg\wp$ by $\neg\wp_2$ in q, obtaining $\hat{q}$. Then check $\hat{q}$ by CTGDOWN.

In practice, ℓ is repeatedly replaced by weaker literals as long as CTGDOWN returns $true$. The modified MIC algorithm is given is algorithm 9.

Geometric generalization does not yield a better performance in all cases. There are benchmarks for which it degrades performance, but also cases for which it results in a significantly shorter run-time. We do not have a clear idea why geometric generalization hurts performance in some cases; it is not the case that the additional code in MIC is responsible for all of the slowdown. As far as we see currently, enabling geometric generalization can lead to both a higher and a lower k until convergence. However, in some cases geometric generalization leads to the termination of benchmarks that suffer from a diverging domain otherwise.

We experimented with bounding the number of geometric generalization attempts for each MIC call by a constant, but bounding the number of attempts does not make a great difference. The main difference in the algorithm performance is determined by whether geometric generalization is enabled at all or not.

3.6 Various Other Performance Improvements

Refinement with Inequalities

In practice, many refinement predicates from an interpolation call are linear equations. Performance is drastically improved if for every linear equation $\sum_i a_i x_i + c = 0$, a logically equivalent conjunction of linear inequality $\sum_i a_i x_i + c \leq 0 \wedge \sum_i a_i x_i + c \geq 0$ is taken instead. This is then split up into the inequalities $\sum_i a_i x_i + c \leq 0$ and $\sum_i a_i x_i + c \geq 0$, which are then added to the abstract domain predicates.

Experiments have revealed that in case geometric generalization in enabled, it is beneficial to refine the domain with the original equality $\sum_i a_i x_i + c = 0$ and one additional inequality, namely $\sum_i a_i x_i + c \leq 0$.

Algorithm 9 MIC with geometric generalization. $\mathbb{P}$ is the lattice of predicates of the abstract domain. The operations on this lattice of predicates are described in section 3.1.

```
function VOID MIC($q$: cube ref, $i$: level)
    MIC($q$, $i$, 1)
end function
function VOID MIC($q$: cube ref, $i$: level, $d$: recDepth)
    for each Literal $\ell$ in $q$ do
        $\hat{q} \leftarrow q \setminus \ell$
        if CTGDOWN($\hat{q}$, $i$, $d$) then
            $q \leftarrow \hat{q}$
        else
            if $\ell = \wp$ then
                for each $\wp_2 \in$ PARENTS($\wp$) do
                    $\hat{q} \leftarrow (q \setminus \{\ell\}) \cup \wp_2$
                    if CTGDOWN($\hat{q}$, $i$, $d$) then
                        $q \leftarrow \hat{q}$
                        break
                    end if
                end for
            else if $\ell = \neg\wp$ then
                for each $\wp_2 \in$ CHILDREN($\wp$) do
                    $\hat{q} \leftarrow (q \setminus \{\ell\}) \cup \neg\wp_2$
                    if CTGDOWN($\hat{q}$, $i$, $d$) then
                        $q \leftarrow \hat{q}$
                        break
                    end if
                end for
            end if
        end if
    end for
end function
```

Initial Abstract Domain

For correctness, it suffices if the initial abstract domain predicate lattice contains only $\top$, $\bot$ and the initial condition I. Any abstract domain used in IC3-CEGAR contains the initial condition by default, as this ensures that the most precise abstraction of any non-initial state does not intersect with the initial condition. (However, including the initial condition as a predicate in the domain is not required for correctness; if I is not included in the initial abstract domain, then a check for initiation of abstract states and subsequent refinement may be necessary.)

Our experiments indicate that performance is markedly improved if the abstract domain predicates are initialized with linear inequalities between pairs of program variables in the same function: That is, the abstract domain predicate lattice is initialized with $\top$, $\bot$, the initial condition, and linear inequalities $v_1 < v_2$ for all pairs of program variables v_1, v_2.

In addition, the abstract domain predicate lattice is set up to contain all Boolean expressions from the original program: The transition relation compiler saves Boolean expressions from the program, and at the beginning of an IC3-CEGAR run, the abstract domain is refined with these Boolean expressions.

Generalization in Consecution

This optimization was described in [10] and can be found in the reference implementation of IC3 ([9]).

Given an abstract state $\overline{s}$, a query for consecution $\neg\overline{s} \wedge F_i \wedge T \Rightarrow \neg\overline{s}'$ is translated to $\neg\overline{s} \wedge F_i \wedge T \wedge \overline{s}'$ and given as input to the SMT solver. If the solver returns UNSAT, the clause $\neg\overline{s}$ satisfies consecution relative to level i. In this case, as a first attempt at clause generalization, the unsatisfiable core of the SMT solver query is extracted. Let c' be a cube made of the constituent literals of $\overline{s}'$ that occur in the unsatisfiable core of the query. The unprimed version $\neg c$ of $\neg c'$ might be a first generalization of $\neg\overline{s}$:

It holds that $\neg\overline{s} \wedge F_i \wedge T \wedge c'$ is unsatisfiable, therefore $\neg\overline{s} \wedge F_i \wedge T \Rightarrow \neg c'$ is valid, and by strengthening of the antecedent, $\neg c \wedge F_i \wedge T \Rightarrow \neg c'$ is also valid. In the case that $\neg c$ satisfies initiation, it is taken as input for the generalization call instead of the possibly much larger $\neg\overline{s}$.

However, $\neg c$ is not a valid generalization of $\neg\overline{s}$ if it does not satisfy initiation: $I \Rightarrow \neg c$ might not be valid, where I is the initial condition. In this case, the unsatisfiable core is not used for preliminary generalization, and MIC is called on the original abstract cube.

In the finite-state case of IC3, the unsatisfiable core is then foregone and $\neg s$ is used as input for the generalization step. However, considering that $\neg\overline{s}$ will already satisfy initiation if it is checked for consecution, the clause $\neg c$ can be enriched by a simple disjunction with the initial condition I, which is a single predicate in the domain. The resulting clause $\neg c \vee I$ is still entailed by the original $\neg\overline{s}$ and will certainly satisfy initiation: $I \Rightarrow (\neg c \vee I)$ always holds.

Thus, GENERALIZE call for a clause $\neg\overline{s}$ that satisfies consecution will already start off with a generalized version $\neg c$ of the original $\neg\overline{s}$. In the process of generalization (notably in CTG-DOWN), the unsatisfiable cores of consecution calls are again used to accelerate the process.

CHAPTER 4

Empirical Evaluation

We implemented IC3-CEGAR in C++, on top of a reference implementation of IC3 available at [9]. As an SMT solver, we used MathSAT5 ([15]).

In order to translate C-like programs to transition relations readable by SMT solvers, we wrote an ANTLR 4 ([39])-based compiler. This very simple translator does not perform any optimizations at all on the resulting control flow graphs. The performance gains after doing very simple manual optimizations (like constant propagation, eliminating unused variables or removing dead code) on the transition relations are sometimes astonishing.

4.1 Benchmark Selection

We chose three benchmark suites to run our model checker on:

- The InvGen benchmark suite, as it is found in [29]. We had to omit some benchmarks:
  - Our translator could not handle the benchmarks `crawl_cbomb.c`, `fragtest.c`, `linpack.c`, `SpamAssassin-loop*.c` as they contain pointers, structures, type definitions or other advanced C constructs that we cannot handle.
  - The benchmarks `half.c`, `heapsort*.c` and `id_trans.c` contain truncating integer divisions, which we cannot handle.
  - The benchmarks `puzzle1.c`, `sort_instrumented.c`, and `test.c` do not contain any safety properties to prove (i.e., they do not contain `assert` statements).
  - The benchmarks `spin*.c` rely on functions that provide mutex functionality (`acquire` and `release`).
- The Dagger benchmarks suite as found in [28]. We omitted the three `p*-ok.c` benchmarks as well as the three `p*-bad.c` benchmarks, since they contain pointers, which we cannot handle.

- The benchmarks suite as found in [1]. We did not duplicate those benchmarks that were taken from the InvGen or Dagger benchmark suites.

4.2 Evaluation Configurations

We let IC3-CEGAR run in multiple configurations:

- **NOOPT**: This is a run where the CTGDOWN parameters maxCTG and maxDepth are each set to 0, effectively disabling CTGDOWN, and geometric generalization as well as refinement state mining are turned off.
- **C**: This is a run where the CTGDOWN parameters maxCTG is set to 3 and maxDepth is set to 1, and geometric generalization as well as refinement state mining are turned off.
- **CG**: This is a run where the CTGDOWN parameters maxCTG is set to 3 and maxDepth is set to 1, the MIC parameter maxGeo is set to 100 and refinement state mining is turned off.
- **CR**: This is a run where the CTGDOWN parameters maxCTG is set to 3 and maxDepth is set to 1, geometric generalization is disabled and refinement state mining is triggered as soon as three possible points occur at a given program counter location.
- **CGR**: This is a run where the CTGDOWN parameters maxCTG is set to 3 and maxDepth is set to 1, the MIC parameter maxGeo is set to 100 and refinement state mining is triggered as soon as three possible points occur at a given program counter location.
- **CRL**: This is a run where the CTGDOWN parameters maxCTG is set to 3 and maxDepth is set to 1, geometric generalization is disabled and refinement state mining is triggered as soon as three possible points occur at a given program counter location. Additionally, lazy refinement is turned on with the maxSpurious parameter set to 3.
- **CGRL**: This is a run where the CTGDOWN parameters maxCTG is set to 3 and maxDepth is set to 1, the MIC parameter maxGeo is set to 100 and refinement state mining is triggered as soon as three possible points occur at a given program counter location. Additionally, lazy refinement is turned on with the maxSpurious parameter set to 3.

CTGDOWN refers to the algorithm in section 3.5. Geometric generalization refers to the technique described in section 3.5. Refinement state mining refers to the algorithm in section 3.4. Lazy refinement refers to the extension described in section 3.4 and is turned off unless noted otherwise.

We compare our implementation of IC3-CEGAR against CPAChecker [7], the winner of the 2nd software verification competition. CPAChecker is a highly configurable verifier. The last column, "CPAChecker", refers to a run of CPAChecker that was started with the parameters `-config config/sv-comp13--combinations-predicate.properties -timeout 1200.`

The benchmarks were performed on "AMD Opteron(TM) Processor 6272" CPUs at 1400 MHz. We did not set a memory treshold. The timeout set for the benchmarks is 1200 seconds, wall time. However, if IC3-CEGAR or CPAChecker do not run into the timeout, we report the run times in the operating systems's user mode used for the benchmark, as they are more accurate than the wall time.

4.3 Run Time Results

A table listing the run time results for out seven configurations and the CPAChecker run on the benchmark suites is presented here. A graphical representation of the run time results for the IC3-CEGAR configurations with Refinement state mining and CPAChecker is presented below in figure 4.1.

Regarding the values in the CPAChecker column:

- All cells that are empty designate a benchmark in which CPAChecker did not terminate within 1200 seconds.
- All cells marked with an asterisk (*) designate a benchmark which CPAChecker failed to prove because of an incomplete analysis. In such cases, CPAChecker returns with a `Verification result: UNKNOWN, incomplete analysis` or a `Analysis incomplete: no errors found, but not everything could be checked.` message.

Benchmark	NOOPT	C	CG	CR	CGR	CRL	CGRL	CPAChecker
apache-escape-absolute.c	502.36	710.19	761.89	714.08	769.28	1181		*
apache-get-tag.c		309.72	744.83	313.78	729.66	378.37	536.01	*
barbr.c								28.95
barbrprime.c								13.28
bind_expands_vars2.c	3.09	5.28	4.46	5.27	4.18	6.79	6.97	4.09
bkley.c	264.5	238.85	328.01	232.28	323.35	399.3	708.2	4.78
bk-nat.c	352.45	304.35	613.13	300.66	635.11	287.43	403.34	5.03
bound.c	10.94	21.7	47.38	20.41	44.77	17.85	195.81	*
cars.c								177.60
dillig01.c	10.5	6.12	13.3	6.15	13.36	4.76	4.86	3.92
dillig03.c	0.53	0.6	0.74	0.57	0.76	0.7	2.18	*
dillig05.c		107.43	1098.71	497.5		76.25		*
dillig07.c	20.72	2.64	3.74	2.64	3.72	2.88	3.29	4.03
dillig12.c				312.63				
dillig15.c				30.19	49.68		199.61	3.90
dillig17.c	207.5	53.43	61.63	53.36	60.79	40.59	44.72	4.18
dillig19.c		214.49	913	241.73	373.63	47.95	129.45	3.86
dillig20.c	1031.46	210.92	290.45	212.92	292.9	163.53	268.25	4.57
dillig25.c	10.92	6.02	75.93	6	73.4	10.61	15.76	4.10
dillig28.c	17.44	23.32	42.63	23.48	33.98	39.95	69.71	4.32
dillig32.c	52.07	23.84	22.19	22.28	22.77	41.54	280.56	4.07
dillig33.c	961.32	528.46		1128.04		1049.34		4.33
dillig37.c	25.86	13.75	46	12.14	44.51	3.98	17.32	4.31
down.c	15.3	123.41	38.19	342.61	37.97	55.12	501.7	
efm.c								7.29
ex1.c	11.68	14.13	14.5	13.33	15.39	312.49	605.05	4.05
ex2.c		336.12	1110.2	335.6		604.16		4.23
fig1a.c				495.58	1123.47	164.21	70.46	*
fig2.c			612.8		623.12	899.2		*
fragtest_simple.c								*

Benchmark	NOOPT	C	CG	CR	CGR	CRL	CGRL	CPAChecker
gulv.c								*
gulv_simp.c	57.21	8.75	22.6	8.44	20.94	8.02	17.36	4.84
gulwani_cegar1.c	5.06	7.48	7.88	7.12	7.66	7.74	7.94	4.15
gulwani_cegar2.c	9.44	4.97	12.72	5.06	12.18	11.5	12.01	4.80
gulwani_fig1a.c	0.28	0.45	0.37	0.46	0.37	0.36	0.38	7.19
hsort.c								4.80
hsortprime.c	78.13	473.21	1082.7	477.58	1116.91	1157.15		4.22
id_build.c	14.62	9.26	12.87	9.14	11.39	4.98	5.66	*
ken-imp.c		999.82	202.98	138.69		13.12	31.55	4.29
large_const.c								*
lifnat.c								27.02
lifnatprime.c								25.25
lifo.c								8.50
MADWiFi-encode_ie_ok.c								4.82
mergesort.c								3.95
nested1.c	1.03	1.22	1.73	1.1	1.69	1.09	1.67	4.04
nested2.c	1.01	1.1	1.74	1.13	1.67	1.12	1.72	4.02
nested3.c	3.67	3.5	3.52	3.62	3.38	2.69	4.45	*
nested4.c	5.08	4.08	6.05	3.73	5.37	3.26	4.04	*
nested5.c	13.36	12.32	9.8	11.18	8.92	15.13	31.8	4.20
nested6.c	212.45	188.59	301.15	183.55		63.56		*
nested7.c								*
nested8.c		500.83		130.72		31.87	170.28	10.11
nested9.c		87.52	114.71	92.19		126.24	140.8	
nested.c	0.11	0.13	0.12	0.12	0.12	0.12	0.12	4.12
nest-if1.c	6.16	2.67	10.74	2.71	12	2.64	8.25	*
nest-if2.c	6.73	11.17	13.32	11	13.19	2.9	5.3	*
nest-if3.c	6.11	4.84	7.44	4.19	7.16	4.64	5.18	*
nest-if4.c	12.58	7.22	12.67	6.79	12.21	4.54	5.34	*
nest-if5.c	2.9	11.94	4.86	11.66	4.9	9.66	13.42	*

