Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Applying Incremental, Inductive Model Checking to Software

Bachelor's Thesis

Author
Tamás Tegzes

Advisor
Tamás Tóth

December 7, 2018

Contents

Kivonat i

Abstract ii

1 Introduction 1

2 Background 3

2.1 Formal Logic . . . 3

2.1.1 Propositional Logic . . . 3

2.1.1.1 Syntax . . . 3

2.1.1.2 Semantics . . . 4

2.1.2 First Order Logic . . . 4

2.1.2.1 Syntax . . . 4

2.1.2.2 Semantics . . . 5

2.1.3 Satisfiability Modulo Theories . . . 7

2.2 Modeling Formalisms . . . 7

2.2.1 Transition Systems . . . 7

2.2.1.1 Transition Systems in General . . . 7

2.2.1.2 Symbolic Transition Systems . . . 9

2.2.2 Control Flow Automata . . . 10

2.2.3 Syntax . . . 10

2.2.4 Semantics . . . 11

3 The Algorithm 13

3.1 Property Directed Reachability . . . 13

3.1.1 High Level Description . . . 13

3.1.1.1 Trace . . . 13

3.1.1.2 Proof Obligations . . . 14

3.1.1.3 Main Loop . . . 14

3.1.1.4 Processing Proof Obligations . . . 14

3.1.1.5 Initial States in Proof Obligations 15
3.1.1.6 Termination 16
3.1.2 Lower Level Description 16
3.1.2.1 Trace 16
3.1.2.2 Proof Obligations 16
3.1.2.3 Main Loop 16
3.1.2.4 Processing Proof Obligations 18
3.1.2.5 Termination 21
3.2 Implicit Predicate Abstraction 21
3.2.1 Predicate Abstraction 22
3.2.2 Abstraction Refinement 23
3.3 Application to Software 24

4 Implementation 27
4.1 The Theta Framework 27
4.2 Kotlin 27
4.2.1 Null pointer safety 28
4.2.2 Operator overloading 28
4.2.3 Type inference 29
4.3 Our Implementation 29
4.3.1 Data classes 30
4.3.2 Handling the Solver 31
4.4 Evaluation 32

5 Conclusions 33

Bibliography 34

Appendix 35
A.1 Evaluation results 35

HALLGATÓI NYILATKOZAT

Alulírott *Tegzes Tamás*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2018. december 7.

Tegzes Tamás
hallgató

Kivonat

Egyre nagyobb szerepet kapnak az életünkben a különböző szoftvereszközök. Ma már olyan, biztonságkritikus alkalmazási területeken is elterjedten alkalmaznak szoftvereket, ahol egy esetleges meghibásodás végzetes következményekkel is járhat: szoftverhibák hatására vagyonok úszhatnak el, repülőgépek zuhanhatnak le, atomerőművek állhatnak le. Az ilyen, kritikus szoftverek helyességének biztosítása tehát kiemelten fontos feladat.

A szoftverek megfelelő működésének ellenőrzése a verifikációjuk során történik. Kritikus szoftverek ellenőrzésére gyakran alkalmaznak formális módszereket, melyekkel – ellentétben a hagyományos ellnőrzési módszerekkel (pl. tesztelés) – nemcsak hibák jelenléte mutatható ki, hanem akár a rendszer helyessége is bizonyítható. Az egyik legelterjedtebben alkalmazott formális módszer a modellellenőrzés, ahol a szoftver állapotterének bejárásával próbáljuk a vele szemben megfogalmazott formális követelmények teljesülését bizonyítani, vagy éppen cáfolni.

Az IC3 egy korszerű, eredetileg hardvermodellek ellenőrzésére alkalmas modellellenőrző algoritmus. Az IC3 algoritmus egyedi jellemzője, hogy a rendszer helyességének ellenőrzését inkrementálisan, számos egyszerű lemma indukciós bizonyításán keresztül végzi. Amennyiben a rendszer helyes, a felfedezett lemmák összessége a rendszer helyességének bizonyítékát (az állapottér egy biztonságos felülbecslését) adják; ellenkező esetben pedig a keresést egy ellenpélda irányába irányítják, ezáltal hatékonyan vezérelve az állapottérfelderítést. Az eredeti algoritmus hátránya, hogy elsősorban véges állapoterű rendszerek (pl. sorrendi hálózatok) hatékony kezelését teszi lehetővé. Az algoritmus így jelenleg is aktív kutatás tárgyát képezi, és számos kiterjesztéssel bír.

Dolgozatomban az IC3 algoritmusnak egy olyan kiterjesztését vizsgálom, amely a végtelen állapottér hatékony kezeléséhez (implicit) predikátumabsztrakciót használ. Ez a módszer a végtelen állapotteret véges sok partícióra osztja fel a változókon értelmezett logikai állítások (predikátumok) mentén. A partíciók egy véges absztrakt állapotteret hoznak létre, melynek állapotait az határozza meg, hogy mely predikátumok teljesülnek. A futás során az algoritmus új predikátumokat tanul, ezzel szükség szerint pontosítja a képét a rendszerről, mégis megtartja az absztrakció általánosságát.

Annak érdekében, hogy az algoritmus alkalmas legyen programokat leíró vezérlésifolyam-automaták ellenőrzésére, egy olyan transzformációt is megadok, amely az automata alapján egy vele ekvivalens viselkedésű szimbolikus tranzíciós rendszert állít elő. Az algoritmust ezen a szimbolikus tranzíciós rendszeren futtatva lehetőség nyílik az automata helyességének ellenőrzésére.

Az algoritmus, illetve a transzformáció implementációját a Theta nyílt forráskódú modellellenőrző keretrendszer moduljaként készítem el, ami további kiterjesztési lehetőségeket biztosít. Az implementált algoritmus különböző konfigurációinak hatékonyságát mérésekkel vizsgálom.

Abstract

Software plays an ever increasing role in our lives. Nowadays, software is used even in safety critical fields where a potential failure can lead to significant consequences: software failures can cause fortunes to go to waste, airplanes to crash, or nuclear power plants to stop. Thus ensuring the correctness of such critical software is a crucial task.

Verification is the process of checking that the software operates as expected. Formal methods are often used to verify critical systems, which – as opposed to traditional verification methods (such as testing) – is not only able to show the presence of errors, but also to prove their absence. One of the most widely applied formal method is model checking, which constitutes traversing the state space of the software to prove or disprove formal requirements against the system.

IC3 is a state-of-the-art model checking algorithm, originally developed for checking hardware models. The unique property of IC3 is that it proves the correctness of the system in incremental steps by proving several simple lemmas about the system. If the system is correct, the proven lemmas collectively form a proof of that fact (they specify a safe overapproximation of the state space). If the system is not correct, the lemmas direct the search towards a counterexample, thereby making the traversal more efficient. The original algorithm is designed to check finite state systems, and is not effective for infinite state systems. Therefore the algorithm is still the subject of active research and has several extensions.

In my thesis I examine an extension to IC3 that uses (implicit) predicate abstraction to handle infinite state spaces efficiently. This method divides the state space into a finite number of partitions with respect to a set of predicates over the system variables. The partitions form a finite abstract state space, where each state is defined by which of the predicates hold and which do not. During its run, the algorithm learns new predicates to refine its abstract image of the system, while still maintaining the coarseness of the abstraction.

In order to check control flow automata which describe programs, I present a transformation that creates a symbolic transition system for a control flow automaton that behaves equivalently to the automaton. The correctness of the automaton can be checked by running the algorithm on this symbolic transition system.

I implement the algorithm and the transformation as part of the open source model checking framework Theta, which allows for additional ways of extending it. Finally, I evaluate the efficiency of the various configurations of the implemented algorithm.

Chapter 1

Introduction

As we rely more and more on machines and software systems, it is increasingly important to ensure that they operate as we expect them to. *Safety critical systems* are systems whose failure can cause severe damage, possibly even death. Examples of such safety critical systems include cars, aircraft, medical devices, fire alarms, etc.

As development of complex software is error-prone, it is crucial to ensure correctness during development of safety critical systems by applying rigorous verification methods, otherwise remaining design flaws may cause them to fail and be unsafe. While meticulous testing can reveal many such flaws, it can not prove the system to be correct. Even if all the tests are passed, it is not sure if the system is truly safe or if there was a missed test case that would cause it to fail.

Formal verification techniques on the other hand are able to prove the correctness of safe systems, or discover bugs of unsafe ones. *Model checking* is a formal verification technique that is suitable to determine the correctness of a system by exhaustive traversal of its state space. The main concept of model checking can be seen on Figure 1.1. As their input, model checkers receive a formalized model of a system and the formalized property to be checked. They search the state space of the model and either find that the property does not hold, and give an example where it does not hold, or they prove that the model never violates the property.

In this thesis, we present an approach to apply incremental, inductive model checking to software. We begin by introducing the theoretical background of the problem. We have implemented the IC3 algorithm and extended it to handle models whose state space is infinite using (implicit) predicate abstraction. Moreover, we have implemented a transformation that allows the verification of control flow automata with our implementation. It creates a symbolic transition system that behave equivalently to a control flow automaton, and by checking the correctness of the transition system, we can learn if the automaton is correct. We present the details of how the algorithm works and how we implemented it, and we evaluate the implementation by measuring its performance on a set of benchmark instances. Finally we examine possible areas of future improvements.

Related work IC3 ("Incremental Construction of Inductive Clauses for Indubitable Correctness"), introduced in [1], is a model checking algorithm. An optimization of IC3, called PDR, has been proposed in [5]. PDR is created to check finite state transition systems defined by variables, whose values are either `true` or `false`. Implicit predicate abstraction as a method of checking infinite state systems using PDR is suggested in [4]. Checking control flow automata with PDR has been described in [3, 6]. Our approach uses implicit

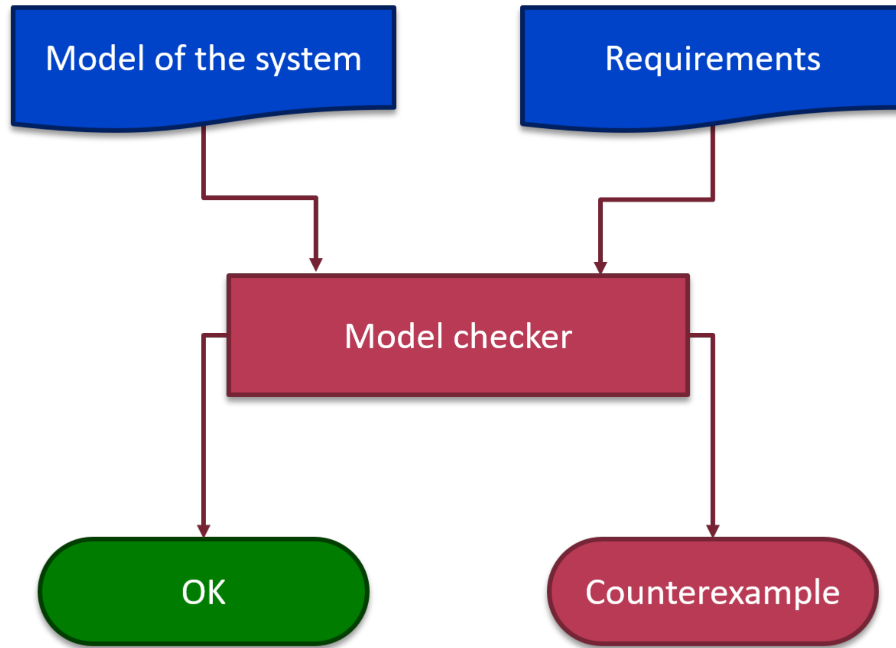

Figure 1.1: The input and output of a model checker

predicate abstraction as well, but handles control flow automata by transforming them to symbolic transition systems.