Benchmark	NOOPT	C	CG	CR	CGR	CRL	CGRL	CPAChecker
nest-if6.c	13.88	13.05	9.18	12.65	8.97	7.16	7.17	4.33
nest-if7.c	16.59	76.24	26.05	74.06	26.07	21.4	65.1	4.08
nest-if8.c	248.64	172.71	564.9	138.48	587.72	99.96	146.52	12.26
nest-if.c	0.12	0.14	0.14	0.12	0.12	0.12	0.13	4.38
nest-len.c	0.25	0.26	0.27	0.25	0.26	0.23	0.26	8.00
NetBSD_g_Ctoc.c								4.04
NetBSD_glob3_iny.c	83.17	344.35	92.54	331.3	99.86		545.04	4.26
NetBSD_loop.c	30.46	49.61	69.48	48.33	67.79	80.44	235.25	4.14
NetBSD_loop_int.c	114.93	177.7	265.72	172.36	270.82	246.69	245.93	3.87
pldi082_unbounded.c								*
pldi08.c	0.16	0.27	0.22	0.3	0.24	0.3	0.24	7.22
rajamani_1.c								*
seesaw.c	791.08	319.37	419.11	315.49	373.04			4.16
sendmail-close-angle.c						498.21	920.09	*
sendmail-mime7to8_ok.c	63.91	82.8	59.91	84.72	60.61	96.85	125.59	4.08
sendmail-mime-fromqp.c	452.72	268.14	1101.93	265.65		96.25	371.78	4.34
seq2.c								3.99
seq3.c								
seq4.c								
seq.c								*
seq-len.c								
seq-proc.c								
seq-sim.c								
seq-z3.c								
simple.c	0.84	0.74	1.37	0.74	1.43	1.06	0.91	*
simple_if.c	0.09	0.09	0.1	0.1	0.09	0.1	0.1	4.06
simple_nest.c	0.08	0.1	0.11	0.11	0.1	0.1	0.1	6.25
split.c								
string_concat-noarr.c								
substring1.c						16	51.5	*

Benchmark	NOOPT	C	CG	CR	CGR	CRL	CGRL	CPAChecker
svd1.c								4.20
svd2.c	292.49	363.93	61.39	381.31	61.07	47.93	47.95	4.45
svd3.c	6.52	18.77	14.49	17.1	15.2	75.32	35.77	4.02
svd4.c								4.06
svd.c								5.11
svd-some-loop.c								9.20
swim1.c								20.14
swim.c								14.99
up2.c					97.94	263.55	699.52	
up3.c				997.4				
up4.c				199.88	103.17	254.66		
up5.c					134.26		773.6	
up.c				45.25	89.2	357.32	635.57	*
up-nd.c				77.3		1093.8		*
up-nested.c	0.11	0.14	0.13	0.11	0.14	0.12	0.12	4.09
xy0.c				89.39	263.1	22.34	40.55	
xy10.c	0.38	0.6	0.86	0.56	0.81	3.58	5.16	4.22
xy4.c				44.76	159.18	968.13		*
xyz2.c				176.69	642.34	775.13		
xyz.c				215.06	623.74			
Solved:	**54**	**61**	**60**	**72**	**64**	**70**	**62**	**63**

Discussion of the Run Time Results

Discussion of Different Configurations

Some observations about specific optimizations can be drawn from the given run time results:

- Enabling CTGDOWN with the given parameters generally leads to significant speedups.
- Refinement state mining hardly impacts performance in a negative way, but can lead to termination (e.g., in `up.c`, `up3.c`, `up4.c`).
- Geometric Generalization mostly leads to higher run times, but can also result in speedups (e.g., in `fig2.c`).
- Lazy refinement can lead to both better and worse performance, but there are benchmarks that can only be solved with lazy refinement (e.g., `substring1.c`).

Comparison with CPAChecker

The comparison with CPAChecker leads to two conclusions:

- IC3-CEGAR is slower than CPAChecker on most benchmarks that both model checkers could solve.
- However, IC3-CEGAR in its best configuration (CR) solved 9 more benchmarks than CPAChecker in total. Additionally, IC3-CEGAR in the CR configuration solved 26 benchmarks that CPAChecker could not solve within the time limit of 1200 seconds. CPAChecker solved only 18 benchmarks that IC3-CEGAR could not solve within the time limit of 1200 seconds.

Discussion of the Reasons of Termination for Selected Benchmarks

In this section, we do a short analysis of why certain benchmarks fail to terminate in some configurations, but terminate (fast) in others.

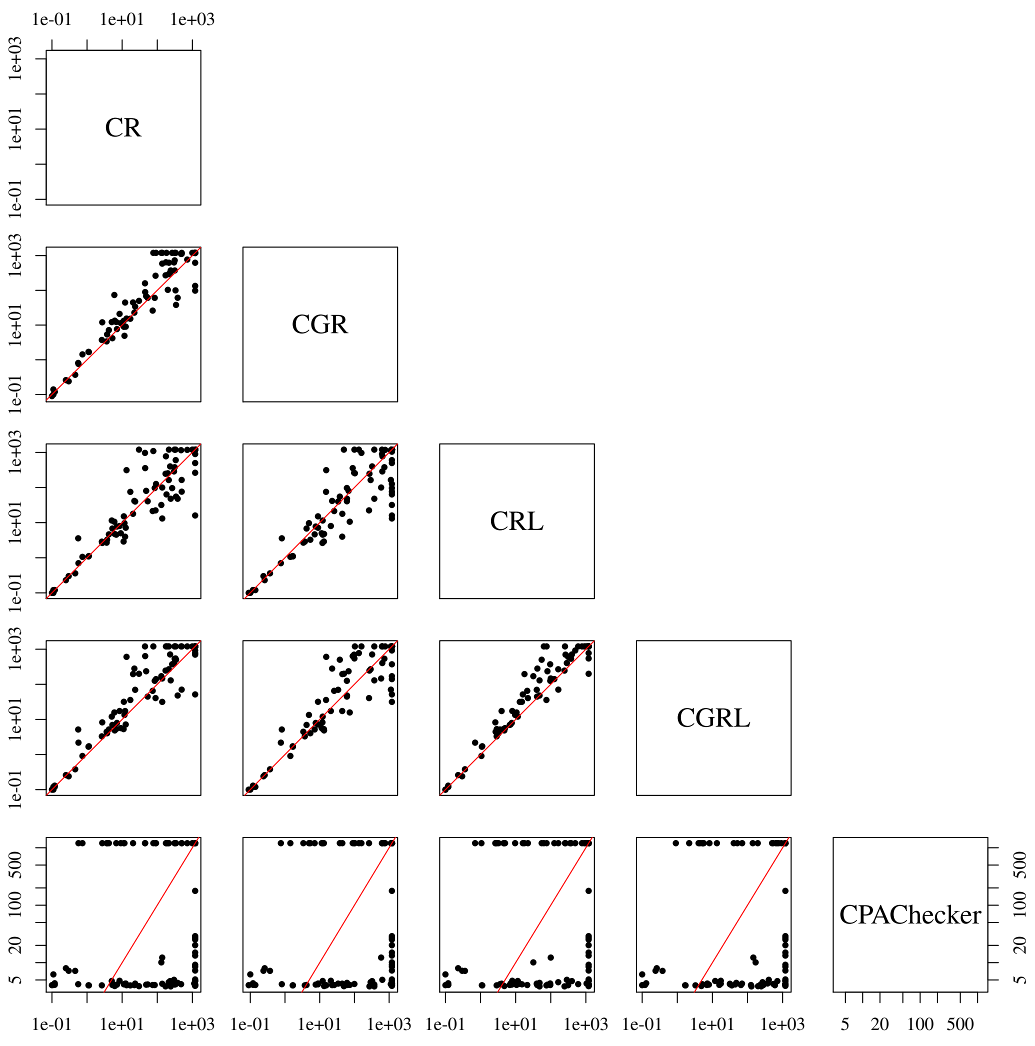
xy0.c

First, we turn the attention to the benchmark `xy0.c`. This benchmarks fails to terminate without refinement state mining, but terminates quite fast in any configuration where the algorithm is enabled.

The reason for non-termination without refinement state mining lies in the fact that the domain diverges without it.

Without refinement state mining, in configuration **C**, the abstract domain gets repeatedly refined with interpolants around concrete states that have a program counter value of 6. These states look like:

Figure 4.1: Scatter plot matrix depicting IC3-CEGAR (all configurations with Refinement state mining and CTGDOWN) and CPAChecker run times. Each sub-plot compares two configurations against each other. Each point corresponds to one benchmark; the position of the point is determined by the run times in the two configurations. The red line is the main diagonal. A point to the upper left side of the main diagonal indicates that the upper configuration performed better on the given benchmark. A point to the lower right side of the main diagonal indicates that the right configuration performed better on the given benchmark. All timeouts and failures are indicated by a point at the upper or right border of the plot.



```
main.x=1, main.y=0, pc=6
main.x=2, main.y=1, pc=6
main.x=3, main.y=2, pc=6
main.x=4, main.y=3, pc=6
...
```

The domain is refined with predicates of the form $x \leq 1$, $x \geq 1$, $y \leq 0$, $y \geq 0$, $x \leq 2$, $x \geq 2, \ldots$. This pattern continues until timeout, without any progress being made.

With refinement state mining turned on (e.g., in configuration **CR**), the state analysis will start when at least three concrete states with a program counter value of 6 are found. The domain will subsequently be refined with two linear inequalities of the form $x - y \leq 1$ and $x - y \geq 1$. Using these predicates, the algorithm will exclude all of the concrete states that turned up without refinement state mining in a single (or very few) strengthenings of a frame. Thus, IC3-CEGAR can finish fast on this instance, provided that the right predicate to refine with is found.

substring1.c

Next, we turn the attention to the benchmark `substring1.c`. This benchmark fails to terminate unless lazy refinement is enabled (which admits a number of spurious transitions in a proof obligations trace).

Without Lazy Refinement (and even with refinement state mining enabled), the abstract domain diverges again. The abstract domain is repeatedly refined with interpolants around concrete states that have a program counter value of 14. These states look like:

```
main.i=-19, main.j=83, main.k=-1,
main.i=-18, main.j=84, main.k=-1,
main.i=-17, main.j=85, main.k=-1,
main.i=-16, main.j=86, main.k=-1,
main.i=-15, main.j=87, main.k=-1,
main.i=-14, main.j=88, main.k=-1,
...
```

The domain is subsequently refined with inequalities describing these concrete states until the program runs into the timeout.

Even refinement state mining does not help in this case. The linear equalities found by the state analysis are of the form $i + k - j = -103$ and $k + j - i = 101$. A more sophisticated algorithm like the one described in [43] might help in this case, but the approach is currently not implemented.

Lazy refinement still leads to termination on this benchmark. With lazy refinement, the algorithm simply admits spurious transitions into the abstractions of such concrete states, and discovers that the trace never reaches the initial condition, thus enabling IC3-CEGAR to exclude all such concrete states without refinement.

Other Statistics

In this section, we present statistics about other aspects of IC3-CEGAR runs. All data sets used in the figures are available in full form in the appendix. All of the following statistics present results for instances where IC3-CEGAR successfully terminates.

Frontier Level at Termination

We present a plot depicting the approximate density of frontier levels for instances where IC3-CEGAR successfully terminates in figure 4.2. About half of the benchmarks terminate with a frontier level lower than 21, but the maximum observed frontier level is 107. The frontier level correlates with the run time, with the correlation to the run time ranging from 68% to 86% on different configurations.

The mean last frontier level across all configurations is 20 (rounded). The standard deviation for the last frontier level across all configurations is 18 (rounded). For the full data for our benchmark set, see appendix A.4.

Number of Predicates in the Domain at Termination

We present a plot depicting the approximate density of the number of predicates in the abstract domain for instances where IC3-CEGAR successfully terminates in figure 4.3. About half of the benchmarks terminate with a number of predicates less than 50, but the maximum observed number of predicates for a terminating instance is 251. The number of predicates in the domain correlates with the run time, with correlations ranging from 68% to 84% for the observed configurations.

The mean final number of predicates across all configurations is 51 (rounded). The standard deviation for the final number of predicates across all configurations is 38 (rounded). For the raw data, see appendix A.6.

In general, a smaller number of predicates in the domain is preferable, as the overhead of abstraction and generalization grows with the number of predicates in the abstract domain.

Number of Interpolating Refinement Calls

The number of interpolating refinement calls elucidates how many times a spurious transition was discovered that led to the refinement of the domain. The number of refinement calls correlates strongly with the number of final predicates in the domain, with correlations between 91% and 95% for the respective configurations.

We present a plot depicting the approximate density of the number of interpolating refinement calls for instances where IC3-CEGAR successfully terminates in figure 4.4. About half of the terminating benchmarks need less than twelve refinements of the domain, but the maximum number of interpolating refinement calls for a terminating instance is 84.

The mean number of interpolating refinement calls across all configurations is 13 (rounded). The standard deviation for the number of interpolating refinement calls across all configurations is 12 (rounded). For the raw data, see appendix A.7.

Figure 4.2: Density plot depicting IC3-CEGAR frontier levels at termination. The plot indicates that about half of the benchmarks terminate with a frontier level lower than 21, but frontier levels can also rise up to 109 on the given benchmark set and benchmarking configuration.