Organization of the thesis We introduce the theoretical background to PDR in Chapter 2. In Chapter 3 we explain how PDR works, describe an extension to it, which allows efficient handling of infinite state spaces by implicit predicate abstraction, and introduce a way to transform control flow automaton to an equivalently behaving symbolic transition systems, such that the correctness of the automaton can be checked by checking the correctness of the symbolic transition system. The details and evaluation of our implementation are presented in Chapter 4. Finally, in Chapter 5 we summarize the thesis and suggest areas to explore in the future.

Chapter 2

Background

In this chapter, we present the theoretical background to PDR. We introduce propositional logic and first order logic. Building upon them we define the types of models which our algorithm operates on, symbolic transition systems and control flow automata.

2.1 Formal Logic

In order to formally prove programs correct, we first have to define a formal system in which we can reason about programs. This section briefly introduces propositional logic and first order logic as they are defined in [2]. They both define a set of symbols (syntax), and a set of rules to assign meaning to those symbols (semantics).

2.1.1 Propositional Logic

Propositional logic (PL) operates over propositions. Propositions can be either `true` or `false`, they cannot take any other value.

2.1.1.1 Syntax

Let $V = \{P, Q, R, P_1, P_2, \ldots\}$ be a countable set of *propositional variables*. A propositional variable $P \in V$ is a *formula*. More complex formulas can be built by using *logical connectives*. Formally, if φ and ψ are formulas, then the following are also formulas:

- $\bot$ (bottom),
- $\top$ (top),
- $\neg\varphi$ (negation),
- $\varphi \wedge \psi$ (conjunction),
- $\varphi \vee \psi$ (disjunction),
- $\varphi \to \psi$ (implication),
- $\varphi \leftrightarrow \psi$ (if and only if, iff).

Nullary connectives $\bot$ and $\top$ are also called *truth symbols*. An *atom* is either a truth symbol or a propositional variable. A *literal* is either an atom α or its negation $\neg\alpha$.

The order of precedence of the connectives is $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. Since $\bot$ and $\top$ take no arguments, precedence is not applicable.

$\wedge$ and $\vee$ are left-associative: by $\varphi \wedge \psi \wedge \gamma$ we mean $(\varphi \wedge \psi) \wedge \gamma$ and by $\varphi \vee \psi \vee \gamma$ we mean $(\varphi \vee \psi) \vee \gamma$. $\rightarrow$ and $\leftrightarrow$ are right-associative, as by $\varphi \rightarrow \psi \rightarrow \gamma$ we mean $\varphi \rightarrow (\psi \rightarrow \gamma)$ and by $\varphi \leftrightarrow \psi \leftrightarrow \gamma$ we mean $\varphi \leftrightarrow (\psi \leftrightarrow \gamma)$

Example 1. $\top, P, \neg P$ *are literals;* $(Q \wedge (P_1 \vee \neg P_2)) \rightarrow (R \leftrightarrow \top)$ *is a formula.*

2.1.1.2 Semantics

In propositional logic all of the elements evaluate to either `true` or `false`.

To evaluate variables, we need an *interpretation*. An interpretation is a function $\mathcal{M} : V \rightarrow \{\texttt{true}, \texttt{false}\}$ that maps each variable symbol to a truth value. We will say that a variable P evaluates to `true` under interpretation $\mathcal{M}$ if $\mathcal{M}[P] = \texttt{true}$. Similarly, a variable P evaluates to `false` under interpretation $\mathcal{M}$ if $\mathcal{M}[P] = \texttt{false}$.

Example 2. *Under interpretation* $\mathcal{M} = \{P \rightarrow \texttt{true}, Q \rightarrow \texttt{false}\}$*, variable* P *evaluates to* `true` *and variable* Q *evaluates to* `false`*, as* $\mathcal{M}[P] = \texttt{true}$ *and* $\mathcal{M}[Q] = \texttt{false}$*.*

To evaluate more complex formulas, we have to define the semantics of the connectives. We will write $\mathcal{M} \models \varphi$ if φ evaluates to `true` under $\mathcal{M}$, and $\mathcal{M} \not\models \varphi$ if φ evaluates to `false` under $\mathcal{M}$.

$\mathcal{M} \not\models \bot$;
$\mathcal{M} \models \top$;
$\mathcal{M} \models \neg\varphi$ iff $\mathcal{M} \not\models \varphi$;
$\mathcal{M} \models \varphi \wedge \psi$ iff $\mathcal{M} \models \varphi$ and $\mathcal{M} \models \psi$;
$\mathcal{M} \models \varphi \vee \psi$ iff $\mathcal{M} \models \varphi$ or $\mathcal{M} \models \psi$;
$\mathcal{M} \models \varphi \rightarrow \psi$ iff $\mathcal{M} \not\models \varphi$ or $\mathcal{M} \models \psi$;
$\mathcal{M} \models \varphi \leftrightarrow \psi$ iff $\mathcal{M} \models \varphi$ iff $\mathcal{M} \models \psi$.

We say that a formula φ is *satisfiable* if there exists such an interpretation $\mathcal{M}$ that $\mathcal{M} \models F$, otherwise, we say that it is *unsatisfiable*.

Example 3. *The formula* $Q \wedge R$ *is satisfiable since for interpretation* $\mathcal{M} = \{Q \rightarrow \texttt{true}, R \rightarrow \texttt{true}, \ldots\}$*, we have* $\mathcal{M} \models Q \wedge R$*.*

The formula $P \wedge \neg P$ *is unsatisfiable, because every interpretation assigns to* P *either* `true` *or* `false`*, and the formula evaluates to false in both of those cases.*

2.1.2 First Order Logic

From our perspective, the principal advantage of first order logic (FOL) compared to propositional logic is that FOL can handle non-logical values (values that are not `true` nor `false`).

2.1.2.1 Syntax

In FOL, non-logical values are represented with *terms*. Let $\mathbb{V} = \{x, y, z, x_1, x_2, \ldots\}$ be a countable set of *variable symbols*. Let $\mathbb{F} = \{f, g, h, f_1, f_2, \ldots\}$ be a countable set of *function symbols*. The arity of function symbols is fixed. A term is

- a variable symbol $x \in \mathbb{V}$,
- or an n-ary function symbol $f \in \mathbb{F}$ applied to n terms $f(t_1, t_2, \ldots, t_n)$.

Let moreover $\mathbb{P} = \{p, q, r, p_1, p_2, \ldots\}$ be a countable set of *predicate symbols*. The arity of predicate symbols is fixed. An n-ary predicate symbol $p \in \mathbb{P}$ applied to n terms $p(t_1, t_2, \ldots, t_n)$ is a *formula*. More complex formulas can be built by using the same logical connectives as in PL, and by using quantifiers. If φ and ψ are formulas and $x \in \mathbb{V}$ is a variable symbol then the following are also formulas:

- $\bot$ (bottom),
- $\top$ (top),
- $\neg\varphi$ (negation),
- $\varphi \wedge \psi$ (conjunction),
- $\varphi \vee \psi$ (disjunction),
- $\varphi \rightarrow \psi$ (implication),
- $\varphi \leftrightarrow \psi$ (if and only if, iff),
- $\exists x.\varphi$ (existential quantification),
- $\forall x.\varphi$ (universal quantification).

An *atom* is a truth symbol or an n-ary predicate symbol $p \in \mathbb{P}$ applied to n terms $p(t_1, t_2, \ldots, t_n)$. A *literal* is an atom α or its negation $\neg\alpha$.

Symbols $\exists$ and $\forall$ are called *quantifiers*. In a formula $\exists x.\varphi$, x is the *quantified variable* and φ is the *scope* of the quantifier. Variable x is called a *bound* variable, while all other variables occurring in φ are *free*.

Example 4. *$p(f(x, y)) \vee \exists z.q(g(z), h)$ is a formula. The structure of the formula is the following:*

- *p is a unary predicate symbol applied to a term $f(x, y)$,*
- *f is a binary function symbol applied to two terms x and y,*
- *$g(z)$ is a unary function symbol applied to a term z,*
- *h is a nullary function symbol, and*
- *x, y and z are variable symbols.*

2.1.2.2 Semantics

Formulas in FOL are evaluated based on an interpretation. An *interpretation* is a pair $\mathcal{M} = (\mathbb{D}, \alpha)$. Here, $\mathbb{D}$ is an (either finite or infinite) nonempty set called the *domain*. It is the set of values that a term can take according to the interpretation. Moreover, α is a function called the *assignment* that assigns meaning to terms and formulas. It maps

- each variable symbol $x \in \mathbb{V}$ to an element $\alpha[x] \in \mathbb{D}$ of the domain,

- each n-ary function symbol $f \in \mathbb{F}$ to an n-ary function $\alpha[f] : \mathbb{D}^n \to \mathbb{D}$ over the domain,
- each n-ary predicate symbol $p \in \mathbb{P}$ to an n-ary relation $\alpha[p] \subseteq \mathbb{D}^n$ over the domain.

Values then can be assigned to more complex terms recursively. For the application of an n-ary function symbol f to terms $t_1, t_2, \ldots, t_n$, let $\alpha[f(t_1, t_2, \ldots, t_n)] = \alpha[f](\alpha[t_1], \alpha[t_2], \ldots, \alpha[t_n])$. We will say that the value of a term t under interpretation $\mathcal{M}$ is v iff $\alpha[t] = v$.

Given an interpretation $\mathcal{M} = (\mathbb{D}, \alpha)$, a variable $x \in \mathbb{V}$, and element $v \in \mathbb{D}$ of the domain, let $\mathcal{M} \triangleleft \{x \to v\}$ stand for the interpretation $(\mathbb{D}, \alpha')$ where $\alpha'[e] = v$ if $e = x$ and $\alpha'[e] = \alpha[e]$ otherwise. Formulas then can be evaluated recursively according to the following rules.

$$
\begin{array}{lll}
\mathcal{M} \models p(t_1, t_2, \ldots, t_n) & \text{iff} & (\alpha[t_1], \alpha[t_2], \ldots, \alpha[t_n]) \in \alpha[p]; \\
\mathcal{M} \not\models \bot; & & \\
\mathcal{M} \models \top; & & \\
\mathcal{M} \models \neg\varphi & \text{iff} & \mathcal{M} \not\models \varphi; \\
\mathcal{M} \models \varphi \land \psi & \text{iff} & \mathcal{M} \models \varphi \text{ and } \mathcal{M} \models \psi; \\
\mathcal{M} \models \varphi \lor \psi & \text{iff} & \mathcal{M} \models \varphi \text{ or } \mathcal{M} \models \psi; \\
\mathcal{M} \models \varphi \to \psi & \text{iff} & \mathcal{M} \not\models \varphi \text{ or } \mathcal{M} \models \psi; \\
\mathcal{M} \models \varphi \leftrightarrow \psi & \text{iff} & \mathcal{M} \models \varphi \text{ iff } \mathcal{M} \models \psi. \\
\mathcal{M} \models \exists x.\varphi & \text{iff} & \mathcal{M} \triangleleft \{x \to v\} \models \varphi \text{ for some } v \in \mathbb{D}; \\
\mathcal{M} \models \forall x.\varphi & \text{iff} & \mathcal{M} \triangleleft \{x \to v\} \models \varphi \text{ for all } v \in \mathbb{D}.
\end{array}
$$

As in the case of PL, we will say that a formula φ is *satisfiable* if there exists an interpretation $\mathcal{M}$ such that $M \models \varphi$, otherwise, we say that it is *unsatisfiable.*

For simplicity, we will often write $\alpha \models \varphi$ for $(\mathbb{D}, \alpha) \models \varphi$.