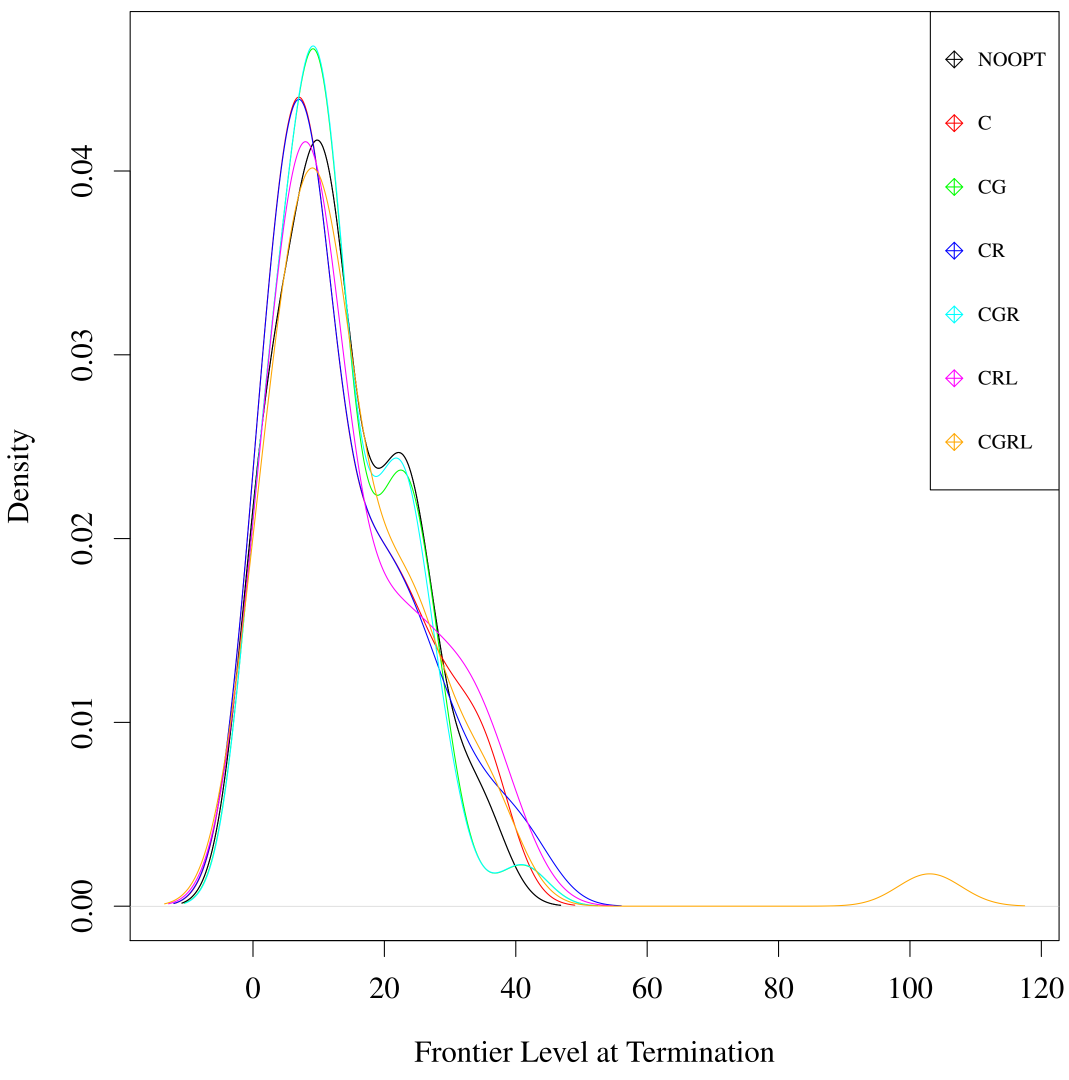


Figure 4.3: Density plot depicting the number of abstract domain predicates for instances where IC3-CEGAR terminates. The plot indicates that about half of the benchmarks terminate with a number of predicates less than 50, but the number of predicates in the final abstract domain can exceed 200.

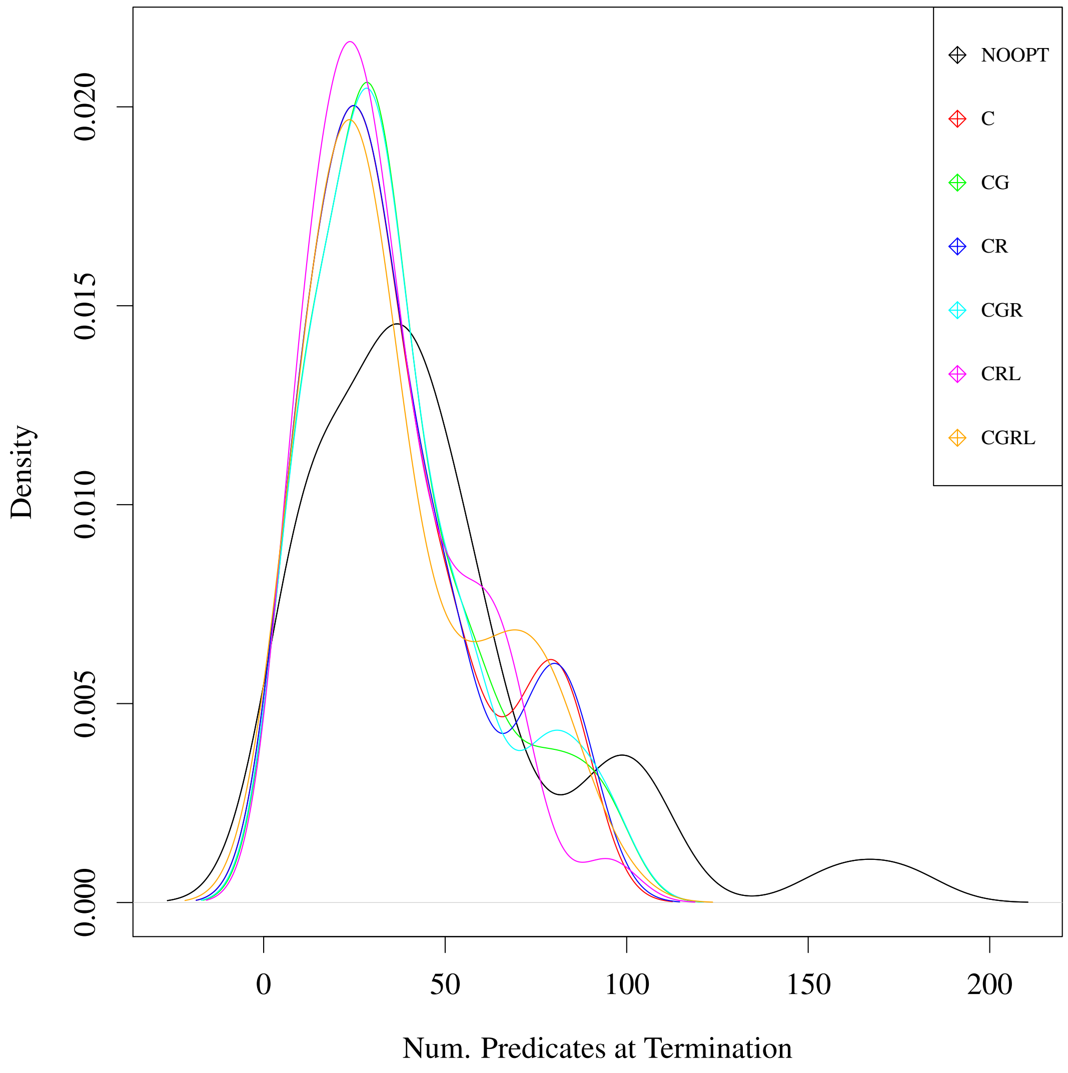
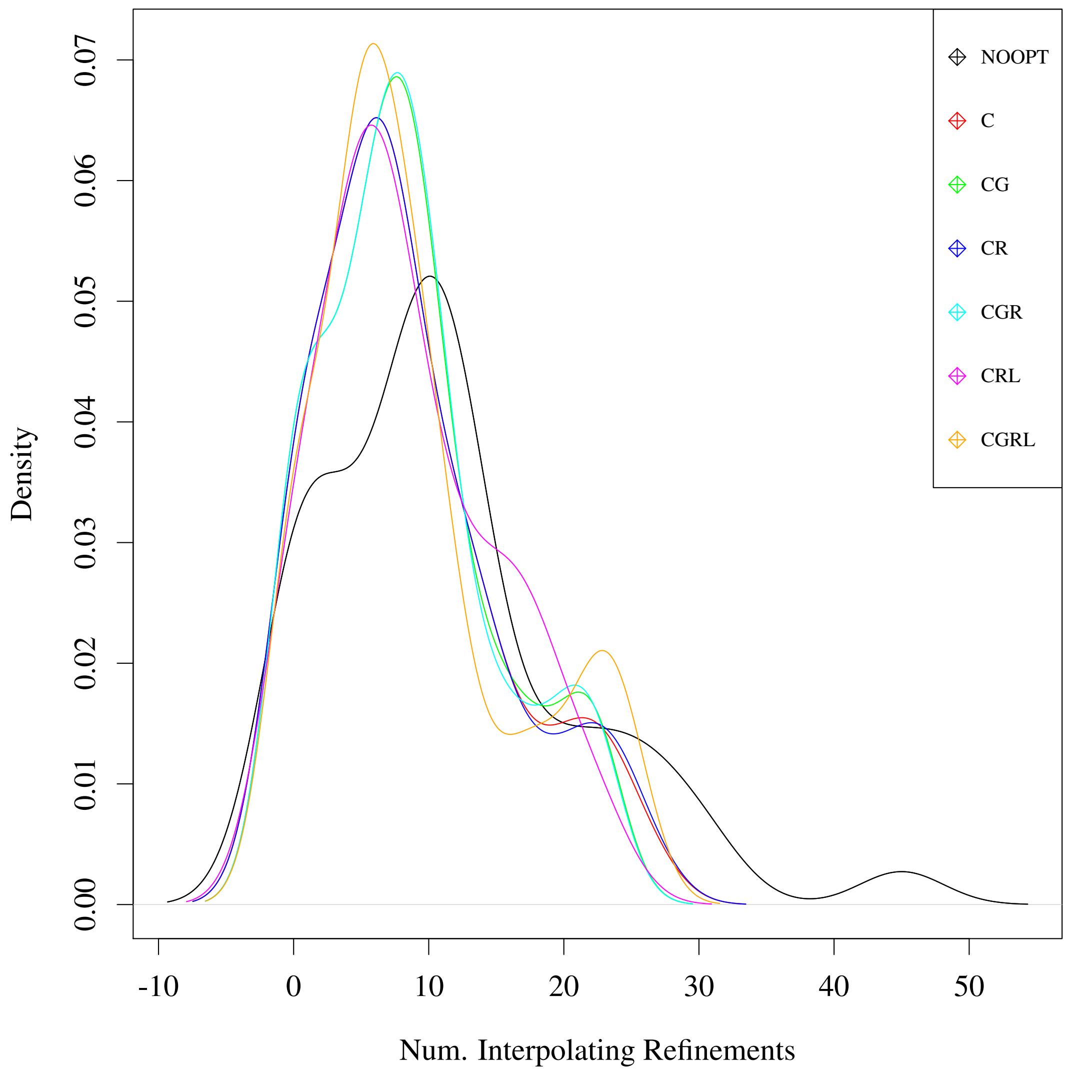


Figure 4.4: Density plot depicting the average number of interpolating refinement calls for instances where IC3-CEGAR terminates. The plot indicates that about half of the terminating benchmarks use less than twelve interpolating refinement calls, but the number of refinement calls can exceed 80 in our benchmark set.



Average Number of Literals/Clause in the Fixed Point

The average number of literals per clause in an important factor for judging the performance of the generalization procedure (such as MIC with CTGDOWN). The less number of literals per clause, the better the generalization procedure works. Good generalization procedures will generally lead to fewer literals per clause and to speed-ups of the algorithm.

We present a plot depicting the approximate density of the average number of literals for instances where IC3-CEGAR successfully terminates in figure 4.5. The average number of literals per clause in the fixed points is about three. Considering that the average number of predicates in the domain is around 50 and goes as high as 251 in our set of benchmarks, this shows that the used generalization procedure (MIC with CTGDOWN) does indeed serve their purpose well.

As seen in figure 4.5, the configurations with geometric generalization enabled emit a lower average number of literals per clause in the fixed point than those configurations without geometric generalization (save NOOPT). Still, these configurations are, in general, slower. One reason for this might be that the additional overhead for geometric generalization leads to significantly greater run times that are not balanced by the higher quality of the clause generalization.

The mean average number of literals per clause in the fixed point across all configurations is 2.96. The standard deviation for the average number of literals per clause in the fixed point across all configurations is 0.59. For the raw data see appendix A.2

Number of Successful Refinement State Mining Attempts

We present a plot depicting the number of successful refinement state mining attempts (i.e., a mining attempt wherefrom a predicate was extracted) for different configurations on our benchmark set in figure 4.6. The number of refinement state minings is surprisingly small for most instances; still, the technique sometimes as a decisive impact on the performance of the algorithm. For the raw data, see appendix A.8.

Other Statistics

Graphics and tables that present other statistical figures about the IC3-CEGAR runs are presented in the appendix (A). We sum up the most important values about these data here:

- The **maximal depth of the proof obligations trace** correlates with the run time (correlation $> 51\%$ on all configurations) and can exceed 150. The mean maximal depth of a proof obligations trace across all configurations is 15 (rounded), and the corresponding standard deviation is 17 (rounded). About half of all successful IC3-CEGAR runs on the benchmark set have a maximal proof obligations trace depth of less than 10.

  Proof obligation trace depths beyond 100 prove that IC3-CEGAR is able to find deep counterexamples. For the raw data, see appendix A.5.

- The **percent of the run time spent in the SMT solver** correlates weakly with the total run time (correlation $> 24\%$ across all configurations). On average, about 70% of the total run time is spent in the SMT solver (arithmetic mean weighted with the run time).

Figure 4.5: Density plot depicting the average number of literals per clause in the fixed point for instances where IC3-CEGAR terminates. The plot indicates that the fixed points' clauses usually comprise around three literals. Observe that the configurations with geometric generalization enabled emit a lower average number of literals per clause in the fixed point.