Example 5. *Consider the interpretation* $\mathcal{M} = (\mathbb{Z}, \alpha)$ *where*

- $\alpha[x] = 3$,
- $\alpha[y] = 9$,
- $\alpha[z] = 0$,
- $\alpha[f] = +_{\mathbb{Z}}$,
- $\alpha[g] = x \mapsto x +_{\mathbb{Z}} (-100)$,
- $\alpha[h] = 42$
- $\alpha[p] = \{x \mid x <_{\mathbb{Z}} 0\}$,
- $\alpha[q] = <_{\mathbb{Z}}$.

$\mathcal{M} \not\models p(f(x, y))$, *as* $\alpha[f](\alpha[x], \alpha[y]) = 3 +_{\mathbb{Z}} 9 = 12$, *and* $12 \notin \alpha[p]$.

$\mathcal{M} \models \exists z.q(g(z), h)$, *as* $\mathcal{M}' \models q(g(z), h)$ *for* $\mathcal{M}' = (\mathbb{Z}, \alpha') = \mathcal{M} \triangleleft \{z \to 100\}$, *since* $\alpha'[g](\alpha'[z])] = 100 +_{\mathbb{Z}} (-100) = 0$ *and* $(0, \alpha'[h]) = (0, 42) \in \alpha'[q]$.

$\mathcal{M} \models p(f(x, y)) \lor \exists z.q(g(z), h)$ *as* $\mathcal{M} \models \exists z.q(g(z), h)$.

Unless we say otherwise, the formulas we use are quantifier-free, meaning that they contain no quantifiers.

2.1.3 Satisfiability Modulo Theories

As general first order logic is undecidable [2], applications often restrict satisfiability checking to some background theory $\mathcal{T}$. A theory can be defined by the class of $\mathcal{T}$-interpretations. Solving *satisfiability modulo theories* (SMT) amounts to checking whether, given a theory $\mathcal{T}$, a formula φ is satisfied under some $\mathcal{T}$-interpretation $\mathcal{M}$, denoted by $\mathcal{M} \models_{\mathcal{T}} \varphi$. Theories commonly used in practice have efficient decision procedures for satisfiability of their quantifier-free fragments.

A theory typically fixes the interpretation of certain symbols. Such symbols are called *interpreted symbols*, while symbols whose interpretation may vary between $\mathcal{T}$-interpretations are called *uninterpreted symbols*.

The theory we are going to use is linear integer arithmetic, or $\mathcal{LA}(\mathbb{Z})$ for short. In $\mathcal{LA}(\mathbb{Z})$-interpretations, the domain is fixed as $\mathbb{Z}$, and constant symbols 0, 1, $\ldots$, function symbols $+$ and $-$, and predicate symbols $<$, $>$, $\leq$, $\geq$, $=$ and $\neq$ are interpreted as usual for integers.

Example 6. *Consider the formula* $\varphi = x+y \leq x-y$. *Let* $\mathcal{M}$ *be the* $\mathcal{LA}(\mathbb{Z})$*-interpretation with* $\alpha[x] = 2$ *and* $\alpha[y] = -3$. *Then* $\mathcal{M} \models_{\mathcal{LA}(\mathbb{Z})} \varphi$, *because* $2 + (-3) \leq 2 - (-3)$.

In the future we will use $\models$ instead of $\models_{\mathcal{T}}$, unless the context makes it ambiguous.

2.2 Modeling Formalisms

The purpose of formal verification is to prove the safety of systems in a formal manner. To achieve this, we formalize both the behavior and the requirements of the system under consideration. The two kinds of models we use are transition systems and control flow automata.

2.2.1 Transition Systems

Transition systems are models of discrete systems. They are mostly used to describe hardware such as sequential logic circuits.

2.2.1.1 Transition Systems in General

A transition system is a quadruple: $W = (S, I, T, P)$, where

- S is the set of states,
- $I \subseteq S$ is the set of initial states,
- $T \subseteq S \times S$ is the set of allowed transitions and
- $P \subseteq S$ is the set of safe states.

Example 7. $W = (\{a, b, c, d\}, \{a\}, \{(a, b), (c, a), (c, d), (d, b)\}, \{a, b, c\})$ *is the transition system visualized on Figure 2.1. It has four states:* a, b, c *and* d. *The single initial state is* a. *The transitions of the system are* $(a, b), (c, a), (c, d)$ *and* (d, b). *The safe states are* a, b *and* c, *thus* d *is unsafe.*

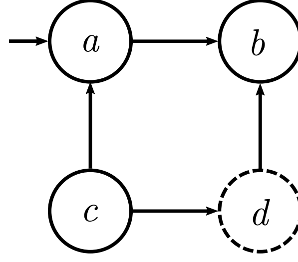

Figure 2.1: The transition system described in Example 7.

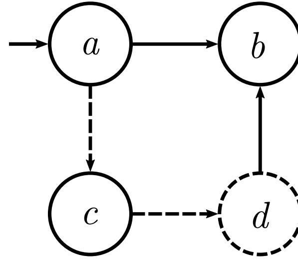

Figure 2.2: The transition system described in Example 9, with the counterexample highlighted with dashes.

A *path* is a sequence of states $\pi = s_0 s_1 s_2 \dots s_n$ such that $(s_i, s_{i+1}) \in T$ for $0 \leq i < n$. An *initial path* is a path where $s_0 \in I$. An *error path* is a path where $s_n \in \overline{P}$. We call a state s *reachable* iff there exists an initial path such that $s_n = s$.

Example 8. *In the transition system W of Example 7, (cdb) is a path, as $(c, d) \in T$ and $(d, b) \in T$. (acd) is not a path, as $(a, c) \notin T$.*

P is the set of states that are considered safe or “good”. States outside P are called unsafe or “bad” states. We call the transition system *safe* if all reachable states are safe. Similarly, the system is *unsafe* if it has an initial error path. Such a path is called a *counterexample.*

Example 9. *The system $W' = (\{a, b, c, d\}, \{a\}, \{(a, b), (a, c), (c, d), (d, b)\}, \{a, b, c\})$, which we obtained by flipping the transition (c, a) in W, is unsafe. A counterexample, as shown in Figure 2.2, is the path (acd).*

A set of states $R \subseteq S$ is an invariant iff

1. $I \subseteq R$, and
2. R is closed under T: for all $(s, s') \in T$, if $s \in R$, then $s' \in R$.

If also $R \subseteq P$, then we call it a *safety invariant.*

Example 10. *In the transition system W of Example 7, the set of reachable states is $\{a, b\}$. The set $R_1 = \{a, b, d\}$ is an invariant, as $I \subseteq R$ and there are no transitions pointing out of it. Although it is an invariant, it is not a safety invariant, since $d \in R \cap \overline{P}$. $R_2 = \{a, b\}$, however is a safety invariant, since $I \subseteq \{a, b\}$ and $\{a, b\} \cap \overline{P} = \emptyset$.*

Proposition 1. *A transition system W has a safety invariant R if and only if W is safe.*∎

Proof. First we prove that if the system is safe, then it has a safety invariant. Since it is safe, all of the reachable states are safe and the set of reachable states are obviously an invariant, thus it is a safety invariant.

For the other direction, assume that the statement is false. That is W has a safety invariant R, and W is unsafe: it has a reachable bad state $s_B \notin P$. Because s_B is reachable, it has an initial path leading to it: $s_0 s_1 \dots s_B$. Since $s_0 \in I$ and $I \subseteq R$ (because R is a safety invariant), $s_0 \in R$. $R \subseteq P$, therefore $s_B \notin R$. The first element of the path is in R, and the last element is not, hence there must be two consecutive elements s_i, s_{i+1} such that $s_i \in R$ and $s_{i+1} \notin R$. $(s_i, s_{i+1}) \in T$ as they are subsequent elements of a path, but that contradicts with R being an invariant, since there is an edge between an element of R and a state outside R. □

2.2.1.2 Symbolic Transition Systems

A symbolic transition systems (STS) over a $\mathcal{T}$ is a transition systems where sets of states and transitions are represented by $\mathcal{T}$-formulas. Let $V = \{v_1, v_2, \dots, v_n\}$ be the set of variable of the system, and $V' = \{v'_1, v'_2, \dots, v'_n\}$ be the set of primed variables.

A state of the system is an assignment of values to all of these variables.

Formally a symbolic transition system over a set of variables V is a triple $(\varphi_I, \varphi_T, \varphi_P)$ where φ_I and φ_P are formulas over V and φ_T is a formula over $V \cup V'$.

Semantically, the symbolic transition system $(\varphi_I, \varphi_T, \varphi_P)$ represents the transition system $W' = (S, I, T, P)$ where

- S is the set of assignments over V,
- $I = \{s \mid s \models \varphi_I\}$,
- $T = \{(s, s') \mid s \cup s' \models \varphi_T\}$,
- $P = \{s \mid s \models \varphi_P\}$.

With this definition we can represent a set of states as a formula and vice versa. We say that a formula φ holds in a state s, or that s is a φ-state, iff $s \models \varphi$. The set of φ-states is then $\{s \mid s \models \varphi\}$.

In the future we will denote the formulas of an STS with I, T and P, instead of φ_I, φ_T and φ_P, respectively.

Example 11. *Let the theory $\mathcal{T}$ fix the domain as $\mathbb{D} = \{\circ, \bullet\}$, and fix the unary predicate symbol p as the set $\{\circ\}$. Take a symbolic transition system (I, T, P) over $\mathcal{T}$ with variables $V = \{x, y\}$ where*

- $I(x, y) = \neg p(x) \wedge \neg p(y)$,
- $T(x, y, x', y') = (\neg p(x) \wedge \neg p(y) \wedge p(x') \wedge \neg p(y')) \vee (p(y) \wedge \neg p(y'))$, *and*
- $P(x, y) = \neg p(x) \vee \neg p(y)$.

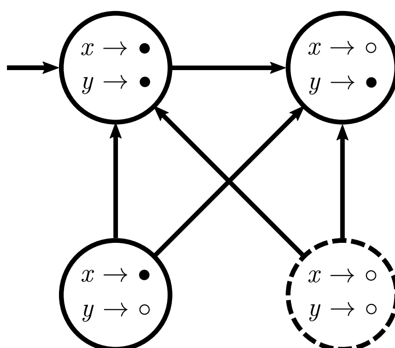

Figure 2.3: The symbolic transition system described in Example 11

The system is visualized on Figure 2.3.

In this case the initial formula is $I(x, y) = \neg p(x) \wedge \neg p(y)$*, which is only true for the state in which variables are assigned* $\bullet$*.*

The transition formula is $T(x, y, x', y') = (\neg p(x) \wedge \neg p(y) \wedge p(x') \wedge \neg p(y')) \vee (p(y) \wedge \neg p(y'))$*. The first part is true for the edge leading from the state where both variables are* $\bullet$ *to the state where* x *is* $\circ$ *and* y *is* $\bullet$*. The second part is true for all the edges leading from states where* y *is* $\circ$ *to states where* y *is* $\bullet$*. The two parts are connected by a disjunction, therefore the whole formula is true for the union of the two sets.*

The property is $P(x, y) = \neg p(x) \vee \neg p(y)$*. This is only false for the state where both variables are* $\circ$*, therefore that state is unsafe, and all the other states are safe.*

2.2.2 Control Flow Automata

Control flow automata (CFA) are structurally more complex than transition systems. They model software systems, and therefore their basis is a control flow graph. The automaton operates in discrete time, and at each point in time it is in one of the nodes of its control flow graph. Transitions can change the node the automaton is at, and they execute a statement. CFA also have a set of variables which the statements manipulate.

2.2.3 Syntax

Formally a control flow automaton over a theory $\mathcal{T}$ and set of variables V is tuple $A = (L, E, \ell_0, \ell_e)$ where

- L is a finite set of locations of the control flow graph,
- $E \subseteq L \times \mathit{Stmt} \times L$ is a finite set of edges, where for an edge $(\ell, s, \ell') \in E$,
  - $\ell \in L$ is the source location,
  - $s \in \mathit{Stmt}$ is a statement to be executed upon traversing the edge,
  - $\ell' \in L$ is the target location;

- $\ell_0 \in L$ is the initial location,
- $\ell_e \in L$ is the error location.

A statement $s \in \mathit{Stmt}$ is

- an assignment $x \coloneqq t$ where $x \in V$ is a variable and t is a term, or
- an assumption $\texttt{assume}\, \varphi$ where φ is a formula, or
- A statement of the form $\texttt{havoc}\, x$ where $x \in V$ is a variable.

A *path* (of length k) in a control flow automaton is a sequence of edges $\mu = (\ell_0, s_1, \ell_1)\,(\ell_1, s_2, \ell_2) \dots (\ell_{k-1}, s_k, \ell')$. ℓ_0 is the initial node of the automaton, as all CFA paths start in the initial node. An *error path* is a path where $\ell' = \ell_e$.

2.2.4 Semantics

At any point in time, the state of the automaton is characterized by a pair (ℓ, α) where $\ell \in L$ is its current location and α is an assignment over V that encodes the current value of the variables of the automaton.

For a statement $s \in \mathit{Stmt}$, we define δ_s to be its transition formula. Let $\mathit{same}(X) \mathrel{\hat{=}} \bigwedge_{x \in X} x' = x$. Then

$$\delta_s = \begin{cases} x' = t \wedge \mathit{same}(V \setminus \{x\}) & \text{if } s = (x \coloneqq t) \\ \varphi \wedge \mathit{same}(V) & \text{if } s = \texttt{assume}\, \varphi \\ \mathit{same}(V \setminus \{x\}) & \text{if } s = \texttt{havoc}\, x \end{cases}$$

We say that a path $\mu = (\ell_0, s_0, \ell_1)\,(\ell_1, s_1, \ell_2) \dots (\ell_{n-1}, s_{n-1}, \ell_n)$ is feasible iff the formula $\bigwedge_{i=0}^{n-1} \delta_{s_i}^{(i)}$ (where $\delta_{s_i}^{(i)}$ is δ_{s_i} with i primes applied to its variables) is satisfiable under $\mathcal{T}$. That is there exist assignments $\alpha_i^{(i)} : X^{(i)} \to \mathbb{D}$ such that $\bigcup_{i=1}^{n} \alpha_i^{(i-1)} \models \bigwedge_{i=0}^{n-1} \delta_{s_i}^{(i)}$

We say that a node ℓ is reachable in a control flow automaton iff there is a feasible path $e_1 e_2 \dots e_n$ for which the target node of e_n is ℓ.

The decision problem of control flow automata is whether the error location ℓ_f is reachable. A counterexample is a feasible path to the final node.

Example 12. *Let the theory $\mathcal{T}$ fix the domain as $\mathbb{Z}$, fix the value of the function $+$ as $+_{\mathbb{Z}}$, $\cdot$ as $\cdot_{\mathbb{Z}}$ and fix the value of predicate $<$ as $<_{\mathbb{Z}}$. The CFA $A = (\{a, b, c, d, f\}, \{e_1, e_2, e_3, e_4, e_5\}, a, f)$ where*

- $e_1 = (a, x \coloneqq 0, b)$,
- $e_2 = (b, \texttt{havoc}\, y, c)$,
- $e_3 = (c, y \coloneqq x \cdot y, d)$,
- $e_4 = (d, x \coloneqq x + 1, b)$,
- $e_5 = (d, \texttt{assume}\, y > 20, f)$.

Table 2.1: Example assignments to the variables of A in Example 12, as it runs through the path μ_1

Location	x	y	Next statement
a	36	42	$x := 0$
b	0	42	$\texttt{havoc}\, y$
c	0	10	$y := x \cdot y$
d	0	0	$x := x + 1$
b	1	0	$\texttt{havoc}\, y$
c	1	21	$y := x \cdot y$
d	1	42	$\texttt{assume}\, y > 20$
f	1	42	

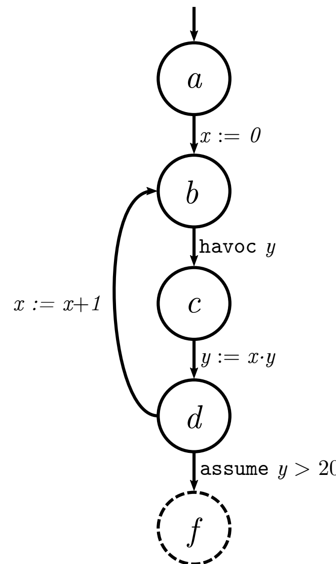

Figure 2.4: The control flow automaton described in Example 12

is visualized on Figure 2.4.

$\mu_1 = e_1e_2e_3e_4e_2e_3e_5$ *is a feasible path, a sample assignment to the variables along its execution can be seen in Table 2.1.*

μ_1 *starts from the initial node and leads to the finial node, therefore this is an initial error path, thus the final node is reachable and the system is unsafe.*

$\mu_2 = e_1e_2e_3e_5$ *is not a feasible path, because when statement* $y := x \cdot y$ *is executed we have* $x = 0$*, and* y *cannot have a value such that* $0 \cdot y > 20$*. Thus the statement on* e_5 *cannot be executed.*

Chapter 3

The Algorithm

In this chapter we present PDR, first in a high level, then in a lower level. We introduce implicit predicate abstraction, and describe how we extend PDR with it. Finally we present our transformation of control flow automata to symbolic transition systems, and prove that it is correct.

3.1 Property Directed Reachability

Property directed reachability (PDR) [5] is a model checking algorithm. It checks reachability properties in transition systems. It analyzes the system incrementally, through the proof of several lemmas, which eventually either form an inductive property that proves the system correct, or which direct the search towards a counterexample.

3.1.1 High Level Description

This chapter introduces PDR in a high level, discussing how it operates over a general transition system.

3.1.1.1 Trace

The algorithm has very few information of the system initially, then as it runs, it deduces more and more. It stores its current view of the state space in a list $[F_0, F_1, \cdots, F_N]$ called the *trace*. An element F_i of the trace is called a *frame*, and is a set of states.

Let R_i denote the set of states reachable from the initial states in at most i transitions. At the beginning this information is known only for $i = 0$, because R_0 is the set of initial states, and that is part of the given STS model. During the run of PDR, the i-th frame is maintained so that it is an overestimation of R_i. More specifically $F_0 = R_0$ and $\forall i. R_i \subseteq F_i$. In other words all the states reachable in i steps or less are in F_i, but there may be other states in it as well. PDR initializes all frames (except F_0) to include the whole state space. It learns later to exclude states by disproving their reachability.

PDR also maintains the relation that $\forall i < j. F_i \subseteq F_j$. The idea comes naturally: if a state is reachable in at most i steps, it is also reachable in at most j steps. However the overestimations do not intrinsically satisfy that.

3.1.1.2 Proof Obligations

When PDR finds that one of the frames intersect with bad states, it looks for a specific state in the intersection, and creates a *proof obligation.* Upon processing the proof obligation it tries to prove the states unreachable: it tries to *block* them. A proof obligation includes the state (or set of states) to be blocked and the frame number to block them from.

A proof obligation shows a lack of knowledge, and a target to learn: according to the information PDR has, a bad state may or may not be reachable. It is obligated to learn which case it is and it does so by proving the state reachable or unreachable.

3.1.1.3 Main Loop

PDR initially only knows about the initial states (F_0) and assumes that all further frames include the whole state space, because that is an obvious overestimation of the reachable states. It refines this knowledge in a directed way, by learning about the reachability of bad states and their predecessors through proof obligations.

PDR iterates over the frames and tries to disprove the reachability of bad states in each frame.

First of all it checks if there are any bad states in F_0, i.e. whether $F_0 \cap \overline{P} = \emptyset$. If there is a bad state among the initial states, then that is an obvious proof of the system being unsafe, and the algorithm can terminate.

If the initial states are all good states, it continues looking at further frames. When PDR looks at F_i, it looks for a state s in the intersection of F_i and $\overline{P}$. If it finds such a bad state, it needs to learn whether it is actually reachable, so it creates a proof obligation and processes it. The algorithm continues looking for bad states until $F_i \cap \overline{P} = \emptyset$.

When PDR reaches the point that F_i contains no more bad states, it has learned that

- no bad states are reachable in at most i steps
- no state in F_{i-1} are bad states or direct predecessors of bad states i.e. such that a bad state is reachable from them in at most one step
- no state in F_{i-2} is such that a bad state would be reachable from it in at most two steps
- etc.

3.1.1.4 Processing Proof Obligations

When a proof obligation arises, it means that according to the current knowledge the algorithm has, a bad state s seems to be reachable in at most i steps. When the proof obligation is processed, either a counterexample is produced, proving the system unsafe, or s is proven to be unreachable in at most i steps and is taken out of F_i.

To find out which is the case, PDR checks if s has any predecessors r in F_{i-1}. If such a state r is found, then it can be treated the same way as s: should r prove to be reachable, s would also be reachable, and they would be two steps in a counterexample. Hence another proof obligation is made from r and $i-1$, and it is processed the same way recursively: PDR tries to block r from F_{i-1}. To block it, it might have to raise even more proof

obligations. If r is blocked, the algorithm has to look for other predecessors, and process them the same way.

If there are no more predecessors of s in F_{i-1}, then s is proven to be unreachable in i steps or less, because F_{i-1} contains all the states reachable in at most $i-1$ steps, and none of them is a predecessor of s. If s intersected with the initial states, it could be reachable without having any predecessors. It is shown in Section 3.1.1.5 that this does not happen. Hence, s can be excluded from F_i, and it will remain an overestimation of the states reachable in at most i steps. To ensure that the earlier frames are subsets of the latter, s is also blocked from all earlier frames. This is sound, as the previous frames are subsets of F_{i-1}, and F_{i-1} does not contain any predecessors of s, therefore the previous frames do not either.

If a proof obligation would be created to block the state q with the frame number being 0, it means that q is the first state in a counterexample. We know that q is an initial state, because we intend to block it from the initial frame. The initial states are known to be reachable, q is too. All proof obligations are created because if an initial path lead to the state to be blocked, then it would form a counterexample. Therefore a path exists from q to a bad state, and that path is a counterexample proving the unsafety of the system.

Optimization: pushing proof obligations forward in the trace When we prove a set of states unreachable in a certain frame, we see that it is a direction of learning that is useful. Those frames might become reachable eventually, and since they have a path leading from them to a bad state, we will have to prove them unreachable in every frame eventually. Experiments show that we can make the algorithm quicker, especially for unsafe systems, if we create a proof obligation to block the same set of states from the next frame.

3.1.1.5 Initial States in Proof Obligations

Proposition 2. *PDR never creates proof obligations that overlap initial states but have a non-zero frame number.* ▪

Proof. PDR creates proof obligations three ways.

The first way is when it finds a bad state s in a frame. State s cannot be initial, because if the intersection of initial states and error states is not empty, then the algorithm terminates before the main loop.