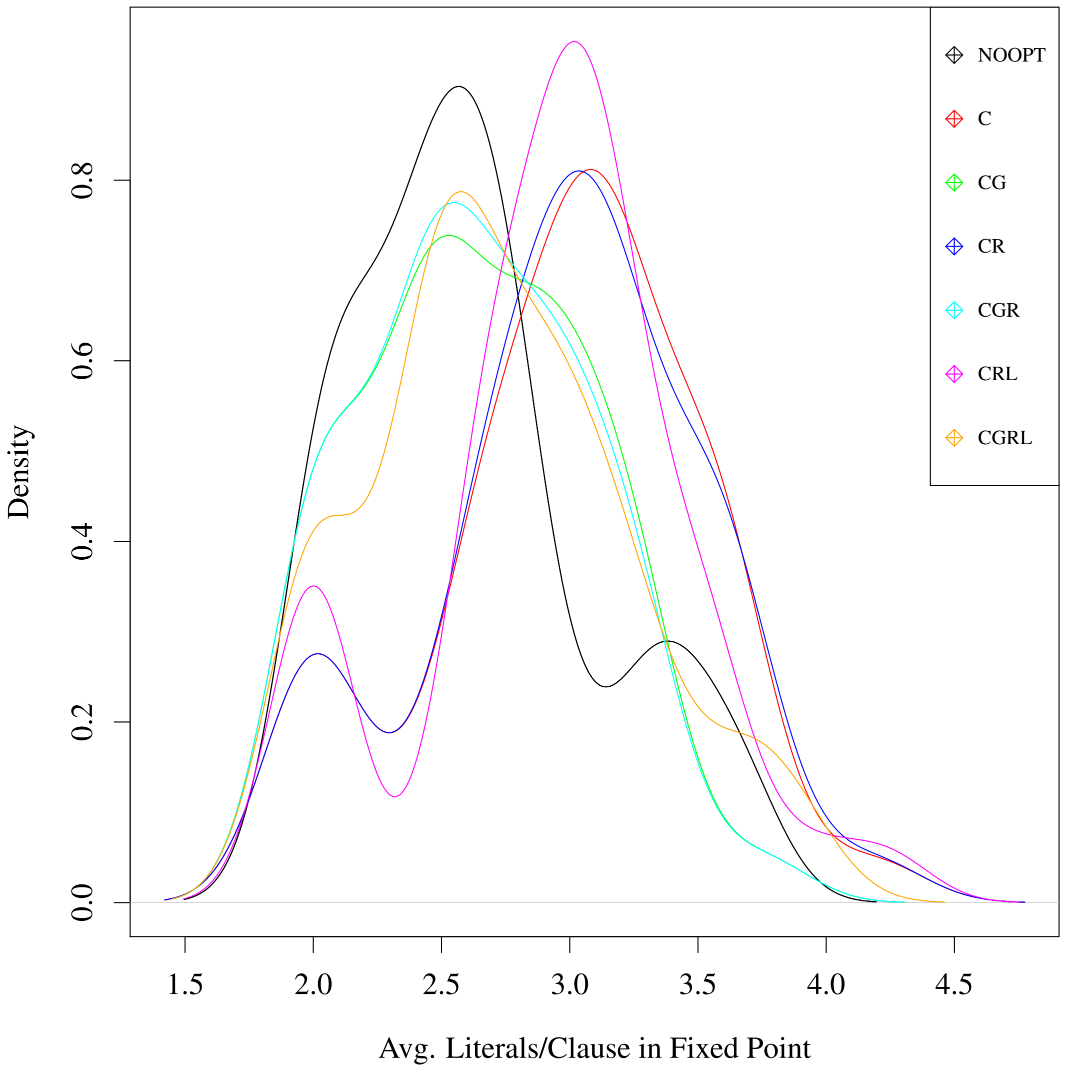
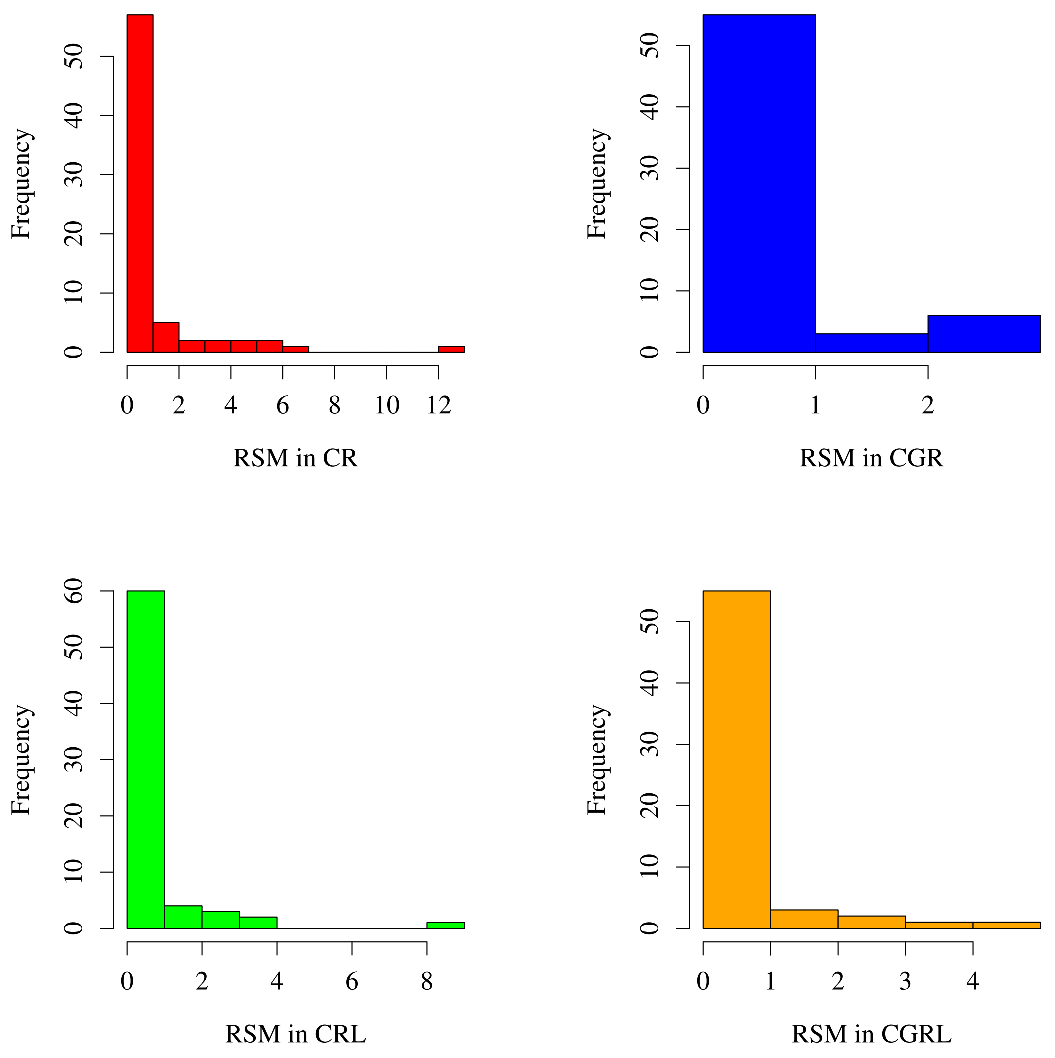


Figure 4.6: Histograms depicting the number of successful refinement state mining attempts for different configurations.



Since the burden of the proof lies on the SMT solver, this indicates that there is room for a performance acceleration of up to 30% by optimization of the IC3-CEGAR framework.

For the raw data, see appendix A.9.

CHAPTER 5

Related Work

5.1 QF_BF Model Checking with Property Directed Reachability

The article [48] deals with extending Property Directed Reachability (a synonym for IC3) to model checking transition relations over quantifier-free bit-vector systems. Our approach is different in that it deals with transition relations over quantifier-free linear integer arithmetic.

Additionally, the article [48] does not use an explicit abstract domain or counterexample-guided abstraction refinement. This alone sets our approach quite far apart from [48].

The approach extends IC3 in the finite-state case by reasoning over sets of states that are described by integer polytopes. The simplest case of an integer polytope is a conjunction of static interval constraints on variables. In the QF_BV approach, such a conjunction is called a cube.

In order to generalize abstract states in case that initiation and consecution hold, the authors use a technique called “interval simulation”. In the simplest version of the algorithm, this consists of a set of rules that aid in determining maximum possible bounds for the intervals of the cubes such that initiation and consecution still hold.

The approach is then extended to abstract states that are conjunctions of arbitrary linear inequalities, i.e., general polytopes. In order to generalize polytopes, the authors present the RESHAPE algorithm, which expands the area covered by the polytope such that it covers more concrete states.

Since the authors deal with programs that contain a finite number of variables, they can reduce their problems to classical finite-state IC3. They compare an implementation of their approach to the classical IC3, showing that the implementation with polytopes can beat the original IC3 implementation.

5.2 Software Model Checking via IC3

The article [14] deals with extending IC3 to model checking transition relations over quantifier-free linear integer arithmetic, similarly to our approach. However, the approach does not use an explicit abstract domain or counterexample-guided abstraction refinement.

In this approach, IC3 is lifted to SMT by defining a cube as a conjunction of theory atoms. The IC3 algorithm then proceeds in the standard manner, trying to find a proof obligation by seeing if the error can be reached from the frontier. In case this holds, a proof obligation is extracted at the level of the frontier.

While in our approach, a new proof obligation is generated by extracting a concrete predecessor of the current proof obligation and adding it at the previous level, the approach described in [14] does not deal with "concrete" states at all. Instead, a proof obligation is a tuple $\langle s, i \rangle$ where s is an cube (i.e., an arbitrary conjunction of theory atoms) and i is a level. Proof obligations are generally extracted by calculating under-approximations of pre-images of cubes or the error. A procedure that calculates under-approximates of pre-images is described in the paper.

The authors then continue to modify the IC3 algorithm in order to improve performance, resulting in an algorithm called TREE-IC3. To this end, they abandon the concept of checking single clauses for consecution. Instead, they perform an unrolling of the abstract reachability tree. Each node in the abstract reachability tree is associated with a set of clauses that describe an over-approximation of the states reachable at this node. Proof obligations by trying to exclude possible paths from the initial condition to each branch of the abstract reachability tree, one branch at a time. When the algorithm determines that a node on a single branch contains a subset of the clauses of a previous node on the same branch, this branch is assumed to be closed.

The sets of clauses that are computed on a closed branch are actually interpolants for the path, like the interpolants in McMillan's "lazy abstraction with interpolants" approach ([38]). Since the consecution checks and fixed point checks are abandoned in this highly modified algorithm, the approach is actually more akin to this approach than to IC3.

The authors compare their implementation with various other model checkers, among them CPAChecker. They show that an adaptation of their TREE-IC3 algorithm, which combines the algorithm described above with proof-based interpolants generated by an SMT solver, beats the other model checkers in terms of the number of benchmarks solved.

5.3 Generalized Property Directed Reachability

The article [32] describes a modification of IC3 that can be used for model checking Boolean programs, generalizing IC3 to non-linear predicate transformers and generalizing IC3 to linear real arithmetic. (IC3 is called "Property Directed Reachability", or "PDR", in the paper.)

A non-linear predicate transformer is a predicate transformer (such as the $post$ operator) that requires two or more "input" states in order to calculate one "output" state. Such transformers are necessary in order to find fixed points for programs whose reachable state space is described using recursive predicates. The article [32] describes how to modify the IC3 algorithm such that it finds fixed points for also works on non-linear predicate transformers. This is done by constructing a proof obligations tree instead of a proof obligations priority queue.

The authors first extend IC3 in order to find fixed points for systems (recursive) Boolean constraints. Then they go on to show how the approach could be modified in order to find fixed points for systems of constraints over the theory of linear real arithmetic. In order to generalize clauses that exclude states from certain levels, the authors use an interpolation approach that is based on Farkas' lemma.

The authors go on to show that their generalized version of IC3, together with interpolation based on Farkas' lemma, is powerful enough to decide safety for timed push-down systems.

CHAPTER 6

Conclusion

In this thesis, we presented IC3-CEGAR, an incremental, inductive software model checker based on IC3 ([10]).

We presented methods for obtaining a symbolic transition relation from a software program.

In order to apply the principle of IC3 on the infinitely large state space, we described how a Boolean predicate abstraction framework could be used to reduce the state space to the finite-state case.

In order to refine the abstract domain, we described a CEGAR-based framework ([18, 19]) that uses interpolation ([20, 21]) in order to eliminate spurious transitions.

Additionally, we presented the technique of Refinement State Mining, which extracts predicates by attempting to describe states previously used for refinement by single predicates.

We described how to adapt the method of lifting proof obligations as described in [13, 24] to IC3-CEGAR, including the handling of non-deterministic assignments.

Existing as well as novel approaches for generalizing abstract states were discussed: We adapted the generalization methodologies described in [10, 30] to IC3-CEGAR and extended them by geometric generalization techniques.

We implemented an abstraction framework based on AllSAT calls ([33]) and adapted it so it could take full advantage of the generalization techniques mentioned above.

Finally, we evaluated our model checker on three benchmark suites (as far as the benchmarks were suitable for linear integer arithmetic software model checking) and showed that IC3-CEGAR performs competitively in comparison with the verification tool CPAChecker ([7]).

Future Challenges

Future challenges include:

- Extending IC3-CEGAR to verify transition relations over non-linear or bit-vector arithmetics: Currently, IC3-CEGAR is cut out for linear integer arithmetic or linear rational arithmic.

- Extending IC3-CEGAR to the verification of parallel or distributed algorithms: The transition relations we currently verify using IC3-CEGAR are strictly single-threaded.
- Implementing optimizations on the transition relation that will allow the verification of much larger code bases: Large code-bases result in large transition relations, which overwhelm the SMT solver. Future work will need to address the problem of how to reduce the size of the transition relation such that an SMT solver can handle it efficiently. Usually, it is not necessary to load all of the transition relation for a single consecution check.
- Implementing different refinement techniques for the abstract domain: Currently, there are two main refinement strategies: Interpolation-based refinement and Refinement State Mining. Other techniques might yield even better results.
- Extending IC3-CEGAR such that it can handle recursive function calls: Recursive function calls are currently not supported.
- Implementing more sophisticated invariant detection techniques. An example would be the approach in [43].

Bibliography

[1] Aws Albarghouthi and Kenneth L. McMillan. Beautiful Interpolants. In *Proceedings of the 25th international conference on Computer Aided Verification*, CAV'13, pages 313–329, Berlin, Heidelberg, 2013. Springer-Verlag.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough Static Analysis of Device Drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006.

[3] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough Static Analysis of Device Drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 73–85, New York, NY, USA, 2006. ACM.

[4] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In EerkeA. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2004.

[5] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

[6] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In John Launchbury and John C. Mitchell, editors, *POPL*, pages 1–3. ACM, 2002. ACM SIGPLAN Notices 37(1), January 2002.

[7] Dirk Beyer and M. Erkan Keremoglu. CPAChecker: a tool for configurable software verification. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 184–190, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[9] Aaron R. Bradley. IC3 Reference Implementation. `https://github.com/arbrad/IC3ref/`.

[10] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Aaron R. Bradley. Understanding IC3. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 1–14, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[13] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 135–143, Austin, TX, 2011. FMCAD Inc.

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

[15] Cimatti, Alessandro and Griggio, Alberto and Schaafsma, Bastiaan and Sebastiani, Roberto. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[16] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[17] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[18] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[19] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[20] William Craig. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *The Journal of Symbolic Logic*, 22(3):pp. 250–268, 1957.

[21] William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic*, 22(3):pp. 269–285, 1957.

[22] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *Proceedings of the 11th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'10, pages 129–145, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[25] E. Allen Emerson and Edmund M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, UK, 1980. Springer-Verlag.

[26] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.

[27] Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. In *JSAT*, volume 8, pages 1–27. Delft University in cooperation with IOS Press, 2012.

[28] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Dagger Benchmarks Suite. `http://www.cfdvs.iitb.ac.in/~bhargav/dagger.php`.

[29] Ashutosh Gupta and Andrey Rybalchenko. InvGen Benchmarks Suite. `http://pub.ist.ac.at/~agupta/invgen/`.

[30] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better Generalization in IC3. In *FMCAD*, 2013.

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

[32] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 157–171, Berlin, Heidelberg, 2012. Springer-Verlag.

[33] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT techniques for fast predicate abstraction. In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 424–437, Berlin, Heidelberg, 2006. Springer-Verlag.

[34] Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3(3):243–263, 1974.

[35] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[36] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[37] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, November 2005.

[38] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 123–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[39] Terence Parr. ANTLR4. `http://www.antlr.org/`.

[40] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 1977.

[41] Amir Pnueli. The temporal semantics of concurrent programs. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 1979.

[42] Jean-Pierre Queille and Joseph. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982.

[43] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 574–592, Berlin, Heidelberg, 2013. Springer-Verlag.

[44] Joseph Sifakis. Global and local invariants in transition systems. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 510–522. Springer Berlin Heidelberg, 1982.

[45] Joseph Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, 18(3):227 – 258, 1982.