The second way proof obligations are created is to block a set B of predecessors from F_i (assume $i > 0$) during the recursive blocking of a bad state s from F_k. When we get to frame F_k with the main loop, we are already done with F_{k-1}, so we know, that no bad state is reachable in at most $k-1$ steps. The proof obligations created during the process of recursively blocking the predecessors of s go back at most $k-1$ steps, because each step decreases the frame number by one, and if it reaches 0, a counterexample is found. If B overlapped with the initial states, then s would be reachable in $k-i$ steps, and the algorithm would terminate when processing the frame F_{k-i}, before it would get to block s from F_k.

The third way a proof obligation is created is when we push a blocked proof obligation forward: we try to block from F_{k+1} a set of states B, that was successfully blocked from F_k. In that case the the main loop might be behind the F_{k+1}. However pushing the obligations happens sequentially increasing the frame number one-by-one, so by the time

it gets pushed to frame F_{k+1}, it is proven to be unreachable in at most k steps from the initial states. Therefore if B overlapped with the initial states, then it would have been reachable in k steps, and the algorithm would terminate before pushing it forward. □

3.1.1.6 Termination

PDR terminates either when it finds a counterexample or when two consecutive frames become equal: $F_i = F_{i+1}$, and the main loop is past them, so $F_i \cap \neg P = \emptyset$. That happens when all the states that are not in F_i are blocked from F_{i+1}. A state is blocked from F_{i+1} if it has no predecessors in F_i. This is true for all the states outside of F_i since they were blocked. Thus the successors of all the states in F_i are also in F_i. That means that no transitions lead out of F_i and that the system can not get out of F_i. Because $F_1 \subseteq F_i$, all of the initial states are in F_i. This means that F_i is an invariant. In other words all the reachable states are in F_i. Since $F_i \cap \overline{P} = \emptyset$, all the states in F_i are safe, therefore F_i is a safety invariant. According to Proposition 1 this means that the system is safe.

3.1.2 Lower Level Description

Sets of states discussed in the previous section are represented as predicates in this section. As its input PDR receives an STS model $W = (I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'), P(\mathbf{x}))$ interpreted over $\mathcal{LA}(\mathbb{Z})$. STS models are described in Section 2.2.1.2.

3.1.2.1 Trace

The trace is a list of frames. Each frame $F_i(\mathbf{x})$ is a formula, that represents an overestimation of the states reachable in at most i steps. The other property of the frames is that for all $i < j$, $F_i(\mathbf{x})$ implies $F_j(\mathbf{x})$.

3.1.2.2 Proof Obligations

Proof obligations are a pair of a number and a formula: $(i, B(\mathbf{x}))$. The states that satisfy B are states that need to be blocked from F_i.

3.1.2.3 Main Loop

Initially $F_0(\mathbf{x}) = I(\mathbf{x})$ and for all $i > 0$, $F_i(\mathbf{x}) = \top$. It is checked first if $F_0(\mathbf{x}) \wedge \neg P(\mathbf{x})$ is satisfiable using an SMT solver. If the formula is satisfiable, the solver can give a model of the formula, which is a counterexample itself, and the algorithm terminates. If the formula is unsatisfiable, that means that there is no bad initial state, and the algorithm can continue checking further frames.

When looking at the i-th frame, PDR checks if $F_i(\mathbf{x}) \wedge \neg P(\mathbf{x})$ is satisfiable. If it is, then its interpretation $\mathcal{M}$ is turned into a formula $B(\mathbf{x})$, and a proof obligation $(i-1, B(\mathbf{x}))$ is created and processed. Then either a counterexample is found, or F_i is refined. The algorithm continues checking $F_i(\mathbf{x}) \wedge \neg P(\mathbf{x})$ and blocking states until it is no longer satisfiable, than proceeds to the next frame.

Algorithm 1: Main loop

Input: A symbolic transition system $W = (I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'), P(\mathbf{x}))$

Output: Whether the transition system is safe

```
1  if I(x) ∧ ¬P(x) is satisfiable then
2  |   return false
3  end
4  F_0 ← I(x)
5  F_1 ← ⊤
6  n ← 1
7  while true do
8  |   if F_n ∧ ¬P(x) is satisfiable then
9  |   |   Let M = (𝔻, α) be a satisfying interpretation.
10 |   |   B ← ⋀_{x∈x} x = α[x]
11 |   |   Block the proof obligation (n, B)
12 |   |   if it cannot be blocked then
13 |   |   |   return false
14 |   |   end
15 |   end
16 |   else
17 |   |   Propagate blocked states
18 |   |   if equal frames found then
19 |   |   |   return true
20 |   |   end
21 |   |   else
22 |   |   |   n ← n + 1
23 |   |   |   F_n ← ⊤
24 |   |   end
25 |   end
26 end
```

3.1.2.4 Processing Proof Obligations

When a proof obligation $(i, B(\mathbf{x}))$ arises, it means that the states satisfying $B(\mathbf{x})$ appear to be reachable in i steps, but if they are indeed reachable, then the system is unsafe. To find out if the states are reachable, PDR tries to see if the states have reachable predecessors. It checks whether $F_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \wedge B(\mathbf{x}')$ is satisfiable.

If it is, then it needs to learn if the found predecessor, the assignment of values to $\mathbf{x}$ is reachable. Therefore it creates and processes a proof obligation from that and $i - 1$.

If it is not satisfiable, then it is proven that $F_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \implies \neg B(\mathbf{x}')$. In other words, the states satisfying $B(\mathbf{x}')$ have no direct predecessors in any of the first $i - 1$ frames and as shown in Section 3.1.1.5, the set $B(\mathbf{x})$ represents does not overlap with initial states. Thus B states are not initial and not a successor of any state reachable in at most $i - 1$ steps, so they are not reachable in at most i steps.

Optimization: Strengthen query When processing proof obligations, the original query

$$F_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \wedge B(\mathbf{x}')$$

can be optimized by adding $\neg B(\mathbf{x})$ to it:

$$F_{i-1}(\mathbf{x}) \wedge \neg B(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \wedge B(\mathbf{x}').$$

This formula is more likely to be unsatisfiable, yet it remains correct.

Proof. Four cases are possible:

1. both queries are satisfiable;
2. both queries are unsatisfiable;
3. the strengthened query is satisfiable, but the original one is not;
4. the original query is satisfiable, but the strengthened one is not.

In Case 1, we find a predecessor with the strengthened query, and we proceed by blocking it.

In Case 2, we block B from F_i, and since the original query is unsatisfiable, that is correct.

Case 3 is impossible, since if there is an interpretation $\mathcal{M}$, for which

$$\mathcal{M} \models_{\mathcal{LA}(\mathbb{Z})} F_{i-1}(\mathbf{x}) \wedge \neg B(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \wedge B(\mathbf{x}')$$

then

- $\mathcal{M} \models F_{i-1}(\mathbf{x})$ and
- $\mathcal{M} \models \neg B(\mathbf{x})$ and
- $\mathcal{M} \models T(\mathbf{x}, \mathbf{x}')$ and
- $\mathcal{M} \models B(\mathbf{x}')$,

therefore

$$\mathcal{M} \models F_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \wedge B(\mathbf{x}')).$$

Case 4 is where the strengthened query works better. We prove by induction, that in this case B is unreachable in i steps or less. Since the original query is satisfiable, B states do have predecessors in F_{i-1}, but since the strengthened query is unsatisfiable, all of the predecessors are also B states. For $1 \leq j \leq i$, F_{j-1} is a subset of F_{i-1}, therefore B states in F_i have only B state predecessors in F_{i-1}. Since $B(\mathbf{x})$ is part of a proof obligation, according to Proposition 2, it does not intersect with the initial states. Thus there are no B states in F_0, i.e. B states are unreachable in 0 steps. Assume that B states are unreachable in $j < i$ steps. F_{j+1} is an overapproximation of the states reachable in $j+1$ steps, and as shown before, B states in F_{j+1} have only B state predecessors in F_j, none of which is reachable in j steps, thus none of the B states is reachable in $j+1$ steps. Therefore B states are unreachable in i steps and can be blocked from F_i.

In conclusion if the strengthened query is unsatisfiable, $B(\mathbf{x})$ can be blocked, and if it is satisfiable, then the found predecessors are truly predecessors of B states. □

Optimization: Generalize blocked states Blocking the states one by one can take a really long time. When a state is blocked, it is blocked because a query

$$F_{i-1}(\mathbf{x}) \wedge \neg B(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \wedge B(\mathbf{x}').$$

is UNSAT. As a consequence we can block all states satisfying B from F_i. However, sometimes we can block a larger set of states. Many SMT and SAT solvers can give a core contradiction to show why the query is unsatisfiable, Z3, the solver we use is one of those. This core contradiction is called an UNSAT core and it is an unsatisfiable subformula of the query. The subformula B_U of B that is in the UNSAT core, is satisfied by more states than B. While the states satisfying B_U do not have predecessors in F_{i-1}, they might be initial states, and initial states must not be blocked, even if they have no predecessors, since they are reachable. Therefore we leave out parts of B that are not in B_U until we get to a formula that intersects with the initial states or until we get to B_U.

We can also use the UNSAT core to learn which frame to block B from. Normally we would block it from F_i, however if the subformula of F_{i-1} used by the UNSAT core is also in an F_{j-1} where $j > i$, then we can block B from F_j.

Optimization: Generalize predecessors When the query is satisfiable, the solver gives an interpretation $\mathcal{M}$ that satisfies the query. $\mathcal{M}$ assigns value to all of the symbols in the query, however, there can be variables to which any value from the domain can be assigned, and the modified interpretation still satisfies the query. An example is $x < 0 \vee y > 0$: if we only assign x to be less than 0, then whatever value is assigned to y, the query is satisfied. By finding such variables, the algorithm can create a more general proof obligation, and learn quicker or find a counterexample quicker. The idea is to try to remove the assignments one by one, and if the result of the query is still unequivocally true, we don't use that assignment in the interpretation. After we try to remove all assignments, we end up with a local minimum of the number of assignments used, not a global one, but heuristics can be applied to the order in which we choose the variables.

Optimization: Propagate blocked states When states are blocked, they are blocked using the information available at the time. As the algorithm progresses, it learns more and more about the system and blocks more and more states. It is possible, that a formula $B(\mathbf{x})$ that was blocked from F_i, but was not blocked from F_{i+1} can be blocked at the later

Algorithm 2: Proof obligation handling

Input: A proof obligation (n, B), a symbolic transition system $W = (I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'), P(\mathbf{x}))$, and the trace $F_0 \dots F_n$

Output: Whether the proof obligation was blocked and possible modifications to the trace

```
1  if n = 0 then
2  |  return false
3  end
4  else
5  |  while F_{n-1} ∧ ¬B(x) ∧ T(x, x') ∧ B(x') is satisfiable do
6  |  |  Let M = (D, α ∪ α') be a satisfying interpretation where α assigns values
   |  |   to x and α' assigns values to x'
7  |  |  V ← {}
8  |  |  for x ∈ x do
9  |  |  |  if x can be assigned such a value that the formula would not hold then
10 |  |  |  |  V ← V ∪ {x}
11 |  |  |  end
12 |  |  end
13 |  |  D ← ⋀_{x∈V} x = α[x]
14 |  |  Block the proof obligation (n − 1, D)
15 |  |  if it cannot be blocked then
16 |  |  |  return false
17 |  |  end
18 |  end
19 |  Let U be the UNSAT core
20 |  C ← a subformula of B such that U is a subformula of C and C ∧ I(x) is
   |   unsatisfiable.
21 |  k ← the smallest frame number that blocks the part of F_{n-1} that is in U
22 |  F_k ← F_k ∧ ¬C
23 |  if k is not too large then
24 |  |  Block the proof obligation pairk + 1B
25 |  |  if it cannot be blocked then
26 |  |  |  return false
27 |  |  end
28 |  end
29 |  return true
30 end
```

frame after new information becomes available. If the query[1]

$$F_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \wedge \neg B(\mathbf{x}')$$

is unsatisfiable then it can be blocked. Using the UNSAT core we can possibly block a more general formula at a later frame. It can be beneficial to check occasionally during the run of the algorithm if there are such formulas and block them from the later frames. It can also lead to faster termination, as frames can become equal.