[46] Alan M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[47] Georg Weissenbacher. Interpolant strength revisited. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 312–326, Berlin, Heidelberg, 2012. Springer-Verlag.

[48] Tobias Welp and Andreas Kuehlmann. QF_BV model checking with property directed reachability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 791–796, San Jose, CA, USA, 2013. EDA Consortium.

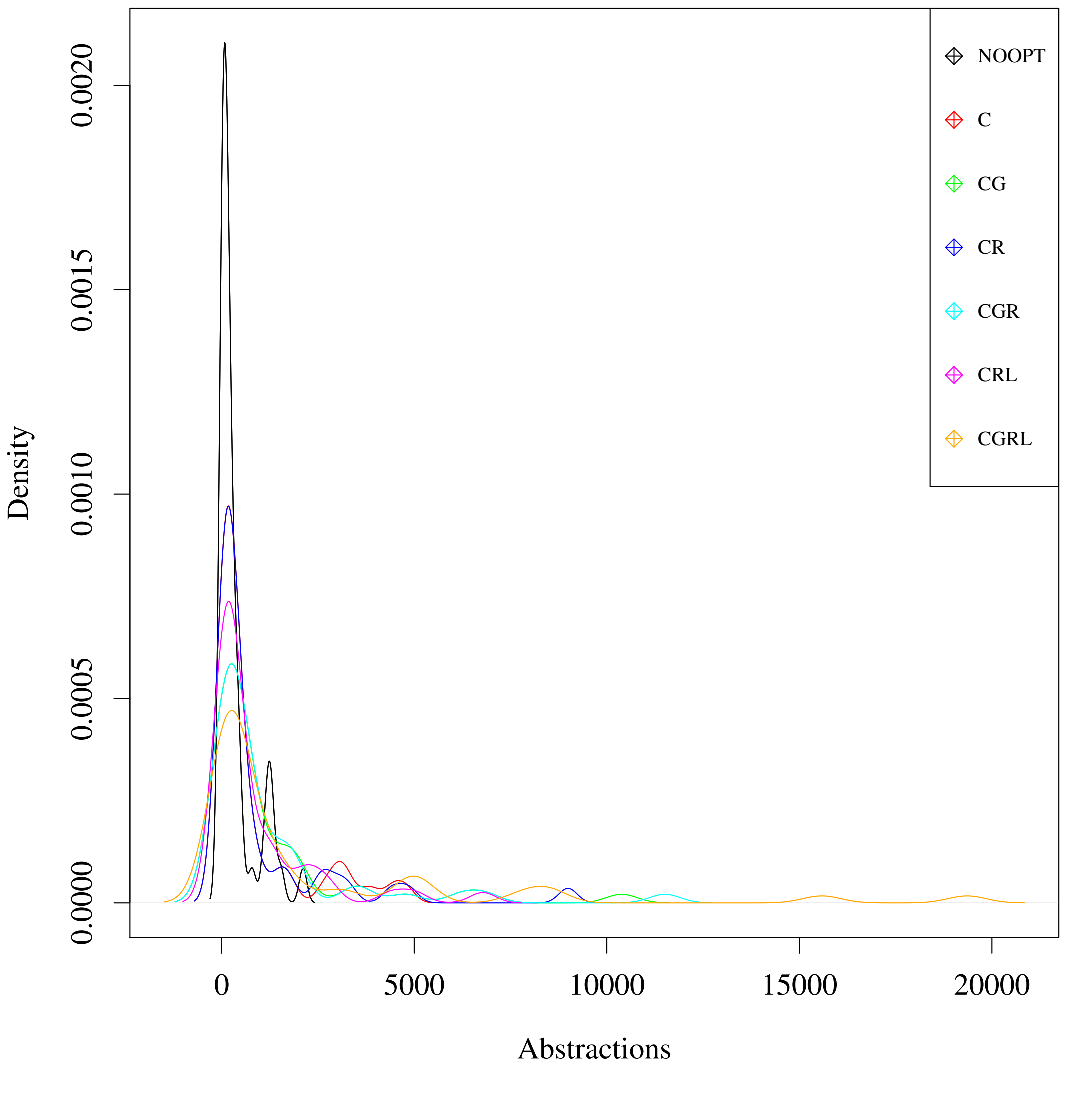
APPENDIX A

Additional Statistics

A.1 Number of Abstractions

The **number of abstractions of concrete states** correlates highly with the run time (correlation $> 84\%$ on all configurations) and can exceed 28000. The mean number of abstractions across all configurations is 2500 (rounded), and the corresponding standard deviation is 4343 (rounded). About half of all successful IC3-CEGAR runs on the benchmark set use less than 586 abstractions. See figure A.1 for a graphical representation and table A.1 for the raw data.

Figure A.1: Density plot depicting the number of abstractions of concrete states for instances where IC3-CEGAR terminates.



Num. Abstractions	NOOPT	C	CG	CR	CGR	CRL	CGRL
apache-escape-absolute.c	656	2121	4038	2121	4038	3755	
apache-get-tag.c		1889	5127	1889	5127	2426	5909
barbr.c							
barbrprime.c							
bind_expands_vars2.c	54	242	203	242	203	343	403
bkley.c	1286	2664	3599	2664	3599	4405	8598
bk-nat.c	1201	2992	6846	2992	6846	2729	4961
bound.c	159	501	1596	501	1596	449	7744
cars.c							
dillig01.c	126	297	773	297	773	291	335
dillig03.c	20	37	55	37	55	40	152
dillig05.c		3161	24616	7195		2382	
dillig07.c	394	142	219	142	219	175	244
dillig12.c				4789			
dillig15.c				1195	2172		7111
dillig17.c	1532	1596	1762	1596	1762	1159	1363
dillig19.c		4734	18247	5352	8632	1385	4128
dillig20.c	2113	3289	4774	3289	4774	2684	5141
dillig25.c	273	244	3423	244	3423	491	834
dillig28.c	339	916	2031	916	1657	1491	2900
dillig32.c	788	752	796	752	796	1148	5062
dillig33.c	1747	8652		14791		16986	
dillig37.c	486	506	1811	506	1811	213	952
down.c	167	3847	1802	8993	1802	2025	19353
efm.c							
ex1.c	83	503	606	503	606	6782	15586
ex2.c		4278	10657	4279		5614	
fig1a.c				13500	28101	6505	3736
fig2.c			14261		14261	13912	
fragtest_simple.c							

Num. Abstractions	NOOPT	C	CG	CR	CGR	CRL	CGRL
gulv.c							
gulv_simp.c	1108	372	980	372	980	320	732
gulwani_cegar1.c	110	273	304	273	304	296	326
gulwani_cegar2.c	225	235	600	235	600	498	642
gulwani_fig1a.c	13	46	35	46	35	46	35
hsort.c							
hsortprime.c	362	4781	14288	4781	14288	10704	
id_build.c	327	287	389	287	389	256	262
ken-imp.c		24409	8623	5827		846	2250
large_const.c							
lifnat.c							
lifnatprime.c							
lifo.c							
MADWiFi-encode_ie_ok.c							
mergesort.c							
nested1.c	33	48	99	48	99	48	99
nested2.c	33	48	99	48	99	48	99
nested3.c	103	168	192	168	192	129	262
nested4.c	146	224	321	224	321	202	263
nested5.c	217	423	420	423	420	577	1538
nested6.c	748	3430	3775	3430		1445	
nested7.c							
nested8.c		5854		2566		754	3401
nested9.c		2443	2790	2443		2964	3030
nested.c	1	2	3	2	3	2	3
nest-if1.c	117	95	475	95	475	95	357
nest-if2.c	122	412	535	412	535	134	249
nest-if3.c	125	195	335	195	335	224	255
nest-if4.c	208	276	482	276	482	194	245
nest-if5.c	65	468	229	468	229	378	586

Num. Abstractions	NOOPT	C	CG	CR	CGR	CRL	CGRL
nest-if6.c	114	107	85	107	85	62	75
nest-if7.c	230	1762	621	1762	621	618	1785
nest-if8.c	1237	3081	10387	2536	11509	2205	3254
nest-if.c	1	2	3	2	3	2	3
nest-len.c	1	2	3	2	3	2	3
NetBSD_g_Ctoc.c							
NetBSD_glob3_iny.c	339	3950	1629	3950	1629		4908
NetBSD_loop.c	267	1375	1788	1375	1752	2206	8320
NetBSD_loop_int.c	427	4427	6235	4427	6235	5006	4753
pldi082_unbounded.c							
pldi08.c	7	26	19	26	19	26	19
rajamani_1.c							
seesaw.c	2961	3281	5265	3281	4863		
sendmail-close-angle.c						5937	10692
sendmail-mime7to8_ok.c	446	882	821	882	821	1228	1664
sendmail-mime-fromqp.c	1462	3286	15397	3286		1375	7064
seq2.c							
seq3.c							
seq4.c							
seq.c							
seq-len.c							
seq-proc.c							
seq-sim.c							
seq-z3.c							
simple.c	48	53	118	53	118	83	80
simple_if.c	1	2	3	2	3	2	3
simple_nest.c	1	2	3	2	3	2	3
split.c							
string_concat-noarr.c							
substring1.c						529	1767

Num. Abstractions	NOOPT	C	CG	CR	CGR	CRL	CGRL
svd1.c							
svd2.c	1294	4778	1207	4858	1207	892	1006
svd3.c	85	378	422	378	422	1291	935
svd4.c							
svd.c							
svd-some-loop.c							
swim1.c							
swim.c							
up2.c					3619	5674	16133
up3.c				14290			
up4.c				4949	3208	5034	
up5.c					3901		15672
up.c				1597	2905	7087	16748
up-nd.c				1928		22105	
up-nested.c	1	2	3	2	3	2	3
xy0.c				3858	9345	1271	2080
xy10.c	15	38	65	38	65	245	401
xy4.c				2191	7289	22949	
xyz2.c				4331	19966	16384	
xyz.c				5069	17417		

A.2 Average Number of Literals/Clause in the Fixed Point

In table A.2, we present the raw data used for the graphics in section 4.3.

Avg. Num. Literals/Clause	NOOPT	C	CG	CR	CGR	CRL	CGRL
apache-escape-absolute.c	2.14573	2.76834	2.46067	2.76834	2.46067	2.82967	
apache-get-tag.c		3.34965	3.32	3.34965	3.32	3.14719	2.98261
barbr.c							
barbrprime.c							
bind_expands_vars2.c	2.61905	3.42308	2.5	3.42308	2.5	3.55319	2.8
bkley.c	3.3324	3.53358	3.16	3.53358	3.16	3.94709	3.6445
bk-nat.c	2.80287	3.59912	3.34375	3.59912	3.34375	3.47619	3.26812
bound.c	2.44444	3.1875	3.03704	3.1875	3.03704	3.22222	3.2
cars.c							
dillig01.c	2.73684	3.19512	3.04167	3.19512	3.04167	2.93939	2.37931
dillig03.c	2.14286	2.33333	2.16667	2.33333	2.16667	2	2.5
dillig05.c		3.64903	3.60838	3.61728		3.22222	
dillig07.c	3.03333	3.15	2.70588	3.15	2.70588	3.125	2.52941
dillig12.c				3.70796			
dillig15.c				3.56731	3.51111		3.04523
dillig17.c	3.46154	3.50311	3.22137	3.50311	3.22137	3.00694	2.92308
dillig19.c		4.40509	3.934	4.34594	3.75234	4.00568	3.74545
dillig20.c	3.69304	3.5991	3.75602	3.5991	3.75602	3.65581	3.68639
dillig25.c	2.77778	2.61538	2.99375	2.61538	2.99375	2.725	2.66038
dillig28.c	2.66216	2.9898	2.77451	2.9898	2.65979	3.09565	3.21831
dillig32.c	3.30104	3.0087	2.90625	3.0087	2.90625	2.65432	3
dillig33.c	3.84274	4.01308		4.32951		3.868	
dillig37.c	3.53365	3.55738	3.25926	3.55738	3.25926	2.66667	3.125
down.c	2.8125	3.78947	3.12069	3.70146	3.12069	3.45109	3.79118
efm.c							
ex1.c	3.36765	4.19403	3.34783	4.19403	3.34783	4.24213	3.91264
ex2.c		3	2.7601	3		3	
fig1a.c				3.52933	4.0214	3.37987	3.26241
fig2.c			3.49577		3.49577	3.59057	
fragtest_simple.c							

Avg. Num. Literals/Clause	NOOPT	C	CG	CR	CGR	CRL	CGRL
gulv.c							
gulv_simp.c	3.24497	3.08696	2.82222	3.08696	2.82222	2.93333	2.94737
gulwani_cegar1.c	2.66667	3.17241	2.58824	3.17241	2.58824	3.23333	2.58824
gulwani_cegar2.c	2.70667	2.8125	2.89552	2.8125	2.89552	3.05263	2.86538
gulwani_fig1a.c	2.5	3	2.5	3	2.5	3	2.5
hsort.c							
hsortprime.c	3.28	3.51648	3.30483	3.51648	3.30483	3.85808	
id_build.c	2.34483	3	2.39286	3	2.39286	3	2.8
ken-imp.c		4.75662	4.5343	5.08549		3.79747	3.91589
large_const.c							
lifnat.c							
lifnatprime.c							
lifo.c							
MADWiFi-encode_ie_ok.c							
mergesort.c							
nested1.c	2.14286	2.66667	2	2.66667	2	2.66667	2
nested2.c	2.14286	2.66667	2	2.66667	2	2.66667	2
nested3.c	2.59259	2.625	2.53333	2.625	2.53333	2.6875	2.38889
nested4.c	2.51282	2.86364	2.42105	2.86364	2.42105	2.90476	2.45
nested5.c	2.95833	3.24138	2.65517	3.24138	2.65517	3.30769	3.15217
nested6.c	3.175	3.28257	2.91051	3.28257		3	
nested7.c							
nested8.c		4.09091		3.72388		3.3125	3.50732
nested9.c		3.804	3.99259	3.804		3.63251	3.05747
nested.c	2	2	2	2	2	2	2
nest-if1.c	2.43243	2.63636	2.51852	2.63636	2.51852	2.63636	2.6087
nest-if2.c	2.15	3	2.37931	3	2.37931	2.6	2.26667
nest-if3.c	2.25	2.71429	2.56522	2.71429	2.56522	2.76471	2.47059
nest-if4.c	2.72222	2.84211	2.71875	2.84211	2.71875	2.82353	2.42105
nest-if5.c	2.17647	2.9	2.35294	2.9	2.35294	2.88235	2.48718