3.1.2.5 Termination

The algorithm can terminate in two ways: finding a counterexample or finding an inductive invariant proving safety. The first way is obvious to notice. The second way happens when two frames are equivalent: $F_i(\mathbf{x}) \iff F_{i+1}(\mathbf{x})$, as discussed before. Since it is always true that $F_i(\mathbf{x}) \implies F_{i+1}(\mathbf{x})$, it only remains to be checked that $F_i(\mathbf{x}) \impliedby F_{i+1}(\mathbf{x})$. This can be done by checking if $\neg F_i(\mathbf{x}) \wedge F_{i+1}(\mathbf{x})$ is satisfiable.

Algorithm 3: Propagate blocked states

Input: A symbolic transition system $W = (I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'), P(\mathbf{x}))$ and the trace $F_0 \dots F_n$

Output: Modifications to the trace and whether there are equal frames

```
1  for F_i ∈ (F_1 ... F_n) do
2      for parts G of F_i do
3          if F_i(x) ∧ T(x, x') ∧ G(x') is unsatisfiable then
4              Let U be the UNSAT core;
5              C ← a subformula of G such that U is a subformula of C and C ∧ I(x)
                 is unsatisfiable.;
6              k ← the smallest frame number that blocks the part of F_i that is in U;
7              F_k ← F_k ∧ ¬C;
8          end
9      end
10     if F_{i-1} = F_i then
11         return true
12     end
13 end
14 return false
```

3.2 Implicit Predicate Abstraction

Handling infinite state spaces is not efficient using simple PDR. When using the model given by the solver, we assign a single value to all of the variables. However if the domain is not finite, as e.g. in the case of $\mathcal{LA}(\mathbb{Z})$, then there is an infinite number of values that can be assigned to a variable. Therefore enumerating models one by one is in general not an option.

[1] Using the strengthened query would yield benefit here, since $B(\mathbf{X})$ is already blocked from F_{i-1}.

3.2.1 Predicate Abstraction

The idea of *predicate abstraction* [4] is to divide the infinite state space into a finite number of partitions. Partitions are defined by which predicates hold in them and which predicates do not.

Example 13. *If the set of predicates is* $\{(x < 10), (y < x)\}$ *then the state space is divided into 4 partitions:*

- *states for which* $(x < 10) \wedge (y < x)$ *holds,*
- *states for which* $\neg(x < 10) \wedge (y < x)$ *holds,*
- *states for which* $(x < 10) \wedge \neg(y < x)$ *holds,*
- *states for which* $\neg(x < 10) \wedge \neg(y < x)$ *holds.*

The partitions are obviously mutually exclusive, since every predicate in every state either holds or does not hold.

From the concrete STS $W = (I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'), P(\mathbf{x}))$, using the predicate set $\Pi = \{\Pi_1(\mathbf{x}), \Pi_2(\mathbf{x}), \ldots, \Pi_n(\mathbf{x})\}$ we create an abstract STS $\hat{W} = (\hat{I}(\hat{\mathbf{x}}), \hat{T}(\hat{\mathbf{x}}, \hat{\mathbf{x}}'), \hat{P}(\hat{\mathbf{x}}))$. The abstract STS operates in the abstract state space defined by n propositional variables in $\hat{\mathbf{x}}$ corresponding to the n predicates. We note $(\bigwedge_{i=1}^{n} \hat{\mathbf{x}}_i \leftrightarrow \Pi_i(\mathbf{x}))$ by $\Lambda(\hat{x}, x)$.

- $\hat{I}(\hat{\mathbf{x}}) = \exists \mathbf{x}. I(\mathbf{x}) \wedge \Lambda(\hat{x}, x)$.
- $\hat{T}(\hat{\mathbf{x}}, \hat{\mathbf{x}}') = \exists \mathbf{x}, \mathbf{x}' \wedge (\mathbf{x}, \mathbf{x}')\Lambda(\hat{x}, x) \wedge (\bigwedge_{i=1}^{n} \hat{\mathbf{x}}_i' \leftrightarrow \Pi_i(\mathbf{x}'))$.
- $\hat{P}(\hat{\mathbf{x}}) = \forall \mathbf{x}. \Lambda(\hat{x}, x) \rightarrow P(\mathbf{x})$.

We can deduce that $\neg\hat{P}(\hat{\mathbf{x}}) = \neg\left(\Lambda(\hat{x}, x) \rightarrow P(\mathbf{x})\right) = \Lambda(\hat{x}, x) \wedge P(\mathbf{x})$

In the algorithm we are only interested in the satisfiability of these formulas, thus we use them without the quantifier. Since $\exists x.\varphi$ is satisfiable iff φ is satisfiable, it is sound to do so.

When it starts, the algorithm extracts the predicates from $I(\mathbf{x})$ and $P(\mathbf{x})$, and proceeds to run on the abstract automaton characterized by those predicates.

When an abstract counterexample $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n$ is found, and the set of predicates is $Pi = \{\Pi_1, \Pi_2, \ldots \Pi_m\}$ it is checked in the concrete state space. The "refute all paths" strategy is to disregard the counterexample and check if an error state is reachable in n steps, by checking the satisfiability of the formula

$$I(\mathbf{x}) \wedge \left(\bigwedge_{i=0}^{n-1} T\left(\mathbf{x}^{(i)}, \mathbf{x}^{(i+1)}\right)\right) \wedge \neg P\left(x^{(n)}\right).$$

The "refute specific path" strategy is to check if the specific n step long path, the counterexample is a path in the concrete state space by checking the satisfiability of the formula

$$I(\mathbf{x}) \wedge \left(\bigwedge_{i=0}^{n-1} \Lambda(\hat{s}_i[\hat{x}], x) \wedge T\left(\mathbf{x}^{(i)}, \mathbf{x}^{(i+1)}\right)\right) \wedge \Lambda(\hat{s}_n[\hat{x}], x) \wedge \neg P\left(x^{(n)}\right)$$

where $\hat{s}_i[\hat{\mathbf{x}}]_j$ is $\top$ if $\hat{s}_i$ assigns `true` to $\hat{\mathbf{x}}_j$, and it is $\bot$ otherwise.

If the query is satisfiable, that means that the concrete system is unsafe, and the algorithm terminates.

If the query is unsatisfiable, then the abstraction must be refined, because the abstract system is unsafe, but the concrete one may not be. The algorithm needs to add predicates to Pi so that it could block the states of the counterexample.

3.2.2 Abstraction Refinement

There are several methods for learning new predicates to exclude spurious counterexamples from the abstract state space. One of the most prominent ways for abstraction refinement is based on interpolation [8].

An *interpolant* for a sequence of n formulas $\Gamma = \psi_1, \psi_2, \dots, \psi_n$ is a sequence of $n+1$ formulas $\Theta = I_0, I_1, \dots, I_n$ such that

- $I_0 = \top$,
- for all $1 \leq i \leq n$, $I_{i-1} \wedge \psi_i$ implies I_i,
- $I_n = \bot$,
- for all $0 \leq i < n$, all of the uninterpreted symbols of I_i are both in I_{i-1} and in I_i.

Abstraction refinement is not addressed in detail in [4], as the algorithm is orthogonal to the specific refinement strategy used. In our work, we use two interpolation-based abstraction refinement strategies. For that, we use an interpolating SMT solver that has the ability to compute an interpolant for unsatisfiable formulas of certain theories, e.g. $\mathcal{LA}(\mathbb{Z})$.

In the case of the "refute all paths" strategy the Γ sequence we create an interpolant for is

$$I(\mathbf{x}), T\left(\mathbf{x}, \mathbf{x}'\right), T\left(\mathbf{x}'', \mathbf{x}^{(3)}\right), \dots, T\left(\mathbf{x}^{(n-1)}, \mathbf{x}^{(n)}\right), \neg P(x^{(n)}).$$

In the case of the "refute specific path" strategy it is

- $I(\mathbf{x}) \wedge \Lambda(\hat{s}_0[\hat{x}], x))$,
- $T\left(\mathbf{x}, \mathbf{x}'\right) \wedge \Lambda(\hat{s}_1[\hat{x}], x)$,
- $T\left(\mathbf{x}', \mathbf{x}''\right) \wedge \Lambda(\hat{s}_2[\hat{x}], x)$,
- $T\left(\mathbf{x}'', \mathbf{x}^{(3)}\right) \wedge \Lambda(\hat{s}_3[\hat{x}], x)$,
- $\dots$
- $T\left(\mathbf{x}^{(n-1)}, \mathbf{x}^{(n)}\right) \wedge \Lambda(\hat{s}_n[\hat{x}], x)$,
- $\neg P\left(\mathbf{x}^{(n)}\right)$

In both cases the common elements of the consecutive formulas of Γ have the same number of primes. Therefore in the interpolant Θ, I_i will refer only to $x^{(i-1)}$.

Then adding the predicates in Θ to Π refines the abstraction enough that the bad state at the end of the counterexample becomes unreachable in n steps and in the "refute all paths" case all of the bad states become unreachable in n steps.

3.3 Application to Software

Imperative programs can be conveniently modeled using control flow automata. The original version of PDR however works on symbolic transition systems. To use PDR for proving the safety of CFA, we convert them to STS first.

Given the CFA $A = (L, E, \ell_0, \ell_e)$ we say that the STS $W = (I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'), P(\mathbf{x}))$ constructed such that

- $I(\mathbf{x}) = (l = \ell_0)$,
- $T(\mathbf{x}, \mathbf{x}') = \left(\bigvee_{(\ell,t,\ell')\in E} l = \ell \wedge \delta_t \wedge l' = \ell'\right)$ where δ_t is the transition formula of the statement t as defined in Section 2.2.4,
- $P(\mathbf{x}) = (l \neq \ell_e)$,

where l is the element of $\mathbf{x}$ used to represent control flow location, is the equivalent STS of A.

Given a path $\mu = e_1 e_2 \dots e_n$ where $e_i = (\ell_i, t_i, \ell_{i+1})$ in the CFA $A = (L, E, \ell_0, \ell_e)$, we say that a path $\pi_\mu = s_1 s_2 \dots s_{n+1}$ in the equivalent STS of A is a corresponding path to μ iff

1. for all $1 \leq i \leq n+1$, $s_i \models l = \ell_i$, and
2. for all $1 \leq i \leq n$, $s_i \cup s'_{i+1} \models \delta_{t_i}$.

Proposition 3. *A path $\mu = e_1 e_2 \dots e_n$ in the CFA $A = (L, E, \ell_0, \ell_e)$ is feasible iff it has a corresponding path $\pi_\mu = s_1 s_2 \dots s_{n+1}$ in the equivalent STS.* ▪

Proof. Let all $e_i = (\ell_i, t_i, \ell_{i+1})$.

First we prove that if it has a corresponding path π_μ, then μ is feasible. According to the definition of feasibility in Section 2.2.4, μ is feasible iff the formula

$$\bigwedge_{i=1}^{n} \delta_{t_i}^{(i-1)}$$

is satisfiable. Since π_μ is a path corresponding to μ,

$$\bigcup_{i=1}^{n+1} s_i \models \bigwedge_{i=1}^{n} \delta_{t_i}^{(i-1)},$$

as for all $1 \leq i \leq n+1$, $s_i \cup s'_{i+1} \models \delta_{t_i}$.

Now we prove that if μ is feasible, then there is a corresponding path π_μ. Since μ is feasible, there is an assignment α under which

$$\alpha \models \bigwedge_{i=1}^{n} \delta_{t_i}^{(i-1)}.$$

α assigns value to $\mathbf{x}, \mathbf{x}', \mathbf{x}'', \dots, \mathbf{x}^{(n)}$. Divide α into n separate assignments $\alpha_1, \alpha'_2, \alpha''_3, \dots, \alpha_n^{(n-1)}, \alpha_{n+1}^{(n)}$ such that $\alpha_i^{(i-1)}$ assigns value to $\mathbf{x}^{(i-1)}$. For all $1 \leq i \leq (n+1)$ let $s_i = \alpha_i \cup \{l \to \ell_i\}$. Now condition 1 of the corresponding path definition holds, as for all $1 \leq i \leq n+1$, $s_i \models l = \ell_i$, since $s_i[l] = \ell_i$. Moreover, condition 2 holds, as for all $1 \leq i \leq n$, $s_i \cup s'_{i+1} \models \delta_{t_i}$, since they were created from a satisfying assignment.

What remains to be proven is that $\pi_\mu = s_1 s_2 \dots s_{n+1}$ does form a path in the STS equivalent to A. s_i are states, in the STS, as they are assignments, and they assign value to the variables of A and the location variable l. For a sequence of states to be a path in a transition system, $(s_i, s_{i+1}) \in T$ has to be true for $1 \leq i \leq n$, where T is the set fof allowed transitions in the transition system. The transition set of the transition system underlying the equivalent STS is defined as

$$T = \left\{ (s, s') | s \cup s' \models \left(\bigvee_{(\ell,t,\ell') \in E} l = \ell \wedge \delta_t \wedge l' = \ell' \right) \right\}.$$

Since, as a consequence of condition 1 and 2, for each $1 \leq i \leq n$,

$$s_i \cup s'_{i+1} \models (l = \ell_i \wedge \delta_{t_i} \wedge l' = \ell_{i+1}),$$

where $(\ell_i, \delta_{t_i}, \ell_{i+1}) \in E$, the subsequent elements of π_μ are in the transition set. □

Proposition 4. *Given the CFA $A = (L, E, \ell_0, \ell_e)$ and its equivalent STS, $W = (I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'), P(\mathbf{x}))$, every* initial *path $\pi = s_1 s_2 \dots s_{n+1}$ is the corresponding path of a feasible path μ in A.* ▪

Proof. Since π is a path, for $1 \leq i \leq n$,

$$(s_i, s_{i+1}) \in \left\{ (s, s') \mid s \cup s' \models \left(\bigvee_{(\ell,t,\ell') \in E} l = \ell \wedge \delta_t \wedge l' = \ell' \right) \right\}.$$

Which means that

$$s_i \cup s'_{i+1} \models \left(\bigvee_{(\ell,t,\ell') \in E} l = \ell \wedge \delta_t \wedge l' = \ell' \right).$$

From the semantics of $\vee$, this means that there is an edge $(\ell_i, t_i, \ell_{i+1}) \in E$ for all subsequent states (s_i, s_{i+1}) such that $s_i \cup s'_{i+1} \models (l = \ell_i \wedge \delta_{t_i} \wedge l' = \ell_{i+1})$.

The path is an initial path, therefore $s_1 \models l = \ell_0$ and the source location of e_1 is ℓ_0. In all of the states, l can only be assigned a single value, and since $=$ is transitive, the target location of ith edge is the same as the source location of the $(i+1)$-th edge. These are the two conditions for $\mu = e_1 e_2 \dots e_n$ to be a CFA path, therefore it is a CFA path.

Since for all $1 \leq i \leq n$, $s_i \cup s'_{i+1} \models (l = \ell_i \wedge \delta_{t_i} \wedge l' = \ell_{i+1})$,

1. for all $1 \leq i \leq n+1$, $s_i \models l = \ell_i$ and
2. for all $1 \leq i \leq n$, $s_i \cup s'_{i+1} \models \delta_{t_i}$,

which means that π is the corresponding path to μ. □

Proposition 5. *The CFA $A = (L, E, \ell_0, \ell_e)$ is safe iff its equivalent STS, $W = (I(\mathbf{x}), T(\mathbf{x}, \mathbf{x}'), P(\mathbf{x}))$ is safe.* ▪

Proof. If A is safe, then there are no feasible error paths in it, therefore there are no initial error paths in W, which means that it is safe. Assume that there is an initial error path π in W. π is an error path, therefore its last state s_n is an error state, which means that $s_n \models (l = \ell_e)$. There is a path μ in A to which π corresponds. Thus, the target location of the last edge of μ is ℓ_e, which means that μ is an error path, and since it has an STS path corresponding to it, it is feasible. Therefore A is not safe.

If W is safe, then there are no initial error paths in it, therefore there are no feasible error paths in A, which means that it is safe. Assume that there is a feasible error path μ in A. Since μ is feasible, it has a corresponding path π in W. μ is an error path, which means that the target node of its last edge is ℓ_e. π is the path corresponding to μ, therefore its last state s_n must be such, that $s_n \models (l = \ell_e)$, which means that s_n is an error location, π is an initial error path, and W is not safe. $\square$

Therefore we can check the safety of a CFA, by creating its equivalent STS, and running PDR on that.

Chapter 4

Implementation

We have chosen to implement the algorithm in the THETA framework in Kotlin.

4.1 The Theta Framework

THETA is a framework for model checking. It offers frontends to various languages and translates those languages to inner representations. It has a versatile library of tools for the inner representations. It also provides an interface to the SMT solver, implemented as a facade over the Z3 solver.

An overview of THETA can be seen in Figure 4.1.

4.2 Kotlin

Kotlin is a programming language that offers modern syntax and features, and can be compiled to run on multiple platforms: JVM, JavaScript, and they are working on a native compiler. The JVM version is fully compatible with Java with minor nuisances, mostly when it comes to dealing with static methods or objects. We exploit this compatibility, because THETA is implemented in Java.

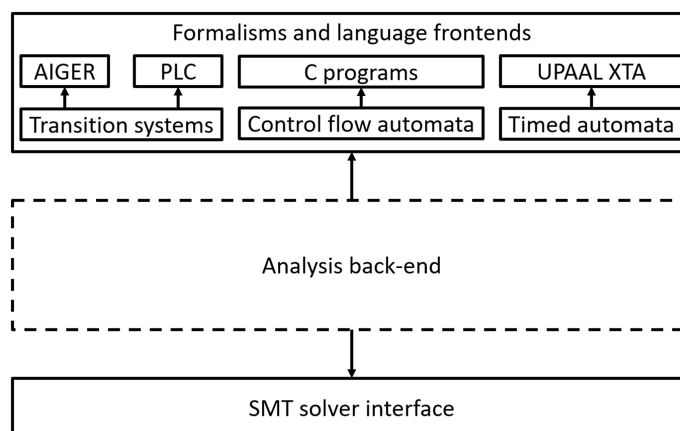

Figure 4.1: An overview of the architecture of THETA
Based on similar figure in [9]

```
//Note the absence of semicolons, they are usually not necessary in Kotlin
var a : String //Non-nullable type
a = null //Compile error
var b : String? //Nullable type
println(b.length()) //Compile error

if(b != null){
    println(b.length()) //No error
}

fun process(command : String){ //Function with non-nullable parameter
    execute(command)
}

fun printMessage(message : String?){ //function with nullable parameter
    println(message?:"Message has not arrived.") //Replaces message with the given string if null
}
```

Listing 4.1: Nullable and non-nullable types in Kotlin

```
fun <T>countSmaller(bound: T, list: List<out Comparable<T>>): Int {
    var count = 0;
    for (e in list) {
        if (e < bound) { //The compiler knows to call the Comparable interface
        ++count;
        }
    }
    return count;
}
```

Listing 4.2: Operator overloading in Kotlin

4.2.1 Null pointer safety

Kotlin tackles null pointer exceptions on the type level. It has separate separate types for references that can and can not be null. To call a method on a variable of a nullable type, one has to check if the variable is null before calling the method. The compiler can smart-cast the variable to a non-nullable type (if the variable cannot be accessed from another thread) after a non-null check, and its methods can be invoked.

Another benefit is the ability to write methods that take non-null parameters. That way the programmer gets a compile-time error if they pass a nullable variable as parameter.

An example can be seen in Listing 4.1.

4.2.2 Operator overloading

Kotlin allows operator overloading. It can improve the readability of the code, and make it less of a hassle to write. One such case can be seen in Listing 4.2.

Kotlin overloads more operators than Java out of the box, and programmers can overload, or define new infix operators themselves.

```
//Creating and initializing a mutable list of mutable lists of immutable lists of expressions.
//Kotlin infers the type of trace, and in most cases the template parameter of the functions.
val trace = mutableListOf(
    mutableListOf(listOf(sts.init)),
    mutableListOf(listOf<Expr<BoolType>>())
)
```

Listing 4.3: Type inference in Kotlin

4.2.3 Type inference

Kotlin can infer the type of the variable being created from the right-hand side of the assignment. This reduces the amount of code to manage, without substantial loss of readability (IDEs also support it well). It can also infer the template parameter of a function from the type of the parameter passed. An example can be seen in Listing 4.3.

4.3 Our Implementation

We represent sets of states by cubes and clauses. A cube is a conjunction of literals $l_1 \wedge l_2 \wedge \cdots \wedge l_n$, and a clause is a disjunction of literals $l_1 \vee l_2 \vee \cdots \vee l_n$. As a consequence of De Morgan's law, the negation of a cube is a clause: $\neg(l_1 \wedge l_2 \wedge \cdots \wedge l_n) = (\neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_n)$, and vice versa: $\neg(l_1 \vee l_2 \vee \cdots \vee l_n) = (\neg l_1 \wedge \neg l_2 \wedge \cdots \wedge \neg l_n)$.

To ease predicate abstraction, we create a representing propositional variable for every atom used in the abstraction. When the abstraction is refined with a predicate p , a propositional variable R_p is created to represent it, and add the formula $p \leftrightarrow R_p$ to the solver. This formula remains in the solver until the program terminates. Thus in queries we can use R_p in place of the predicate.

From the model of a satisfiable query, which is an assignment to the variables in the query, we create a cube. The variables in the query include only the propositional variables created to represent formulas, therefore all the variables are assigned either `true` or `false`. For each predicate p, if its representative variable R_p is assigned `true`, then we add p to the cube, if R_p is assigned `false`, then we add $\neg p$ to the cube.

When a cube is blocked, its negation is added to the element of the trace it is blocked from. An element of the trace is therefore a conjunction of clauses, it is in conjunctive normal form (CNF).

When the algorithm proves the system correct, one of the elements of the trace become a safety invariant. Since that is in CNF, the algorithm produces the safety invariant in CNF.

We represent the trace in a way that exploits the property that earlier frames imply the latter. When we block a cube, we only add it to the latest frame from which it is blocked. Therefore the formula of the ith frame is the conjunction of all the clauses in the ith and later frames. The initial frame is an exception, as its formula is known exactly and cubes are not blocked from it. We also have an "invariant" frame, where we add blocked clauses when they are proven unreachable without restriction to the number of steps. New frames are always added before the invariant frame, so that it stays the last element of the trace.