Avg. Num. Literals/Clause	NOOPT	C	CG	CR	CGR	CRL	CGRL
nest-if6.c	2.4	3	2.25	3	2.25	3	2.875
nest-if7.c	2.47945	3.33514	2.88	3.33514	2.88	3.03371	3.44355
nest-if8.c	2.73854	3.21073	3.13827	2.90262	3.02311	3.20106	2.6
nest-if.c	2	2	2	2	2	2	2
nest-len.c	2	2	2	2	2	2	2
NetBSD_g_Ctoc.c							
NetBSD_glob3_iny.c	1.94949	2.9901	2.10769	2.9901	2.10769		2.51592
NetBSD_loop.c	2.6129	3.64706	2.91096	3.64706	2.75207	3.17931	3.31902
NetBSD_loop_int.c	2.40141	3.5303	3.24684	3.5303	3.24684	3.4726	2.89326
pldi082_unbounded.c							
pldi08.c	2.5	3	2.5	3	2.5	3	2.5
rajamani_1.c							
seesaw.c	2.97802	3.62121	3.32013	3.62121	3.29617		
sendmail-close-angle.c						3.35544	3.02071
sendmail-mime7to8_ok.c	2.78676	3.22047	2.82258	3.22047	2.82258	3.24706	3.02817
sendmail-mime-fromqp.c	3.17213	3.24567	3.3474	3.24567		3.24576	3.04348
seq2.c							
seq3.c							
seq4.c							
seq.c							
seq-len.c							
seq-proc.c							
seq-sim.c							
seq-z3.c							
simple.c	2.54545	3.22222	2.90909	3.22222	2.90909	3.18182	2.75
simple_if.c	2	2	2	2	2	2	2
simple_nest.c	2	2	2	2	2	2	2
split.c							
string_concat-noarr.c							
substring1.c						3.11111	3.13684

Avg. Num. Literals/Clause	NOOPT	C	CG	CR	CGR	CRL	CGRL
svd1.c							
svd2.c	3.66126	3.57095	2.4375	3.80707	2.4375	3.10714	2.67647
svd3.c	2.26316	3.19355	2.38095	3.19355	2.38095	3.54286	3.01887
svd4.c							
svd.c							
svd-some-loop.c							
swim1.c							
swim.c							
up2.c					4.2303	4.17972	3.89916
up3.c				3.93799			
up4.c				3.95032	3.59375	4.31504	
up5.c					3.82081		3.9344
up.c				3.41985	3.47541	3.58901	3.72396
up-nd.c				3.53659		4.14217	
up-nested.c	2	2	2	2	2	2	2
xy0.c				3.45756	3.609	3.09722	2.90769
xy10.c	2.4	2.625	2.2	2.625	2.2	2.9375	2.55556
xy4.c				3.41304	3.55556	3.70431	
xyz2.c				3.42541	3.4	3.49	
xyz.c				3.48039	3.44221		

A.3 Number of Successful Geometric Generalizations

The **number of successful geometric generalizations** (where successful means that one literal was successfully replaced by one weaker literal) correlates highly with the run time (correlation $> 90\%$ on all configurations with geometric generalization) and can exceed 5400. The mean number of successful geometric generalizations across all configurations where it is enabled is 760, and the corresponding standard deviation is 1244. About half of all successful IC3-CEGAR runs on the benchmark set have less than 122 successful geometric generalizations. See figure A.2 for a graphical representation and table A.3 for the raw data.

Num. Geometric Gen.	CG	CGR	CGRL
apache-escape-absolute.c	1802	1802	
apache-get-tag.c	2516	2516	2613
barbr.c			
barbrprime.c			
bind_expands_vars2.c	19	19	73
bkley.c	1816	1816	4221
bk-nat.c	3664	3664	2586
bound.c	210	210	2692
cars.c			
dillig01.c	64	64	3
dillig03.c	5	5	16
dillig05.c	3948		
dillig07.c	19	19	22
dillig12.c			
dillig15.c		326	1476
dillig17.c	330	330	380
dillig19.c	3205	1946	827
dillig20.c	1517	1517	1856
dillig25.c	205	205	35
dillig28.c	189	156	273
dillig32.c	125	125	748
dillig33.c			
dillig37.c	199	199	41
down.c	395	395	2072
efm.c			
ex1.c	31	31	3026
ex2.c	5403		
fig1a.c		2219	458
fig2.c	1749	1749	
fragtest_simple.c			
gulv.c			
gulv_simp.c	122	122	85
gulwani_cegar1.c	70	70	78

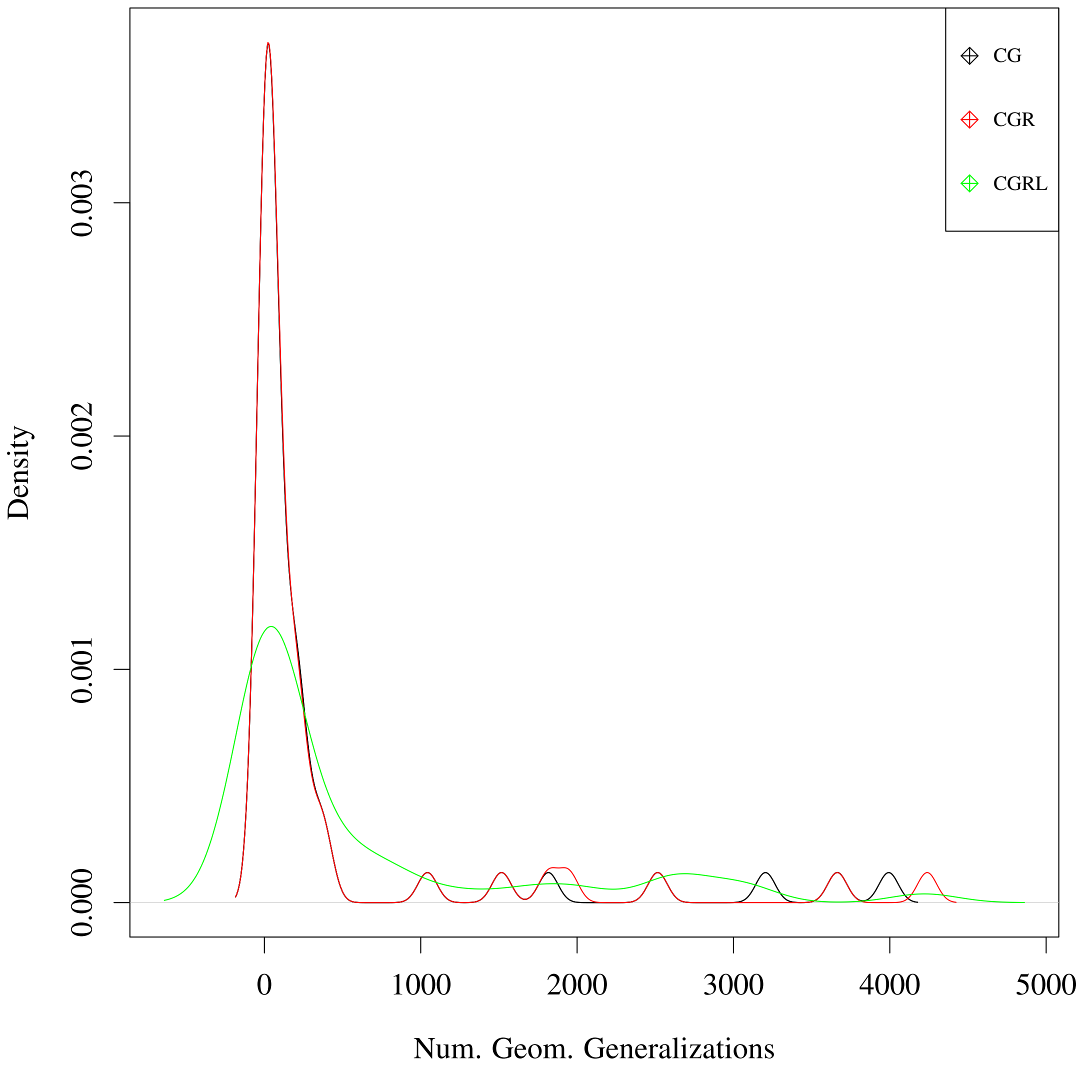
Num. Geometric Gen.	CG	CGR	CGRL
gulwani_cegar2.c	67	67	86
gulwani_fig1a.c	1	1	1
hsort.c			
hsortprime.c	4872	4872	
id_build.c	79	79	40
ken-imp.c	1210		165
large_const.c			
lifnat.c			
lifnatprime.c			
lifo.c			
MADWiFi-encode_ie_ok.c			
mergesort.c			
nested1.c	3	3	3
nested2.c	3	3	3
nested3.c	10	10	9
nested4.c	16	16	12
nested5.c	76	76	318
nested6.c	945		
nested7.c			
nested8.c			1194
nested9.c	375		804
nested.c	0	0	0
nest-if1.c	33	33	29
nest-if2.c	67	67	11
nest-if3.c	17	17	14
nest-if4.c	36	36	15
nest-if5.c	24	24	103
nest-if6.c	8	8	9
nest-if7.c	201	201	537
nest-if8.c	3993	4239	1621
nest-if.c	0	0	0
nest-len.c	0	0	0
NetBSD_g_Ctoc.c			
NetBSD_glob3_iny.c	252	252	3082
NetBSD_loop.c	113	118	1244
NetBSD_loop_int.c	1043	1043	901
pldi082_unbounded.c			
pldi08.c	0	0	0
rajamani_1.c			
seesaw.c	1852	1737	
sendmail-close-angle.c			5274
sendmail-mime7to8_ok.c	226	226	572

Num. Geometric Gen.	CG	CGR	CGRL
sendmail-mime-fromqp.c	3757		1553
seq2.c			
seq3.c			
seq4.c			
seq.c			
seq-len.c			
seq-proc.c			
seq-sim.c			
seq-z3.c			
simple.c	9	9	11
simple_if.c	0	0	0
simple_nest.c	0	0	0
split.c			
string_concat-noarr.c			
substring1.c			386
svd1.c			
svd2.c	367	367	522
svd3.c	29	29	260
svd4.c			
svd.c			
svd-some-loop.c			
swim1.c			
swim.c			
up2.c		665	3452
up3.c			
up4.c		516	
up5.c		711	4846
up.c		486	1865
up-nd.c			
up-nested.c	0	0	0
xy0.c		1018	299
xy10.c	3	3	19
xy4.c		521	
xyz2.c		1744	
xyz.c		2456	

A.4 Frontier Level at Termination

In table A.4, we present the raw data used for the graphics in section 4.3.

Figure A.2: Density plot depicting the number of successful geometric generalizations for instances where IC3-CEGAR terminates.



Frontier Level	NOOPT	C	CG	CR	CGR	CRL	CGRL
apache-escape-absolute.c	51	46	43	46	43	54	
apache-get-tag.c		27	36	27	36	30	30
barbr.c							
barbrprime.c							
bind_expands_vars2.c	8	8	9	8	9	10	10
bkley.c	35	28	29	28	29	41	37
bk-nat.c	24	36	28	36	28	37	29
bound.c	15	15	11	15	11	14	14
cars.c							
dillig01.c	13	11	11	11	11	11	10
dillig03.c	3	3	3	3	3	5	9
dillig05.c		17	31	39		23	
dillig07.c	23	7	8	7	8	7	7
dillig12.c				17			
dillig15.c				29	25		33
dillig17.c	23	26	24	26	24	23	18
dillig19.c		27	53	28	36	22	25
dillig20.c	31	32	23	32	23	22	23
dillig25.c	13	6	19	6	19	10	8
dillig28.c	12	19	15	19	15	28	26
dillig32.c	25	21	15	21	15	35	103
dillig33.c	22	31		38		40	
dillig37.c	9	13	23	13	23	10	15
down.c	19	35	14	43	14	26	35
efm.c							
ex1.c	9	8	8	8	8	35	36
ex2.c		26	32	26		47	
fig1a.c				63	96	56	28
fig2.c			29		29	65	
fragtest_simple.c							

Frontier Level	NOOPT	C	CG	CR	CGR	CRL	CGRL
gulv.c							
gulv_simp.c	28	17	23	17	23	22	25
gulwani_cegar1.c	12	11	12	11	12	11	12
gulwani_cegar2.c	10	10	10	10	10	14	10
gulwani_fig1a.c	5	5	4	5	4	5	4
hsort.c							
hsortprime.c	19	40	36	40	36	33	
id_build.c	20	20	22	20	22	11	10
ken-imp.c		79	39	35		17	19
large_const.c							
lifnat.c							
lifnatprime.c							
lifo.c							
MADWiFi-encode_ie_ok.c							
mergesort.c							
nested1.c	7	7	7	7	7	7	7
nested2.c	7	7	7	7	7	7	7
nested3.c	11	8	8	8	8	8	11
nested4.c	13	10	12	10	12	10	11
nested5.c	18	18	11	18	11	20	20
nested6.c	17	20	29	20		22	
nested7.c							
nested8.c		82		35		22	29
nested9.c		15	31	15		22	26
nested.c	1	1	1	1	1	1	1
nest-if1.c	9	7	10	7	10	7	11
nest-if2.c	9	8	9	8	9	7	10
nest-if3.c	11	8	12	8	12	10	10
nest-if4.c	12	8	11	8	11	8	7
nest-if5.c	8	11	8	11	8	9	10

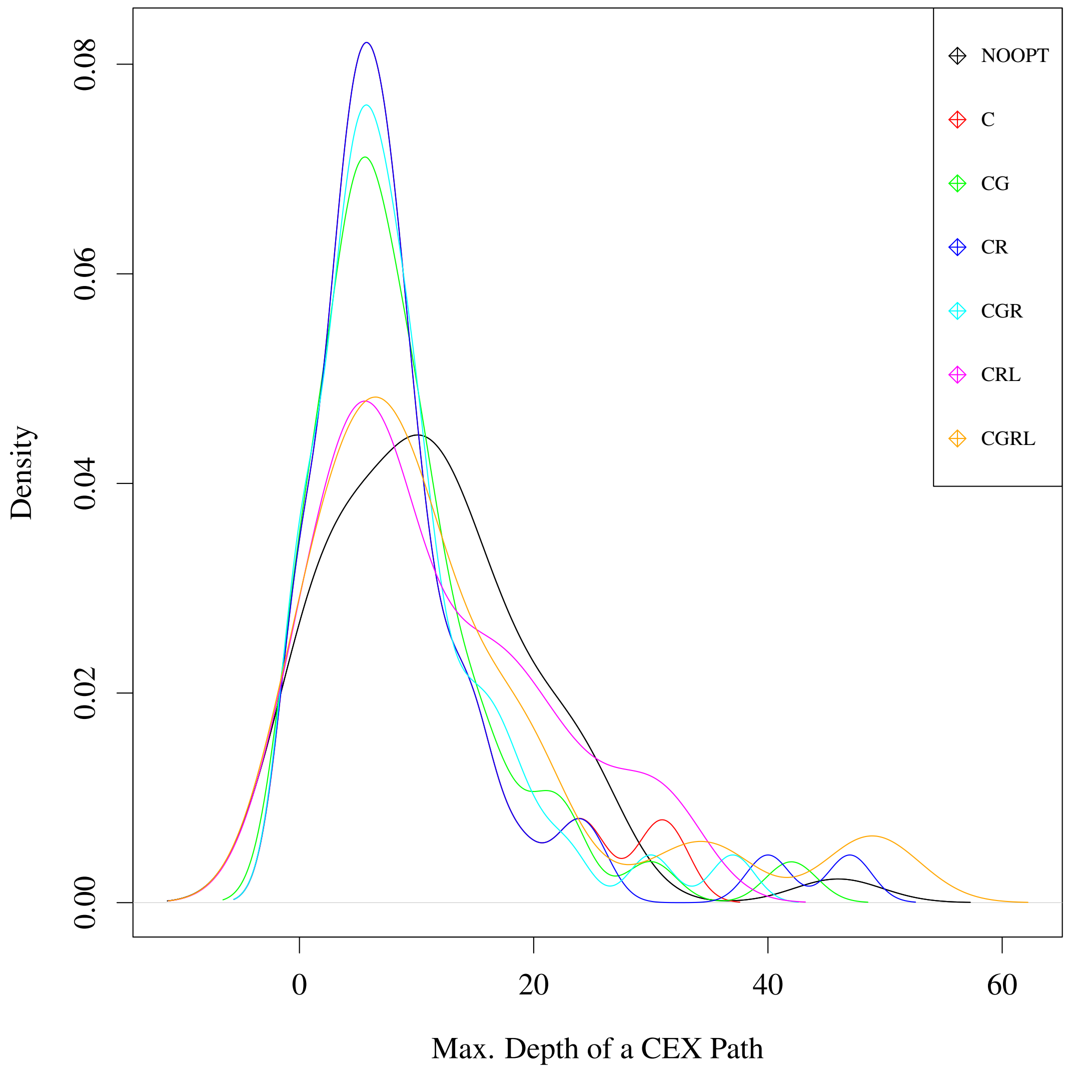
Frontier Level	NOOPT	C	CG	CR	CGR	CRL	CGRL
nest-if6.c	13	7	7	7	7	6	6
nest-if7.c	21	25	20	25	20	21	24
nest-if8.c	25	26	27	26	22	17	25
nest-if.c	1	1	1	1	1	1	1
nest-len.c	1	1	1	1	1	1	1
NetBSD_g_Ctoc.c							
NetBSD_glob3_iny.c	27	23	18	23	18		36
NetBSD_loop.c	23	21	26	21	28	30	22
NetBSD_loop_int.c	24	23	41	23	41	32	23
pldi082_unbounded.c							
pldi08.c	3	4	4	4	4	4	4
rajamani_1.c							
seesaw.c	83	50	51	50	48		
sendmail-close-angle.c						57	80
sendmail-mime7to8_ok.c	23	31	22	31	22	31	31
sendmail-mime-fromqp.c	43	31	65	31		21	31
seq2.c							
seq3.c							
seq4.c							
seq.c							
seq-len.c							
seq-proc.c							
seq-sim.c							
seq-z3.c							
simple.c	6	6	9	6	9	6	4
simple_if.c	1	1	1	1	1	1	1
simple_nest.c	1	1	1	1	1	1	1
split.c							
string_concat-noarr.c							
substring1.c						15	19