4.3.1 Data classes

Instances of the `Literal` class represent literals. They contain the represented atom, its representative variable, and a boolean of whether it is negated or not. They are immutable, but have a property that returns their negated version.

Instances of the `Clause` and `Cube` classes represent clauses and cubes. They contain a set of literals. They are immutable, they implement the (immutable) `Set` interface [1] by delegating it to the contained set of literals. They can return their negation, an instance of the other class with each of the literals negated. Clauses have a method `subsumes` that checks syntactically if a clause implies another. If c_1 contains all the literals in c_2, or in other words c_1 subsumes c_2, then c_2 implies c_1, as for every interpretation $\mathcal{I}$, if $\mathcal{I} \models c_2$, then there is a literal l in c_2, for which $\mathcal{I} \models l$, and since l is also a literal of c_1, $\mathcal{I} \models c_1$.

All three of these classes are immutable, therefore their properties can be `lazy`. "Lazy" properties are calculated only when they are first accessed, then their value is stored.

Instances of the class `ProofObligation` represent proof obligations. They contain a cube, the number of steps the cube must be proven unreachable in, and a reference to the `ProofObligation` instance that caused it. The reference is used to create counterexamples: when a proof obligation reaches the initial frame, the path leading from it to a bad state can be traced. When a proof obligation is created from a bad state, the reference is set to null.

Instances of the `Frame` class represent elements of the trace. They contain a mutable list of clauses, which are the negations of blocked cubes. They implement the `MutableList` interface by delegating it to the contained list of clauses. There are three noteworthy methods in the class. `add` is overridden from the `MutableList` interface and not delegated. It iterates over the already learned clauses and if any of them subsumes the element to be added, then they are dropped. Dropping them simplifies the formula, but does not change the set of interpretations satisfying it, as c_1 implies c_2, therefore $c_1 \wedge c_2$ is equivalent to c_1. `dropIfSubsumes` simplifies the frame when a more general clause is to be added to a later frame. It drops all the clauses that are subsumed by the clause to be added. `ensures` is used to check syntactically if a cube is blocked, so that the more expensive semantic check using the solver can be possibly spared. It takes the negation of the cube to be blocked, a clause. If that clause subsumes any of the clauses already learned, then the cube is blocked in that frame, because the frame's formula implies the negation of the cube.

An instance of the `Trace` class represents the trace. It stores a list of `Frame` instances, separately from it, the initial formula and a special `Frame` instance, the invariant frame. Its method `blockObl` takes a proof obligation that is proven unreachable and adds the cube's negation to the appropriate frame. It also removes unnecessary clauses from earlier frames using `dropIfSubsumes`. The `safetyInvariant` method calculates the list of clauses in the safety invariant, when the system is proven safe. Since we add the learned clauses to only the last frame at which they hold, the method looks for the last empty frame, as that has the same clauses as the next. The safety invariant is the conjunction of all the clauses in that frame and all frames after it, including the invariant frame.

[1] Kotlin has both immutable and mutable versions of every collection. The name of the mutable versions start with `Mutable`, such as `MutableSet`.

4.3.2 Handling the Solver

The solver we use (Z3) has a stack-like behaviour. We can add multiple formulas to it, and it will check if their conjunction is satisfiable. We can remove formulas in a last-in-first-out way as layers of the stack.

In our implementation we use two layers. On the bottom layer are the formulas that are kept there long-term, such as $R_p \leftrightarrow p$ for predicates and their representative variables. On the top layer are the formulas of the specific query.

For elements of the trace, we create propositional variables, we call activation variables. The ith frame's activation variable is F_i. When a clause c is added to the ith frame, we add the formula $F_i \rightarrow c$ to the bottom level of the solver's stack. And when we want to use the ith frame in a query, we refer to it by F_i. Because of the way we represent frames, we have to also add every F_j where $i < j$. The invariant frame does not have an activation literal, clauses added to the invariant frame are added to the bottom level, since they always hold. The `getBadCube` method returns either a bad cube at the given frame, or `null` if there is none. It adds to the top level of the solver all activation literals after the requested one, and adds the negation of the property formula. If the formula is satisfiable, then it creates a cube from the assignment of the representative variables. The cube has a literal for every predicate in the current abstraction, if its representative variable is assigned `false`, then the literal is negated, otherwise it is not. If the formula is unsatisfiable, it returns `null`.

The `solveRelative` method checks if a proof obligation can be blocked, and returns either a general version that can be blocked, or a general predecessor. It adds the formula corresponding to the chosen query to the solver, and checks it.

If the formula is satisfiable, then it creates a cube from the assignment to the predecessor variables, and then generalizes it by attempting to leave out the literals one-by-one. It can leave a literal from the cube, iff every state satisfying the remaining cube has a successor satisfying the cube of the proof obligation. This is checked syntactically, using a method similar to ternary simulation. If the formula can be simplified to $\top$ even after leaving a variable out from the assignment, then the variable is removed from the cube. The simplification of the formula is done with general rules such as $x \wedge \bot \leftrightarrow \bot$ and $x \vee \top \leftrightarrow \top$ regardless of the value of x.

If the formula is unsatisfiable, then it generalizes the cube of the proof obligation. It tries to leave out literals from the cube one-by-one, that are not in the UNSAT core. If leaving a variable causes the remaining cube to intersect with the initial states, then the variable is kept, since blocking a cube that intersects with the initial states would break the invariant that the earlier states are subsets of the latter. It also checks which frame's activation variable is in the UNSAT core. The generalized cube is blocked from the frame after the earliest used.

The `refine` method is used to refine the abstraction. It takes an abstract counterexample as parameter. Depending on the strategy used, it either checks the counterexample, or all paths of the given length in the concrete state space using a different solver instance. If the path is feasible, then it returns false. If the path is not feasible, then it extracts the predicates from the interpolant of the path, refines the abstraction with them and returns true.

Table 4.1: Results of running the algorithm with “Refute all paths” strategy.

	All	Safe	Unsafe
Average	2.56 s	2.61 s	0.77 s
Median	2.07 s	2.13 s	0.76 s
Min	0.74 s	1.3 s	0.74 s
Max	14.06 s	14.06 s	0.77 s

Table 4.2: Results of running the algorithm with “Refute specific path” strategy.

	All	Safe	Unsafe
Average	5.9 s	6.04 s	0.74 s
Median	3.62 s	3.67 s	0.74 s
Min	0.72 s	1.35 s	0.72 s
Max	32.4 s	32.4 s	0.77 s

4.4 Evaluation

We ran our implementation on a set of benchmarks developed for the International Competition on Software Verification (SV-COMP) [7]. We measured the time it took to complete each instance using the “refute all paths” and using the “refute specific path” refinement strategy. The results are presented in Table 4.1, Table 4.2, Figure A.1.1 and Figure A.1.2. The “refute all paths” strategy seems to perform better in our benchmarks, but we do not draw further conclusions from that.

Chapter 5

Conclusions

In this thesis, we have examined IC3, a hardware model checking algorithm, that checks reachability properties in symbolic transition systems. We have implemented PDR, an optimized version of IC3. We have also implemented an extension of IC3 that enables the efficient handling of infinite state-space models using implicit predicate abstraction. Moreover, to enable the algorithm to check reachability properties in control flow automata, we have implemented a transformation that constructs a symbolic transition system equivalent to the input automaton. Finally, we evaluated our implementation by measuring the time it takes to solve a set of benchmarks developed for SV-COMP [7].

Future work Our implementation is single-threaded, and thus cannot exploit the advantages of a multi-processor system. PDR can be parallelized, either by blocking multiple bad states from a frame simultaneously, or blocking multiple predecessors simultaneously. This could potentially improve the performance of the algorithm on multi-processor systems.

Our approach of running the algorithm on control flow automata creates a less structured model and checks the properties on that. Comparing the performance of our algorithm to other IC3-like algorithms that exploit the structural information in CFA, such as in [3, 6] would also be interesting.

Bibliography

[1] Aaron R. Badley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011. DOI: 10.1007/978-3-642-18275-4_7.

[2] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer, 2007. DOI: 10.1007/978-3-540-74113-8.

[3] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *Computer Aided Verification*, volume 7358 of *LNCS*, pages 277–293. Springer, 2012. DOI: 10.1007/978-3-642-31424-7_23.

[4] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 46–61. Springer, 2014. DOI: 10.1007/978-3-642-54862-8_4.

[5] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design*, pages 125–134. FMCAD Inc, 2011.

[6] Tim Lange, Martin R Neuhauber, and Thomas Noll. IC3 software model checking on control flow automata. In *Formal Methods in Computer-Aided Design*, pages 97–104. FMCAD Inc, 2015. DOI: 10.1109/FMCAD.2015.7542258.

[7] Ludwig-Maximilians-Universität München Software and Computational Systems Lab. Collection of verification tasks. https://github.com/sosy-lab/sv-benchmarks/tree/d16580bde3c8e47e12f8fc3a8843178e841ac83f/c/locks. Accessed on 07.12.2018.

[8] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006. DOI: 10.1007/11817963_14.

[9] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In *Formal Methods in Computer-Aided Design*, pages 176–179. FMCAD Inc, 2017. DOI: 10.23919/FMCAD.2017.8102257.

Appendix

A.1 Evaluation results

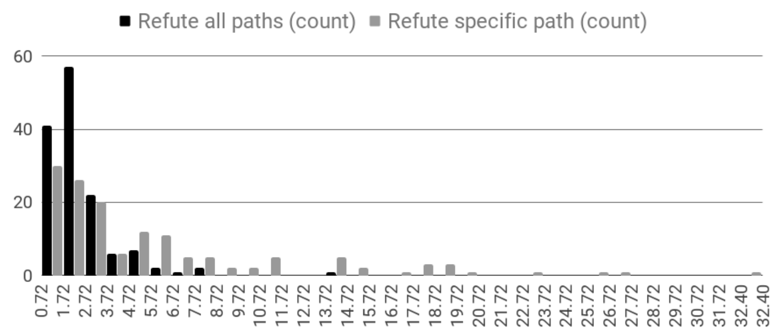

Figure A.1.1: Histogram of the time taken to run the algorithm with both configurations

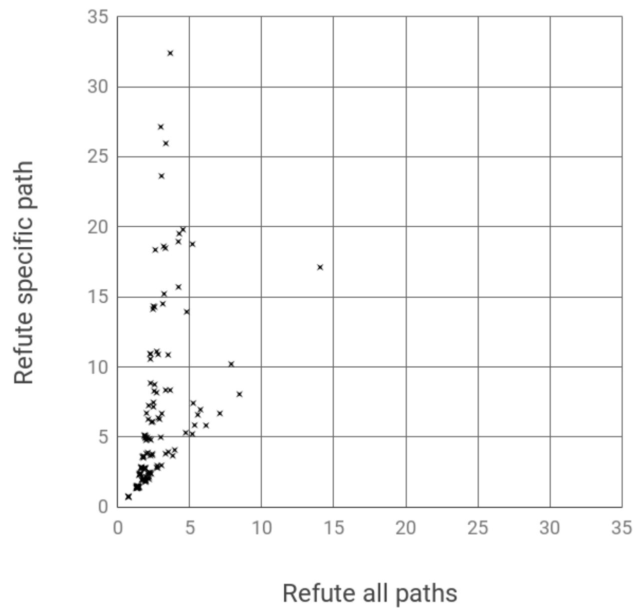

Figure A.1.2: Scatterplot of the time taken to run the algorithm with the "refute all paths" strategy vs. the "refute specific path strategy" on certain instances