Frontier Level	NOOPT	C	CG	CR	CGR	CRL	CGRL
svd1.c							
svd2.c	35	36	22	39	22	16	15
svd3.c	12	16	14	16	14	26	16
svd4.c							
svd.c							
svd-some-loop.c							
swim1.c							
swim.c							
up2.c					26	56	69
up3.c				64			
up4.c				33	42	60	
up5.c					39		59
up.c				24	30	71	60
up-nd.c				32		68	
up-nested.c	1	1	1	1	1	1	1
xy0.c				37	59	29	28
xy10.c	2	3	5	3	5	14	15
xy4.c				31	38	109	
xyz2.c				60	72	78	
xyz.c				62	107		

A.5 Maximal Depth of the Proof Obligations Trace

In table A.5, we present the raw data used for the calculations in section 4.3. In figure A.3, we present a density plot for this data.

Figure A.3: Density plot depicting the maximal depth of the proof obligations trace for instances where IC3-CEGAR terminates.



Max. Trace Depth	NOOPT	C	CG	CR	CGR	CRL	CGRL
apache-escape-absolute.c	14	14	10	14	10	46	
apache-get-tag.c		15	15	15	15	39	31
barbr.c							
barbrprime.c							
bind_expands_vars2.c	4	4	4	4	4	5	5
bkley.c	23	12	13	12	13	32	48
bk-nat.c	23	15	15	15	15	30	35
bound.c	11	10	10	10	10	10	9
cars.c							
dillig01.c	10	6	6	6	6	8	8
dillig03.c	5	5	5	5	5	6	7
dillig05.c		11	11	13		23	
dillig07.c	17	4	4	4	4	5	5
dillig12.c				37			
dillig15.c				16	16		29
dillig17.c	17	25	10	25	10	17	20
dillig19.c		22	22	21	26	13	13
dillig20.c	46	14	30	14	30	17	12
dillig25.c	15	16	20	16	20	17	11
dillig28.c	14	14	22	14	18	32	36
dillig32.c	19	12	10	12	10	19	51
dillig33.c	21	19		51		52	
dillig37.c	24	10	10	10	10	9	11
down.c	10	30	10	40	10	23	48
efm.c							
ex1.c	6	4	4	4	4	13	16
ex2.c		34	34	34		33	
fig1a.c				98	61	92	28
fig2.c			17		17	36	
fragtest_simple.c							

Max. Trace Depth	NOOPT	C	CG	CR	CGR	CRL	CGRL
gulv.c							
gulv_simp.c	22	8	16	8	16	17	20
gulwani_cegar1.c	11	9	10	9	10	11	11
gulwani_cegar2.c	18	9	14	9	14	21	12
gulwani_fig1a.c	3	3	2	3	2	3	2
hsort.c							
hsortprime.c	10	10	10	10	10	20	
id_build.c	9	7	7	7	7	9	8
ken-imp.c		39	19	23		20	20
large_const.c							
lifnat.c							
lifnatprime.c							
lifo.c							
MADWiFi-encode_ie_ok.c							
mergesort.c							
nested1.c	4	4	4	4	4	4	4
nested2.c	4	4	4	4	4	4	4
nested3.c	8	6	6	6	6	6	6
nested4.c	11	7	7	7	7	6	6
nested5.c	11	5	5	5	5	19	18
nested6.c	16	14	14	14		20	
nested7.c							
nested8.c		58		58		16	17
nested9.c		19	26	19		20	21
nested.c	0	0	0	0	0	0	0
nest-if1.c	12	6	6	6	6	6	6
nest-if2.c	12	9	12	9	12	7	7
nest-if3.c	12	6	7	6	7	6	6
nest-if4.c	18	7	7	7	7	7	7
nest-if5.c	7	7	7	7	7	7	9

Max. Trace Depth	NOOPT	C	CG	CR	CGR	CRL	CGRL
nest-if6.c	11	4	4	4	4	3	3
nest-if7.c	13	7	9	7	9	14	18
nest-if8.c	26	32	42	47	37	15	19
nest-if.c	0	0	0	0	0	0	0
nest-len.c	0	0	0	0	0	0	0
NetBSD_g_Ctoc.c							
NetBSD_glob3_iny.c	12	12	12	12	12		28
NetBSD_loop.c	16	8	12	8	8	28	22
NetBSD_loop_int.c	14	19	23	19	23	24	15
pldi082_unbounded.c							
pldi08.c	2	3	2	3	2	3	2
rajamani_1.c							
seesaw.c	29	11	9	11	9		
sendmail-close-angle.c						37	43
sendmail-mime7to8_ok.c	8	10	8	10	8	30	31
sendmail-mime-fromqp.c	31	23	22	23		21	27
seq2.c							
seq3.c							
seq4.c							
seq.c							
seq-len.c							
seq-proc.c							
seq-sim.c							
seq-z3.c							
simple.c	4	3	4	3	4	5	5
simple_if.c	0	0	0	0	0	0	0
simple_nest.c	0	0	0	0	0	0	0
split.c							
string_concat-noarr.c							
substring1.c						15	15

Max. Trace Depth	NOOPT	C	CG	CR	CGR	CRL	CGRL
svd1.c							
svd2.c	25	23	17	23	17	8	11
svd3.c	8	7	7	7	7	23	10
svd4.c							
svd.c							
svd-some-loop.c							
swim1.c							
swim.c							
up2.c					7	33	32
up3.c				45			
up4.c				25	9	70	
up5.c					12		39
up.c				14	14	84	64
up-nd.c				14		77	
up-nested.c	0	0	0	0	0	0	0
xy0.c				73	54	28	28
xy10.c	4	4	4	4	4	14	14
xy4.c				37	33	155	
xyz2.c				44	80	82	
xyz.c				48	72		

A.6 Number of Predicates in the Domain at Termination

In table A.6, we present the raw data used for the calculations in section 4.3.

Num. Domain Predicates	NOOPT	C	CG	CR	CGR	CRL	CGRL
apache-escape-absolute.c	147	143	117	143	117	157	
apache-get-tag.c		105	106	105	106	97	94
barbr.c							
barbrprime.c							
bind_expands_vars2.c	35	34	29	34	29	33	31
bkley.c	93	80	86	80	86	63	81
bk-nat.c	105	86	78	86	78	63	84
bound.c	52	51	51	51	51	45	44
cars.c							
dillig01.c	37	24	28	24	28	18	19
dillig03.c	17	17	17	17	17	16	18
dillig05.c		79	152	140		60	
dillig07.c	65	23	24	23	24	21	22
dillig12.c				170			
dillig15.c				53	55		65
dillig17.c	176	49	58	49	58	47	55
dillig19.c		90	158	100	90	49	55
dillig20.c	158	73	92	73	92	68	75
dillig25.c	46	37	40	37	40	33	35
dillig28.c	53	44	50	44	51	37	51
dillig32.c	58	43	45	43	45	56	78
dillig33.c	111	99		112		92	
dillig37.c	43	36	41	36	41	23	30
down.c	47	56	37	53	37	40	63
efm.c							
ex1.c	36	30	27	30	27	48	73
ex2.c		180	241	182		251	
fig1a.c				128	94	62	37
fig2.c			87		87	113	
fragtest_simple.c							

Num. Domain Predicates	NOOPT	C	CG	CR	CGR	CRL	CGRL
gulv.c							
gulv_simp.c	58	26	31	26	31	28	34
gulwani_cegar1.c	30	29	29	29	29	29	29
gulwani_cegar2.c	33	27	33	27	33	37	29
gulwani_fig1a.c	8	8	8	8	8	8	8
hsort.c							
hsortprime.c	55	55	58	55	58	77	
id_build.c	44	38	40	38	40	32	29
ken-imp.c		158	86	83		45	43
large_const.c							
lifnat.c							
lifnatprime.c							
lifo.c							
MADWiFi-encode_ie_ok.c							
mergesort.c							
nested1.c	21	21	21	21	21	21	21
nested2.c	21	21	21	21	21	21	21
nested3.c	29	25	25	25	25	23	23
nested4.c	32	27	27	27	27	23	23
nested5.c	37	25	27	25	27	29	29
nested6.c	100	73	73	73		47	
nested7.c							
nested8.c		92		80		56	70
nested9.c		44	66	44		61	68
nested.c	12	12	12	12	12	12	12
nest-if1.c	37	23	29	23	29	23	23
nest-if2.c	38	38	36	38	36	29	29
nest-if3.c	37	25	29	25	29	23	23
nest-if4.c	54	27	31	27	31	25	25
nest-if5.c	27	29	27	29	27	30	31

Num. Domain Predicates	NOOPT	C	CG	CR	CGR	CRL	CGRL
nest-if6.c	100	78	77	78	77	73	73
nest-if7.c	42	50	40	50	40	34	37
nest-if8.c	105	75	69	78	77	68	57
nest-if.c	9	9	9	9	9	9	9
nest-len.c	12	12	12	12	12	12	12
NetBSD_g_Ctoc.c							
NetBSD_glob3_iny.c	137	128	120	128	120		128
NetBSD_loop.c	62	53	62	53	60	58	63
NetBSD_loop_int.c	99	83	96	83	96	95	94
pldi082_unbounded.c							
pldi08.c	8	8	8	8	8	8	8
rajamani_1.c							
seesaw.c	129	92	105	92	104		
sendmail-close-angle.c						79	98
sendmail-mime7to8_ok.c	61	63	56	63	56	59	67
sendmail-mime-fromqp.c	90	75	87	75		67	72
seq2.c							
seq3.c							
seq4.c							
seq.c							
seq-len.c							
seq-proc.c							
seq-sim.c							
seq-z3.c							
simple.c	13	13	13	13	13	15	15
simple_if.c	9	9	9	9	9	9	9
simple_nest.c	9	9	9	9	9	9	9
split.c							
string_concat-noarr.c							
substring1.c						44	57

Num. Domain Predicates	NOOPT	C	CG	CR	CGR	CRL	CGRL
svd1.c							
svd2.c	83	85	56	88	56	40	45
svd3.c	40	34	34	34	34	49	41
svd4.c							
svd.c							
svd-some-loop.c							
swim1.c							
swim.c							
up2.c					48	85	103
up3.c				155			
up4.c				86	63	73	
up5.c					64		108
up.c				52	60	94	102
up-nd.c				63		99	
up-nested.c	11	11	11	11	11	11	11
xy0.c				88	75	31	34
xy10.c	16	16	16	16	16	20	21
xy4.c				55	59	115	
xyz2.c				62	85	84	
xyz.c				67	83		

A.7 Number of Interpolating Refinement Calls

In table A.7, we present the raw data used for the calculations in section 4.3.

Interpolating Refinements	NOOPT	C	CG	CR	CGR	CRL	CGRL
apache-escape-absolute.c	49	49	39	49	39	55	
apache-get-tag.c		39	38	39	38	41	40
barbr.c							
barbrprime.c							
bind_expands_vars2.c	7	7	5	7	5	8	7
bkley.c	25	23	23	23	23	18	23
bk-nat.c	29	26	22	26	22	20	22
bound.c	14	14	13	14	13	11	10
cars.c							
dillig01.c	10	5	6	5	6	4	4
dillig03.c	2	2	2	2	2	4	5
dillig05.c		23	38	32		16	
dillig07.c	13	5	5	5	5	6	6
dillig12.c				41			
dillig15.c				16	15		26
dillig17.c	45	12	14	12	14	13	13
dillig19.c		23	31	23	21	13	15
dillig20.c	31	15	17	15	17	13	15
dillig25.c	13	9	11	9	11	8	8
dillig28.c	12	12	13	12	12	16	11
dillig32.c	13	10	10	10	10	16	23
dillig33.c	26	26		29		24	
dillig37.c	8	6	7	6	7	6	8
down.c	11	14	8	14	8	11	18
efm.c							
ex1.c	8	5	5	5	5	17	25
ex2.c		68	76	68		84	
fig1a.c				34	21	16	9
fig2.c			19		19	37	
fragtest_simple.c							

Interpolating Refinements	NOOPT	C	CG	CR	CGR	CRL	CGRL
gulv.c							
gulv_simp.c	20	9	11	9	11	9	11
gulwani_cegar1.c	8	8	8	8	8	8	8
gulwani_cegar2.c	9	7	9	7	9	13	8
gulwani_fig1a.c	1	1	1	1	1	1	1
hsort.c							
hsortprime.c	13	13	14	13	14	19	
id_build.c	11	10	10	10	10	7	5
ken-imp.c		41	21	17		10	10
large_const.c							
lifnat.c							
lifnatprime.c							
lifo.c							
MADWiFi-encode_ie_ok.c							
mergesort.c							
nested1.c	4	4	4	4	4	4	4
nested2.c	4	4	4	4	4	4	4
nested3.c	8	6	6	6	6	5	5
nested4.c	9	6	7	6	7	5	5
nested5.c	8	6	7	6	7	11	11
nested6.c	24	20	19	20		15	
nested7.c							
nested8.c		24		19		15	16
nested9.c		7	15	7		13	15
nested.c	0	0	0	0	0	0	0
nest-if1.c	11	5	8	5	8	5	5
nest-if2.c	11	12	10	12	10	7	7
nest-if3.c	11	6	8	6	8	5	5
nest-if4.c	14	7	9	7	9	6	6
nest-if5.c	7	8	7	8	7	8	9

Interpolating Refinements	NOOPT	C	CG	CR	CGR	CRL	CGRL
nest-if6.c	17	7	6	7	6	4	4
nest-if7.c	11	13	10	13	10	8	8
nest-if8.c	27	20	19	20	20	16	17
nest-if.c	0	0	0	0	0	0	0
nest-len.c	0	0	0	0	0	0	0
NetBSD_g_Ctoc.c							
NetBSD_glob3_iny.c	36	34	32	34	32		36
NetBSD_loop.c	21	17	21	17	20	19	22
NetBSD_loop_int.c	25	19	22	19	22	22	19
pldi082_unbounded.c							
pldi08.c	1	1	1	1	1	1	1
rajamani_1.c							
seesaw.c	39	25	30	25	30		
sendmail-close-angle.c						24	28
sendmail-mime7to8_ok.c	19	23	17	23	17	23	25
sendmail-mime-fromqp.c	25	24	26	24		24	26
seq2.c							
seq3.c							
seq4.c							
seq.c							
seq-len.c							
seq-proc.c							
seq-sim.c							
seq-z3.c							
simple.c	3	3	3	3	3	4	4
simple_if.c	0	0	0	0	0	0	0
simple_nest.c	0	0	0	0	0	0	0
split.c							
string_concat-noarr.c							
substring1.c						11	11

Interpolating Refinements	NOOPT	C	CG	CR	CGR	CRL	CGRL
svd1.c							
svd2.c	23	22	15	23	15	10	11
svd3.c	11	9	9	9	9	16	9
svd4.c							
svd.c							
svd-some-loop.c							
swim1.c							
swim.c							
up2.c					10	20	25
up3.c				48			
up4.c				24	17	16	
up5.c					16		22
up.c				12	15	24	24
up-nd.c				16		27	
up-nested.c	0	0	0	0	0	0	0
xy0.c				24	17	10	10
xy10.c	2	2	2	2	2	6	6
xy4.c				21	20	42	
xyz2.c				23	25	23	
xyz.c				17	18		

A.8 Number of Successful Refinement State Mining Attempts

In table A.8, we present the raw data used for the calculations in section 4.3.

RSM	CR	CGR	CRL	CGRL
apache-escape-absolute.c	0	0	0	
apache-get-tag.c	0	0	1	0
barbr.c				
barbrprime.c				
bind_expands_vars2.c	0	0	0	0
bkley.c	0	0	0	0
bk-nat.c	0	0	0	0
bound.c	0	0	0	0
cars.c				
dillig01.c	0	0	0	0
dillig03.c	0	0	0	0
dillig05.c	7		2	
dillig07.c	0	0	0	0
dillig12.c	6			
dillig15.c	2	1		5
dillig17.c	0	0	0	0
dillig19.c	5	3	0	1
dillig20.c	0	0	0	0
dillig25.c	0	0	0	0
dillig28.c	1	1	1	1
dillig32.c	0	0	2	3
dillig33.c	2		1	
dillig37.c	0	0	0	1
down.c	2	0	0	0
efm.c				
ex1.c	0	0	0	1
ex2.c	3		4	
fig1a.c	6	3	1	1
fig2.c		0	4	
fragtest_simple.c				
gulv.c				
gulv_simp.c	0	0	0	1
gulwani_cegar1.c	0	0	0	0
gulwani_cegar2.c	0	0	0	0
gulwani_fig1a.c	0	0	0	0
hsort.c				
hsortprime.c	0	0	0	
id_build.c	0	0	0	0
ken-imp.c	3		1	1

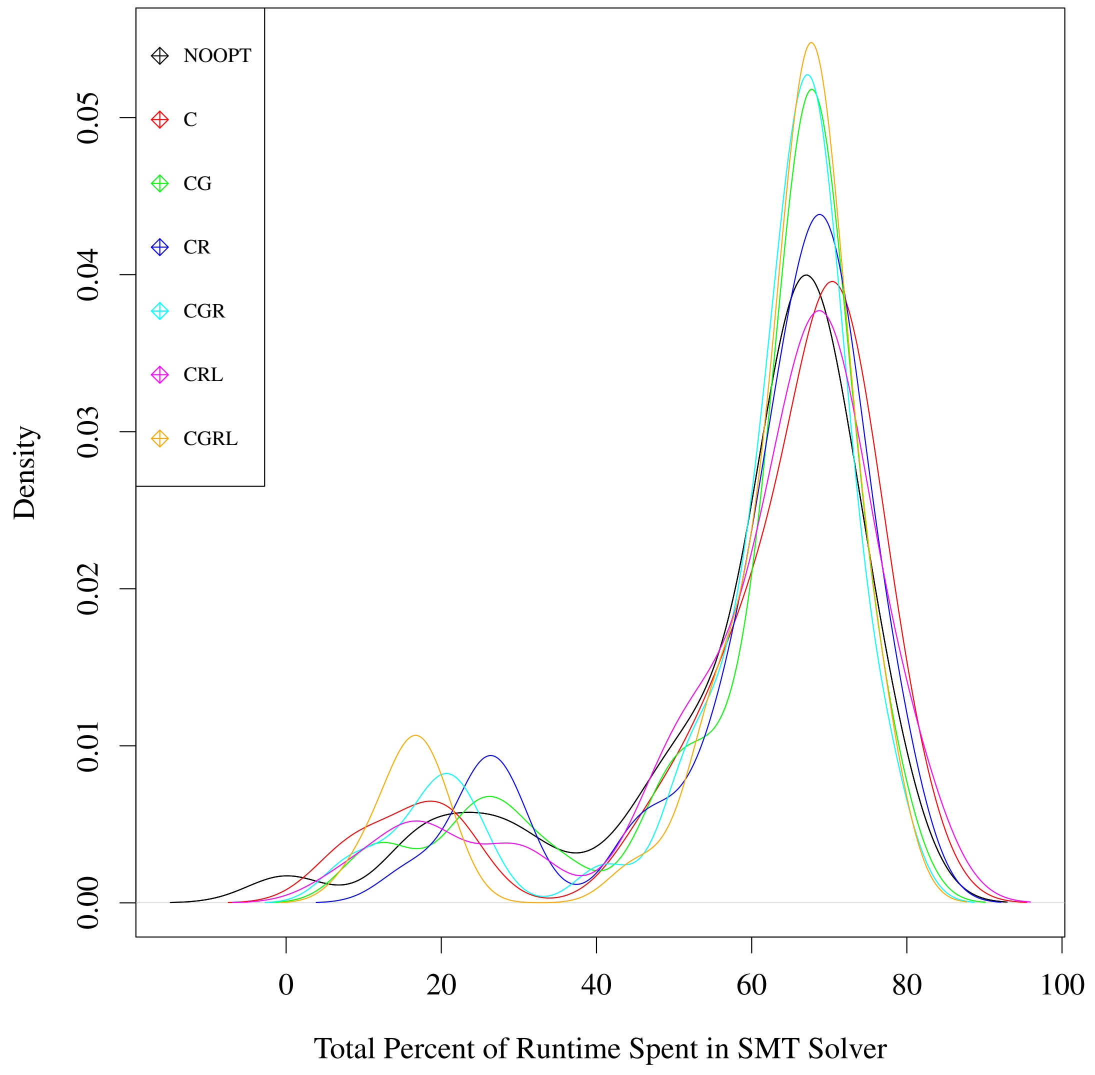
RSM	CR	CGR	CRL	CGRL
large_const.c				
lifnat.c				
lifnatprime.c				
lifo.c				
MADWiFi-encode_ie_ok.c				
mergesort.c				
nested1.c	0	0	0	0
nested2.c	0	0	0	0
nested3.c	0	0	0	0
nested4.c	0	0	0	0
nested5.c	0	0	0	0
nested6.c	0		0	
nested7.c				
nested8.c	2		0	2
nested9.c	0		0	2
nested.c	0	0	0	0
nest-if1.c	0	0	0	0
nest-if2.c	0	0	0	0
nest-if3.c	0	0	0	0
nest-if4.c	0	0	0	0
nest-if5.c	0	0	0	0
nest-if6.c	0	0	0	0
nest-if7.c	0	0	0	0
nest-if8.c	1	1	0	0
nest-if.c	0	0	0	0
nest-len.c	0	0	0	0
NetBSD_g_Ctoc.c				
NetBSD_glob3_iny.c	0	0		0
NetBSD_loop.c	0	1	0	1
NetBSD_loop_int.c	0	0	0	0
pldi082_unbounded.c				
pldi08.c	0	0	0	0
rajamani_1.c				
seesaw.c	0	3		
sendmail-close-angle.c			1	0
sendmail-mime7to8_ok.c	0	0	0	0
sendmail-mime-fromqp.c	0		0	1
seq2.c				
seq3.c				
seq4.c				
seq.c				
seq-len.c				

RSM	CR	CGR	CRL	CGRL
seq-proc.c				
seq-sim.c				
seq-z3.c				
simple.c	0	0	0	0
simple_if.c	0	0	0	0
simple_nest.c	0	0	0	0
split.c				
string_concat-noarr.c				
substring1.c			0	0
svd1.c				
svd2.c	1	0	0	0
svd3.c	0	0	0	0
svd4.c				
svd.c				
svd-some-loop.c				
swim1.c				
swim.c				
up2.c		1	3	3
up3.c	13			
up4.c	4	3	2	
up5.c		2		4
up.c	1	2	3	2
up-nd.c	1		3	
up-nested.c	0	0	0	0
xy0.c	5	3	1	1
xy10.c	0	0	0	0
xy4.c	4	2	9	
xyz2.c	1	3	2	
xyz.c	2	1		

A.9 Percent of the Run Time Spent in the SMT Solver

In table A.9, we present the raw data used for the calculations in section 4.3. In figure A.4, we present a density plot for this data.

Figure A.4: Density plot depicting the percent of the run time spent in the SMT solver for instances where IC3-CEGAR terminates.



% Time in SMT Solver	NOOPT	C	CG	CR	CGR	CRL	CGRL
apache-escape-absolute.c	82	87	81	87	80	85	
apache-get-tag.c		86	80	85	80	85	79
barbr.c							
barbrprime.c							
bind_expands_vars2.c	55	58	61	60	60	55	61
bkley.c	76	78	76	77	76	79	76
bk-nat.c	78	79	77	79	77	81	76
bound.c	65	74	69	72	68	74	69
cars.c							
dillig01.c	68	71	69	70	67	69	68
dillig03.c	43	50	48	50	52	52	64
dillig05.c		68	63	67		70	
dillig07.c	59	62	65	62	63	62	62
dillig12.c				68			
dillig15.c				64	65		63
dillig17.c	65	70	68	68	68	69	68
dillig19.c		67	64	65	66	72	70
dillig20.c	74	72	70	73	70	73	71
dillig25.c	65	64	67	62	66	69	68
dillig28.c	67	67	64	68	65	68	68
dillig32.c	69	68	66	69	66	68	56
dillig33.c	74	68		65		70	
dillig37.c	67	68	66	68	65	62	68
down.c	67	68	68	68	66	69	64
efm.c							
ex1.c	73	72	73	72	71	67	65
ex2.c		76	70	76		72	
fig1a.c				57	56	60	66
fig2.c			68		68	64	
fragtest_simple.c							

% Time in SMT Solver	NOOPT	C	CG	CR	CGR	CRL	CGRL
gulv.c							
gulv_simp.c	65	71	67	71	68	68	69
gulwani_cegar1.c	66	73	69	73	69	71	68
gulwani_cegar2.c	68	68	67	64	66	64	66
gulwani_fig1a.c	71	55	48	45	59	52	55
hsort.c							
hsortprime.c	80	83	78	82	78	81	
id_build.c	67	62	65	63	65	63	62
ken-imp.c		51	62	61		62	65
large_const.c							
lifnat.c							
lifnatprime.c							
lifo.c							
MADWiFi-encode_ie_ok.c							
mergesort.c							
nested1.c	47	54	53	57	57	55	65
nested2.c	51	48	59	55	52	46	56
nested3.c	57	60	59	63	62	61	63
nested4.c	62	65	62	65	61	65	66
nested5.c	69	72	68	70	70	71	70
nested6.c	77	73	77	73		74	
nested7.c							
nested8.c		57		66		70	70
nested9.c		73	68	73		70	69
nested.c	27	15	25	16	25	16	16
nest-if1.c	63	64	69	64	69	66	68
nest-if2.c	62	73	68	67	70	62	68
nest-if3.c	65	70	68	64	65	71	68
nest-if4.c	64	71	71	70	70	69	67
nest-if5.c	62	73	63	73	63	72	70

% Time in SMT Solver	NOOPT	C	CG	CR	CGR	CRL	CGRL
nest-if6.c	51	60	53	58	51	49	52
nest-if7.c	70	72	70	70	70	72	72
nest-if8.c	71	74	70	72	70	74	71
nest-if.c	16	7	14	25	8	8	15
nest-len.c	28	19	22	24	19	30	19
NetBSD_g_Ctoc.c							
NetBSD_glob3_iny.c	73	78	71	78	72		76
NetBSD_loop.c	69	69	67	69	67	69	66
NetBSD_loop_int.c	75	69	66	69	66	71	70
pldi082_unbounded.c							
pldi08.c	50	59	36	56	41	50	45
rajamani_1.c							
seesaw.c	69	76	74	76	74		
sendmail-close-angle.c						75	72
sendmail-mime7to8_ok.c	78	81	78	80	78	81	77
sendmail-mime-fromqp.c	76	80	74	80		80	72
seq2.c							
seq3.c							
seq4.c							
seq.c							
seq-len.c							
seq-proc.c							
seq-sim.c							
seq-z3.c							
simple.c	57	55	68	64	60	57	56
simple_if.c	22	22	10	30	22	30	10
simple_nest.c	0	10	27	27	20	20	20
split.c							
string_concat-noarr.c							
substring1.c						67	67

% Time in SMT Solver	NOOPT	C	CG	CR	CGR	CRL	CGRL
svd1.c							
svd2.c	74	77	76	77	75	80	76
svd3.c	68	78	72	77	72	78	74
svd4.c							
svd.c							
svd-some-loop.c							
swim1.c							
swim.c							
up2.c					68	60	60
up3.c				56			
up4.c				62	64	60	
up5.c					63		62
up.c				67	65	57	58
up-nd.c				69		64	
up-nested.c	18	21	30	27	14	16	16
xy0.c				62	63	68	67
xy10.c	34	45	52	46	55	61	59
xy4.c				63	64	51	
xyz2.c				63	63	61	
xyz.c				63	61		

Index